The Lemmings Puzzle: Computational Complexity of an Approach and Identification of Difficult Instances

by Kristian Spoerer, BSc

Thesis submitted to The University of Nottingham
for the degree of Doctor of Philosophy, 2007

Abstract

Artificial Intelligence can be thought of as the study of machines that are capable of solving problems that require human level intelligence. It has frequently been concerned with game playing. In this thesis we shall focus on the areas of search and complexity, with respect to single-player games. These types of games are called puzzles.

Puzzles have received research attention for some decades. Consequently, interesting insights into some of these puzzles, and into the approaches for solving them, have emerged. However, many of these puzzles have been neglected by the artificial intelligence research community. Therefore, we survey these puzzles in the hope that we can motivate research towards them so that further interesting insights might emerge in the future.

We describe research on a puzzle called LEMMINGS that is derived from a game called Lemmings, which itself is in NP-Complete. We attempt to find the first successful approach for LEMMINGS. We report on a successful approach to a sub-problem called α-LEMMINGS, and an investigation into different initialisation schemes for the approach. We also report a similar approach for the general LEMMINGS problem, and an investigation in order to find the best parameters to use for this approach to LEMMINGS. Furthermore, we outline the time and space requirements for this approach.

Previous work has shown that difficult problems, which are contained in the set of NP-Complete problems, are sometimes easy to solve in general. The existence of a small number of hard instances of these problems yields their high difficulty. Previous work has also shown where the difficult instances are located. We report on an investigation that aims to find difficult instances of LEMMINGS. Future investigations, for example algorithm testing on benchmark problems, might make use of our difficult LEMMINGS instances.

Acknowledgements

I acknowledge the following people for their contributions to my research. My supervisor Professor Graham Kendall has offered sound advice throughout my research. Dr Louise Dennis originally refered me to the use of game of Lemmings as an artificial intelligence problem. Martin Geiger brought to my attention the work on Sokoban that helped me to change my angle of approach to research. Dr Yong Yang and Dr Steven Gustafson originally introduced me to NP-Complete problems. Ross O'Brien has aided my mathematical understanding. Dr Andrew Parkes has provided plenty of insight into my research and pointed me in the right direction with regards to hard satisifiability problems. I have had many helpful discussions with Jools Greensmith. Many thanks to the reviewers of conferences to which I have submitted articles for their helpful advice. Many thanks also to Professor Edmund Burke for giving me the opportunity to carry out this work. Thank you also to all of the other students, researchers and members of staff at the University of Nottingham who have been a part of this work.

This thesis is dedicated to Patrick, Glenalee, Charlotte, Harriet, Fredrick and Frauke.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

An introduction to the thesis

"Let's go!"
Lemming

1.1 Context of the Thesis

This thesis is framed within the general context of **Artificial Intelligence (AI)**. AI can be thought of as the study of machines that are capable of solving problems that require human level intelligence. People have enquired of AI for centuries (for example, see Mary Shelley's story about Frankenstein). However, AI as we know it today emerged in the 1940s when computers were built in order to decode communications sent by an enemy. AI encompasses many areas of study, including search, logic, planning, learning and complexity, amongst others. See [1] for a good AI textbook, and also [2] and [3]. In this thesis we shall focus on search and complexity with respect to games.

Search is a set of methods for finding solutions to complicated problems. One method involves a tree of actions and looks ahead in order to ascertain the outcome of different sequences of actions in order to locate a sequence that yields a good outcome. Other methods work over a single solution and attempt to optimise it. Some of the earliest work in search was carried out in the 1950s with the Logic Theorist (see Newell et al. [4]), which found proofs to logical theorems. Research focused on search is still active (see, for example, [5] and [6]). This thesis tends towards the tree view of search.

Complexity is the study that involves analysing the efficiency of algorithms. That is, algorithms are judged by how much time and memory they require when they are executed on a computer in order to solve a problem. An important area of complexity is concerned with complex problems that are thought to be difficult for any algorithm to solve. These problems are contained within a set of problems called NP-Complete, which was first recognised by Cook [7] in 1971. Interestingly, these problems are often only made difficult by a few special instances. That is, these problems are often simple to solve, except for a small number of difficult cases. Therefore, research is focused towards locating these difficult instances (see for example Cheeseman et al. [8]).

Games have been a central theme of AI since the 1950s, when Shannon [9] developed a program for playing chess. Research on Games is still active [10]. Schaeffer [11] provides a good survey of research towards two player games like chess. Other multi player games have been researched, for example, Othello [12], Hex [13], Shogi [14], Go [15], Backgammon [16], Poker [17], Scrabble [18] and Checkers [19]. This thesis is focused on single player games, which are sometimes called puzzles. Puzzles have been researched for some decades (see [20]) and are often contained in NP-Complete.

1.2 Basis of the Thesis

This thesis is motivated by the following three quotes.

"*The computer game Lemmings can serve as a new Drosophila for AI research*", John McCarthy [21].

Here, McCarthy suggests that a popular computer game called Lemmings™ (trademark of Sony Computer Entertainment) can be a new Drosophila for AI research. This title has traditionally been held by chess. Yet despite McCarthy's introduction of Lemmings, we do not know of any empirical evidence that a computer can produce solutions for this problem. Lemmings arguably deserves the kind of research attention that chess has received.

"*The fact that deciding whether a level* (of Lemmings) *is completable is NP-Complete suggests that it will be a big challenge for any system to guarantee to solve any level*", Graham Cormode [22].

Here, Cormode concludes that Lemmings is an NP-Complete problem, and therefore is an interesting and challenging problem for AI.

"*'Where are the really hard instances of NP problems?' Can a subclass of problems be defined that is typically (exponentially) hard to solve, or do worst cases appear as rare 'pathological cases' scattered unpredictably in the problem space?*" Cheeseman et al. [8].

Here, Cheeseman et al. are motivating their work that attempts to locate hard instances of problems.

In this thesis we shall describe research on a puzzle called LEMMINGS, which is derived from Lemmings, whereby we apply a search algorithm in order to locate difficult instances. It is hoped that this will provide motivation for other researchers to take up the challenge and tackle this puzzle.

1.3 Goals of the Thesis

There is very little work surrounding Lemmings other than McCarthy's proposal of the problem as a new Drosophila of AI and Cormode's NP-Completeness illustration. Lemmings, and its derivatives, are relatively new problems, which are ready to challenge AI researchers. This thesis is focused on the Lemmings game as a problem class.

We have the following research goals

1. We aim to motivate work on NP-Hard puzzles, in particular the game of Lemmings.

   It is our hope that, by motivating work on puzzles, further research might provide more insight than has already been gained.

2. There are no empirical results to show that a computer can produce solutions to LEMMINGS. Therefore, we aim to show the first results for producing feasible solutions to instances of LEMMINGS.

   If we are able to find a successful approach to LEMMINGS then future work on the problem would have a basis from which to work.

3. Very little is understood about the LEMMINGS landscape, by which we mean the landscape of instances that are captured by the LEMMINGS definition. Therefore, we aim to learn something about that part of the LEMMINGS instance landscape, consisting of those instances that have been solved by a computer, in terms of the difficulty for the given algorithm to produce a solution to these instances. That is, we aim to show, for the first time, the location of the more difficult LEMMINGS instances. Moreover, we aim to describe a method for generating the difficult instances.

   Future work, for example testing the performance of algorithms, might find the ability to generate difficult LEMMINGS instances helpful.

4. We aim to evaluate the amount of time and memory required by our algorithm for producing feasible solutions to instances of LEMMINGS.

   If we are able to evaluate the first algorithm for approaching LEMMINGS, then future approaches would have a comparison to measure against.

1.4 Outline of the Thesis

The thesis is organised as follows. In chapter 2 we provide a theoretical basis on which to support the arguments and claims of the thesis. We define the basic concepts of problem, solution, and algorithm that will be used throughout the thesis, and provide an outline of algorithm analysis and problem analysis. We describe a background on the theory of NP-Completeness, a description of the $P = NP$ (or $P \neq NP$) debate, which is very relevant to contemporary mathematical science, and probably the most important open question in computer science. We also discuss various approaches to solving the difficult NP-Complete problems.

In chapter 3 we outline search algorithms. We describe various uninformed algorithms and detail their complexities, using algorithm analysis as defined in chapter 2. We then discuss some informed algorithms, including constructive and local varieties. We also define the concepts of state space and search space, and we highlight issues relating to heuristic functions. We also briefly discuss the No Free Lunch Theorem.

In chapter 4 we provide a survey of NP-Hard puzzles. This survey is intended to motivate research in the area of NP-Hard puzzles by highlighting some gaps in that area. Furthermore we intend to motivate research on the Lemmings game by classifying it as a self updated puzzle. The puzzles are categorised into player updated and self updated. The difference between the two types of puzzle is that in self updated puzzles the state will change even if the player chooses to pass, whereas in a player updated puzzle the state will remain unchanged until the player moves. For each puzzle we provide a brief description of the rules, and outline its inclusion, and also the inclusion of its sub-problems, in NP-Hard, NP-Complete or P. We provide a brief outline of some of the work that has been published around each puzzle in order to highlight which of those constitute a gap in the literature and might benefit from future research, and we offer additional sources of information that might be useful to a researcher who is interested in tackling one of these problems. Chapter 4 can be seen as an extension of Demaine's survey of puzzles from [23] and, as such, this chapter is a contribution to the existing body of knowledge.

In chapter 5 we outline the LEMMINGS problem definition, and how it relates to

Lemmings, from which it is derived. In particular, we shall define action, instance, solution, feasible solution and outcome with respect to LEMMINGS. We also highlight a history of Lemmings, and suggest that LEMMINGS is also in NP-Complete.

In chapter 6 we discuss various possible approaches to computing feasible solutions for LEMMINGS instances. The motivation for this chapter is to find a successful way to approach LEMMINGS, because there is no known successful approach for LEMMINGS. We restrict ourselves to search and offer a theoretical discussion of some approaches. We describe a successful approach to a sub-problem of LEMMINGS, called α-LEMMINGS, and describe an investigation into different initialisation schemes for the approach to α-LEMMINGS. This work has been published in [24]. We also describe a similar approach for the general LEMMINGS problem. The approach for LEMMINGS will receive a small investigation in order to find the best parameters to use.

In chapter 7 we show where the more difficult LEMMINGS instances are. We use our approach to LEMMINGS, that is detailed in chapter 6, in order to find difficult instances. Another motivation for this chapter is to analyse the performance of our approach for LEMMINGS.

Finally, in chapter 8, we conclude the thesis.

Chapter 2

Problems, Algorithms and Complexity

A theoretical grounding for our thesis

"I would not give a fig for the simplicity this side of complexity,
but I would give my life for the simplicity on the other side of complexity."
Oliver Wendell Holmes

In this chapter we provide a theoretical basis on which to support the arguments and claims of the thesis. We shall define the basic concepts of problem, solution, and algorithm that will be used throughout the thesis. We then provide an outline of algorithm analysis that allows algorithms to be analysed in terms of the time and space complexity required in order to find a solution. We also outline problem analysis, which allows us to analyse the time and space complexity of problems. Algorithm analysis is used particularly in chapter 3 in order to analyse the time and space required by some search algorithms, and also in chapter 6 so that we can analyse the potential success of applying some algorithms to the LEMMINGS puzzle. This chapter then describes a background on the theory of NP-Completeness, which will provide a foundation upon which to place our survey of NP-Complete puzzles in chapter 4, the definition of LEMMINGS in chapter 5 and investigation of LEMMINGS in chapters 6 and 7. This chapter then leads into a description of the P = NP debate, which is very relevant to contemporary mathematical science. Finally we discuss various approaches to solving the difficult NP-Complete problems, which can provide clues as to how we might approach LEMMINGS. The concepts in this chapter can be found in more detail in [25], [1] (Appendix A), [26] (Ch. 10) and [27].

2.1 Basics

A **problem** is a question that can be answered over some variables. For example, "what is $x + y$?". This particular problem is restricted to mathematics, and furthermore it is a restricted form of a more general problem, such as "what is $x\ z\ y$?", which itself is a restricted form of the more general five variable problem "$a\ b\ x\ z\ y$?". A general problem can be described by a finite set of variables "$v_1 \dots v_k$?", but we can more easily reason over problems that have been restricted, so some of the variables are made constant. An **instance** is then an assignment of values to the problem variables so that all terms are constant. For example, an instance of the problem "what is $x + y$?" might contain $x = 2$ and $y = 3$, giving "what is 2 + 3?". Figure 2.1 shows the genealogy tree of our simple problem, each depth in the tree represents a restriction of some variable to a constant. This is similar to Darwin's depiction of a tree of species in [28]. Note that any node that has no successors represents a problem instance, for example "what is 2 + 3?", and also that the root of the tree "$v_1 \dots v_k$?" has little meaning because it is simply a list of k variables. The root represents all of the problems and instances.

A **solution** is a possible answer to the problem question. Two solutions to "what is 2 + 3?" are 5 and 8. The first solution is correct. An **algorithm** is a step-by-step routine that takes a problem instance as its input and returns a solution to that instance. For example, an algorithm to solve "what is $x + y$?" might be a simple computer program that takes two inputs, x and y and returns $x + y$. An algorithm is said to **solve** a problem if it is guaranteed to return a correct solution to every

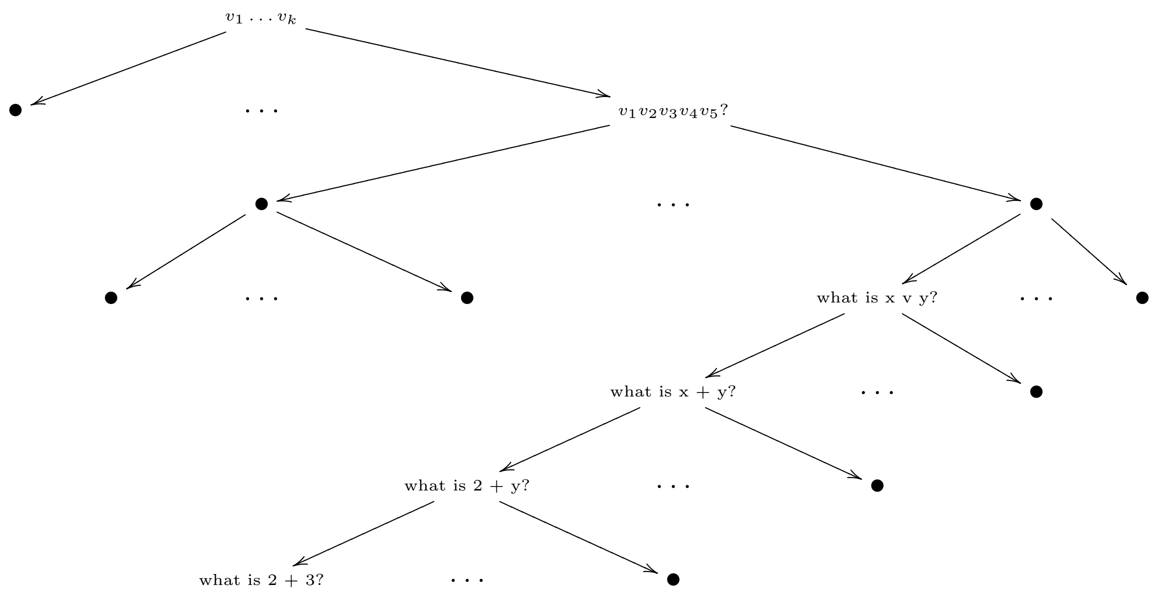


FIGURE 2.1: A condensed view of the genealogy of the problem "what is 2 + 3?".

instance of that problem.

2.2 Algorithm Analysis

Algorithm analysis is a method of evaluating the time and space required by an algorithm in order to solve a problem. We use **big-O notation** for this analysis. It is written $O()$. First, we abstract over the size of the instance, that is, the size of the input to an algorithm, and call this n. For example, if our problem is "how many elements are contained in the set S?", then S is passed as the input to an algorithm, and $n = |S|$. The algorithm performs one counting operation on each element in the set. We can easily see that this algorithm, with an input of size n, will not take more than n steps to return the number of elements in the set. We say that this algorithm, to solve this problem, has $O(n)$ complexity, or that the time taken by this algorithm to solve this problem grows linearly over n. This is a **linear-time algorithm**. Note that we say nothing about the exact time measure. Instead we offer an abstract indication of the number of computation steps required based on the size of the input n. We can interchange between the terms 'time' and 'steps' so

Instance size n	Linear-time $O(n)$	Polynomial-time $O(n^k)$ $k = 2$	Exponential-time $O(2^n)$
1	1	1	2
10	10	100	1,024
20	20	400	1,048,576
30	30	900	1,073,741,824

TABLE 2.1: Examples of algorithm complexity classes.

long as we understand that a step that requires one second to compute today, might take only one-hundredth of a second to compute in the future.

Suppose we have a different problem, for example, "for some set S, what is the sum of each element with each of the other elements?", then S is passed as the input to a new algorithm, and $n = |S|$. Consider the following algorithm

```
total = 0
for each s in S
     for each s' in S, such that s' nequal s
          total = total + s + s'
```

Notice the two `for each` loops. We can easily see that this algorithm, with an input of size n, will not take more than n^2 steps to return the correct solution. We say that this algorithm to solve this problem has $O(n^k)$ complexity for some known constant k, or that the time taken by this algorithm to solve this problem grows polynomially over n. This is a **polynomial-time algorithm**.

Table 2.1 shows the number of steps required by different classes of algorithms against a growing instance size. In $O()$ we analyse over an increasing n that asymptotically approaches infinity. Here we see that $O(n)$ algorithms eventually emerge to be faster than $O(n^k)$ algorithms (where $k > 1$). This table also introduces **exponential-time algorithms**. These algorithms have $O(2^n)$ complexity. Polynomial-time algorithms are much more desirable than exponential-time algorithms. As n increases and asymptotically approaches inifinity, the time required by any exponential-time algorithm to solve a problem instance of size n will eventually be more than that taken by any polynomial-time algorithm.

The analysis of the space required by an algorithm is done in the same way, except that we offer an abstract indication of the number of memory locations required based on the size of the input n.

2.3 Problem Analysis

There will often be more than one algorithm to solve a given problem and each algorithm will have its own unique complexity. Moreover, we might want to be able to analyse the difficulty of solving a given problem without restricting ourselves to a specific algorithm for solving it. We can perform this analysis over decision problems.

A **decision problem** is a problem that has only two possible solutions, either "yes" or "no". For example, "is there a z such that $3 + 2 > z$?". The correct solution is "yes". Despite having only two possible solutions, some decision problems might be very complex and require many algorithmic steps in order to find a correct solution. We can use algorithm analysis to asymptotically evaluate the time required to solve a decision problem using some algorithm. When we refer to the complexity of a problem, we are talking about the fastest known algorithm that solves it. In other words, all known algorithms but one might solve the problem in exponential-time, but if that one algorithm solves the problem in polynomial-time, then that problem can be solved in polynomial-time.

We can now identify an important class of decision problems called **P**. Refer to Figure 2.2, which shows the universe of decision problems. This figure is clearly not to scale, and is not likely to be in proportion. P contains those decision problems for which there exists a **deterministic** polynomial-time algorithm for finding a correct solution to any instance. Determinism in this sense is best described as that which is captured when any computer executes an algorithm step-by-step. P stands for polynomial. The decision problems in P are known as **tractable** problems, or efficiently solvable problems.

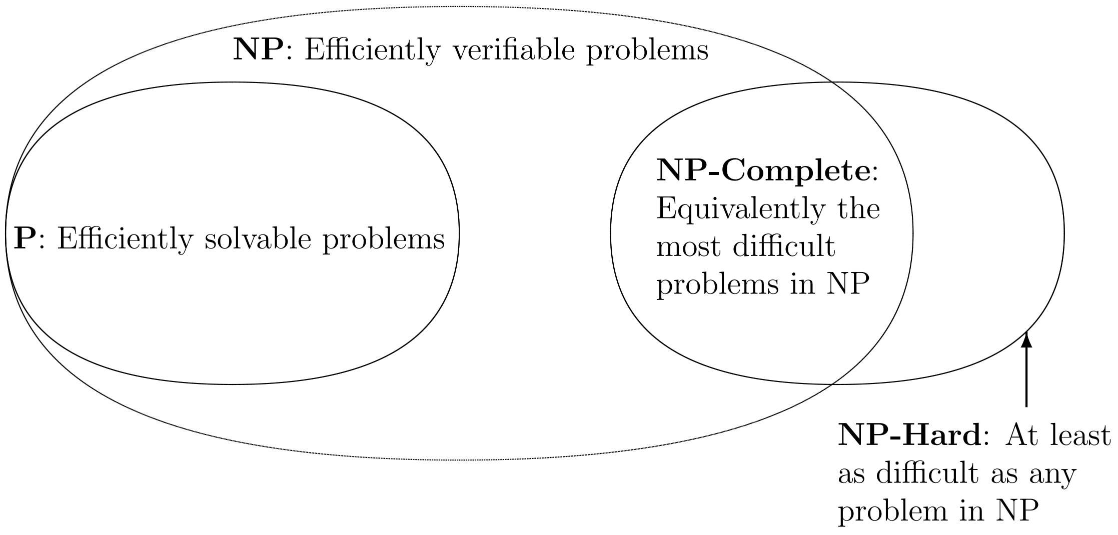

NP: Efficiently verifiable problems
P: Efficiently solvable problems
NP-Complete: Equivalently the most difficult problems in NP
NP-Hard: At least as difficult as any problem in NP

FIGURE 2.2: The universe of decision problems.

2.4 Problem Verification

Recall that a solution to some problem instance is either correct or incorrect. For example, given the problem instance "what is $2,005,675 + 3,543,672$?", and the solution "5,549,347", we do not know at a glance whether or not this is correct. We can **verify** the correctness of a solution using a verification algorithm. Once again, we can use algorithm analysis in order to asymptotically evaluate the time required by a verification algorithm, so that the fastest known verification algorithm defines the complexity of verifying a problem.

We can now identify another important class of decision problems called **NP**. Refer to Figure 2.2. NP contains those decision problems for which there exists a polynomial-time verification algorithm. Verifying a problem is very different from solving one. When we talk about polynomial-time verifiability, we say nothing about the amount of time that is required to find a solution. Even if a problem is verifiable in polynomial-time, or efficiently verifiable, it might actually require exponential-time to find a solution for that problem. Importantly, finding a correct solution actually involves iteratively sampling different solutions and verifying the correctness of each one.

Consider this problem. “Given a tree T, can a blind caterpillar, starting from the base of the tree locate an apple growing on one of the branches in the tree?”. Let us assume that the number of branches in the tree grows exponentially against the size, by which we mean height, of the tree. The blind caterpillar must try all of the branches in the worst case before possibly finding an apple. On the other hand, suppose that the blind caterpilar has a non-blind friend, who claims to have found the apple and to know the way to reach it. Further suppose that his friend shouts directions as to which path to take through the tree. Given some set of trees that increase linearly in size, if we record the time that it takes the blind caterpillar to climb each tree in order to verify that the route shouted to him by his non-blind friend takes him to an apple, and the time taken to verify each solution grows polynomially against the size of each tree, then this problem is in NP. That is, the problem is in NP because it can be verified in polynomial time. Importantly, the blind caterpillar is not searching for the apple, this would take exponential time, he is simply following his friends directions in order to verify that they lead him to an apple.

There are some problems that are not in NP. Consider this new problem. “Given a tree T, can a blind capterpillar, starting from the base of the tree, locate an apple on every branch in the tree?”. Let us assume again that the number of branches in the tree grows exponentially against the size of the tree. Even if the blind caterpillar is told the route to take through the tree, in the worst case he must visit all of the branches in the tree in order to verify the solution. There are an exponential number of branches, so this problem cannot be verified in polynomial time, and therefore it is not in NP.

Another way of describing NP is that it contains those decision problems for which there exists a **non-deterministic** polynomial-time algorithm for finding a correct solution. NP stands for non-deterministic polynomial-time. Non-determinism in this sense is best described as guessing a correct solution, or alternatively being given one, and then verifying it.

2.5 The P = NP Problem

Clearly, P $\subseteq$ NP. That is, if we can find the correct solution to a problem in deterministic polynomial-time (which involves iteratively verifying solutions until we find a correct one), then we can certainly verify that a solution to that problem is correct in deterministic polynomial-time. However, what is not so certain is whether P = NP.

An algorithm can be seen as an enumeration through some ordering of solutions until a correct one is sampled, at which point the problem is solved. Good algorithms order the solutions in such a way that the correct solution to the problem can be found in as few operations as possible. It follows then that the very best algorithm is the one that is guaranteed to sample a correct solution first, of all solutions, every time, for every instance of every problem[1]. If such an algorithm existed then P = NP because any problem that can be verified in deterministic polynomial-time can also be solved in deterministic polynomial-time by simply verifying the first solution, which will be a correct one. However, this is generally considered improbable and most people assume that P $\neq$ NP. That is, there is at least one problem that can be verified in polynomial-time but that cannot be solved in polynomial-time. Note that this has been neither proved nor disproved.

Theoretical computer scientists use two further subclasses of decision problems in order to reason about whether P = NP. These subclasses are defined as the class of NP-Hard problems and the class of NP-Complete problems.

2.6 NP-Complete Problems

A decision problem is in **NP-Hard** if it is at least as hard as any of the problems in NP. Refer to Figure 2.2. This can be proved using a **transformation** algorithm. A transformation algorithm maps every instance of one problem onto a corresponding instance of a different problem. Once again, we can use algorithm analysis in order

[1]In chapter 3 we talk about a search algorithm called A* that uses a heuristic function to decide which solution to sample next. If that heurstic function was fully accurate then A* would always try the best solution first of all solutions.

to asymptotically evaluate the time required by a transformation algorithm, so that the fastest known transformation algorithm defines the complexity of transforming a problem. If we can transform a problem Q_1 into a problem Q_2 in polynomial-time then we say that Q_2 is at least as hard as Q_1 since Q_2 contains Q_1. Then, if we can transform each of the problems in NP to a target problem in polynomial-time, it follows that the target problem is at least as hard as any problem in NP, and thus that problem is in NP-Hard. However, transforming each problem in NP to a target problem in order to prove that it is in NP-Hard will probably take a long time. Instead, we use the class of NP-Complete problems to help us.

A decision problem is in **NP-Complete** if it is in NP-Hard and also in NP. Refer to Figure 2.2. It follows that the NP-Complete problems are equivalently the most difficult problems in NP. This means that if we can transform an arbitrary problem from NP-Complete to some target problem in polynomial time, then that target problem is at least as hard as any problem in NP, therefore it is in NP-Hard. If we also show that this target problem is in NP then it follows that it is in NP-Complete. The problems in NP-Complete are generally regarded as being **intractable**. For each problem in NP-Complete, there is no known algorithm that can solve it in deterministic polynomial-time. That is, with our current knowledge, P is disjoint from NP-Complete.

NP-Complete was once empty, so there must have been one decision problem that was the first to be proven to be in NP-Complete, which was the **SATISFIABILITY** problem. SATISFIABILITY is specified in [25] as follows. Let $U = \{u_1, \dots, u_m\}$ be a set of Boolean **variables**, then a **truth assignment** for U is a function $t : U \rightarrow \{T, F\}$ such that $t(u) = T$ means that u is true under t, and $t(u) = F$ means that u is false under t. Given a variable u, we then have **literals** u and $\bar{u}$. The literal u is true under t if, and only if, the variable u is true under t. Equally, the literal $\bar{u}$ is true under t if, and only if, the variable u is false under t. A **clause** over U is a set of literals, for example $\{u_2, u_3, \bar{u}_5\}$. It represents the disjunction of those literals. Therefore, a clause is **satisfied** by t if, and only if, at least one of its members is true under t. In the preceeding example, the clause is not satisfied if, and only if, the variable u_2 is false under t and the variable u_3 is false under t and the variable u_5 is true under

t. A set C of clauses over U is **satisfiable** if, and only if, there exists some truth assignment t for U that simultaneously satisfies all of the clauses in C. This is called a **satisfying truth assignment** for C. The SATISFIABILITY problem is defined as:

INSTANCE: A set U of variables and a set C of clauses.
QUESTION: Is there a satisfying truth assignment for C?

Cook [7] proved that SATISFIABILITY was the first problem in NP-Complete by showing that every problem in NP can be transformed in polynomial-time to SATISFIABILITY and further that SATISFIABILITY is in NP. Afterwards it became more straightforward to prove that other problems are in NP-Complete. SATISFIABILITY could be transformed to some target problem that is also in NP in order to show that the target problem is in NP-Complete.

Recall that, for each problem in NP-Complete, there is no known algorithm that can solve it in deterministic polynomial-time. NP-Complete problems may help us to answer the question of whether P = NP. If a single problem in NP-Complete can be proved to be solvable in deterministic polynomial-time then P will intersect with NP-Complete, and since all of the problems in NP can be transformed in polynomial-time to any NP-Complete problem then any problem in NP can be solved in polynomial-time and NP $\subseteq$ P. Hence, P = NP.

The P = NP problem is one of the seven **Millenium Problems**. The Millenium Problems were internationally recognised as the seven most difficult and most important open problems of mathematics in May 2000. A $1 million prize is offered to the person who provides the first solution to any of these problems. The Millenium Problems reflect the **Hilbert Problems** that were defined as the 23 most significant unsolved problems in mathematics in 1900. The Hilbert Problems can be seen as guiding mathematicians through the twentieth century, and in the same way the Millenium Problems are hoped to guide research through the twenty-first century. These problems are so important that one of the Hilbert Problems that was not solved is also included in the list of Millenium Problems. The P = NP problem is regarded as very important and a solution would have significant impact. A description of the seven Millenium Problems can be found in [29].

2.7 Approaching NP-Complete Problems

Recall that the NP-Complete problems are the most difficult problems in NP. We can show that a given problem is as difficult to solve as these other intractible problems. With this information we have a better idea about how to approach solving that difficult problem. Importantly, we expect that P $\neq$ NP, so we should place a lower priority on finding an algorithm that guarantees to solve that problem efficiently, and instead focus on different approaches.

We would not attempt to use a systematic approach like uninformed search (see section 3.2) for one of these difficult problems. These approaches would guarantee that we solve the problem, but would probably end up being far too inefficient to be applicable. We might relax some of the criteria in the problem definition so that the problem itself becomes slightly easier to solve for these approaches.

Alternatively, we might use heuristic constructive search (see section 3.3). This would also guarantee a solution and although it would improve upon uninformed search, it would probably still take too long. Or we might attempt to find the best possible solution within the time available. This is called **Optimisation**, and we shall look at some local search algorithms in more detail in section 3.4. Optimisation is more efficient but does not guarantee to solve the problem.

Another approach is to try to solve those instances of the problem that might be the easier ones to solve. There has been some work done on finding the more difficult instances of an NP-complete problem, and this information can be used to avoid attempting to solve these difficult instances.

2.8 Finding Difficult Instances

There are some problems that are in NP-Complete whose instances are mostly easy to solve. For example, Turner [30] shows that almost all of the K-Coloring problem instances are easy to solve. K-Coloring has been shown to be in NP-Complete by

Karp [31]. Hard instances of these problems are sparse. There has been some work that focuses on locating these hard instances.

Cheeseman et al. [8] perform empirical studies on various NP-Complete problems and conject that:

> "*All NP-Complete problems have at least one order parameter and the hard to solve problems are around a critical value of this order parameter. This critical value (a phase transition) seperates one region from another, such as overconstrained and underconstrained regions of the problem space. In such cases, the phase transition occurs at the point where the solution probability changes abruptly from almost zero to almost 1.*"

A simple analogy is a tourist arriving for the first time in a town and wishing to see a monument. In one case they might have a very specific type of monument that they want to see, in which case they can ask the locals where it is and the locals can direct them (if it exists) and they have easily solved their problem. On the other hand they might want to see anything that the town has to offer, in which case they can also very easily achieve their goal. However, in the seperating case, where they want to see something specific, of which the town has a few close (but not exact) matches, they can ask the locals who will direct them to the near matches but the tourist will have to search back and forth between these near matches. This last case encapsulates the idea of the phase transition, where there is enough of a constraint so that search is necessary, and where there is so much constraint that the solutions located are not feasible. Thus, local search algorithms (see section 3.4) tend to have difficulty on the phase transition because they are fooled by local optima that are far apart. See chapter 3 for more information.

For one of the problems, Hamiltonian Circuit, Cheeseman et al. first show that the probability of there being a solution varies with the average connectivity of the graph. When this order parameter is low then there is not likely to be any solutions, but when it is high, that is for a fully connected graph, then a solution is highly likely. At the critical value of the order parameter, the probability changes from near to zero to near to 1. Cheeseman et al. then apply a backtrack search procedure to a set of problems and plot the number of steps required by the procedure to find a solution.

They report the existence of a phase transition at the same point as the critical value of the order parameter.

Cheeseman et al. show that in some cases this phase transition is preserved when the target problem is mapped to a second problem. They further show that there are hard instances that lie away from the phase transition, but the instances on the transition are even harder. In the discussion of the results of the work, they admit that their phase transition results might be due to the algorithm choice rather than being intrinsic to the problem, (see the No Free Lunch theorem description in chapter 3). Therefore, other algorithms might be used in an attempt to produce similar results. They finally open the question: "*Do other types of problems, such as optimisation problems, games, etc. have the same properties?*" This leads to our investigation to locate hard LEMMINGS instances in chapter 7.

2.9 Conclusions

In this chapter we have provided a theoretical basis for the thesis, upon which we can support our arguments. We have defined the basic concepts of problem, solution, and algorithm that will be used throughout the thesis. We have provided an outline of algorithm and problem analysis. Algorithm analysis will be used in chapter 3 in order to define the complexities of various search algorithms, and also in chapter 6 so that we can analyse the potential success of applying some of these search algorithms to solving the LEMMINGS puzzle. We have described a background on decision problems and the theory of NP-Completeness, which have relevance to our survey of NP-Complete puzzles in chapter 4 and the definition and investigations of our LEMMINGS puzzle in chapters 5, 6 and 7. We have described the P = NP problem, which is very relevant to contemporary mathematical science and is included in the Millenium Problems. We have discussed various approaches to tackling a difficult NP-Complete problem, and this will form a basis for some of our discussion on various approaches to LEMMINGS in chapter 6. Finally, we discussed some work focussing on the location of difficult instances, and this leads to our investigation of difficult LEMMINGS instances in chapter 7.

In the next chapter we shall outline various search algorithms that can be used for solving various problems, and we shall also provide the time and space complexities for some of these algorithms.

Chapter 3

Search

A background of search algorithms

"He that climbs the tall tree has won right to the fruit."
Sir Walter Scott

In this chapter we outline search algorithms. In chapter 2 we introduced definitions for problem, algorithm and complexity, and here we provide more insight into what an algorithm does to solve problems, and what different types of algorithms there are.

Our main motivation for this chapter is to detail those algorithms that we might apply to our LEMMINGS problem in chapter 6. LEMMINGS is a new problem and so it is relevant for us to detail the standard algorithms in the literature. As we have seen in chapter 2 NP-Complete problems should generally not be tackled using uninformed algorithms. So it is important to know which algorithms are likely to be more successful for one of these difficult problems. Therefore, we outline the informed algorithms that are more likely to be successful. The details from this chapter will also have relevance to chapter 4 where we shall survey some NP-complete puzzles and some of the approaches taken towards them.

Search is a method of moving from state to state in a search space, by applying a search operator, in the hope of finding a solution to a problem. The algorithms that we describe in this chapter are seperated into constructive search, both uninformed and informed, and local search. Constructive search builds up a solution, whereas local search starts from a complete solution. Uninformed search enumerates over all of the problem states, whereas informed search is guided through them. We outline when it is good to use some of these algorithms so that we can apply that information to LEMMINGS in chapter 6.

We describe various uninformed algorithms and detail their complexities, using algorithm analysis as defined in chapter 2. We then discuss some informed algorithms, including constructive and local varieties. We also define the concepts of state space and search space that will be relevant to our discussion of our difficulty measure for LEMMINGS in chapter 7, and we highlight issues relating to heuristic functions that has relevance to parts of our puzzle survey in chapter 4.

Finally, we shall briefly discuss the No Free Lunch Theorem. This will be relevant to our difficulty measure of LEMMINGS instances in chapter 7.

The information in this chapter can be found in any good undergraduate text book, for example [1], and also in [32] and in [33].

3.1 Constructive Search

In this section we shall define a general constructive search method, so that in the following sections we can outline some uninformed and informed constructive search algorithms.

A problem can be described using **states**. The problem instance that we are trying to solve is described by the **initial state**, and then one or more **goal states** each describe the instance when in a solved state. The **operator** in constructive search generates the successors of each state, such that each successor is the consequence of an action on that state. An action describes the requirements that the state must meet in order for that action to be applicable, and also the consequences of that action on that state. Thus, each successor describes the new problem state brought about as a consequence of applying each action on the original state. Each action is associated with a **cost**. A **state space** is then implicitly defined by the initial state and the actions for each state, and consists of all of the states that a problem can be in from the initial state. The state space is actually a graph of nodes (states) joined by arcs (actions). A **solution** to a problem instance is a **path** of actions from the initial state to one of the goal states. The **optimal** solution is the one with the lowest cost in terms of its actions[1].

For example, suppose that our problem is to dive a submersible into a pool of water to retrieve a quarry whilst avoiding some static obstacles. We assume that this is in a 2-dimensional world. Each state in the state space describes the position of our sub and whether it is holding the quarry. The initial state consists of the sub in its initial position above the surface of the pool without the quarry, and assuming that the quarry can be grabbed if it is adjacent to the sub, then there are four goal states and each describes the sub in an adjacent position to, and holding, the quarry. The actions include grab, descend, ascend, move-left and move-right. The problem is discrete, so the position of the sub and each of the movement actions are also discrete. The cost of each action can be defined as the amount of fuel that is consumed by the sub. A solution to an instance of this problem is a sequence of actions that navigate the sub from its initial position above the pool, down into the pool avoiding the obstacles in order to grab the quarry. The optimal solution is the one that consumes the least amount of fuel.

One process of finding the optimal path through the state space is called **constructive search**. Constructive search essentially builds up different sequences of actions, keeping some in memory and expanding one sequence at a time, until it finds a sequence whose outcome is a goal. Constructive search builds a **search space (tree)** of **nodes**, and each node represents a state, such that the **root node** represents the initial problem state. Constructive search **expands** the initial state by

[1]It is common to assign every action a cost of one, therefore making the optimal solution the one with the fewest actions.

performing the actions for that state. When a state is expanded then its successors are **generated**. One state is generated for each action. These new states are added as nodes to the search tree and are linked to the root node with **branches**, and each branch represents an action. The generated nodes that have not yet been expanded are called **open** nodes. The open nodes are then expanded one-by-one and the generated successors are then added to the tree and open list. Constructive search continues either until there are no more open nodes remaining to be expanded, or until a goal state is located, at which point a solution has been found. The **depth** of a node in the tree is defined as the number of actions that were applied from the root to reach that node. As the depth of the tree increases the number of new branches at that depth generally increases, but there is always a unique path from every node back to the root. This path represents a solution, which is the sequence of actions that yield the node from the root node.

This process is called **tree search**. Note that two nodes in the search tree might actually represent the same state. In other words the same state can be reached via two different sequences of actions. For example, if the submersible moved left, descended, and then moved right, it would be in the exact same state if it moved right, descended, and then moved left. Of course the submersible could just descend and it would also be in the same state. This means that the search space (tree) is usually bigger than the state space. We can easily augment tree search so that it uses a **closed** list of nodes. This augmented search is called **graph search**. Expanded nodes are then added to this list of closed nodes. To ensure that a state is never expanded more than once, graph search simply compares any state before it is expanded with the closed list, and if it appears then it has already been expanded, and so that state should not be expanded again. That is, if the current node matches one in the closed list then it is discarded, otherwise it is expanded. This would mean that a state is expanded at most once, and this makes the search space (tree) no bigger than the state space. However, there is an overhead involved in graph search. The closed list must be saved in memory. Therefore, we need to be sure that graph search will offer enough of an advantage for our problem to make it worthwhile.

The very first time that the controller of the submersible experiences an instance of

$$\begin{tikzcd}
 & & & a \arrow[dl] \arrow[dr] & \\
 & & b \arrow[dl] \arrow[dr] & & c \arrow[d] \\
 & d \arrow[dl] \arrow[dr] & & e & \underline{f} \\
g & & h & &
\end{tikzcd}$$

FIGURE 3.1: A whole search tree.

the example problem above, they will not know the solution to the problem instance, and so must search through the pool to find the quarry in the way we have just described, remembering all the actions that they have tried and trying new ones until they achieve their goal. Afterwards, if the same instance is experienced again, then the controller of the submersible will know the optimal solution and so can apply the very same sequence of actions in order to achieve the same goal without needing to search.

Figure 3.1 shows a very simple search tree with a root node labelled a and a goal node labeled $\underline{f}$. The intention of search is to find a path through the tree to the goal. Importantly, the search knows the initial state of the problem, and the successors of each state, and it uses this information to build the tree from nothing. The difficulty lies in the size of the tree, which can be very large with billions of nodes. There are different constructive search algorithms that are defined by their choice of which of the open nodes to expand next. They are categorised as follows: **uninformed** search algorithms, which systematically expand every open node, and **informed** search algorithms, which use domain specific information to identify open nodes that are more likely to be on an optimal solution path, but are not guarenteed to be so, and expands these nodes first. Here we outline some of the basic uninformed and informed constructive search algorithms. Unless otherwise stated, the algorithms described perform tree search that does not save the closed list of states and is allowed to expand nodes more than once. This means that a state can appear more than once in the tree and so the search space (tree) will usually be larger than the state space. We judge the performance of these constructive search algorithms by four criteria. An algorithm is **complete** if it is guaranteed to find an existant goal, it is **optimal** if it always finds the goal with the cheapest path, and the algorithm requires at most **time** in terms of the number of nodes generated and at most **memory** in terms of the maximum number of nodes stored, both according to algorithm analysis (see chapter 2).

In this section we have described a general constructive search method. In the next section we shall introduce several uninformed search algorithms including breadth first search, depth first search, and iterated deepening search. There are other similar types of uninformed search algorithms that can be found in [1]. We do not intend to give a full description of all of the various uninformed search algorithms. It is only necessary for the reader to know that they are systematic algorithms.

3.2 Uninformed Constructive Search

In this section we shall outline some uninformed constructive search algorithms. These are systematic algorithms that perform a blind search through the search space.

Breadth First Search (BFS) (see Moore [34]) expands the root node of the tree to all of its successors, and then always expands the shallowest open node in the tree until a goal is located. This means that all of the nodes at a given depth d are expanded to their successors before the nodes at depth $d + 1$. Figure 3.2 depicts the progress of BFS, the next node to be expanded is written in uppercase, the expanded nodes are written as $\bar{n}$.

If the controller of the submersible carried out BFS in real life, then he would search in his submersible in all directions but only a short distance from the start and not search further away until all areas at a given distance from the start point have been searched. He would keep searching further and further away from the initial position until finally locating the quarry.

BFS will always find the optimal solution if all actions have an equal cost because

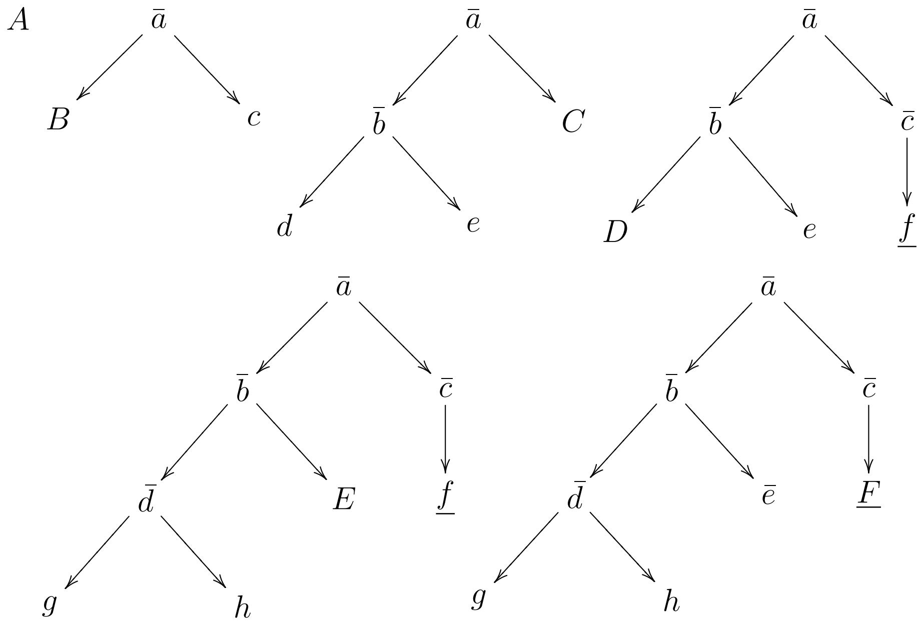


FIGURE 3.2: The progress of breadth first search.

the shallowest goal node in the tree is the first one found. The algorithm can be augmented very easily to **uniform cost search** that will find the optimal solution and the actions can have any cost. Uniform cost search always expands the node with the cheapest path rather than the shallowest node in the tree.

The amount of time and memory used by BFS are equal because every node in the search must be stored. Assuming that every node in the tree has b successors, then we generate b nodes from the root, b^2 nodes in the next level, followed by b^3, and so on, until we generate $b^{d+1} - b$ nodes at the depth of the shallowest goal node d. Recall that we do not expand a goal node. Unfortunately, this is too much memory, so we need an alternative that uses less memory. **Depth First Search (DFS)** always expands the deepest open node in the tree until the deepest extent of the tree is reached. At this point the search 'backs up' to the deepest open node, and again searches until the deepest extent, and so on, until a goal has been found. Figure 3.3 depicts the progress of DFS, the next node to be expanded is written in uppercase, the expanded nodes are written as $\bar{n}$. DFS only requires enough memory to hold the current path, which is at most length m, that is the maximum depth of the tree, along with the

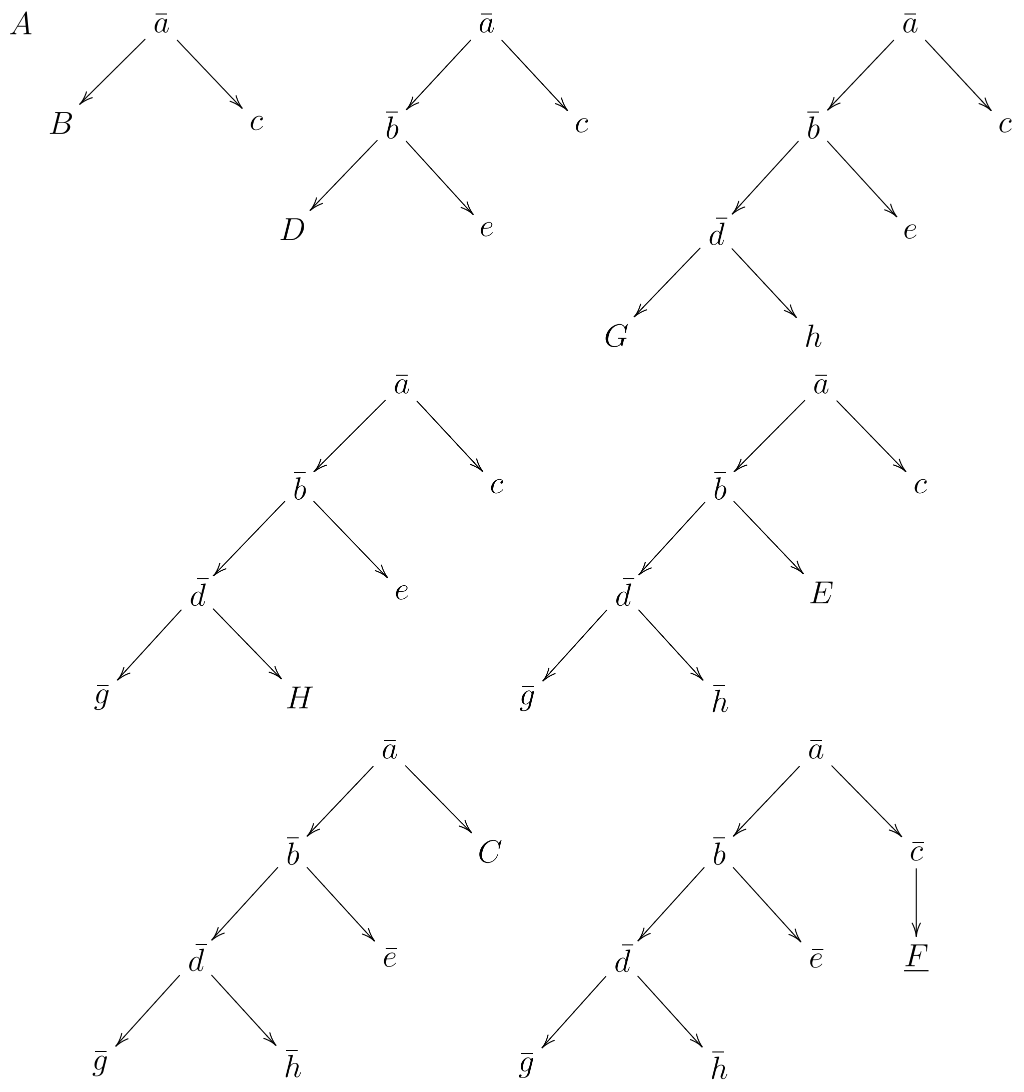


Figure 3.3: The progress of depth first search.

open successors, which is at most b, at every decision point along that path. This is much less memory than BFS.

If the controller of the submersible carried out DFS in real life, then he might dive straight downwards in order to locate the quarry.

DFS has a major drawback that it cannot avoid. If the search tree has unbounded depth, or if the state space contains cycles, then DFS might expand an infinite path without locating a goal. Furthermore, DFS is not guaranteed to find the optimal solu-

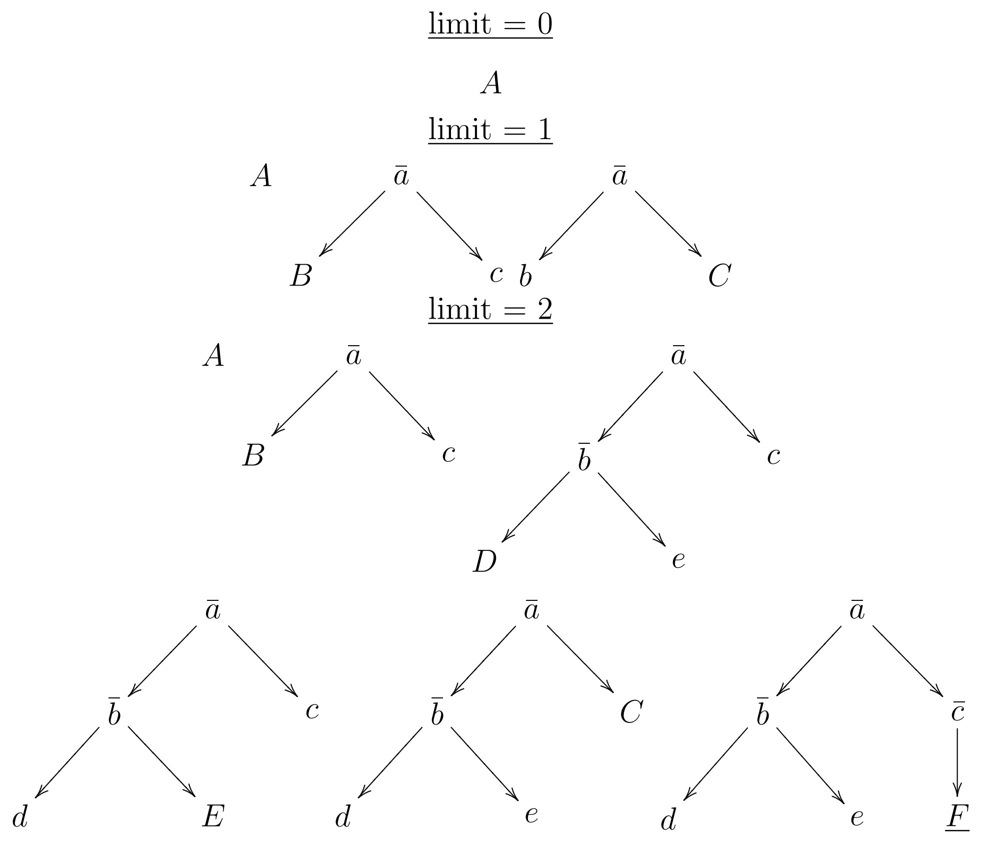


FIGURE 3.4: The progress of iterated deepening search.

tion because the first goal it finds might not be on the cheapest path. These drawbacks can be alleviated by a simple augmentation to make **depth limited search**. Depth limited search performs a DFS alongside a depth limit so that it does not expand any nodes whose depth is equal to that limit. Unfortunately, if a limit is chosen that is less than the depth of the shallowest solution then this algorithm will not find any solution at all. Alternatively, **Iterative Deepening Search (IDS)** (see Slate and Atkin [35] and Korf [36]) iteratively searches with an increasing depth limit starting from zero so that the algorithm succeeds when the depth limit matches the depth of the shallowest goal node. Figure 3.4 depicts the progress of IDS, the next node to be expanded is written in uppercase, the expanded nodes are written as $\bar{n}$. IDS will always find the optimal solution if all actions have an equal cost because the shallowest goal node in the tree is the first one found, just like BFS. IDS only requires enough

	BFS	DFS	IDS
Complete	Yes	No	Yes
Optimal	Yes	No	Yes
Time	$O(b^{d+1})$	$O(b^m)$	$O(b^d)$
Memory	$O(b^{d+1})$	$O(bm)$	$O(bd)$

TABLE 3.1: A comparison of search performance.

memory to hold the current path, which is at most length d, that is the shallowest goal the tree, along with the open successors, which is at most b, at every decision point along that path. This is much less memory than BFS, just like DFS.

If the controller of the submersible carried out IDS in real life, then he would search up to a limited number of actions, repeatedly in all directions, then if he does not find the quarry he would go back to the initial position and search his submersible a little bit further from the initial position repeatedly in all directions, before coming back to the initial position again and searching a little bit further still until he finds the quarry.

Table 3.1 (taken from [1]) shows a summary of the performance of each of the main search methods we have described according to our four criteria. Here b is the maximum number of successors of any node, d is the depth of the goal with the shortest path, and m is the maximum depth of the tree. For example, the tree in Figure 3.1 has the following values, $b = 2$, $d = 2$ and $m = 3$. As can be seen in the table, BFS is optimal and complete but suffers from large memory requirements, DFS has smaller memory requirements but is not optimal or complete (because the depth of the tree might be unbounded), whereas IDS combines the optimality and completeness of BFS and the lower memory requirements of DFS. In general, IDS is the prefered uninformed algorithm when searching through a large tree where the depth of the optimal solution is unknown.

Recall that these algorithms use tree search that does not save the closed list in order to compare the current node before expanding it. This means that the size of the search space (tree) is potentially larger than the size of the state space. If we used graph search that saves the closed list then the memory requirements of the algorithms would be bounded by the size of the state space, which might be lower than $O(b^d)$.

Time vs. memory is generally an issue in search, and there is usually a conflict between them. For example, given the n-queens problem, we might have an algorithm that includes a precomputed hash table of solutions, one for each instance of the problem to be solved, so that solving the problem is simply a matter of parsing the input describing the instance and then referencing the appropriate solution. This algorithm would have constant time but would require a formidable amount of memory in which to store all of the solutions. One example of an algorithm that saves time by sacrificing space is a Checkers playing program called Chinook, which employed opening and endgame databases of moves for billions of piece positions (see [37] and [38]).

An alternative algorithm is a random search called **Iterative Sampling** (see [39]), which starts at the root node and chooses one of its successors at random, and then chooses one of that nodes' successors at random, and so on, so that it forms a random path through to the bottom of the tree. A depth limit can be used to cut off trees of unbounded depth. This procedure is iterated, and each iteration starts back at the root node so that a new random path is generated. Iterative sampling has very low memory requirements because we only save the path through the tree. The algorithm also avoids infinite paths, and it is complete. However, it is not optimal because it is a random sampling. We would favour this type of algorithm when there are lots of goal nodes in the tree so that we might easily stumble upon one of them, when the optimality of the solution is not relevant, and when we do not have enough knowledge about characteristics of the problem in order to employ informed search. This last element is discussed in more detail in the next section.

If we vary our submersible problem so that there is more than one quarry or maybe a very large single quarry, then the controller of the submersible might carry out iterative sampling in real life. In which case we might imagine him trying a random sequence of actions of limited length, then if he doesn't find the quarry he would take his submersible back to the initial position and then try another random sequence of actions, and so on.

In this section we have outlined various uninformed constructive search algorithms. In the next section we shall outline various informed constructive algorithms.

3.3 Informed Constructive Search

The uninformed search algorithms that are described above are generally much too inefficient to be applied to very large state spaces that contain billions of states. For example, if we intend to solve the submersible problem that we defined in the previous section, in particular an instance where there are at most five actions at any state ($b = 5$), where the shortest solution consists of fifteen actions ($d = 15$), and where there is no limit to the length of a solution ($m = \infty$), then BFS will take at most 5^{16} steps, DFS might never halt, and IDS will take at most 5^{15} steps. Therefore, we clearly need to be able to apply more efficient algorithms. Thus, we might use informed search algorithms that use knowledge about characteristics of the problem that they intend to solve in order to find a solution more efficiently. Informed search includes informed constructive algorithms, which are similar to those described in the previous section that start from the initial problem state and build a search tree. Informed search also includes local search algorithms that take an arbitrary state in the search space, which represents a candidate solution, and attempts to move to a better neighbouring state (solution). In this section we describe some informed constructive algorithms, and we describe local search in the next section.

Informed constructive search is based on either tree search or graph search as defined in the previous section, and also incorporates an **evaluation function** $f(n)$. $f(n)$ assigns a value to each open node n in the search according to the probability that the node is on the optimal path to a goal, so that the node with the best evaluation is expanded first. That is, the open node that is most likely to lead eventually to the optimal goal is expanded first. The key to a successful informed constructive search algorithm is to have an evaluation function that is as accurate as possible, so that the search does not make any unnecessary expansions. This turns out to be unlikely and evaluation functions are usually inaccurate and sometimes push the search in the

$$\begin{tikzcd}
 & & & a_1^0 \arrow[dl] \arrow[dr] & \\
 & & b_0^1 \arrow[dl] \arrow[dr] & & c_1^1 \arrow[d] \\
 & d_1^2 \arrow[dl] \arrow[dr] & & e_1^2 & \underline{f_0^2} \\
g_2^3 & & h_2^3 & &
\end{tikzcd}$$

FIGURE 3.5: A whole search tree with costs.

wrong direction.

An essential component of an evaluation function is a **heuristic function**. An heuristic function $h(n)$ returns the estimated cost of reaching the cheapest goal node from node n. We place two restrictions on any heuristic function. Firstly, it must be **admissible**. That is, it can never overestimate the cost of reaching a goal. So a heuristic, in this case, is a lower bound on the exact cost. If the heurisitc function were exact, then only the nodes that need to be expanded are expanded. Secondly, it must be the case that $h(goal) = 0$ for any $goal$. Heuristic functions can be created by **relaxing** the rules of the problem that we intend to solve because "*the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem*" [1]. For example, we might relax our submersible problem that we described above so that we ignore any obstacles in the pool of water, then when we locate the quarry by searching, the cost of the optimal solution to that relaxed problem is in fact a lower bound on the cost of the solution to the original problem that includes obstacles. So an admissible heuristic for the submersible problem is the cost of moving straight to the position of the quary and grabbing it, disregarding the obstacles. Relaxing the problem more results in an heuristic that is easier to compute, but also means that the heuristic is less accurate. A better heuristic will yield a more efficient search, but will take longer to compute. Therfore, we need a balance between quality of heuristic and time required to compute it.

$$A_1^0(1)$$

$$\begin{tikzcd}
& \bar{a} \arrow[dl] \arrow[dr] & \\
B_0^1(1) & & c_1^1(2)
\end{tikzcd}$$

$$\begin{tikzcd}
& & \bar{a} \arrow[dl] \arrow[dr] & \\
& \bar{b} \arrow[dl] \arrow[dr] & & C_1^1(2) \\
d_1^2(3) & & e_1^2(3) &
\end{tikzcd}$$

$$\begin{tikzcd}
& & \bar{a} \arrow[dl] \arrow[dr] & \\
& \bar{b} \arrow[dl] \arrow[dr] & & \bar{c} \arrow[d] \\
d_1^2(3) & & e_1^2(3) & \underline{F_0^2(2)}
\end{tikzcd}$$

FIGURE 3.6: The progress of A* search.

We shall introduce two informed search algorithms called A-star search and iterative deepening A-star search. Other informed search algorithms can be found in [1], here we only give a taster of the concepts involved in these algorithms.

A-star search (A*) (see Hart et al. [40]) is one of the best known informed constructive search algorithms. It uses an evaluation function $f(n) = g(n) + h(n)$. $h(n)$ is an heuristic function as described above, and the extra component, $g(n)$ is a simple measure of the cost to reach node n from the root. Thus, $f(n)$ is the cost to reach n from the root, plus the estimated cost to reach the cheapest goal from n. That is, $f(n)$ returns the estimated cost of the cheapest solution through n. A* always expands the open node with the lowest f-value, so A* expands the node that it expects is on the optimal solution[2]. Figure 3.5 shows a simple search tree with a root node labelled a and a goal node labeled $\underline{f}$, and also with additional information $n_{h(n)}^{g(n)}$. Figure 3.6 depicts the progress of A*, with each node labelled as $n_{h(n)}^{g(n)}(f(n))$, the next node to be expanded is written in uppercase, the expanded nodes are written as $\bar{n}$.

If we use tree search, then A* is optimal if our heuristic function $h(n)$ is admissible. We can prove this. Take any non-optimal goal node g_n that is in the open list. Since

[2]If we can design a perfect heuristic function then A* will only expand those nodes that are on the optimal solution. That is, A* would always know the optimal solution to a problem instance immediately, and if that problem is in NP-Complete then it is also in P, and P = NP. See chapter 2 for more details.

$h(goal) = 0$ for any *goal*, then $f(g_n) = g(g_n) > C*$ where $C*$ is the actual cost of the optimal solution. Then, for any open node n that is on the path to the optimal goal, $f(n) = g(n) + h(n) \leq C* < f(g_n)$. Thus, a non-optimal goal node will never be expanded and A* using tree search will always return the optimal solution as long as $h(n)$ is admissible.

If we use graph search then a suboptimal solution can be returned if there are two paths through the same state and the suboptimal one is the first one found. This is because graph search discards states that are already in the closed list. This can be remedied by always keeping the cheaper of any two paths that arrive at the same state and removing the more expensive path from the search, and then A* using graph search is optimal.

Note that A* will expand all nodes with $f(n) < C*$ and might expand some nodes with $f(n) = C*$, but A* expands none of the nodes with $f(n) > C*$. Thus, since $h(n)$ is an admissible heuristic, A* can **prune** subtrees from the search tree. For example, in Figure 3.6 the subtree underneath node d is pruned from the search because any goal node that appears after d will not be on the optimal solution.

If the controller of the submersible, in the original problem described above, carried out A* in real life, then he would use a tracking device that told him the straight line distance to his quarry but disregarded the obstacles. That way the controller would move the sub in the direction of the quarry in order to move closer to it whilst avoiding the obstacles as and when they get in the way.

Unfortunately, A* is based on breadth first search and stores every node in the search in memory, and so requires lots of space. We need a different algorithm that uses less memory. **Iterative-deepening A-star search (IDA*)** (see Korf [36]) is similar to IDS except that it compares the f-value of the nodes with the limit rather than depth of the nodes in the tree. IDA* performs A* alongside a cost limit so that it discards any nodes whose f-value is greater than the limit. The cost limit starts as the f-value of the root node and is increased each iteration, setting the cost limit to the smallest f-value of any node that was discarded in the previous iteration, so that the algorithm succeeds when the cost limit matches the actual cost of the optimal solution. Figure 3.7 depicts the progress of IDA*, with each node labelled

<u>limit = 1</u>

$A_1^0(1)$

$$\begin{tikzcd}
& \bar{a} \arrow[dl] \arrow[dr] & \\
B_0^1(1) & & c_1^1(2)
\end{tikzcd}$$

$$\begin{tikzcd}
& & \bar{a} \arrow[dl] \arrow[dr] & \\
& \bar{b} \arrow[dl] \arrow[dr] & & C_1^1(2) \\
d_1^2(3) & & e_1^2(3) &
\end{tikzcd}$$

<u>limit = 2</u>

$A_1^0(1)$

$$\begin{tikzcd}
& \bar{a} \arrow[dl] \arrow[dr] & \\
B_0^1(1) & & c_1^1(2)
\end{tikzcd}$$

$$\begin{tikzcd}
& & \bar{a} \arrow[dl] \arrow[dr] & \\
& \bar{b} \arrow[dl] \arrow[dr] & & C_1^1(2) \\
d_1^2(3) & & e_1^2(3) &
\end{tikzcd}$$

$$\begin{tikzcd}
& & \bar{a} \arrow[dl] \arrow[dr] & \\
& \bar{b} \arrow[dl] \arrow[dr] & & \bar{c} \arrow[d] \\
d_1^2(3) & & e_1^2(3) & \underline{F_0^2(2)}
\end{tikzcd}$$

FIGURE 3.7: The progress of IDA* search.

as $n_{h(n)}^{g(n)}(f(n))$, the next node to be expanded is written in uppercase, the expanded nodes are written as $\bar{n}$.

IDA* is optimal if our heuristic is admissible because the cost limit will gradually increase until it is equal to the actual cost of the optimal solution.

If the controller of the submersible carried out IDA* in real life, then he would once again use a tracking device that told him the distance to his quarry whilst disregarding the obstacles. Then he would submerge only as far as he thought that he would need to travel in order to reach the quarry, and then if he didn't find it within that distance because he had to avoid obstacles, then he would go back to the surface and submerge a little bit farther until eventually he would submerge as far as he actually needed to in order to reach the quarry.

There are many other informed constructive search algorithms, for example Weighted A* [41], K-Best-First search [42], Breadth-First Heuristic search [43], and Frontier search [44] that is a general improvement to constructive search.

In this section we have outlined various informed constructive search algorithms and discussed some issues relating to heuristic functions. In the next section we shall introduce informed local search and then outline some local search algorithms.

3.4 Informed Local Search

In this section we shall introduce local search and then outline various local search algorithms.

Local search is a set of algorithms that use a different procedure than constructive search in order to locate solutions to problems. The main difference between constructive search and local search, is that local search disregards the path to the goal that constructive search relies upon in order to find a solution. Instead, local search operates over a single solution state. This is known as a **candidate solution** and we assume that it has the characteristics of a solution without any guarentees that it actually solves the problem instance. A **feasible solution** is one that solves the instance, but it is not as good as the **optimal solution**. From the point of view of a tree, a candidate solution is any sequence of legal actions, a feasible solution is one that yields a goal node, and an optimal solution is the best candidate solution. A candidate solution for the submersible problem described above would be a sequence of actions that the submersible can perform, a feasible solution would be one that located the quarry, and the optimal solution would locate the quarry using the least amount of fuel. Rather than building up a path of actions, local search moves to **neighbours** of the candidate solution in the hope that it can move towards better a solution. The neighbours of a solution are defined by a **neighbourhood function**, such that they are in some way close to the candidate solution. Moving from candidate solution to a neighbour is the operator in local search. Local search is often used to solve hard minimisation or maximisation problems for which there might be no definition of cost or distance with which to build an heuristic function. In this case an **objective function** represents the relative worth of a solution against the others in the solution space, so that the best solution according to the objective function can be sought. Local search has some advantages over constructive search. Firstly, it uses less memory because it does not save the path through the space, and secondly, it can locate feasible solutions in large spaces where constructive search might not be appropriate.

Figure 3.8 (see [1]) shows a visualisation of a **solution space landscape** that is defined by the value of the objective, for a maximisation problem. Here a candidate solution is being improved towards a **local maxima**. A local maxima is a good solution when compared locally against its neighbours, but not as good as the **global maximum**, which is our ultmimate goal. **Plateaux** are neighbourhoods of the landscape that contain solutions of near equal objective value.

In a maximisation problem the objective represents the quality of a solution, so we want a better quality solution. On the other hand we might have a minimisation problem where the objective represents the cost of the solution, so we want a cheaper solution. To convert between a minimisation and maximisation problem we simply negate the objective, which serves to flip the landscape in Figure 3.8. For our discussions we shall assume a maximisation problem.

The struggle in local search is that the neighbourhood of solutions at a given step is generally not large enough to encapsulate the required information contained in the landscape that would push a solution towards the global maximum. Therefore, there are different local search algorithms that each attempt to avoid sticking at the local maxima whilst trying to locate global maximum. That is, the algorithms try to exploit the fact that they have located a good solution, and they simultaneously attempt to explore further.

For example, suppose that our problem is to climb up to the highest point of a mountain that is shrowded in fog so that we can only see a few meters around us. We do not necessarily care about the path that we took up the mountain and do not need to save it in memory, our only goal is to safely reach the top. This is what local search is good for. We simply try to move to a better position as defined by our objective,

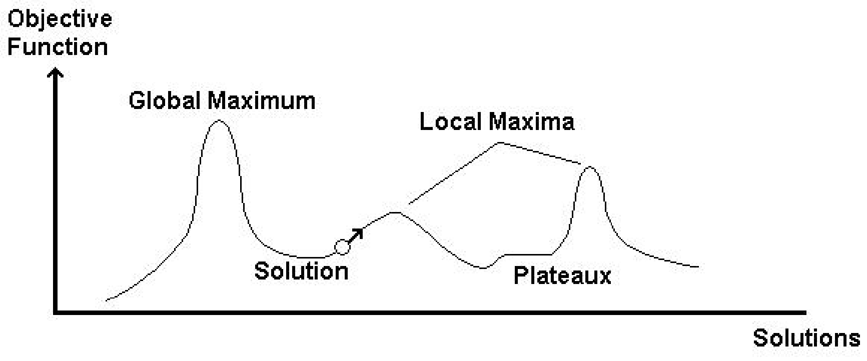


FIGURE 3.8: Solution Space.

which in this case is our height on the mountain. This simple example matches the landscape in Figure 3.8. Note that we cannot see the summit (global maximum) because of the fog (small size of neighbourhood). Each candidate solution to this problem is a vector describing the position of the climber, for example (x, y, z). The operator moves to a neighbouring solution that is close to our current position. That is, the neighbourhood might include the solutions $(x+1, y, z)$, $(x-1, y, z)$, $(x, y+1, z)$, $(x, y-1, z)$, $(x, y, z+1)$, $(x, y, z-1)$. Constructive search would save every position that the climber was in from the base of the mountain to his current position. On the other hand, local search only saves his current position and tries to move to a better one. This example suggests the most basic local searcher called Hill Climbing.

Hill Climbing (HC) is the most simple local search method. A single candidate solution is saved in memory and its objective is compared with those of its neighbouring solutions so that the solution with the highest objective is saved as the new candidate. Consequently, pressure is put on the solution to move up hill. If our mountaineer applied HC then he would simply use the height of his location as an objective and repeatedly move to a new position that is higher. Unfortunately, HC cannot escape local maxima or plateaux. The obvious countermeasure is to sometimes allow a new solution that is worse to replace the current one. The following methods represent different implementations of this countermeasure.

Tabu Search (TS) (see Glover [45, 46, 47] and Gendreau and Potvin [48]) is an extension of HC. The main extension is that TS has a concept of short term memory called **Tabu lists**. The tabu lists contain the previous n solutions located in the search. When a local maxima is encountered, TS allows nonimproving moves to be made and furthermore disallows any solutions that appear in the tabu lists to be sampled again so that search is not repeated. Thus any improvements to the candidate solution are sure to be new unsampled solutions. If the mountaineer applied TS to his problem then he would remember the set of positions that he was in most recently, so that when he was forced to descend he would be sure that he would not retrace any of his footsteps during the climb up again, so that he will try new peaks. Kendall and Mohd Hussin [49] apply a tabu search hyperheuristic to examination timetabling.

Simulated Annealing (SA) (see Kirkpatrick et al. [50] and Aarts et al. [51]), allows a neighbouring solution to be accepted even if its objective value is worse than that of the current solution. SA simulates the process of annealing, whereby a metal is hardened by heating it to melting point and then gradually allowing it to cool and solidify again. SA picks a random neighbour of the current solution. If the neighbour improves on the current solution then it is accepted. However, if the neighbour is worse than the current solution, then it is accepted with a probability. This probability decreases as the differences between solutions increases, and also decreases as the length of the search increases. Thus, at the start of the search solutions that deteriorate are accepted with high probability, and as the search 'cools', then the probabilty of accepting these worse solutions drops until eventually only better solutions are accepted. If our mountaineer applied SA then we can imagine that there are earthquakes on the mountain, such that when the climber has just started the earthquakes are ferocious and shake him all around the mountainside[3]. But then as his expedition continues the earthquakes become less and less angry, shaking him less and less, all the while he is climbing, until eventually, the mountain is calm and he

[3]We assume that the climber is not injured by this shaking.

can climb upwards. Garibaldi and Ifeachor [52] describe an application of SA "*to a fuzzy expert system for the interpretation of the acid-base balance of blood in the umbilical cord of new born infants*".

Variable Neighbourhood Search (VNS) (see Mladenovic and Hansen [53] and Hansen and Mladenovic [54]) uses multiple neighbourhoods during a local search. VNS exploits the following: First, a local maxima for one neighbourhood is not necessarily so for all neighbourhoods. Second, a global maximum is a local maxima for all neighbourhoods. Finally, local maxima across different neighbourhoods are relatively close together. The most basic form of VNS is called **Variable Neighbourhood Descent (VND)**. VND performs HC in the first neighbourhood, and continues through each neighoubourhood until it locates an improvement to the candidate solution, at which point it sets this new one to the current solution and continues. The different forms of VNS retain the idea of multiple neighbourhood structures. If the mountaineer applied variable neighbourhood search to his problem, then he would have a helicopter move him to various areas of the mountain so that he could perform local search within each neighbourhood in order to locate the summit. Abdullah et al. [55] apply VNS to university course timetabling.

Genetic Algorithms (GAs) (see Goldberg [56] and Sastry et al. [57]) are based on Natural Selection and Genetics (see Darwin [28], Fraser [58] and Holland [59]). GAs work over a population of candidate solutions. The general flow is as follows:

1. Initialise the population, usually randomly.

2. Evaluate the population using the objective function.

3. Select a set of solutions from the population, favoring the better solutions. The most common selection operator is roulette wheel, where each solution takes a slot on a roulette wheel. The slots have different size proportionate to the solution fitness. The wheel is spun and a solution is selected.

4. Recombine the selected solutions to make new solutions for the next genera-

tion. The most common recombination operator is one point crossover, where a random position is chosen on the solution, which splits the solution in two, so that the sub-solutions from different solutions can be crossed and recombined.

5. Mutate the new solutions. The normal mutation operator is to randomly change each component in the solution according to a probability.

6. Repeat steps 2 to 5 until terminating criteria is met.

There are many operators to choose from, the ones that are described are the most common. See Sastry et al. [57] for more information.

One interesting feature of GAs is that they search over a set of **Building Blocks** (see Holland [59]). Building Blocks are sub solutions that contribute to a high quality outcome. It is thought that GAs are able to locate these sub solutions within the population of solutions, and recombine different ones in order to make even better solutions.

Chaperot and Fyfe [60] use a GA as a technique for improving artificial neural network training for a motocross game.

Artificial Immune Systems (AIS) (see Aickelin and Dasgupta [61]) are inspired by the biological immune system. There are different algorithms for different problems and here we discuss one for the intrusion detection problem. In this problem our goal is to "*detect unauthorized use, misuse and abuse of computer systems.*" AIS deals with multiple points in a space, just like GAs. Each point is an encoding, for example, a data packet that is transferred between computers. A similarity measure is then defined to compare points, for example, we might define a set of trustworthy self points so that non-matching points can be deemed intruders. Each point has a concentration that drops over time until it disappears. However, intruders, that show negative matching to self, increase in concentration, so that eventually an intruder with enough concentration is found. Mutation can also be applied, for instance, a point that matches self can be mutated rather than discarded. AIS are very similar to GAs, as can be seen. Both evolve a set of complex points, for the purpose of forming a solution of closely matched points. Cayzer and Aickelin [62] describe an application of artificial immune networks.

Ant Colony Optimisation (ACO) (see Merkle and Middendorf [63], Dorigo [64] and Dorigo et al. [65]) is based on the natural process of ant coordination, whereby ants follow a pheromone trail and simultaneously lay more pheromone behind them. One visualisation of ACO is as follows. A set of artificial ants repeatedly traverse a decision tree in order to construct solutions. Along the way the ants make each decision based on a probability according to the amount of pheromone that is on each choice. Pheromone evaporates over time, however, any ant that locates a good solution is rewarded by pheromone values being increased along the path that it took to find that good solution. Thus, after some time there will be more pheromone on paths that yield a good solution and more ants will traverse these paths, and hopefully the better solutions will be located. A clear example of ACO for the mountaineering problem is if the climber favoured a route up the mountain that had been explored previously, and so he followed the beaten trail. In this way, the best trail might be used more and more by future climbers. Burke et al. [6] apply an ant algorithm hyperheuristic to a scheduling problem.

Particle Swarm Optimisation (PSO) (see Merkle and Middendorf [63] and Kenedy and Eberhart [66]) is based on the natural phenomenon of flocking birds that are searching for food. Each bird moves based on its own velocity and also based on the movement of the flock as a whole. The algorithm works over a population of points, each of which has a location. The aim is for the population to locate the position that has maximum objective. The points are initialised with random location and random velocity. Each point is then evaluated for its current location, and its best location so far is saved. The global best position that the flock has visited is also saved. The vector describing the next location of each point is updated based on its current movement vector, personal best location, global best location over the flock, and some random noise. Hopefully, the flock will locate the position of maximum objective. PSO can be applied to the mountaineering problem by having a group of climbers with radios, so that they can each radio the locations of their highest point to eachother and all flock towards the highest point. Su and Kendall [67] describe an application of PSO to financial investment.

There are other related algorithms such as **Genetic Programming (GP)** (see Koza [68] and Koza and Poli [69]). GP is similar to GAs, where each individual in the population is no longer a solution, but is instead a computer program. Also **Machine Learning (ML)** (see Yao and Liu [70]) utilises algorithms like GAs in order to optimise a set of parameters that describe, for example, an **Artificial Neural Network (ANN)** (see McCulloch and Pitts [71]). This ANN can then be used for various tasks, even for acting as an heuristic function in one of the informed constructive search algorithms described in the previous section. **Hyper-Heuristics (HH)** (see Ross [72] and Burke et al. [73]) search over a space of heuristics, as opposed to **Meta-heuristics (MH)** (see Glover and Kochenberger [74]), for example GAs, that generally search over a space of solutions. HH "*might be thought of as heuristics to choose heuristics*" [72]. Given some heuristics that work for sub-problems of a problem, HH combines them so that together they work better on the whole problem than individually. The goal with HH is to develop a general algorithm that is widely applicable and successful.

In this section we have introduced local search and described some local search algorithms. In the next section we shall discuss the No Free Lunch theorem.

3.5 No Free Lunch

In the previous sections of this chapter we have described various search algorithms. In this section we shall briefly outline the No Free Lunch theorem.

No Free Lunch (NFL) (see Whitley and Watson [27] and Wolpert and Macready [75]) is a theorem over search algorithms. NFL can be summarised as follows. "*For all possible performance measures, no search algorithm is better than another when its performance is averaged over all possible discrete functions.*" [27]. That is, all search algorithms are equally good for all problems according to all performance measures.

This means that there is no unviersal best algorithm, and there never shall be. That is, even a random sampling is as good as any other search over all problems and performance measures. Therefore, if NFL is true then research to find a search method that outperforms all others will never succeed.

However, what we can do is take some specific algorithm and improve that algorithm according to some performance measure, for a specific problem. Importantly, this algorithm will be improved for our target problem, but shall simultaneously deteriorate for some other problem.

3.6 Conclusions

In this chapter we have outlined the basics of some constructive search algorithms and local algorithms. There are many other search algorithms, but we have only described those that we believe are important and thus relevant to this thesis.

We have described a general constructive search method. This is used for both uninformed and informed constructive search. We detailed some uninformed constructive search algorithms and defined their complexities using algorithm analysis. If we do not have any domain knowledge that we require in order to invent a heuristic and so we must use uninformed algorithms, then IDS combines the optimality and completeness of BFS and the lower memory requirements of DFS, and is the prefered algorithm. The random sampling algorithm that we described is prefered when there are lots of goal nodes in the tree so that we might easily stumble upon one of them, and when the optimality of the solution is not relevant. This random sampling algorithm can be combined with domain knowledge to form one of several local algorithms that search in a random manner, for example a GA.

We have also described informed constructive search and have detailed a number of algorithms. Usually, informed algorithms are better than uninformed if we have the domain knowledge necessary for their use, and IDA* is the prefered algorithm in terms of memory usage. We finally described informed local search and detailed some algorithms.

Our discussion of all of the algorithms in this chapter means that we now know how algorithms actually go about solving problems. We also have a better idea of each of the different types of algorithms, and furthermore which algorithms to apply to which problems. This has relevance to our survey of NP-Complete puzzles in chapter 4 where we describe some of the approaches to each of the puzzles. We shall also have a better idea of which algorithm we might like to use to approach LEMMINGS in chapter 6. We have discussed definitions of state space and search space, which will be useful when describing our difficulty measure for LEMMINGS in chapter 7.

We have outlined when it is good to use each of the algorithms. We can apply what we have learned about the advantages and disadvantages of these algorithms when we come to approach our LEMMINGS problem in chapter 6. We now know that uninformed algorithms have complexity issues, which suggests that we should not apply these algorithms to LEMMINGS. We also see that a random sampling can be good for certain problems, which might include LEMMINGS. Finally, we have seen that generally speaking informed algorithms are better than uninformed when we have the right information about the problem to be solved, which suggests we should attempt to formalise some information about LEMMINGS that can be used for informed search. It turns out that we apply a GA to LEMMINGS, which is a form of informed random sampling.

Our discussion of heuristics hints at an issue that is relevant to other areas of optimal planning. That is, we can often discover the shortest possible solution, but in spite of this, even the optimal solution is longer. Therefore, the key is to find the minimum amount of work that needs to be done on top of the shortest solution in order to find the optimum. For more information see the discussion of Blocks World in chapter 4.

We have also briefly discussed the No Free Lunch Theorem, which will be relevant to our difficulty measure for LEMMINGS in chapter 7.

In the next chapter we shall provide a survey of some NP-Complete puzzles, and discuss some of the work that has been done surrouding each one.

Chapter 4

Puzzles

A survey of puzzles

"*The art of simplicity is a puzzle of complexity*"
Doug Horton

In this chapter we shall provide a survey of the many puzzles that have been designed and become popular. They are contained in the set of those NP-Hard problems that have a single decision maker. In chapter 2 we defined the NP-Hard problems. Puzzles are strongly defined problems that have clear goals and constraints. As such, they have been the topic of artificial intelligence research for some decades (see [20]). This survey is intended to motivate research in the area of NP-Hard puzzles by highlighting some gaps in that area. Furthermore we intend to motivate research on the Lemmings game by classifying it as a self updated puzzle. This leads to our investigations of LEMMINGS in chapters 6 and 7.

The puzzles are categorised into those that are **player updated**, for example Blocks World, and those that are **self updated**, for example Lemmings. The difference between the two types of puzzle is that in self updated puzzles the state will change even if the player chooses to pass, whereas in a player updated puzzle the state will remain unchanged until the player moves. For this reason we must model a pass action as a choice in self updated puzzles. This makes the search space for self updated puzzles larger and also complicates the process of the search, since generally a player of self updated puzzles will mostly choose to pass and only once in a while choose a proper action, which means that a very large portion of the search space is irrelevant to the search. This might require a specific type of search, since a human will intuitively know that he does not need to perform a proper action and he automatically passes.

The puzzles we present below are organised alphabetically within each category. For each puzzle we provide a brief description of the rules. We also provide a reference for the complexity of each puzzle, which the reader is pointed towards in order to gain a precise problem definition. The brief problem description is included so that the reader has an appreciation of the problem. Furthermore, the reader might gain an insight into the similarities between these hard puzzles. For example, the implicit hidden constraints that are only revealed once a solution is attempted. These constraints are maybe what make these games fun.

We detail the inclusion of each puzzle, and also the inclusion of its sub-problems, in NP-Hard, NP-Complete or P[1]. Primarily, we intend to reinforce the appreciation of the difficulty of solving the puzzle. However, we would also like the reader to appreciate what can and cannot be efficiently solved, and thus have a broader understanding of the territory that seperates P and NP-Complete with respect to puzzles.

We also provide a brief outline of some of the work that has been published around each puzzle. This work tends to apply algorithms from chapter 3 to the puzzles. We have not attempted to include all of the research for each puzzle. Our motivation for briefly outlining this research is to provide references to the texts that can be used as starting points for future work on these puzzles. Through this chapter we have been able to analyse each puzzle and highlight which of those constitute a gap in the

[1]Some of the games have been shown to be in other complexity classes but we do not include any further details on this matter.

literature and might benefit from future research. We highlight these puzzles in Table 4.1 in our concluding remarks.

Finally, we offer additional sources of information that might be useful to a researcher who is interested in tackling one of these problems. Often these sources provide software so that the reader can attempt to solve the puzzle and gain an appreciation of the subtleties involved in solving it. Once again, these additional sources might be used as a starting point for any future work.

Erik Demaine has completed some work in the area of **Combinatorial Games**, see `http://theory.lcs.mit.edu/~edemaine/games/` for more information. We are interested in Demaine's survey [23], which details the complexity of some two-player games and also some single-player puzzles. This chapter can be seen as an extension of Demaine's survey of puzzles from [23], and also an extension of Eppstein's survey of puzzles that can be found at `http://www.ics.uci.edu/~eppstein/cgt/hard.html`.

4.1 Player Updated Puzzles

Player updated puzzles are those that remain unchanged until the player moves. That is, if the player continues to pass, then the state of the problem will not change.

Blocks World

Blocks World is played on a table, which we can think of as being infinite in size. There is a finite number of square blocks, such that every block is sat on top of exactly one other block or sat directly on top of the table. Any block that does not have another block on top of it is known as 'clear'. An action moves a clear block from its current place, and puts it either directly onto the table or on top of another clear block. An instance is a description of the blocks in some arbitrary arrangement, and a description of the blocks in their goal positions. A solution is an ordering of moves from the initial arrangement to the goal. The optimal solution is the one with fewest moves. Finding a solution to Blocks World that is no longer than some bound has been shown to be in NP-Complete by Gupta and Nau [76].

Blocks World has recently been studied by Slaney and Thiebaux [77]. They advocate the use of toy problems like Blocks World on the condition that they are properly understood. Therefore, they offer knowledge about Blocks World at a sufficient level so that it can be used as a benchmark and so that it can be successfully tackled. In particular they describe how to generate random instances that can be used for experiments, they detail a number of algorithms to produce solutions for Blocks World along with their time complexity and they explore the structure of hard and easy instances.

In Blocks World the table is not limited in size, which means that for any starting arrangement of blocks, all of those that are out of place can be moved to the table and then moved one-by-one to their goal positions. Out of place blocks must be moved at least once in order to place them correctly, even in the optimal solution. Thus, a simple algorithm can find some arbitrary near-optimal solution that is at most twice the optimal length in time that is linear over the number of blocks [77]. Blocks World might be restricted to a more difficult problem by placing a limit on the size of the table so that it cannot hold all of the blocks, at which point finding any solution might be hard. The difficulty of Blocks World lies in finding the optimal solution. We know that the shortest possible solution must move all out of place blocks to their correct positions, so finding the optimal solution involves finding the minimum number of moves that each place a block where it should not be placed so that blocks can be correctly placed. That is, to find the optimal solution, we must find the minimum number of moves that are not towards the goal.

Further literature surrounding Blocks World can be found in [77]. The interested reader can find more information about Blocks World on the world wide web at `http://users.rsise.anu.edu.au/~jks/bw.html`. This site provides functionality to generate problem instances and their solutions, and also access to some of the literature.

Clickomania

Clickomania (also known as Same Game) is played on a grid of c columns and r rows. The grid contains different coloured square stones, such that there are k colors. Groups are formed by stones of the same colour whose edges are touching. A move deletes a group that contains at least two stones. Stones are constantly pulled downwards until they touch either the bottom of the grid or another stone, so that any gaps made by deleting a group are filled by any stones above it. When a column is deleted all of the stones to the left and right of it move together to fill the space. An instance is a description of the grid that contains a stone in every position. A solution is successful if it removes every single stone. Biedl et al. [78] shows the following. Deciding solvability of 1-column (or 1-row) 2-color Clickomania can be done in linear time. Deciding solvability of 2-column 5-color Clickomania is in NP-Complete. Deciding solvability of 5-column 3-color Clickomania is also in NP-Complete.

Clickomania has not received any other scientific interest, at least as far as the authors are aware. The interested reader can find further information about Clickomania at `http://www.clickomania.ch/click/`, which provides a downloadable version of the game, and `http://theory.lcs.mit.edu/~edemaine/clickomania/`, which also describes some of the history of the game.

Corral Puzzle

Corral Puzzle is played on an $m \times n$ grid. Some of the squares contain a number. The goal is to find a closed loop in the grid such that all of the numbered squares are inside the loop, and for each numbered square the total number of squares horizontally and vertically inline and also inside the loop are equal to that number. See Figure 4.1 for an example.

Deciding solvability of Corral Puzzle has been shown to be in NP-Complete by [79]. This seems to be the only information about Corral Puzzle.

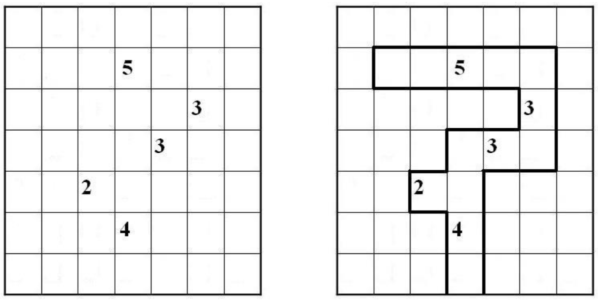


FIGURE 4.1: A Corral Puzzle instance and solution.

Cross Sum

Cross Sum (also known as Kakuro) is a number puzzle played on an $m \times n$ grid of black and white squares. White squares form horizontal and vertical lines of two or more adjoining squares. Each horizontal and vertical line is labeled with a number representing its sum. The goal is to place a digit bewteen $1 \ldots 9$ in every white box so that the sum of every line is equal to the label of that line, and so that each line does not contain repetitions. See Figure 4.2 for an example.

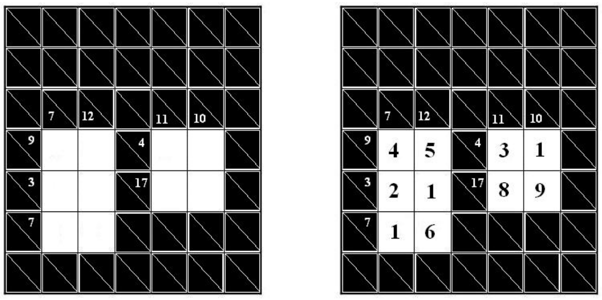


FIGURE 4.2: A Cross Sum instance and solution.

Deciding solvability of Cross Sum has been shown to be in NP-Complete by Takahiro [80]. Takahiro further defines (N, l, L)-CROSS SUM as the same problem except that every line of white has n boxes such that $l \leq n \leq L$, and each digit placed lies between $1 \ldots N$. Then $(9, 2, 6)$-CROSS SUM is NP-Complete, $(N, 2, 5)$-CROSS SUM (with $7 \leq N < \infty$) is NP-Complete, $(N, 1, 3)$-CROSS SUM (with $7 \leq N < \infty$)

is NP-Complete, $(N, l, 2)$-CROSS SUM is linearly sovable, and $(2, l, L)$-CROSS SUM is linearly solvable. See [80] for further information.

We cannot find any other references to Cross Sum. Further information can be found at `http://www.pro.or.jp/~fuji/java/puzzle/crosssum/index-eng.html`, which details some combinations of numbers that can be used to help find a solution, and lists some example problems and their solutions.

Cryptarithms

Cryptarithms (also known as alphametics) is a number puzzle. An alphabet is used to encode some numbers of base k. The goal is to form a one-to-one function between the alphabet and the numbers so that the decoded mathematical formula is satisfied. The size of the alphabet is bounded by k. Numbers cannot have leading zeros. A famous example of Cryptarithms (with k=10) along with its solution is

$$\begin{array}{r} \text{SEND} \\ \underline{+\text{MORE}} \\ \text{MONEY} \end{array} \qquad\qquad \begin{array}{r} 9567 \\ \underline{+1085} \\ 10652 \end{array}$$

Deciding the solvability of a Cryptarithms problem has been shown to be in NP-Complete by Eppstein [81].

There seems to be no other published work on Cryptarithms. More information can be found at `http://www.geocities.com/Athens/Agora/2160/`. This site provides a large number of example problems, some of which can be solved online, a tutorial on Cryptarithms solving, recommendations of books and links to other websites. Also, `http://www.tkcs-collins.com/truman/alphamet/alpha_solve.shtml` provides a solver, and `http://bach.istc.kobe-u.ac.jp/llp/crypt.html` provides downloadable source code for a solver.

Cubic

Cubic is played on an $m \times n$ grid. Each cell can be empty, be solid, or contain a block. Blocks are of different colours. An action moves a block horizontally into an adjacent empty cell. Blocks are pulled downwards by gravity until they are obstructed by another block or by solid. A group of two or more touching blocks that are of the same colour are deleted. A solution is successful if it deletes all of the blocks. Deciding solvability of Cubic has been shown to be in NP-Complete by [82].

There doesn't appear to be any other work on Cubic. More information can be found at `http://www.agon.com/doodle/four.html`, which offers a version of Cubic that can be played online.

Instant Insanity

Instant Insanity is played using cubes, which can be freely rearranged and rotated. The faces of each cube are coloured, such that with n cubes there are n unique colours. Note that it is not the case that each cube is one colour. The goal is to arrange the cubes into a single-file row, such that there are no repeated colours displayed along the top, front, bottom and back of the row. Instant Insanity (with n=4) was a popular game by Parker Brothers. Deciding solvability of Instant Insanity has been shown to be in NP-Complete by Robertson and Munro [20].

Webster [83] offers an approach to Instant Insanity (n=4) that uses a graph to represent the problem, and Walsh and Julstrom [84] attempt to tackle it using a GA. See `http://www.cs.uidaho.edu/~casey931/puzzle/insane/insane.html`, which includes the ability to printout a set of cubes that can be constructed in order to play the game.

KPlumber

KPlumber is a computer puzzle game played on an $m \times n$ grid of tiles. Each tile in the centre of the grid has four adjoining tiles, the corner tiles have two and the edge tiles have three. At the centre of each tile is an intersection of up to four pipes, each of which runs directly to one of the four sides of the tile. Some tiles have no pipes. Water runs through the pipes and leaks from any that are left open. An open pipe at the edge of a tile that touches an open pipe at the edge of the adjacent tile closes both of the pipes. An action rotates one tile, and the pipes on it, 90 degrees. The goal is to arrive at a situation in which all pipes are closed so that there is no leaking water. Deciding solvability of KPlumber has been shown to be in NP-Complete by Kral et al. [85]. Kral et al. also shows that some restrictions to the types of tiles that are allowed results in polynomial-time decidability.

KPlumber has not received any other attention, as far as the author is aware. The interested reader can visit `http://www.list32.com/program_Linkz_648.htm`, which provides a freeware download of a game based on KPlumber called Linkz.

Light Up

Light Up is played on an $m \times n$ grid of black and white squares. An action places a light bulb on a white square. Some of the black squares have a number between 0 and 4 that represents the number of light bulbs that should be placed directly next to that square, horizontally or vertically. A light bulb illuminates the row and column of white squares in each of the four directions away from it up to a black square or to the edge of the grid. The goal is to illuminate every single white square, without any light bulb illuminating any other light bulb. Deciding solvability of Light Up has been shown to be in NP-Complete by [86].

Light Up has not received any other attention, at least as far as the author is aware. See `http://www.puzzle.jp/letsplay/play_bijutsukan-e.html`, for a set of sample problems.

Minesweeper

Minesweeper is a popular computer puzzle game that comes with some Microsoft operating systems. It is played on an $m \times n$ grid of cells, all of which are initially hidden. k mines are randomly distributed over the grid. An action reveals the contents of a cell, at which point, if the cell contains a mine then the player fails and the game ends. Otherwise a number is revealed that corresponds to the number of peripheral cells that contain a mine. The goal is to reveal all of the cells that do not contain a mine, leaving the k cells that do contain a mine still hidden. Deciding solvability of Minesweeper has been shown to be in NP-Complete by [87]. We can define l as the maximum number of mines that surround any cell, and then general Minesweeper has $l = 8$. McPhail [88] shows that deciding solvability of Minesweeper with $l = 3$ is also in NP-Complete.

Castillo and Wrobel [89] describe a system to learn Minesweeper playing strategies. Adamatzky [90] applies Minesweeper to Cellular Automaton. Also see [91] and [92] for evolutionary approaches to Minesweeper. Minesweeper can be found on Microsoft operating systems.

n-Puzzle

n-**Puzzle** is played on an $m \times m$ grid, such that $m^2 - 1 = n$. Within the grid are n square tiles, each of which has a unique number from $1 \ldots n$. Thus, one square in the grid is always empty, called the 'blank' and, depending on its position, the blank has two, three or four adjacent tiles. A move puts one of these tiles into the blank and relocates the blank to that tile's previous position. This is the same as the blank moving, and some solution methodologies model the problem in this way. The tiles are initially set to some random configuration. The goal is to rearrange the tiles into a configuration such as that shown in Figure 4.3 for the 8-Puzzle. Finding a solution to n-Puzzle with the fewest number of moves has been shown to be in NP-Hard by [93].

1	2	3
4	5	6
7	8	

FIGURE 4.3: A goal state of the 8-puzzle

Reinefeld [94] details experiments on 8-Puzzle with IDA* and operator selection, Culberson and Schaeffer [95] apply pattern databases to the 15-puzzle, and Korf and Taylor [96] perform experiments on 24-Puzzle with IDA*. Parberry [97] reports an algorithm for finding sub-optimal solutions in polynomnial-time. Taylor and Korf [98] detail a technique for pruning dupliate nodes from a search and test it on 15-Puzzle and 24-Puzzle. The interested reader can find more information at `http://www-cse.uta.edu/~cook/ai1/lectures/applets/ep/ep.html`, which allows the user to play 8-Puzzle, and also solves 8-Puzzle problems.

Nurikabe

Nurikabe is played on an $m \times n$ grid of white squares. An action shades one of the squares. Some of the squares contain a number and they cannot be shaded. Squares of the same colour are grouped by a sequence with touching edges. The goal is to shade the grid such that each group of white squares contains exactly one number that represents the number of white sqaures in the group, there are no 2×2 or larger shaded sub-groups, and all shaded squares are in the same group. See Figure 4.4 for an example. Deciding solvability of Nurikabe has been shown to be in NP-Complete by [88].

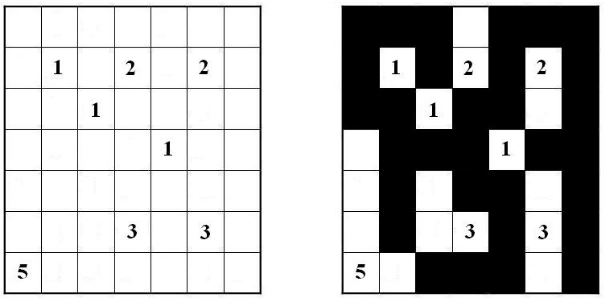


FIGURE 4.4: A Nurikabe instance and solution.

Nurikabe has not received any other attention, at least as far as the author is aware. See `http://www.puzzle.jp/letsplay/play_nurikabe-e.html` for a set of sample problems.

Pearl Puzzle

Pearl Puzzle is played on an $m \times n$ grid of squares. Each square can contain a white pearl, a black pearl or can be empty. The goal is to construct a closed path that does not cross itself, and that passes through every pearl such that the path turns 90° through each black pearl and is straight immediately before and afterwards, and such that the path is straight through every white pearl and turns 90° immediately before or afterwards. See Figure 4.5 for an example. Deciding solvability of Pearl Puzzle has been shown to be in NP-Complete by [99]. This appears to be the only information about Pearl Puzzle.

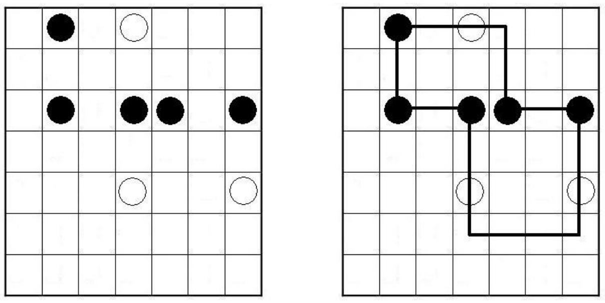


FIGURE 4.5: A Pearl Puzzle instance and solution.

Peg Solitaire

Peg Solitaire is played on an $n \times n$ grid of holes arranged in the shape of a cross. There is a peg in every hole except for the centre one. A move takes a peg over another one that is horizontally or vertically adjacent and places it into the adjacent hole, and removes the peg that was jumped over. The goal is to leave a single peg remaining. Deciding solvability of Peg Solitaire has been shown to be in NP-Complete by [100]. Moore and Eppstein [101] show that the restriction to a single line of pegs is solvable in polynomial-time, and Ravikumar [102] replaces this by showing that Peg Solitaire played on a $k \times n$ grid, for any constant k, is solvable in linear-time.

Kiyomi and Matsui [103] apply integer programming algorithms to Peg Solitaire, and Matos [104] applies Depth First Search to the puzzle. Also see Jefferson et al. [105] for details of various models of Peg Solitaire. The interested reader can find more information at `http://www.myparentime.com/games/games8/games8.shtml` to play Peg Solitaire and see a solution.

Reflections

Reflections is played on an $m \times n$ grid. There is a laser that emits a beam of light, a number of light bulbs, each of which is lit when an odd number of light beams hit it, and a number of bombs that explode if any beam hits them. There are immovable walls that block beams, and movable objects including mirrors that reflect beams and crystals that refract them. An action moves or rotates a mirror or crystal. The goal is to light all of the light bulbs without any of the bombs exploding in the process. Deciding solvability of Reflections has been shown to be in NP-Complete by [106].

There appears to be no other work on Reflections. For more information see `http://www.twilightgames.com/aargondeluxe/aargondeluxe_info.htm`, which provides a free demo download of a similar game called Aargon.

Shanghai

Shanghai is played using Mahjong tiles that are layed out in a number of rows. Some of the rows are stacked on top of others. A tile that does not have another on top of it and that is also at the end of a row is in play. An action removes a pair of matching tiles that are both in play. The goal is to remove all of the tiles. Deciding solvability of Shanghai, where the position of every tile is known, has been shown to be in NP-Complete (see `http://www.ics.uci.edu/~eppstein/cgt/hard.html#shang`).

See `http://www.gamegarage.co.uk/info/shanghai-mahjong/` in order to play a trial version of the puzzle.

Slither Link

Slither Link is played on an $m \times n$ grid, with dots marking the intersections. Some of the cells are numbered between 0 and 3. An action connects two adjacent dots with a vertical or horizontal line. The goal is to form a single loop in the grid of dots that does not cross or branch. The number in each cell represents the number of lines that surround that cell. An emtpy cell can be surrounded by any number of lines. See Figure 4.6 for an example. Slither Link has been shown to be in NP-Complete by Yato [107].

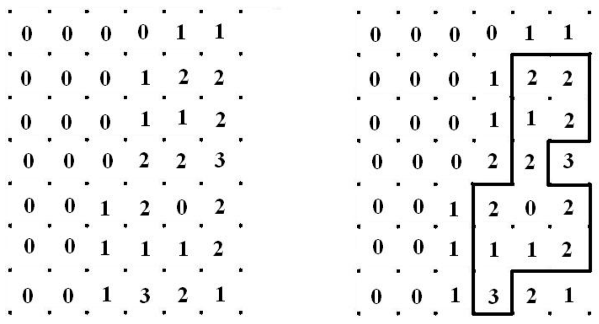


FIGURE 4.6: A Slither Link instance and solution.

Slither Link has received no other research attention. For more information see `http://www.puzzle.jp/letsplay/play_slitherlink-e.html` to attempt some Slither Link problems.

Sokoban

Sokoban is played on an $m \times n$ grid. Each cell is solid or empty, and empty cells can contain a stone or a target, such that there are k stones and targets. In the grid is a moving device. An action moves the device one cell horizontally or vertically as long as there is no solid cell blocking the way. Moving the device towards a stone will also move that stone one cell in the same direction, provided that the stone is not blocked by a solid cell or another stone. The goal is to move each of the stones to an empty target. Deciding solvability of Sokoban has been shown to be in NP-Hard by [108].

Junghanns and Schaeffer have performed various investigations on Sokoban. In [109] they apply techniques to IDA* in order to identify Sokoban states that provably cannot be solved. In [110, 111] they prune moves from the search that are not deemed to be relevant to the recent sequence of moves. In [112, 113] they advocate the use of domain specific techniques over general ones. In [114] they describe results of their Rolling Stone Sokoban solver. Murase et al. [115] describe a Sokoban problem generator. See `http://www.cs.ualberta.ca/~games/Sokoban/` for information relating to the history of Sokoban, a list of publications and a description of the Rolling Stone Sokoban solver, and also see Junghanns PhD thesis [32] that contains references to other work on Sokoban.

Sokoban is included in a larger set of problems that are known similarly as motion planning problems (see [116]) and pushing block problems (see [23]). In general there is a single moving device that can push (or pull) some number of objects simultaneously, of some predefined shape, which have inertia or not, in order to either arrange those objects or to move to a goal position, in 2 or 3-dimensions. These problems have generally been shown to be in NP-Hard. See [116] and [23] for more information about these other problems.

Spiral Galaxies

Spiral Galaxies is played on an $m \times n$ grid. k markers are within the grid, located in the centre of cells or on the partitions between cells. The goal is to seperate the grid into k segments such that each segment contains a marker at its centre, and each segment is rotationally symmetrical. Deciding solvability of Spiral Galaxies has been shown to be in NP-Complete by [117], which seems to be the only published work on this puzzle.

Sudoku

Sudoku (also known as Number Place) was first published in Dell Magazines in 1979[2]. It is currently attracting more attention than other puzzles, perhaps because it is featured regularly in daily newspapers. It is played on an $n^2 \times n^2$ grid that is divided into $n \times n$ equal segments. Initially some cells in the grid contain a number between $1 \ldots n^2$. An action inserts a number between $1 \ldots n^2$ in an empty cell. The goal is to fill the grid with numbers such that every row, column and segment contains the numbers $1 \ldots n^2$ without any repetitions. Deciding solvability of Sudoku has been shown to be in NP-Complete by [118].

Pereira et al. [119] apply constraint programming techniques to Sudoku. Nicolau and Ryan [120] offer an evolutionary approach to Sudoku. See [121] and [122] for other approaches to Sudoku. The world wide web is littered with Sudoku solvers. See `http://www.sudoku.org.uk/` for some sample problems that can be attempted online.

In this section we have detailed player updated puzzles. In the next section we shall detail self updated puzzles.

4.2 Self Updated Puzzles

Self updated puzzles, also known as interactive computer games, are generally simulated on a computer. They are different from player updated puzzles in that they will change their state even when the player chooses to pass. For this reason, if the player continues to pass then the problem might eventually solve itself or become unsolvable. Thus, a pass in these problems is explicitly modelled as an action.

Some possible techniques for dealing with self updated puzzles are as follows. A constructive search algorithm like A* might benefit from a move ordering scheme where the algorithm simply chooses to pass first of all choices at every decision point. This might help search by avoiding much of the irrelevant space consisting of non-pass

[2]see `http://en.wikipedia.org/wiki/Sudoku`.

choices that it might have searched otherwise. A local search algorithm such as GAs might benefit from an initialisation scheme that ensures that every solution starts off as a sequence made up of pass actions.

Lemmings

Lemmings is a popular computer game from the early 1990s. The rules of the game are complex, therefore this description shall be brief and not definitive. See chapter 5 for a definition of LEMMINGS, which is a derivative of Lemmings. Lemmings is played over a two-dimensional display of lemming entities that move around an environment. The player interacts with the lemmings. Each of the lemmings enter the environment at some specified position and must be guided to an exit. Each one that exits is counted. A lemming will continue to walk until either it is blocked whereby it turns around, or until it falls down a hole. An action assigns a type to a single lemming. There are eight types, which are

1. Climber. This makes the lemming climb up all vertical walls that it comes into contact with.
2. Floater. This makes the lemming float slowly down instead of falling, thus stopping it from dying on impact with the ground.
3. Bomber. This makes the lemming explode after a countdown. The explosion kills the lemming and destroys the peripheral environment, but it does not affect other lemmings.
4. Blocker. This makes the lemming stand still and block the movement of the other lemmings, so that they reverse direction.
5. Builder. This makes the lemming create a bridge.
6. Basher. This makes the lemming bash a horizontal tunnel through solid wall.
7. Miner. This makes the lemming mine a diagonal tunnel downwards through solid wall.

8. Digger. This makes the lemming dig a vertical tunnel downwards through solid platform.

Each type has a specified quantity, such that the number of times that a type is used during a play does not exceed its quantity. The goal is to guide a specified number m of the n lemmings through the environment to an exit within a specified time limit. Deciding solvability of Lemmings has been shown to be in NP-Complete by Cormode [22]. Cormode also shows that deciding solvability of a restriction to $m = n = 1$ is still in NP-Complete, and further that a restriction to climber and floater types only is decidable in P time.

McCarthy [21] offers Lemmings as a Drosophila for AI research. Recall that John McCarthy originally named the AI discipline at the Dartmouth conference in 1956 (see [123]). The game of Chess has traditionally been cited as the Drosophila of AI research. That is, any insight gained in AI for the game of chess might provide researchers with some understanding in AI for other games or maybe some other types of problems. Chess has recieved decades of attention from researchers until eventually Deep Blue [124] defeated the World Chess Champion Garry Kasparov in 1997. Research in Chess is still very active (see for example [125] and [126]) but it no longer attracts the media attention it did in the mid 90s. McCarthy advocates the use of Lemmings as a model for Logical AI and also suggests that other approaches might benefit from using Lemmings as a test domain. McCarthy suggests approaching Lemmings using Situation Calculus (see [1] p329). He further highlights the concurrent nature of the game that arises because there are many lemmings with which the player interacts and which are integral to the game. In [24] we approach a sub-problem of a derivative of Lemmings using a GA, and this work is detailed in chapter 6. More information about Lemmings can be found at `http://lemmings.deinonych.com/`, which offers information about the various Lemmings iterations and links to other sites.

Lemmings is interesting because of the complexity of its rules. It is clear from just this simple description that the puzzle is complicated to understand and describe. Unfortunately this also makes it complicated to implement and maintain a Lemmings simulator. On the plus side, these complexities make the problem much closer to real life whilst retaining the tight constraints and clearly defined goal of the other puzzles. It is for this reason, coupled with the classification of Lemmings as a self updated puzzle, that we advocate Lemmings as an AI research domain. We investigate LEMMINGS, that is derived from Lemmings, in chapters 5, 6 and 7.

Pac-Man

Pac-Man is a popular computer puzzle from the early 1980s that is played on an $m \times n$ grid. The grid is seperated into coridors, throughout which are distributed dots and pills. A single Pac-Man constantly moves around the grid and an action changes the direction of the Pac-Man. A number of ghosts also move around the grid and kill the Pac-Man upon contact. However, if one of the pills is eaten by the Pac-Man then, for a brief time, he kills the ghosts upon contact, whereby they regenerate in a central area. The goal is to navigate the Pac-Man through the grid in order to eat all of the dots before being killed three times. Currently, the complexity classification of Pac-Man is unknown.

Gallagher and Ryan [127] apply an evolutionary approach to learn to play a simple game of Pac-Man. Yannakakis and Hallam [128] report work on generating behaviour for the ghosts. Lucas [129] applied evolutionary neural networks to a version called Ms. Pac-Man.

Tetris

Tetris is a popular computer puzzle game that is played on an $m \times n$ grid. Each cell is either empty or contains a solid. The game is played with pieces that are made up of combinations of four squares of solid. The pieces fall downwards from the top of the grid until they either hit the bottom of the grid or hit another solid. A row of solids is deleted from the grid when it is formed, and everything above falls downwards. A tetris is the simulatneous deletion of four consecutive rows. An action moves the falling piece once to the left or right or rotates the piece. Demaine et al. [130] have reasoned about the offline version of tetris, that is where the sequence of pieces that will drop from the top of the grid are included as part of the problem instance. They show that the following are all NP-Complete - maximising the number of deleted rows, maximising the number of tetrises, maximising the number of pieces that are placed, or minimising the maximum height of a solid cell.

Siegel and Chaffee [131] and Yurovitsky [132] use Tetris to test an evolutionary algorithm. Germundsson [133] does some work on a variation of Tetris. For more information see `http://www.ebaumsworld.com/tetris.html` in order to play a version of tetris.

In this section we have outlined some self updated puzzles. In the next section we shall conclude the chapter.

4.3 Conclusions

In this chapter we have expanded Demaine's and Eppstein's surveys of puzzles by adding extra puzzles. We can analyse the puzzles in terms of the perceived amount of research effort towards them. Table 4.1 summarises our findings. The reason why some of these puzzles have received lots of research attention and others have received very little might be because the latter are not in the most easily accessible places. One motivation for this chapter is to bring all of the puzzles together in one place so that the reader knows where to look for them, if they wish to research them.

Most research	Some research	Research gaps	
Blocks World	Instant Insanity	Corral Puzzle	Nurikabe
n-Puzzle	Lemmings	Clickomania	Pearl Puzzle
Sokoban	Minesweeper	Cross Sum	Reflections
	Pac-Man	Cryptarithms	Shanghai
	Peg Solitaire	Cubic	Slither Link
	Sudoku	KPlumber	Spiral Galaxies
	Tetris	Light Up	

TABLE 4.1: An analysis of research effort towards puzzles.

As we can see from Table 4.1, Blocks World, n-Puzzle and Sokoban have received the most research effort. On the other hand, we have identified thirteen puzzles that constitute a research gap. This is over half of the surveyed puzzles. It is our hope that by highlighting these puzzles that researchers might be motivated to investigate them.

Maybe we can learn something from investigating puzzles. John McCarthy felt that Lemmings offered enough to be considered a Drosophila of AI research. The research carried out on those problems to the left of Table 4.1 have yielded some interesting findings. For example, the work by Slaney and Thiebaux [77] on Blocks World resulted in an appreciation of the structure of hard and easy instances, the work by Culberson and Schaeffer [95] on n-Puzzle resulted in the effective use of pattern databases, and the work by Junghanns and Schaeffer [110, 111] on Sokoban resulted in the relevance cuts enhancement. A direct research effort on the problems constituting a gap in the research might yield even more interesting results. Moreover, problems like Blocks World could yield even more insights than have already been revealed.

Any researcher who is interested in devoting some time to a puzzle might find Table 4.1 helpful. Those problems that have received the most research, up until now, will have a strong starting point from which further research might proceed. On the other hand, for those problems constituting a gap, the only information we have from which to start future work is the problem definition and the inclusion in NP-Hard or NP-Complete.

In this chapter we have motivated research on NP-Hard puzzles, and have also motivated our research on the Lemmings game. Therefore, we have achieved thesis goal 1 as outlined in section 1.3. We feel that it offers something different because it is a self updated puzzle, and furthermore we feel that its rules are substantially more complex than the other puzzles, so much so that it is closer to real life problems, whilst retaining the tight constraints and clearly defined goal of the other puzzles. We advocate the use of Lemmings as an AI research model.

In the next chapter we shall define the LEMMINGS problem, which is derived from Lemmings. Then in chapters 6 and 7 we shall describe our investigations of

LEMMINGS.

Chapter 5

LEMMINGS

An interesting problem for artificial intelligence

"*What is an intellectual problem?*
...a goal...an initial state...a set of operations
...constraints...(and)*...an outcome.*"
Patrick Dunleavy - Authoring a PhD Thesis

In this chapter we outline the **LEMMINGS** problem definition. We have seen a general definition of a problem in chapter 2, and our LEMMINGS definition here will be used in the investgations in chapters 6 and 7. In chapter 3 we detailed some algorithms that can be used for solving problems. In this chapter we shall define action, instance, solution, feasible solution and outcome with respect to LEMMINGS. These concepts shall be useful when it comes to discussing approaches to LEMMINGS in chapter 6. LEMMINGS is governed by rules that are abstracted from rules that dictate the popular computer game called Lemmings, which was described in chapter 4. In this chapter we shall highlight a history of Lemmings. As we have seen in chapter 4, there is a set of NP-Complete puzzles that includes Lemmings. In this chapter we suggest that LEMMINGS is in NP-Complete along with them. This advocates research on LEMMINGS.

5.1 The Problem Definition

In this section we shall describe the LEMMINGS problem, including actions, instance variables and solutions.

LEMMINGS is a one-player, deterministic game. It is displayed as a two dimensional image of discrete width and height, but no depth. Figure 5.1 shows the display of the problem, which consists of entities called **lemmings**, depicted here as 'L' or 'R', that move around an **environment**. Throughout this thesis we will use LEMMINGS when refering to our problem, Lemmings when citing the popular computer game and lemmings when speaking of the entities.

For convenience we tolerate a degree of anthropomorphism when describing the behaviour of lemmings. They appear to be alive and the player can interact with them. If the player ignores the lemmings then they will continue their lives and move around the environment in some way.

The environment is arranged as an $m \times n$ grid[1]. The environment is surrounded by a **boundary**. Each **cell** in the environment can be either an **entrance** that admits the lemmings, an **exit** that offers the lemmings a safe means of escape, or **empty space** through which the lemmings can move. Entrances are depicted as small hollow squares and exits are depicted as small grey-filled sqaures in Figure 5.1. The cells are seperated by **partitions**, depicted as thick black lines, with which the lemmings collide. Partitions are structured horizontally and vertically so that they form platforms and walls. A lemming can move outside of the environment in one of two ways, either by escaping through an exit, or by crossing the boundary of the environment. A lemming that escapes through an exit is counted and then removed from the environment. A lemming that crosses the environment boundary is killed. A lemming that dies is removed and is not counted.

[1]In this thesis we restrict to a 20×20 grid.

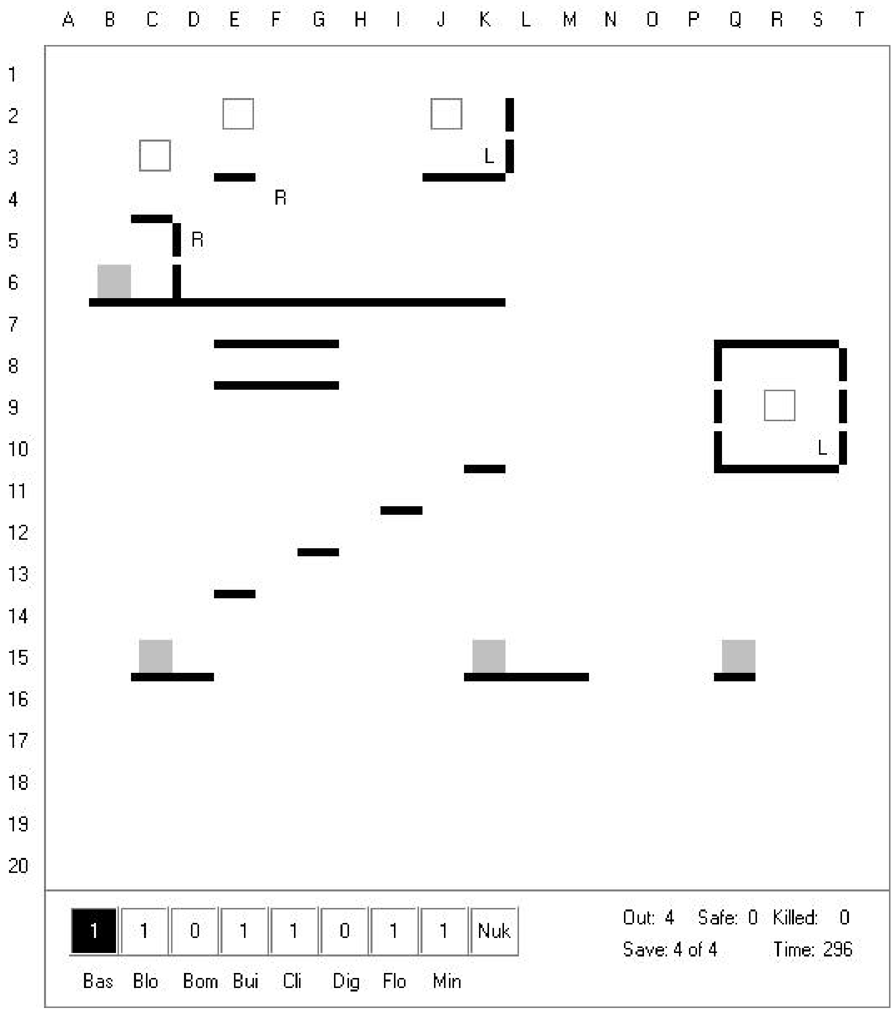


FIGURE 5.1: The display of the LEMMINGS problem.

The lemmings do not think for themselves. They react to instructions given by the player and to the rules of the LEMMINGS problem. Each lemming has a **type** and they enter the environment as a **walker** type. The lemmings are affected by gravity. A walker drops through empty space until it lands on a platform beneath it. If a walker drops further than five cells then, when it lands on a platform, it dies, otherwise when it lands it starts to move along the top of the platform in the direction that it faces. Walkers initially move to the right, depicted as 'R' in Figure 5.1. They will swap direction if they are blocked, at which point they start to move to the left, depicted as 'L', and will continue to do so until they are blocked again. A blockage is a wall that is at least two cells high, or a lemming of blocker type (see the list below for a description of the types that can be assigned to a lemming). A lemming will move through a wall that is only one cell high. A walker cannot stand still, and if one reaches the end of a platform it will continue over the edge and start to fall through empty space. A walker moves one cell in the direction that it is facing every time step if it is on a platform. A walker moves one cell downwards every time step if it is falling. A walker moves diagonally up onto a platform if the platform is at most one grid cell higher than the lemming and there is no blockage.

There are eight types that can be assigned to individual lemmings by the player. These types offer the lemmings a means of forming a path through the environment to an exit. There are two seperate stages that a lemming must complete in order to change its type. Firstly the lemming recieves a type assignment from the player, and secondly the lemmings type is actually changed to the type that has been assigned. There are different constraints for each stage. The following gives a brief description of the behaviour of a lemming of each type and the constraints imposed upon that type.

1. A **basher** lemming makes a tunnel of empty space one grid cell high as it moves horizontally through walls. If there are two consecutive walls, one wall next to the other one, then the lemming will bash through both of them. It will continue to bash whilst it is on a platform and there is a wall in front of it, otherwise its type changes back to a walker whereby the lemming forgets that it was assigned a basher type. A basher moves one cell in the direction that it is facing every two time steps.

   For a lemming to recieve this type it must be on a platform and have a current type of either walker, builder, digger or miner. A lemming changes its type to basher immediately upon receiving that type assignment, but only if it is not about to die from falling too far.

2. A **blocker** lemming stands still on top of a platform and blocks the lateral movement of the other lemmings, making them change direction. If a lemming

falls from above then a blocker does not block that downwards movement, and furthermore, the blocker is hidden from view by that falling lemming and so does not block any lemmings at all. It remains a blocker until it is removed from the environment, and inexperienced players assume that it is sacrificed[2].

For a lemming to recieve this type it must be on a platform and have a current type of either walker, basher, builder, digger or miner. A lemming changes its type to blocker immediately upon receiving that type assignment, but only if it is not about to die from falling too far.

3. A **bomber** continues to move as it did before being assigned this type, but counts down five seconds then explodes and dies, destroying the partitions, but not the lemmings, immediately surrounding it. The lemming is sacrificed when it explodes. A lemming can be a bomber and recieve an assignment of any one of the other seven types described.

   For a lemming to be assigned a bomber it can have any current type, but must not currently be a bomber. A lemming starts to count down immediately upon receiving this assignment.

4. A **builder** lemming creates platforms one grid cell wide behind it as it moves diagonally upward through empty space. It will continue to build until it has built three platforms at which point its type changes back to a walker, it stands still for one time step and the lemming forgets that it was assigned a builder type. If the platform is removed from under the lemming whilst it is building then it changes back to a walker and the lemming forgets that it was assigned a builder type. If a partition obstructs its way in front or above at any point, then its type changes back to a walker and the lemming forgets that it was assigned a builder type. A builder moves one cell in the direction that it is facing and also one cell upwards every two time steps.

   For a lemming to recieve this type it must be on a platform and have a current

[2]Experienced players know that if they destroy the platform underneath a blocker it will change back to a walker, whereby the lemming forgets that it was assigned the blocker type.

type of either walker, basher, digger or miner. A lemming changes its type to builder immediately upon receiving that type assignment, but only if it is not about to die from falling too far.

5. A **climber** lemming climbs up walls. It remembers that is was assigned a climber type and climbs all walls until it is removed from the environment. It will continue to climb up its current wall until there is no wall to climb, at which point it steps over the top of the wall and changes back to a walker type. If at any point it is blocked from above by a platform and is still climbing, then it changes to a walker type, turns around and starts to fall through empty space. A climber moves upwards one cell every time step.

   For a lemming to recieve this type it can have any current type except a climber, and cannot have ever been assigned a climber type. A lemming continues to move as it did before it was assigned the climber type. It changes its type to climber only when it touches a wall in order to climb up it, but only if it is walking along a platform, is not stood at an exit and is not about to die from falling too far, and only if the player has not just assigned it a basher, blocker, builder, digger or miner type.

6. A **digger** lemming makes a tunnel of empty space one cell wide as it moves downwards through platforms. If there are two consecutive platforms, one platform above the other one, then the lemming will eventually dig through both of them. It will continue to dig whilst there is a platform directly beneath it, otherwise its type changes to a walker whereby the lemming forgets that it was assigned a digger type. A digger moves downwards one cell every two time steps.

   For a lemming to recieve this type it must be on a platform and have a current type of either walker, basher, builder or miner. A lemming changes its type to digger immediately upon receiving that type assignment, but only if it is not about to die from falling too far.

7. A **floater** lemming drops slowly down through empty space until lands safely

on a platform whereby its type changes to a walker. It remembers that it was assigned a floater type and survives drops of any height until it is removed from the environment. A floater falls downwards through empty space one cell every two time steps.

For a lemming to recieve this type it can have any current type except floater, and cannot have ever been assigned a floater type. A lemming continues to move as it did before it was assigned the floater type. It changes its type to floater only when it has a walker type and has been falling through empty space for at least two cells.

8. A **miner** lemming makes a tunnel of empty space, two grid cells high and two grid cells wide, as it moves diagonally downwards through platforms and walls. A miner is like a digger except that it moves diagonally downwards and makes a wider tunnel. It will continue to mine whilst there is a platform beneath it, otherwise its type changes to a walker whereby the lemming has forgotten that it was assigned the miner type. A miner alternately moves one cell in the direction that it is facing every two time steps and one cell downwards every two time steps.

   For a lemming to recieve this type it must be on a platform and have a current type of either walker, basher, builder or digger. A lemming changes its type to miner immediately upon receiving that type assignment, but only if it is not about to die from falling too far.

Lemmings move at a constant speed depending on their current type. In other words, there is no acceleration or deceleration. For example, a walker constantly moves faster than a basher.

The player interacts with the lemmings in the following way. He first selects a type to assign, and then selects a lemming to which it will be assigned. The eight types are depicted towards the lower-left of Figure 5.1 where basher is the currently selected type. The default type is basher and it remains so until the player choses a different type to assign to a lemming, at which point that type remains the one that will be assigned to lemmings until a different type is chosen once again. Each type is associated with a **type-quantity**. When a type is assigned to a lemming, its associated type-quantity is decremented by one. These type-quantities are depicted in Figure 5.1 towards the lower-left of the figure, where all of the types have an associated type-quantity of one except for the bomber and digger types that have an associated type-quantity of zero. The player cannot assign any type if its associated type-quantity is zero. The player cannot assign any type to any lemming that is already of that type.

The player can **nuke** the lemmings once, which sets all of those already in the environment to a bomber type, and stops any more lemmings from entering. The nuke option is depicted in Figure 5.1 immediately to the right of the types.

Time in the lemmings world is discrete so that exactly one **action** occurs at each time step. An action amounts to either 1) the assignment of some type to a lemming[3] that is in the environment, 2) a nuke of all the lemmings simultaneously, or 3) nothing, which we call a **pass**. The current state of the game is displayed at every discrete time step so that the player can see the consequences of his actions.

The player is presented with LEMMINGS **instances**. Each instance consists of the environment[4], a finite number of lemmings that will enter, a set of initial type-quantities, a **time limit**, a **target** that defines a specified fraction of the lemmings that must be saved, and a **lemmings frequency** that defines the time between each lemming entering the environment.

The **initial state** of the game is when all of the lemmings are waiting to enter the environment, the type-quantities are reset, the partitions are in their initial arrangement and the time is at zero. Performing an action is called a **move**. The player can move at all time steps until the **outcome** of the game. There are two criteria for an outcome. The first criterion is that there are no more lemmings. That is, there are no more lemmings waiting to enter the environment and also that there are no more lemmings in the environment. The second criterion is that the time-limit is reached.

[3]Conceptually, we can combine a type selection and a type assignment into one single action since both can be performed within one discrete time step.

[4]The partitions are initially arranged so that the lemmings cannot navigate from entry to exit.

If either of these criteria are matched then the game ends with the outcome.

A **solution** to a LEMMINGS instance consists of a sequence of actions from the initial state to an outcome. A **feasible solution** yields an outcome such that the number of lemmings that have reached an exit is at least equal to the target. Note that the player continues to move until an outcome emerges at the end of the game, which might be after the target has been achieved. Therefore, the nuke action is used to force the end of the game once we have achieved the target, in order to get an outcome sooner.

In this section we have defined the LEMMINGS problem, including action, instance, outcome, solution and feasible solution. In the next section we shall describe some related issues.

5.2 Related Issues

In this section we shall discuss some issues that are related to the problem definition for LEMMINGS. First we shall describe some history of the popular computer game called Lemmings from which our LEMMINGS problem is derived. Then we shall discuss some complexity issues.

The Game of Lemmings

Lemmings is a well known computer puzzle game that was developed by DMA Design. It first went on sale for the Commodore Amiga on 14th February 1991 and was later converted for other home computer markets, and eventually games consoles. A number of sequels also appeared. The game is based on a popular myth that a type of arctic rodent, called a lemming, commits suicide by jumping from cliffs. The reader is pointed to `http://lemmings.deinonych.com/` for a general source of information about Lemmings on the web. Lemmings includes 120 levels[5], which range from easy to difficult and are distributed evenly over four stages, called 'fun', 'tricky', 'taxing'

[5]A Lemmings level is synonomous with a LEMMINGS instance.

and ’mayhem’. If the player solves the current level then they advance to the next one, otherwise they must retry. Our work is loosely based on the ’fun’ stage of Lemmings for Windows^{TM} (trademark of Psygnosis Limited), which consists of 28 levels from the original 30 for that stage.

Our LEMMINGS problem is derived from Lemmings. Lemmings included different kinds of material, for example material that was not permeable from a given direction, whereas LEMMINGS partitions are permeable from any direction and at all times. Also, Lemmings incorporates various hazards that kill the lemmings, for example water and fire, whereas LEMMINGS does not. Otherwise, the rules governing LEMMINGS are derived from the rules governing Lemmings, as the author understands them.

Complexity Issues

Cormode’s [22] illustration that Lemmings is in NP-Complete suggests that our derived LEMMINGS problem is also in NP-Complete. So we might say that our LEMMINGS problem is intractable. That is, there is no known exact algorithm that can solve it in polynomial-time. Therefore, we would be foolish to attempt to solve LEMMINGS using exact methods and we talk more about this in chapter 6. This also suggests that if we can find an exact algorithm that can solve LEMMINGS in polynomial-time, then we might have showed that P=NP (see chapter 2 for more details).

It should be noted that there are a few differences between our LEMMINGS problem and the Lemmings game that Cormode showed to be in NP-Complete. Firstly, Cormode allows multiple moves at each time-step. That is, at most one move on each lemming at each time-step, whereas we restrict the player to only one move per time-step regardless of the number of lemmings. Secondly, Cormode allows two different types of partitions[6], the permeable kind that can be destroyed, and the impermeable kind that cannot be destroyed at all. On the other hand, we restrict our

[6]Cormode refers to material that exists in place of air, but since the purpose of that material is the same as our partitions, that is to restrict the movement of the lemmings, then we can interchange between terms partition and material.

problem instances to use only the permeable kind of partitions that can all be destroyed. Thirdly, Cormode restricts his instances to only one exit, whereas we allow any number of exits. Regardless of these differences, we believe that our LEMMINGS problem is probably as complicated as Lemmings and thus in NP-Complete. Note that we have not proved this, and do not intend to, in our thesis.

In this section we have discussed the history of the popular computer game called Lemmings from which we derive LEMMINGS, and we have highlighted some complexity issues surrounding LEMMINGS. In the next section we shall detail the conclusions of the chapter.

5.3 Conclusions

In this chapter we have defined a relatively new problem called LEMMINGS that is derived from a popular computer puzzle game called Lemmings. This means that we can now investigate LEMMINGS in the following chapters. We have described how LEMMINGS is deterministic and single player, which means that in chapter 6 we can focus our approaches to LEMMINGS towards offline search. We have defined action, instance, solution, feasible solution and outcome with respect to LEMMINGS. These can be used in describing our approaches to LEMMINGS in chapter 6. We have described some related issues, including a history of Lemmings, and we have suggested that LEMMINGS is in NP-Complete and therefore is an interesting and challenging problem for AI. Note that we do not prove membership of LEMMINGS in NP-Complete.

In the next chapter we shall discuss different approaches to LEMMINGS, focusing on search. We shall also describe a successful evolutionary approach to LEMMINGS. In chapter 7 we shall describe an investigation in order to locate difficult instances of LEMMINGS, and we shall describe our results.

Chapter 6

Approaching LEMMINGS

A discussion of various approaches towards LEMMINGS

"*The solution of every problem is another problem*"
Johann Wolfgang von Goethe

In this chapter we discuss various possible approaches to computing feasible solutions for LEMMINGS instances. We have introduced the NP-Complete decision problems in chapter 2 and have defined the LEMMINGS problem in chapter 5. Our survey in chapter 4 motivates work on NP-Complete puzzles especially Lemmings. We have highlighted the fact that there is currently very little work surrounding LEMMINGS, in particular there is no known approach for LEMMINGS. The motivation for this chapter is to find a successful way to approach LEMMINGS, so that in chapter 7 we can investigate the problem in order to find difficult instances.

In chapter 3 we described various search algorithms and we shall restrict ourselves to these approaches, but there are many other possible approaches to the LEMMINGS problem. For example, we might employ an agent based approach. In this case each lemming shall make a decision about an action to perform, and shall do so at each time-step. This approach would be more powerful than LEMMINGS requires, because it would handle nondeterminism. For example, it might be suitable for an extension of LEMMINGS that contains environment that randomly changes during a play of the game. One way to achieve this approach might be to use a neural network whose input is a portion of the environment of predefined size surrounding the lemming in question. The output might be an integer representing an action for that lemming. The neural network can be trained using an evolutionary algorithm. This is similar to the technique used in [60].

We start off with a theoretical discussion of search applied to LEMMINGS (see section 6.1). However, we shall also describe an approach to a sub-problem of LEMMINGS, called α-LEMMINGS, in section 6.2, where we shall investigate different initialisation schemes for the approach to α-LEMMINGS. We shall also describe a similar approach for the general LEMMINGS problem in section 6.3. The approach will receive a small investigation in order to find the best parameters to use. This is intended to give a more detailed description of how best to approach LEMMINGS. In this chapter we do not provide details of how 'well' our approach works on LEMMINGS. In chapter 7 we shall give a detailed analysis of the time required by our approach.

6.1 Theoretical Discussion

In this section we shall discuss the theoretical possibility of approaching LEMMINGS in different ways.

LEMMINGS is deterministic so we can search offline. We might use an uninformed search algorithm such as one of those detailed in section 3.2 to build a search tree corresponding to a LEMMINGS instance. A part of the tree might look something like that shown in Figure 6.1. Each node in a search tree corresponds to a problem state and each branch corresponds to an action. Recall, from chapter 5 that an action in LEMMINGS amounts to either 1) the assignment of some type to a lemming, 2) a nuke of all the lemmings simultaneously, or 3) pass. To this end we represent an

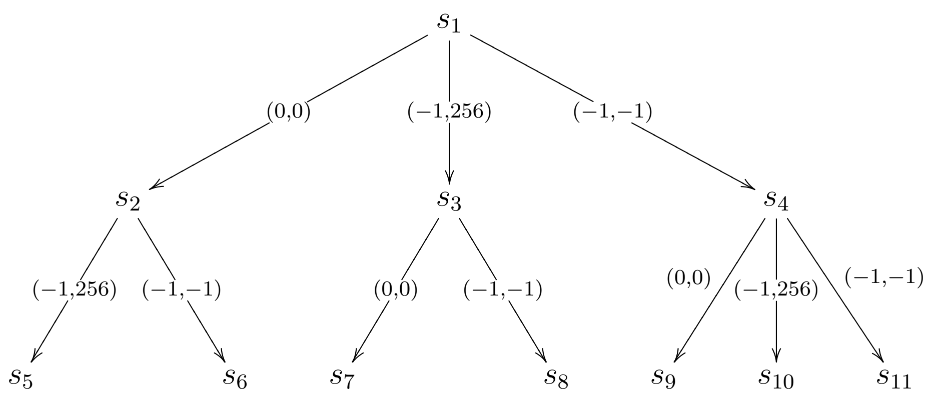


FIGURE 6.1: A LEMMINGS Search tree.

action as a pair (l, t), where l corresponds to a lemming and lies in the range $L = \{-1, 0, \ldots, nl - 1\}$ such that nl is the number of lemmings, and where t corresponds to some type and lies in the range $T = \{-1, 256, 0, \ldots, 7\}$. L contains an index to each lemming and also an index pointing to nothing (-1), and T contains an index to each of the eight LEMMINGS types where 0 corresponds to the basher type and 7 corresponds to the miner type. T also contains an index to nothing (-1) and an index to the nuke command (256). This representation allows us to model an action. The assignment of some type t to a lemming l can be modelled where t lies in the range $T - \{-1, 256\}$, and where l lies in the range $L - \{-1\}$. A nuke is modelled as (-1, 256), and a pass as (-1, -1).

The tree in Figure 6.1 shows that in state s_1 the player can assign the basher type to the lemming at index zero, or the player can nuke, or the player can pass. If the player choses to assign the basher type to the lemming then the next state will be s_2, and the player can nuke or pass. The rules of LEMMINGS forbid assigning a type to a lemming that is already of that type, which in this case means that the player cannot assign the basher type to a lemming that is already a basher. If the player had originally chosen to nuke in s_1 then the next state will be s_3, and the player can assign the basher type to the lemming at index zero or pass. The rules of LEMMINGS also state that the player can only nuke once, so the player cannot nuke again in s_3. Finally, if the player had originally chosen to pass in s_1 then the next state will be s_4 and the player can perform all of the actions that he could perform at s_1.

It is easy to see that if the LEMMINGS instance is solvable then there will be at least one goal node in the search tree that corresponds to the problem in a solved state, and that it is reached via a path of actions from the root. This list of actions is a solution that will solve that instance. LEMMINGS is bounded by a time limit, and since one action occurs each time step, any tree will have bounded depth. Consequently any of the uninformed search algorithms detailed in section 3.2 are guaranteed to find a goal node where one exists. In other words, for any solvable LEMMINGS instance, any uninformed search algorithm will eventually find a feasible solution. Unfortunately, if we look at the time and space complexity of these algorithms (see Table 3.1) then we shall see that it is not wise to call upon any uninformed algorithm to perform a search for LEMMINGS.

We can bound the number of lemmings by 10. It might be possible to conceive a LEMMINGS instance where the player can assign any one of the eight LEMMINGS types to any of the lemmings at some game state. Then, since the player can also nuke and pass at each game state, this gives us the maximum number of successors of any node $b = 12$. The depth of the goal with the shortest path for this instance might conceivably be the maximum depth of the tree. The maximum depth of the tree is equal to the time limit for that instance. If we set a bound for the time limit at 100, then we have $d = m = 100$. The instance that we talk about might correspond to the one shown in Figure A.50. Then, refering to Table 3.1, we see that the number of nodes generated by BFS is $O(12^{101})$ and for DFS and IDS is $O(12^{100})$. We also see that the maximum number of nodes stored in memory by BFS is $O(12^{101})$, and for DFS and IDS is $O(12*100)$. Clearly, it would be foolish to attempt to approach LEMMINGS using one of these uninformed search algorithms. It would take too much time. Therefore, we cannot use one of these uninformed algorithms to approach LEMMINGS. Alternatively, LEMMINGS might be approached successfully using a random sampling technique like that described in section 3.2. We hope that given the time available to us, a random sampling will give us more feasible solutions than a full enumeration.

Cormode [22] showed that Lemmings is an NP-Complete problem. LEMMINGS is derived from Lemmings. This also provides evidence that it would be foolish to approach LEMMINGS using exact methods like these uninformed search algorithms. Therefore, we might employ an informed algorithm. The GA that we successfully apply to LEMMINGS (described later) is a form of informed random sampling.

McCarthy [21] himself suggests that Lemmings might be solved using search, but argues that the branching is high enough to make a brute-force search of the full tree infeasible. Therefore, McCarthy proposes some methods for reducing the size of the tree. Firstly, we might heuristically group the individual lemming entities into equivalence classes so that assigning a type to any one of the lemmings in a given class has the same outcome as assigning that type to any of the lemmings in that class. Secondly, we might seperate the environment into regions that are heuristically different, so that assigning a type to a lemming at one point in a given region has the same outcome as assigning that type to that lemming at any point in that region. Thirdly, we might heursitically order the moves that are examined so that, for example, since the player of Lemmings does nothing most of the time, the pass move is examined first for the majority of the states. These methods comprise an heuristic approach, and one that we shall not take in this investigation. The reason that we use a GA rather than an heuristic approach is because the GA does not require very much detail about LEMMINGS in order for us to implement the approach. That is, we can more easily implement the GA because it uses random operators.

We use a GA that starts with a random candidate solution to a LEMMINGS instance and optimises that solution whilst it still has time in the hope that eventually it can produce a feasible solution. Recall that a solution to LEMMINGS is a sequence of actions from the start of the game until an outcome, with one action for each time step. Thus, a candidate solution corresponds to a sequence of decisions made from the root node of a search tree that arrives at some terminal node.

In Figure 6.2 we see a search tree on the left that a search algorithm like A* might construct, and a complete solution on the right that lies within the same tree. We use the representation on the right to optimise a solution for a LEMMINGS

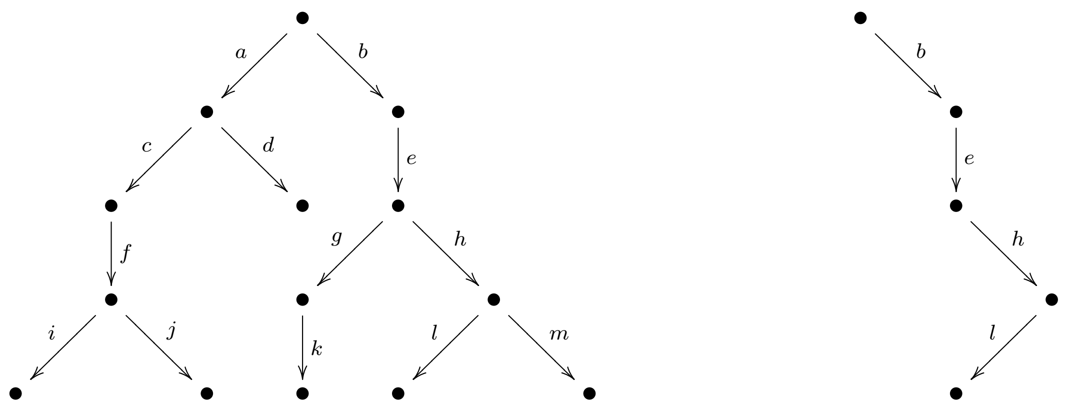


FIGURE 6.2: A search tree on the left and a complete solution on the right.

instance. That is, we start with some arbitrary solution that begins at the initial state and results in some outcome, and we iteratively change the actions making up that solution until the resultant outcome is a goal state. We provide the algorithm with information as to the quality of the current solution. That way the algorithm can continue to improve the solution.

We use a genetic algorithm in order to approach LEMMINGS because a GA is a form of informed random sampling. "*Genetic algorithms can be used where more convergent techniques dare not tread ... do sort out interesting areas of a space quickly, but they are a weak method, without the guarantees of more convergent procedures.*" [56] (p74). Essentially, a GA is not guaranteed to solve our problem (unlike a full enumeration), but makes up for this by quickly locating the better solutions in the solution space. A recent publication [57] suggests we "*start by using an 'off the shelf' genetic algorithm...GA-LIB is probably seen as the software of choice for many people.*"

In this section we have discussed various approaches to LEMMINGS. We have concluded that we shall not use uninformed search, but instead we shall use a form of informed random sampling called a genetic algorithm. In the next section we describe our approach in greater detail and discuss a preliminary investigation of our approach.

6.2 Approaching α-LEMMINGS

In this section we describe an approach that can successfully produce a feasible solution to instances of a subproblem of LEMMINGS that we call α-LEMMINGS. The purpose of this is simply to show that the approach and problem class are compatable, and thus that we may continue with further investigations. We offer an evolutionary approach to α-LEMMINGS, and perform an investigation into different initialisation schemes for the approach. The work related in this investigation is based on work that has been published (see [24]). Note that our results in this section are different from those published in [24] because we repeated the investigation after publication, and our new results were slightly different. First we shall detail our approach to α-LEMMINGS and then we shall discuss the results.

Method

We restrict ourselves to a subproblem of LEMMINGS. Our subproblem is the same as LEMMINGS except for the following

- The miner type has been ommitted. This means that there are a maximum of seven types.
- A lemming moves at the same speed regardless of its type. This means that, for example, a basher will move at the same speed as a walker.
- A wall of any height will block a lemming. This means that a wall that is only one cell high will block a lemming.
- A floater lemming will start to float immediately upon falling. This means that the lemming will not fall for two cells before floating.

This subproblem is called α-LEMMINGS. Solutions are evolved using a genetic algorithm (see chapter 3). We designed a set of seven α-LEMMINGS instances by hand, using an editor, on which to test the algorithm (see Figures 6.3 to 6.9). We tried to design the instances so that they required the use of each of the seven α-LEMMINGS types. Refer to Figures 6.7 to 6.9. The instances in these figures have a very similar environment surrounding the entrance, and the first part of a feasible solution for these instances will be similar. Our genetic algorithm is based upon the GAlib genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology[1].

Our GA works over a population of possible solutions to an α-LEMMINGS instance. A solution is represented as an array of actions that has length TL, where TL is equal to the time limit imposed by the α-LEMMINGS instance. Each action is an (l, t) pair, where l corresponds to a lemming and and where t corresponds to some type, where 0 corresponds to the basher type and 6 corresponds to the floater type. Thus, there is an action for each time step during a play of α-LEMMINGS. For example, the solution $\{(1,2),(4,1),(2,5),(2,1),(3,1)\}$ for an instance that has a time limit of five, means the following. In the first time step, lemming 1 is assigned the bomber type, in the second time step, lemming 4 is assigned the blocker type, and so on. In this way our solution is like a script, which is referenced at each time-step of a play of α-LEMMINGS. Each action is interpretted by the α-LEMMINGS simulator and the relevant lemming is given the particular order. Illegal actions are interpretted by α-LEMMINGS as 'pass'.

We test four initialisation schemes, called TYPEA, TYPEB, TYPEC and TYPED. In TYPEA, each solution in the population is initialised to pass actions, that is

$$\{(-1,-1),(-1,-1),(-1,-1),(-1,-1),(-1,-1)\ldots\}$$

In TYPEB, each solution in the population is initialised to random actions, that is

$$\{(-1,6),(3,5),(1,-1),(-1,256),(-1,-1)\ldots\}$$

In TYPEC, when we tackle Instance1, each solution in the population is initialised to pass actions, and then for each instance after, each solution in the population is

[1]See `http://lancet.mit.edu/ga/`

initialised to the result given by the GA for the previous instance. In TYPED, when we tackle Instance1, each solution in the population is initialised to random actions, and then for each instance after, each solution in the population is initialised to the result given by the GA for the previous instance.

Each solution is then used to play the α-LEMMINGS instance. The α-LEMMINGS outcome resulting from playing the genotype solution represents the phenotype. The solution is evaluated and assigned a score according to how well it solved the instance. This evaluation is of the form $score = (\alpha + \beta + \delta + \gamma)/10$. We divide the score by 10 to keep it within a preferred range. A higher score indicates a better solution.

$$\alpha = (numStartingLemmings * TL) * numLemmingsSaved * 10$$

$$\beta = (numStartingLemmings * timeRemaining) * 10$$

α rewards every lemming that reaches an exit. β gives a higher reward if the instance is solved in less time. β is set to zero if the number of lemmings to reach an exit is zero. $\alpha + \beta$ is the most intuitive way to reward success. However, it does not reward any lemmings that do not reach an exit.

$$\delta = \sum_i (lemming[i].life)$$

δ rewards lemmings that remain active in the environment for longer. Here i is an index for each lemming that entered the environment. δ sums the total life of all lemmings. A lemming's life is incremented once for every time step that it is active in the environment. δ is never more than ($numStartingLemmings$ * TL). α is deliberately set so that if a single lemming reaches an exit, then the evaluation of α is more than the evaluation of δ if all of the lemmings were to remain in the environment for the duration of the game.

$$\gamma = \sum_{x,y} (explored[x][y]) * 10$$

γ rewards a more diversely explored environment with an emphasis on horizontal platform exploration. Here x and y are the coordinates of the environment grid. Each cell in the environment initially has an explored score of 0. At the end of the game each cell is scored according to the following

$$explored[x][y] = \begin{cases} 0 & \text{if no lemming entered } cell[x][y] \text{ during the game} \\ 1 & \text{if any lemming entered } cell[x][y] \text{ during the game} \\ 2 & \text{if any lemming entered } cell[x][y] \text{ during the game and if} \\ & cell[x][y] \text{ also has a horizontal platform beneath it} \end{cases}$$

Two parent solutions are then selected from the population with a probability according to their score, and two offspring are produced using one-point crossover. For example, the parent solutions p_1 and p_2 produce the offspring c_1 and c_2, such that each parent has a unique cross-point and that the length of the solution is maintained during crossover.

$$\begin{array}{ll} p_1 = \{(1,2)\ \ (4,1) \parallel (2,5)\ \ (2,1)\ \ (3,2)\} & c_1 = \{(1,2)\ \ (4,1)\ \ (8,0)\ \ (1,2)\ \ (4,1)\} \\ p_2 = \{(7,1)\ \ (5,2)\ \ (8,0) \parallel (1,2)\ \ (4,1)\} & c_2 = \{(7,1)\ \ (5,2)\ \ (8,0)\ \ (2,1)\ \ (3,2)\} \end{array}$$

Each action is a pair of gene variables. The set of alleles for a variable is defined as the set of all of its possible values. The set of alleles for the l variable includes an index for each lemming that enters the environment of the α-LEMMINGS instance and also an extra index that has the value -1 that indicates a pass. For example, if five lemmings enter the environment then the set of l alleles is initialised as $lAlleles = \{-1, 0, 1, 2, 3, 4\}$. The set of alleles for the t gene variable includes all of the types in the instance that have an associated type-quantity greater than zero and also two extra types, the first has the value -1 that indicates pass, and the second has the value 256 that indicates a nuke. For example, if the blocker and climber types in the instance have a type-quantity of greater than zero then the set of t alleles is $tAlleles = \{-1, 256, 1, 4\}$.

The two new offspring solutions are then mutated using flip mutation. Random genes are replaced with a random selection from the appropriate variable allele set. For example, the third gene of c_1 is mutated.

$$(pre - mut)c_1 = \{(1,2)\ \ (4,1)\ \ \mathbf{(8,0)}\ \ (1,2)\ \ (4,1)\}$$

$$(post - mut)c_1 = \{(1,2)\ (4,1)\ \textbf{(-1,-1)}\ (1,2)\ (4,1)\}$$

We hope that the best evolved solution will have the optimal number of lemmings being assigned a type. So crossover and mutation are also applied to the set of alleles for the l gene. For example the parent allele sets $lAllelesP_1$ and $lAllelesP_2$ produce the offspring $lAllelesC_1$ and $lAllelesC_2$ such that each parent has a unique cross-point and that the size of the allele set can change during crossover.

$$\begin{array}{ll} lAllelesP_1 = \{-1, 0, \|2, 6, 8\} & lAllelesC_1 = \{-1, 0, 9\} \\ lAllelesP_2 = \{-1, 2, 7, 8, \|9\} & lAllelesC_2 = \{-1, 2, 7, 8, 2, 6, 8\} \end{array}$$

$$(pre - mut)lAllelesC_1 = \{-1, \mathbf{0}, 9\}$$

$$(post - mut)lAllelesC_1 = \{-1, \mathbf{6}, 9\}$$

Thus, there is a single, static set of alleles for the t gene, which is shared by the entire population of solutions. On the other hand, each solution has its own, dynamic, set of alleles for the l gene.

The two new offspring are then placed into the new population and the process is iterated to form the next generation. An elitist approach is taken. That is, if the previous generation best solution has a higher fitness than the new generation best solution, then that previous generation best is copied to the new generation.

The solution with the highest score overall when every generation has evolved is used in an attempt to solve the α-LEMMINGS instance. Note that the GA attempts to optimise the α-LEMMINGS solution, and continues even after it has found a feasible solution until it has exceeded the maximum generation count.

Roulette wheel selection is used with a crossover rate of 0.6 and a mutation rate of 0.001. These values are taken from [134]. The GA was run for 2000 generations. Population size is set to 250 in an attempt to avoid premature convergence of the population [135].

Results

Our genetic algorithm was able to produce a feasible solution to each of the seven α-LEMMINGS instances. The results of this preliminary study show that this approach is a suitable method for producing feasible solutions to α-LEMMINGS.

We ran our GA on each of the seven instances, with each of the four inititialisation schemes, and repeated the run five times. From those five runs, we take the run with the highest scoring result, and then trace the score of the best solution during that run of the GA. Figures 6.10 to 6.16 show the results. The interesting results are in Figures 6.15 where TYPEA achieves a worse final score, and 6.16, where TYPEA and TYPEB achieve a worse final score. We would expect this result. Refer to Figures 6.7 to 6.9. The instances in these figures have a very similar environment surrounding the entrance, and the first part of a feasible solution for these instances will be similar. TYPEC and TYPED are initialised to the result of the run for the previous instance, so the run for instance 6 and instance 7 will have a headstart from the result for instance 5 and 6 respectively. Thus, it seems that TYPEC and TYPED are better when the instances are similar enough, but we cannot say whether it is beneficial overall to use TYPEC or TYPED when attempting to solve arbitrary α-LEMMINGS instances. Also note, the solution provided by TYPEA was good enough to solve instance 6 and 7 despite it underperforming. From these results, we cannot draw a firm conclusion with regards to the best initialisation scheme for our GA on α-LEMMINGS. Each scheme is as good as the other for five instances. For the other two instances, TYPEA is worse but still solves the instance, and TYPEC and TYPED are better, probably because of the similarity between the instances.

The obvious question that arises out of this preliminary study is whether or not our approach can produce feasible solutions to the general LEMMINGS problem. We shall answer this question in the next section.

We witnessed some effects of the evaluation function on the solution that at first seemed as though they were unwanted. For example, the δ term caused the solution to evolve in such a way that lemmings were remaining active in the environment when a human player would guide the same lemmings more directly to an exit. Also, the

γ term caused the solution to evolve in such a way that some lemmings apparently needlessly explored the environment when they had already found the route to an exit. These effects are caused by the algorithm attempting to improve the solution in terms of the evaluation function, which was possibly not as good as it could have been. It turns out that these solution behaviours are not so much unwanted, they are merely unneccessary, and do not actually stand in the way of the solution successfully solving the instances. Thus our approach is successful for α-LEMMINGS.

In this section we have described a GA that can successfully approach α-LEMMINGS, and have discussed various initialisation schemes. In the next section we shall show that a similar approach can be applied to LEMMINGS.

6.3 Approaching LEMMINGS

In this section we describe an approach that can produce a feasible solution to instances of general LEMMINGS. The purpose is to investigate our approach a little further in order to answer the following questions

- Can our approach produce feasible solutions to instances of the general LEMMINGS problem?
- What are the best parameters for our approach?

In this investigation our approach shall have a new emphasis. That is, our approach to α-LEMMINGS was designed to optimise solutions, whereas our approach to LEMMINGS shall now find any feasible solution to an instance. This change of emphasis is in preparation for the invesitgation in chapter 7 to find difficult LEMMINGS instances.

First we shall describe the approach to LEMMINGS, then we shall discuss the best parameters that we found.

Approach

Solutions to LEMMINGS instances are evolved using a genetic algorithm (see chapter 3). We have implemented a new GA for our LEMMINGS problem. This is because our investigation into an approach for LEMMINGS was performed over two years after the preliminary approach to α-LEMMINGS. In this time we updated the LEMMINGS simulator and learned more about GAs, and this is reflected in the new GA. The differences between the two GAs are summarised later in Table 6.1. However, once again the GA is based on GAlib.

Our GA works over a population of possible solutions to a LEMMINGS instance. Our solutions use the same representation as in the previous approach. LEMMINGS has eight types in contrast to the seven for α-LEMMINGS, so t now lies in the range $0 - 7$ and $t = 7$ corresponds to the miner type in an action for LEMMINGS.

Each solution in the population is initialised to pass actions, that is

$$\{(-1,-1),(-1,-1),(-1,-1),(-1,-1),(-1,-1)\ldots\}$$

and then each solution is used to play the LEMMINGS instance, after which the solution is evaluated and assigned a score according to how well it solved the instance. The solution is also told if it successfully solved the instance. The evaluation is of the form $score = \theta + \lambda + \sigma$. A higher score indicates a better solution and our evaluation is always non-negative.

$$\theta = numLemmingsSaved * numStartingLemmings * TL$$

θ rewards every lemming that reaches an exit.

$$\lambda = \sum_i (lemming[i].life)$$

λ rewards lemmings that remain active in the environment for longer. Here i is an index for each lemming that entered the environment. λ sums the total life of all lemmings. A lemming's life is incremented once for every time step that it is active in the environment. λ is never more than ($numStartingLemmings$ * TL). Once again, TL is equal to the time limit for the LEMMINGS instance. θ is deliberately set so that if a single lemming reaches an exit, then the evaluation of θ is more than the evaluation of λ if all of the lemmings were to remain in the environment for the duration of the game. This is done so that θ and λ can both be maximised with a greater emphasis placed on maximising θ.

$$\sigma = \sum_{x,y} (explored[x][y])$$

σ rewards a more diversely explored environment with an emphasis on horizontal platform exploration. Here x and y are the coordinates of the environment grid. Each cell in the environment initially has an explored score of 0. At the end of the game each cell is scored according to the following

$$explored[x][y] = \begin{cases} 0 & \text{if no lemming entered } cell[x][y] \text{ during the game} \\ 1 & \text{if any lemming entered } cell[x][y] \text{ during the game and if} \\ & cell[x][y] \text{ does not have a horizontal platform beneath it} \\ 2 & \text{if any lemming entered } cell[x][y] \text{ during the game and if} \\ & cell[x][y] \text{ also has a horizontal platform beneath it} \end{cases}$$

The evaluation function for the approach to LEMMINGS is different from the evaluation function from the preliminary approach. The main difference is that the β term has been removed. This is because we no longer care about the quality of a feasible solution, all feasible solutions have equal worth, and β differentiated between two feasible solutions, so β is now irrelevant.

The fitness of each solution is scaled according to linear scaling as defined in GAlib (see the parametric investigation below). We did not perform scaling in the preliminary approach. The reason that we have added it now is because we have a more in depth knowledge of genetic algorithms and [56] recommends fitness scaling. Two parent solutions are then selected from the population using roulette wheel selection, and two offspring are produced using one-point crossover. For example, the parent solutions p_1 and p_2 produce the offspring c_1 and c_2, such that both parents use the same cross-point and that the length of the solution is maintained during crossover.

$p_1 = \{(1,1)\ (0,5)\ (2,1) \parallel (0,1)\ (1,1)\}$ $\quad$ $c_1 = \{(1,1)\ (0,5)\ \mathbf{(2,\text{-}1)}\ (1,0)\ (5,1)\}$

$p_2 = \{(0,1)\ (2,3)\ (0,5) \parallel (1,0)\ (5,1)\}$ $\quad$ $c_2 = \{(0,1)\ (2,3)\ (0,5)\ (0,1)\ (1,1)\}$

Once again the set of alleles for the l variable includes an index for each lemming that enters the environment of the LEMMINGS instance and also an extra index that has the value -1 that indicates a pass. For example, if five lemmings enter the environment then the set of l alleles is initialised as $lAlleles = \{-1, 0, 1, 2, 3, 4\}$. The set of alleles for the t gene variable includes all of the types in the instance that have an associated type-quantity greater than zero and also one extra index that has the value -1 that indicates pass. For example, if the blocker and climber types in the instance have a type-quantity of greater than zero then the set of t alleles is $tAlleles = \{-1, 1, 4\}$. In the new approach we no longer include the nuke as an action. This is because a nuke will only result in a feasible solution becoming a better feasible solution. Therefore, a nuke has no use.

During crossover each gene from the parent is mutated, with a probability, to the corresponding gene in the offspring. The gene is first copied from parent to child, and if a mutation occurs then the child gene variables are randomised so that either the child l variable is not equal to the parent l or the child t is not equal to the parent t. Randomising a gene variable involves randomly selecting an item from the appropriate variable allele set. For example, in the crossover illustrated above, the third gene of c_1 is mutated.

In this new GA we do not reimplement the dynamic evolving *lAlleles* from the preliminary approach. This is because the GA for LEMMINGS that is documented here is good enough for what we require (as verified in the next chapter), so we do not need to reimplement any extras or possible improvements, for example the evolving *lAlleles*.

The two new offspring are then placed into the new population, and once again evaluated on their success on solving the instance. The process (Select parents, crossover, evaluate) is repeated to form the next generation. An elitist approach is taken, that is, the previous generation best is copied to the new generation (see the parametric investigation below). The new generation then replaces the old generation.

Component	GA for α-LEMMINGS	GA for LEMMINGS
Evaluation Function	$(\alpha + \beta + \delta + \gamma)/10$	$\theta + \lambda + \sigma$
tAlleles	$\{-1, 256, 0, 1, 2, 3, 4, 5, 6\}$	$\{-1, 0, 1, 2, 3, 4, 5, 6, 7\}$
lAlleles	Dynamic	Static
Stopping criteria	Generation count	Feasible solution/ Generation Count
Scaling	-	Linear Scaling

Table 6.1: A summary of the differences between the two GAs.

Test	Score
Scaling off/Elitism off	3818.9
Scaling on/Elitism off	3810.6
Scaling off/Elitism on	4066.7
Scaling on/Elitism on	4211.5

Table 6.2: Results for scaling and elitism tests.

The best solution that has been found during the run is saved after each generation, and the GA stops once that best solution solves the instance. The best solution found is then taken as the result. The GA for approaching LEMMINGS is different from the preliminary one for α-LEMMINGS in that, the new one stops once a feasible solution has been found whereas the preliminary one continues even after a feasible solution is found.

We tested our approach on the eight LEMMINGS instances shown in Figures A.1, A.11, A.21, A.31, A.41, A.51, A.61, and A.71. Each instance incorporates one of the eight LEMMINGS types. Our approach successfully computed a feasible solution to each of these eight instances. We take this as an indication that our approach is successful for LEMMINGS. In the next chapter we discuss results that show the amount of time required by our approach for LEMMINGS instances of varying difficulty.

Parameter Investigation

The goal of this investigation is to find a set of good parameters with which to execute our approach to LEMMINGS.

Test	Instance 1	Instance 2	Instance 3
pop = 50, mut = 0.05, cross = 0.6	7,425	6,500	124,500
pop = 100, mut = 0.05, cross = 0.6	24,240	12,000	156,300
pop = 50, mut = 0.05, cross = 0.0	30,750	9,250	187,000 (9)
pop = 50, mut = 0.05, cross = 1.0	**5,840**	7,500	**103,250**
pop = 50, mut = 0.1, cross = 1.0	48,075	-	-
pop = 50, mut = 0.1, cross = 0.6	-	28,250	127,500
pop = 50, mut = 0.025, cross = 1.0	7,595	-	-
pop = 50, mut = 0.025, cross = 0.6	-	**5,250**	255,000

Table 6.3: Results for parameter tests.

First we assess the benefit of using an elitist strategy, and also scaling the population. We ran the GA with and without elitism, and with and without scaling, for 1000 generations, with a population of 50, a crossover rate of 0.6 and mutation rate of 0.05, on the LEMMINGS instance shown in Figure 5.1 and we analysed the average of the highest scoring solution that was found over 10 runs. The results are shown in Table 6.2. As can be seen, the best performance was witnessed when both elitism and scaling were on. Therefore, we conclude that elitism and scaling should be used.

We then assess the effect of changing the parameters for the GA. We ran the GA using different values for the population size, mutuation probability and crossover probability parameters, ensuring that we maintained a maximum number of solutions of 500,000, and analysed the number of solutions that were sampled in order to locate a feasible solution. We aim to minimise the number of sampled solutions required to locate a feasible solution. We repeated each test 10 times on three different LEMMINGS instances. The results are shown in Table 6.3. The GA only solved Instance3 9 times out of 10 for the third test. As can be seen, there are different best parameters for the instances, as written in bold. However, the first set of parameters worked second best for all three instances. The conclusion that we came to was that this set of parameters seemed to work the best for our problem. That is, population size is set to 50, crossover rate is 0.6 and mutation rate is 0.05. These values agree with [57]. The source that provided us with parameters for this approach is different from the source for the preliminary approach. This is simply because we now have more knowledge of GAs and this new source is more recent.

In this section we have described the approach that can be successfully applied to LEMMINGS, and described the parameters that work best. In the next section we conclude our chapter.

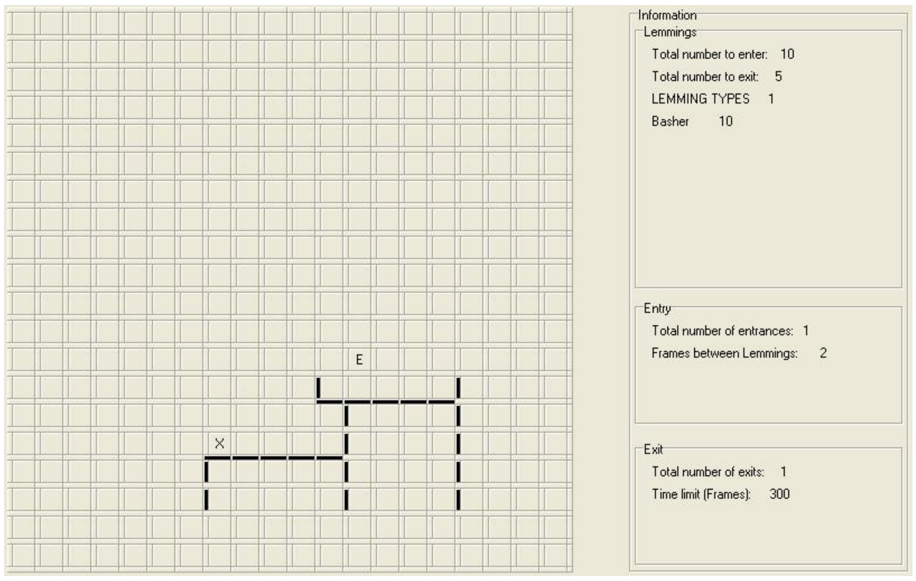
6.4 Conclusions

In this chapter we have discussed various possible approaches to the LEMMINGS problem, focusing on search. There was previously no known approach for LEMMINGS. We have shown that it is theoretically infeasible to approach LEMMINGS using uninformed search. We have described a GA that will successfully approach α-LEMMINGS, and have analysed four different initialisation schemes. We were not able to conclude on the best overall initialisation scheme. However, TYPEC and TYPED initialisation schemes, where the solutions are initialised to the result from the previous instance, are beneficial when the instances are similar enough.

We have also described a successful GA for LEMMINGS. This is the first successful approach for LEMMINGS. Thus, we have achieved thesis goal 2 as defined in section 1.3. Since we have successfully approached LEMMINGS, future investigations will have a basis from which to work. We have also described a set of parameters that seem to work best when a GA is used to approach LEMMINGS. This will also benefit future work. We now have a successful approach to LEMMINGS, and we can use that approach in our investigation in chapter 7 in order to find difficult instances.

One advantage to our approach is that we do not need to know very much detail about the LEMMINGS problem in order for us to implement it. Other approaches might require us to add local knowledge to the operators, whereas in our case we use random operators. The random mutation operator means that our algorithm can move from one side of the genotype solution space to the other side from a single application. Given the very small mutation rate, this is admittedly not very likey. It is a double edged sword. On the one hand our algorithm has global coverage and might sample solutions from a wider area of the solution space. On the other hand, it might move far away from a good solution as a result of the randomness.

A disadvantage to our approach is that the genotype solution space through which the GA searches is larger than the phenotype solution space of each LEMMINGS instance. That is, our GA is searching needlessly through solutions that all represent the same LEMMINGS outcome. One cause of this difference between genotype space and phenotype space is that any illegal action in the GA solution is interpreted by LEMMINGS as a pass action.

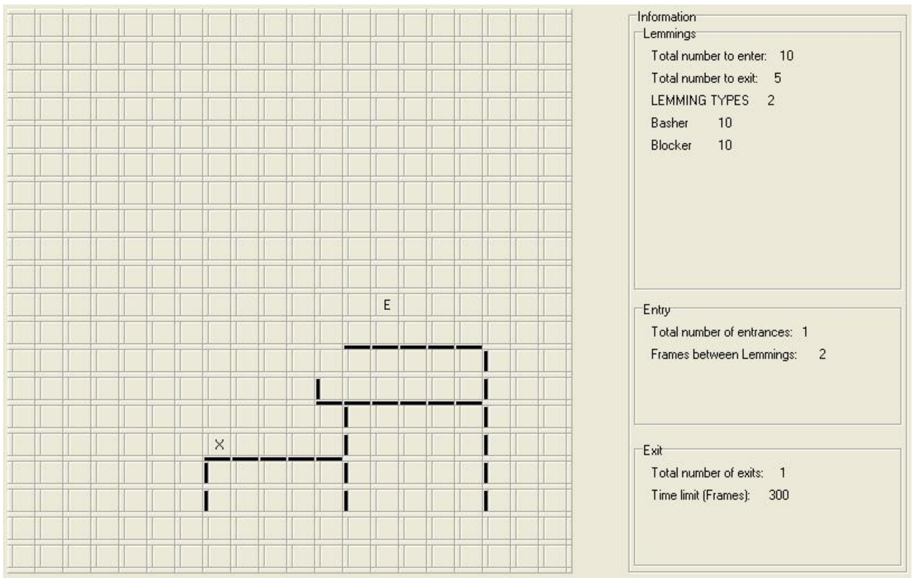


Information
Lemmings
Total number to enter: 10
Total number to exit: 5
LEMMING TYPES 1
Basher 10

Entry
Total number of entrances: 1
Frames between Lemmings: 2

Exit
Total number of exits: 1
Time limit (Frames): 300

Figure 6.3: α-LEMMINGS instance 1.

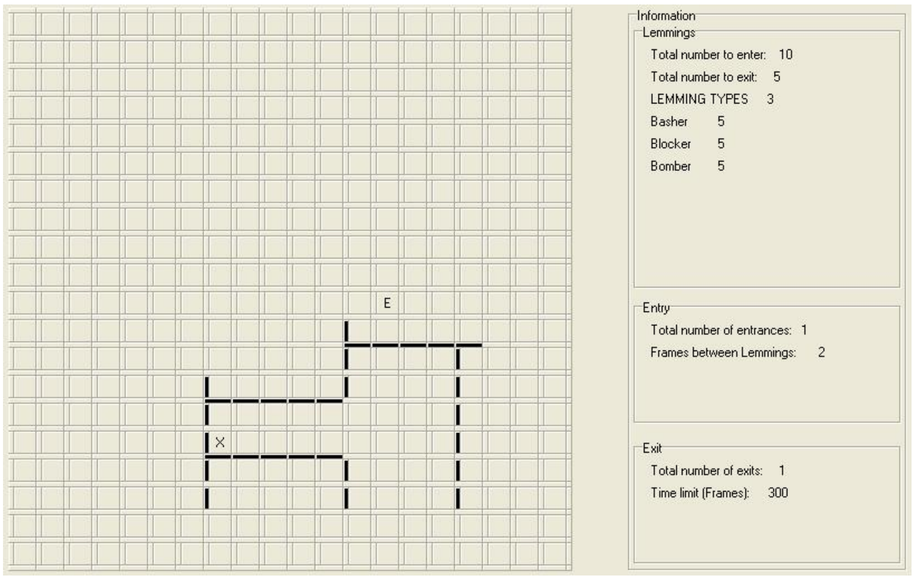


Information
Lemmings
Total number to enter: 10
Total number to exit: 5
LEMMING TYPES 2
Basher 10
Blocker 10

Entry
Total number of entrances: 1
Frames between Lemmings: 2

Exit
Total number of exits: 1
Time limit (Frames): 300

FIGURE 6.4: α-LEMMINGS instance 2.

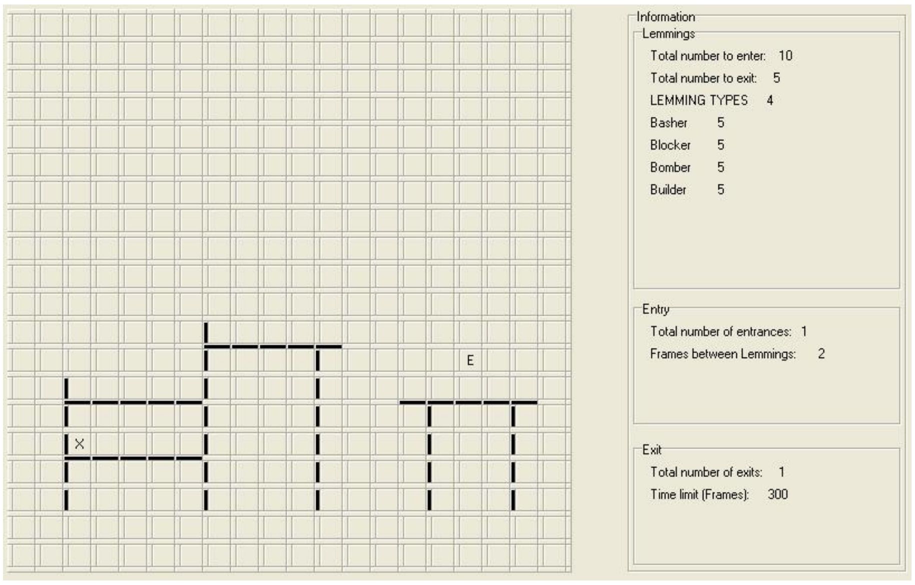


Information
Lemmings
Total number to enter: 10
Total number to exit: 5
LEMMING TYPES 3
Basher 5
Blocker 5
Bomber 5

Entry
Total number of entrances: 1
Frames between Lemmings: 2

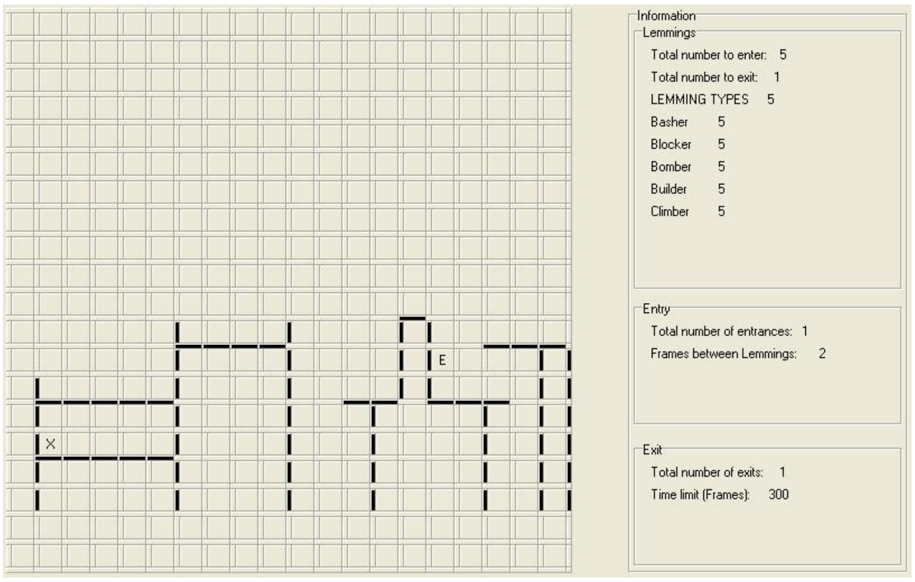
Exit
Total number of exits: 1
Time limit (Frames): 300

FIGURE 6.5: α-LEMMINGS instance 3.



Information
Lemmings
Total number to enter: 10
Total number to exit: 5
LEMMING TYPES 4
Basher 5
Blocker 5
Bomber 5
Builder 5
Entry
Total number of entrances: 1
Frames between Lemmings: 2
Exit
Total number of exits: 1
Time limit (Frames): 300

Figure 6.6: α-LEMMINGS instance 4.



Information
Lemmings
Total number to enter: 5
Total number to exit: 1
LEMMING TYPES 5
Basher 5
Blocker 5
Bomber 5
Builder 5
Climber 5
Entry
Total number of entrances: 1
Frames between Lemmings: 2
Exit
Total number of exits: 1
Time limit (Frames): 300

Figure 6.7: α-LEMMINGS instance 5.

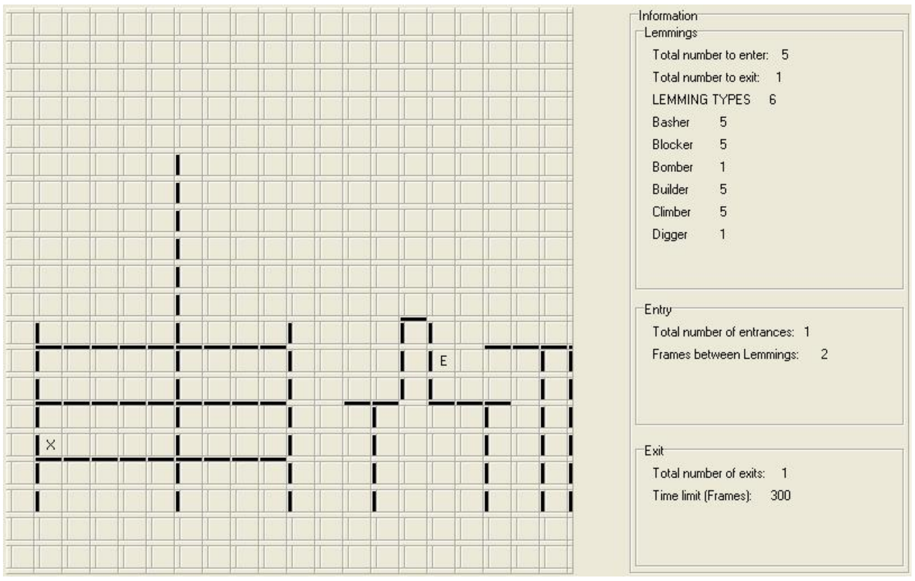

Information
Lemmings
Total number to enter: 5
Total number to exit: 1
LEMMING TYPES 6
Basher 5
Blocker 5
Bomber 1
Builder 5
Climber 5
Digger 1
Entry
Total number of entrances: 1
Frames between Lemmings: 2
Exit
Total number of exits: 1
Time limit (Frames): 300

Figure 6.8: α-LEMMINGS instance 6.

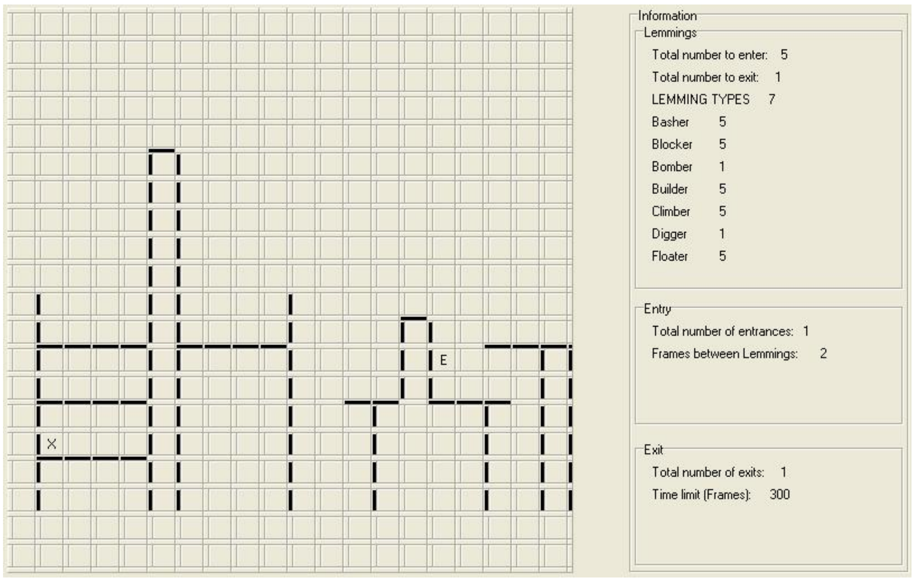

Information
Lemmings
Total number to enter: 5
Total number to exit: 1
LEMMING TYPES 7
Basher 5
Blocker 5
Bomber 1
Builder 5
Climber 5
Digger 1
Floater 5
Entry
Total number of entrances: 1
Frames between Lemmings: 2
Exit
Total number of exits: 1
Time limit (Frames): 300

Figure 6.9: α-LEMMINGS instance 7.

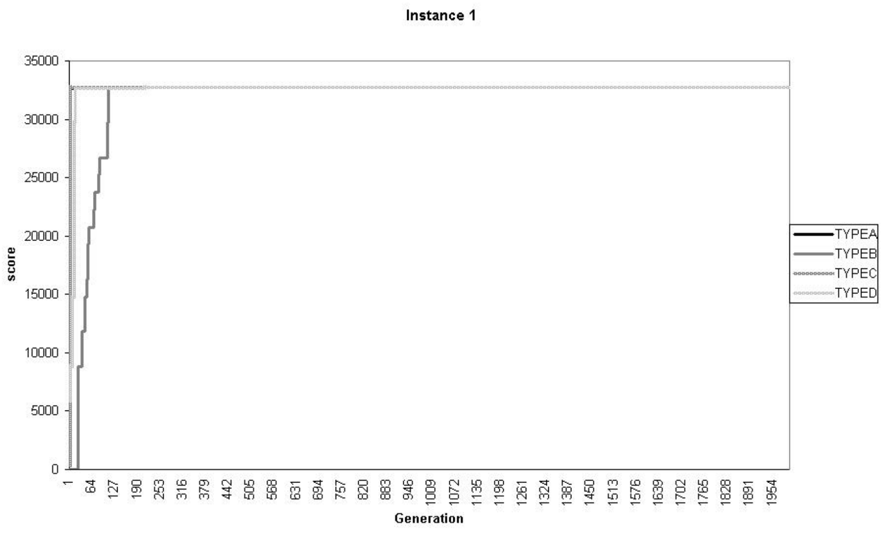


Figure 6.10: Graph showing score against generation for instance 1.

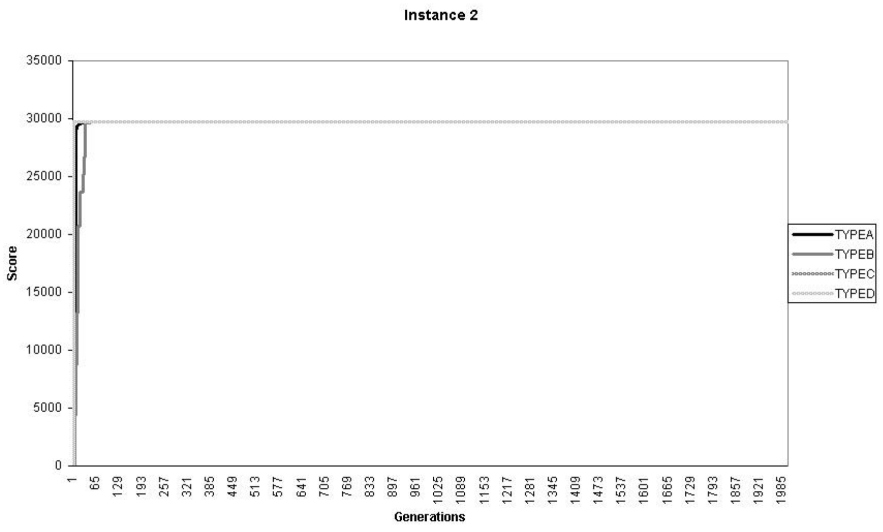


Figure 6.11: Graph showing score against generation for instance 2.

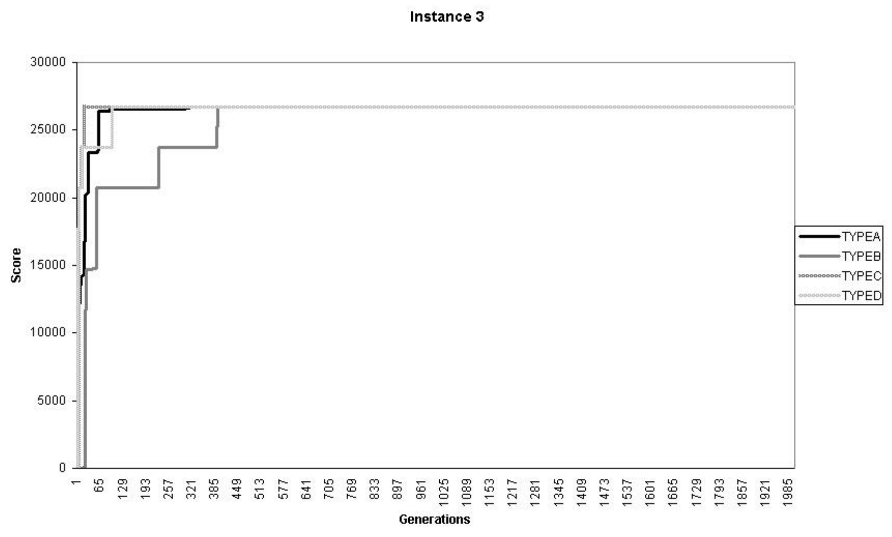


FIGURE 6.12: Graph showing score against generation for instance 3.

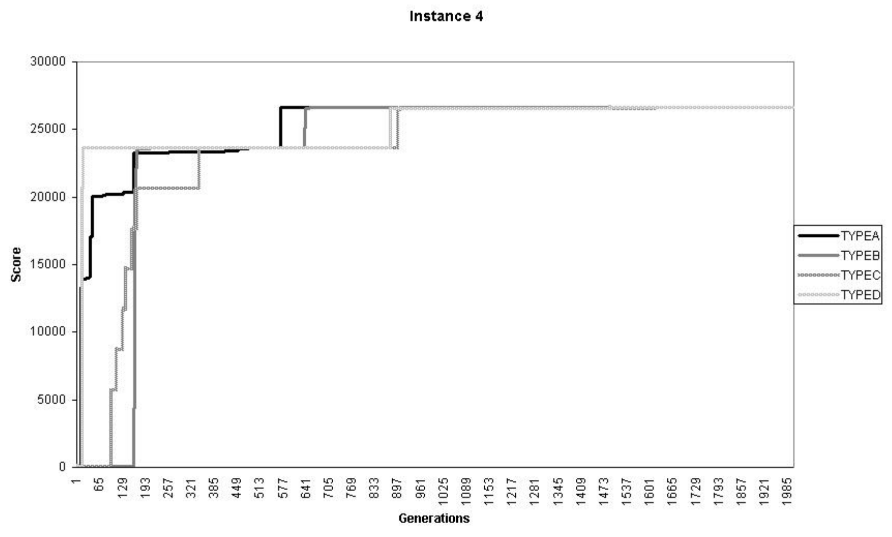


FIGURE 6.13: Graph showing score against generation for instance 4.

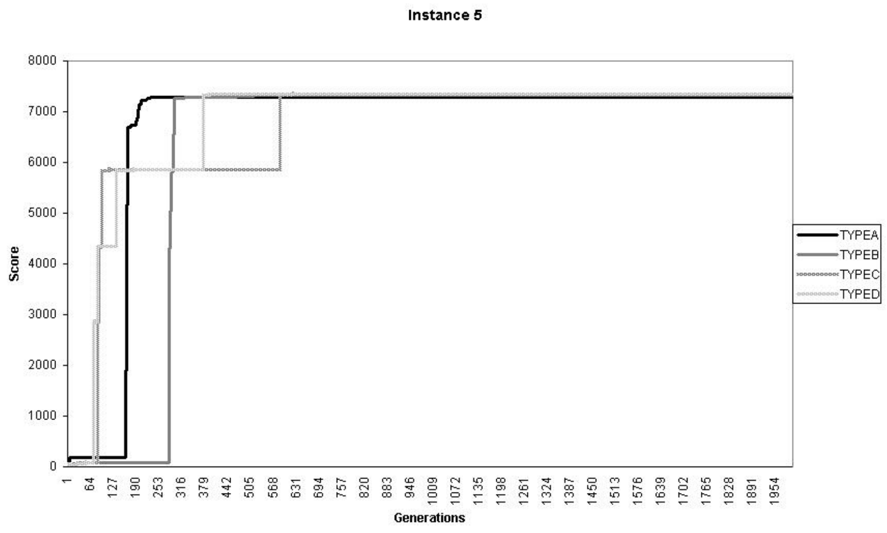


FIGURE 6.14: Graph showing score against generation for instance 5.

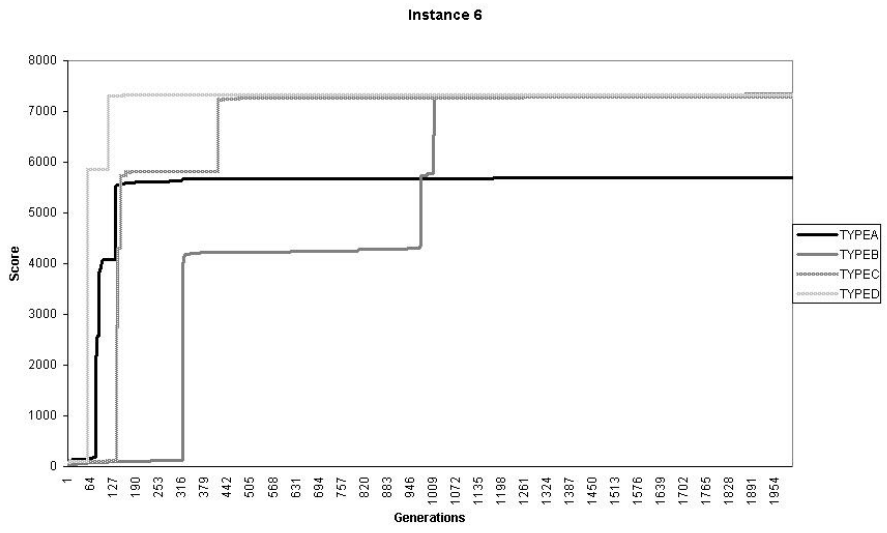


FIGURE 6.15: Graph showing score against generation for instance 6.

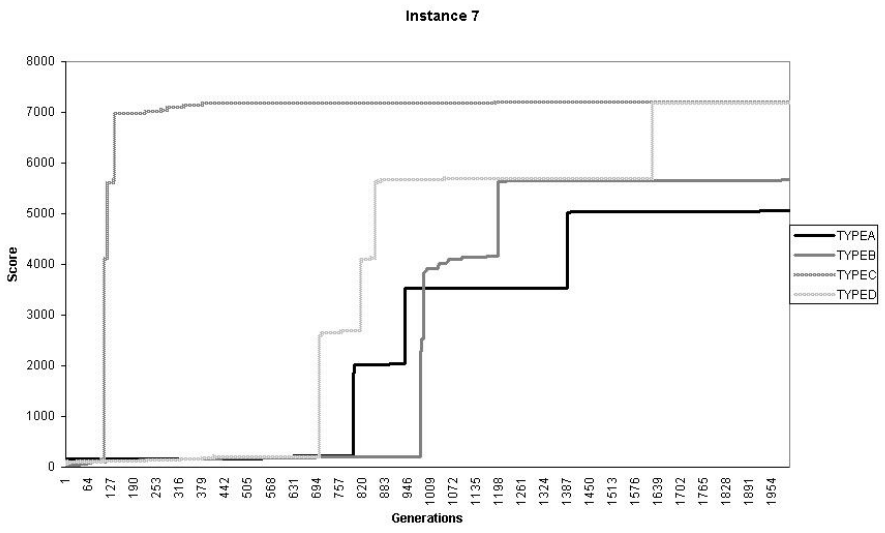


FIGURE 6.16: Graph showing score against generation for instance 7.

Chapter 7

Finding Difficult Instances

An investigation to find difficult LEMMINGS instances

"*A wise man thinks what is easy is difficult*"
John Churton Collins

In this chapter we aim to show where the more difficult LEMMINGS instances are. In chapter 6 we described a successful approach to LEMMINGS. In chapter 2 we described some work that aimed to find difficult instances of an NP-Complete problem. Also in chapter 4 we discussed some work on Blocks World that related to easy/hard instances. Therefore, in this chapter we intend to use our approach to LEMMINGS in order to find difficult instances.

Jones and Forrest [136] describe a problem difficulty measure, specifically for genetic algorithms, called Fitness Distance Correlation (FDC). The method is to "*examine a problem with known optima, take a sample of individuals and compute the correlation coefficient, r, given the set of (fitness, distance) pairs. If we are maximizing, we should hope that fitness increases as distance to a global maxima decreases.*

With an ideal fitness function, r will therefore be -1.0." That is, the genetic algorithm will find a problem difficult if fitness does not correlate with distance from the optimum.

Another motivation for this chapter is to analyse the performance of our GA on LEMMINGS. Mitchell et al [137] have analysed the performance of a GA on a toy landscape. They compared a form of Random Mutation Hill Climbing (RMHC) and a so-called Idealized Genetic Algorithm (IGA) under the expected time to achieve the optimum in One-Max. They modelled the problem as N blocks, each containing k 1s. They found that "*the IGA gives an expected time that is on the order of* $2^k logN$*, where RMHC gives an expected time that is on the order of* $2^k NlogN$".

We intend to show where the difficult instances of LEMMINGS lie within a subset of LEMMINGS instances, that we shall call the approached landscape. That is, from a set of instances that our approach solves, we intend to show which were the more difficult to solve. This might help future research on the LEMMINGS problem by providing an indication as to where the more difficult instances lie. For example, approaches might need to be tested on difficult instances, so we intend to provide a means of generating difficult instances. We are conducting the first investigations into difficult LEMMINGS instances, so our results surrounding the landscape of approached instances will form the early knowledge of LEMMINGS that we hope will be used as a basis for further investigation.

It is important to distinguish between solving LEMMINGS, that is solving every instance, which is beyond the reach of this work, and computing solutions to a subset of instances, such that this subset is large enough so that we can gain insight into the difficulty of the instances contained, and thus answer the research question relating to the location of difficult instances.

An ideal approach would be to fully enumerate a search past the point where we locate a feasible solution in order to build the entire search space for a LEMMINGS instance. Then compare the exact number of feasible solutions against the exact total number of solutions. This is one possible indication of how difficult it would be to solve that instance. That is, a more difficult instance would be characterised by a smaller ratio of feasible solutions against total solutions. Furthermore, if we were to do this for every LEMMINGS instance, then we would have a difficulty landscape of LEMMINGS. Unfortunately, there are problems with this ideal approach. Firstly, we cannot do this for every instance because there are too many. Secondly, this will not be an accurate and universal difficulty measure. We might have an instance for which there exists only one feasible solution among millions, suggesting a very difficult instance. But if we use a clever heuristic then an algorithm might locate that one feasible solution immediately. Thirdly, we have already shown in section 6.1 that a full enumeration will take too much time.

We can relax our ideal of knowing the exact number of solutions as a result of performing a full enumeration, and instead perform a random sampling. Then, instead of comparing the exact number of feasible solutions against the exact number of total solutions for a given instance, we take the amount of time required by the random sampling in order to locate a feasible solution as a difficulty measure. A more difficult instance should take longer to solve, when analysed over a number of runs. The GA that successfully approached LEMMINGS in section 6.3 can be used for this method of locating difficult instances. A GA locates the better solutions in the solution space. Therefore, an instance that is difficult for the GA, actually is an instance for which our GA finds it difficult to locate interesting areas of the solution space. This is still an indication of difficulty.

We should note that different algorithms will provide different results pertaining to the difficulty of solving LEMMINGS instances, when using this measure of difficulty. This does not matter because in this investigation we shall say that one instance is more difficult than another for our given algorithm. However, it is important to remember that difficult instances for one algorithm might actually be easy instances for another, and vice versa. For more information see No Free Lunch Theorem in chapter 3. So our measure of difficulty will probably be relevant to a subset of approaches, maybe even only our approach that gives us the results, not a general measure of difficulty over all approaches. This was also an issue with the work that Cheeseman et al. [8] did on finding difficult instances, that we detailed in chapter 2.

Set Name	Leniency Measure
nClimbers	∞
nFloaters	9
nBuilders	2
nBashers	2
nBombers	2
nBlockers	1
nDiggers	1
nMiners	1

Table 7.1: The design principle for the eight sets of scaled instances.

7.1 Method

In this investigation we shall measure the time our approach requires in order to locate a feasible solution to a linear scaling of LEMMINGS instances. The purpose is to answer the following

- How successfully will our approach tackle instances of LEMMINGS based on how well it handles a controlled landscape of LEMMINGS instances?
- Where are some difficult LEMMINGS instances?

We apply our approach to a series of instances that linearly increase in size and analyse the amount of time it takes to find a feasible solution to each instance. We then find a best-fit curve that matches the growth in time against the growth in instance size, in order to analyse how the time to solution grows as instance size grows linearly.

There are eight sets of instances, such that each set contains instances that require the use of one of the eight LEMMINGS types. See Table 7.1. The leniency measure is in terms of the number of time-steps in which each non-pass action can be played in a feasible solution. This can be explained as a form of restriction under which all feasible solutions exist.

For example, Figure A.51 shows instance 01Diggers, the environment of which one lemming will enter and one lemming must exit where there is only a single digger type available. The distance from the platform (B, 10) underneath the entrance to the next platform (C, 15) is such that the lemming will die if it falls. The only way for the lemming to survive is if it is given a digger type at the single time step at which it stands on the single platform (B, 10) underneath the entrance. The lemming will then exit the environment and the instance is solved. Refering to Figures A.51 to A.60 showing instances 01Diggers to 10Diggers, note that with each successive instance, the number of lemmings that enter and must reach an exit, the number of digger types, and the number of platforms through which to dig, all increase by one. This forms the linear growth of nDiggers instances. Importantly, each instance can only be solved if each of the non-pass actions, in this case setting a lemming to a digger type, is played at a specific time step.

In contrast, Figure A.41 shows instance 01Climbers, the environment of which one lemming will enter and one lemming must exit where there is only a single climber type available. A lemming will not die in this instance but the time limit can be exceeded. The instance can be solved by setting the lemming to a climber type at any time step, with the single proviso that there are enough time steps for the lemming to climb the wall to the right of the entrance in order to reach the exit before the time limit. Once again refering to Figures A.41 to A.50 showing instances 01Climbers to 10Climbers, with each successive instance, the number of lemmings that enter and must reach an exit and the number of climber types all increase by one. This forms the linear growth of nClimbers instances. The difference between this set and the nDiggers set is that each non-pass action, in this case setting a lemming to a climber type, can be placed at one of many time-steps in order to solve the instance. ∞ is used to denote uncountable.

The eight sets of instances all have a similar form of linear growth and Table 7.1 shows the number of time-steps in which each non-pass action can be played. We chose some specific instance variables and altered them in a certain way, as described above, in order to achieve this linear scaling. Note that there might be many other instance variables that can be altered in many different ways in order to achieve different types of linear scaling. There are many instance variables describing a LEMMINGS instance, so encapsulating a linear growth in terms of variables is a difficult task.

Our hypothesis is that a tighter restriction under which non-pass moves must be made yields a more difficult instance, whereas a looser one results in an easier instance. That is, we expect that our approach will take longer to locate a feasible solution to each of the nDiggers instances than it takes to locate a feasible solution to the corresponding nClimbers instances. Moreover, the growth of time to solution against instance size will be steeper for nDiggers than for nClimbers. In general, easier instances will be characterised by more space in which to place non-pass actions, whereas the more difficult instances will be those for which a feasbile solution requires each non-pass action to be located in one of only a few time steps. Note that currently we have very little knowledge of LEMMINGS, thus we can only speculate about the difficulty of a LEMMINGS instance, although intuitively we would expect our hypothesis to hold. Our motivation for this investigation is to gain some insight into the difficulty of LEMMINGS instances. If our hypothesis is shown to be true then we have gained the insight that we desire. These scaling experiments might teach us something about the difficulty of LEMMINGS that can be used as a basis for later investigations.

The set of nMiners instances are identical to the set of nDiggers instances. The only difference between each corresponding instance is that between a digger action and a miner action. Thus, from our hypothesis above, we would expect to attain very similar growth curves for nMiners and nDiggers. This is a control test that we can use to help verify that our results are correct. In future we might investigate LEMMINGS more deeply, at which time we can alter more of the instance variables in order to gain more insight into the difficulty of LEMMINGS instances.

Each set contains ten instances that grow linearly in size, such that $n = \{1, \ldots, 10\}$. All 80 instances can be found in Appendix A. We first verify that each instance has at least one feasible solution. We then use our GA with the default parameters, as described in section 6.3, to locate any feasible solution to an instance, and count the number of generations required. We limit the GA to a maximum of 80,000 generations per run, which means that the GA will sample a maximum of 4,000,000 solutions per run. We repeat this 25 times and take an average of the number of generations required by the GA to find a feasible solution to an instance. In this way we get the average number of generations required by the GA to find a feasible solution to each of the 80 instances.

In this section we have described our method. In the next section we shall describe the results of the investigation.

7.2 Results

Figures 7.1 to 7.9 each show a graph that plots the growth of average number of generations to locate a feasible solution to each instance against a linear growth of instance size for each of the eight sets. We want to analyse the growth of the time to solution as the instance grows linearly, so each graph contains a best fit curve for a quadratic and exponential.

The first thing to note is that none of the sets of instances display a linear growth in generations to solution against a linear growth in instance size. nBashers, nBombers, nBuilders, nClimbers and nFloaters best fit a quadratic growth. nDiggers and nMiners best fit an exponential growth. nBlockers at first displayed very eratic growth, that emerged to fit exponential when the data was split into odd and even for n. We do not know why this was so and we believe that this is worthy of further investigation, but it does not effect our results for this investigation. We have shown that there are LEMMINGS instances that when increased in a certain linear way, as described above, will require a number of generations to find a feasible solution that grows exponentially. This means that our approach does not scale well for this particular type of LEMMINGS instance growth. Later in this section, we shall attempt to pinpoint this type of instance.

nBashers and nBombers have similar growth that is not steep, as expected. nBuilders is slightly less steep than nBashers and nBombers. We expected these three to have similar growths, and this slight difference might just be down to the stochastic nature of our GA.

nDiggers and nMiners are very similar, as expected. This suggests that our results can be trusted. Both graphs contain only four data points. The data for 05Diggers and 05Miners was not included because not all of the 25 runs located a feasible

Perceived Difficulty	Set Name	Best Fit Growth	Leniency
Least Difficult	nClimbers	Quadratic	∞
	nFloaters	Quadratic	9
	nBuilders	Quadratic	2
	nBashers	Quadratic	2
	nBombers	Quadratic	2
	nBlockers	Exponential	1
	nMiners	Exponential	1
Most Difficult	nDiggers	Exponential	1

Table 7.2: Sets of instances from least to most difficult.

solution to these two instances within the generations allowed. None of the larger instances for these two sets were investigated because it was thought that even fewer of the 25 runs would locate a feasible solution. Since the GA could locate a feasible solution within the time available for all 25 runs but for only four of the nDiggers and nMiners instances suggests that our approach is limited. On the other hand, it might be the case that any reasonable approach would find these instances too difficult. Importantly, we must be careful to remember that our approach can find a feasible solution to only a limited set of instances within the available time.

nClimbers had the least steep growth and required the fewest number of generations to locate a feasible solution for the largest instance. This suggests that is the least difficult of the instances, which agrees with our hypothesis. nFloaters has slightly steeper growth than nClimbers, which also agrees with our hypothesis.

nDiggers and nMiners have very similar growths, which agrees with what we expected. The only noticable difference is in the result for 04Diggers in comparison to 04Miners, which is probably due to the stochastic nature of our GA.

Table 7.2 shows the eight sets of instances sorted from easy to difficult, according to the best fit growth function, and then according to the maximum number of generations required to locate a feasible solution. From this table we can see that those instances that require a non-pass action to be placed at exactly one time step in the solution are the more difficult ones. Moreover, if we linearly increase the size of these instances in the way described earlier then the number of generations required to locate a feasible solution increases exponentially. Thus, it seems that the type of instance for which our approach does not scale well is one that requires a non-pass action to be placed in exactly one time step and where the number of these actions increases.

An interesting finding from this table is that nBlockers, nMiners and nDiggers should have similar difficulty. However, there are only four data points for nMiners and nDiggers because the GA could not find a feasible solution for all 25 runs within the time limit for 05Miners and 05Diggers. On the other hand, the GA located a feasible solution 25 times for all 10 of the nBlockers instances. This suggests that nBlockers is easier than nMiners and nDiggers for our GA. We think that this might be caused by the number of time-steps in which a non-pass action can be placed without consequence. That is, we expect that the solution for each nBlockers instance contains more of these time-steps, and thus offers more room for error. A possible future investigation would aim to verify this.

Our hypothesis, that a tighter restriction under which non-pass moves must be made yields a more difficult instance, whereas a looser one results in an easier instance, has been verified. We now know where some of the more difficult instances lie. This gives us a good basis upon which to start further investigations.

There are some questions that arise from this investigation. Why does nBlockers have eratic growth until we split it into odd and even? Why does it appear that nBlockers is easier for our approach than nMiners and nDiggers? Do other approaches to locating difficult instances, for example other linear scalings, yield similar difficult instances to the ones that we have found?

In this section we have described the results of our investigation. We have shown where some difficult instances lie. In the next section we shall conclude the chapter.

7.3 Conclusions

In this chapter we have detailed an investigation that aimed to find some difficult LEMMINGS instances. Refer to Figure 2.1, which shows a condensed tree representing the set of all problems. In chapter 4 we detailed a subset of this containing some NP-Complete puzzles. A further subset (probably) contains the LEMMINGS instances. The landscape of approached instances (see Appendix A) is a subset of LEMMINGS. From this set of approached instances, we have described eight partitions of instances (see Table 7.2) such that the instances contained in each seem to linearly increase in size. We have showed that for some of these partitions of LEMMINGS instances, the time taken to locate a feasible solution grows exponentially. Importantly, these partitions of instances that have exponential growth are the more difficult LEMMINGS instances of those that we have approached. Therefore, we have found some difficult instances. We have attempted to describe these partitions of difficult instances as those for which feasible solutions contain non-pass actions that must be played at precisely one time-step and the number of these actions increases. We have shown that as the number of these actions increases linearly the time taken to locate a feasible solution grows exponentially for these difficult instance partitions. Future investigations might use this description of difficult LEMMINGS instances. That is, difficult instances can be generated by designing one instance such that a non pass action must be placed at exactly one time step in order to solve the instance, then when the number of these actions is increased the instance will become very difficult very quickly due to the exponential growth of the time required to locate a feasible solution. We have provided the first indication of the location of difficult LEMMINGS instances, and we have provided a description of how to generate those difficult instances. Therefore, we have achieved thesis goal 3 as defined in section 1.3.

We have showed that the GA is limited and does not scale well. Importantly, this evolutionary approach to LEMMINGS has exponential time complexity. Thus, LEMMINGS cannot be shown to be in P. This agrees with Cormode's NP-Completeness proof for Lemmings [22]. The approach also has linear space complexity. Since we have shown the complexity of our approach, then we have achieved thesis goal 4 as defined in section 1.3.

Disadvantages to the approach are that it does not scale well and is limited over the number of instances that it can solve. However, we argue that we have solved enough instances to gain some insight into a difficulty landscape of LEMMINGS. Thus, we have not solved LEMMINGS because our approach did not solve some of the instances within the time that we allowed. This means that there are still unapproached areas of LEMMINGS. Consequently, our difficult instances are relative to a subset of LEMMINGS, and so there might be other descriptions of generating even more difficult unapproached instances that we do not know about. Also, our measure of difficulty might not be an accurate one because we use an informed method as a means of locating the feasible solutions, and thereby defining our difficult instances. A difficult instance is only difficult for the given informed approach. Our evaluation function might be so incorrect that the difficult instances are not actually difficult at all.

We realise that one difficult instance for our approach might actually be a relatively simple instance for a different approach. Equally, our approach might have no trouble in locating a feasible solution to an instance, and this same instance might actually pose real difficulty for some other approach. Moreover, many of the decisions we have made in terms of the representation of our LEMMINGS solution might have been wrong decisions, and there might be better alternatives out there. To which we suggest that future research might attempt to capture a different approach to LEMMINGS and compare the difficult instances for that approach with the difficult instances that we have recognised. It might be that our difficult instances are also difficult for some other approaches.

In chapter 2 we saw how Cheeseman et al. [8] described a phase transition, around which hard instances of NP-Complete problems are located. In this chapter we have described a different approach for locating difficult instances of LEMMINGS. The reason that our approach is different is because our local search algorithm uses random operators and so it is not affected by the phase transition. If our algorithm had used local operators then we could have tried to locate a phase transition for LEMMINGS. Both their approach and our approach that is described here are successful in locating difficult instances. The essence of each approach is the same. That is, take a problem whose instances are defined by some number of variables, disregard those variables in favor of a single parameter, and search for particular values of that parameter so that it describes computationally difficult instances. The idea is that with fewer variables it is easier to locate difficult instances.

Recall also that Cheeseman et al. [8] said that the instances on the phase transition are more difficult than those hard instances away from the phase transition. Therefore, it might be the case that if we apply an algorithm with local operators to LEMMINGS and locate a phase transition, then we might find even more difficult instances than we have located in this chapter. Finally, Cheeseman et al. showed that in some cases the phase transition remains unchanged when problems are mapped. This suggests that if we located difficult instances of LEMMINGS surrounding a phase transition, then they might map to difficult instances of other problems. This is a further avenue that might be explored in order to locate difficult LEMMINGS instances in the future. That is, if we have a problem for which there exists a phase transition locating difficult instances, and we map that problem to LEMMINGS, then those difficult instances might map to difficult LEMMINGS instances.

Our investigation in this chapter is the first work of its kind on the LEMMINGS problem. We have described some empirical results and have a basis for future investigations. Therefore, any limitations described can be addressed in future work. We have highlighted some open questions that arise from the investigation in this chapter, which can also be addressed in future work.

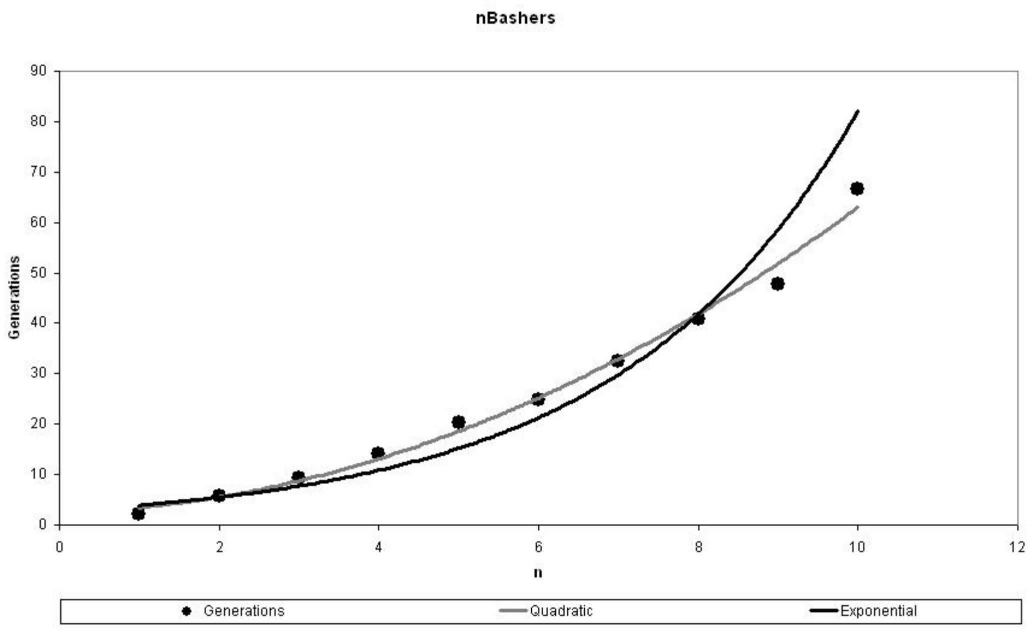


FIGURE 7.1: Generations to solution against instance size for nBashers.

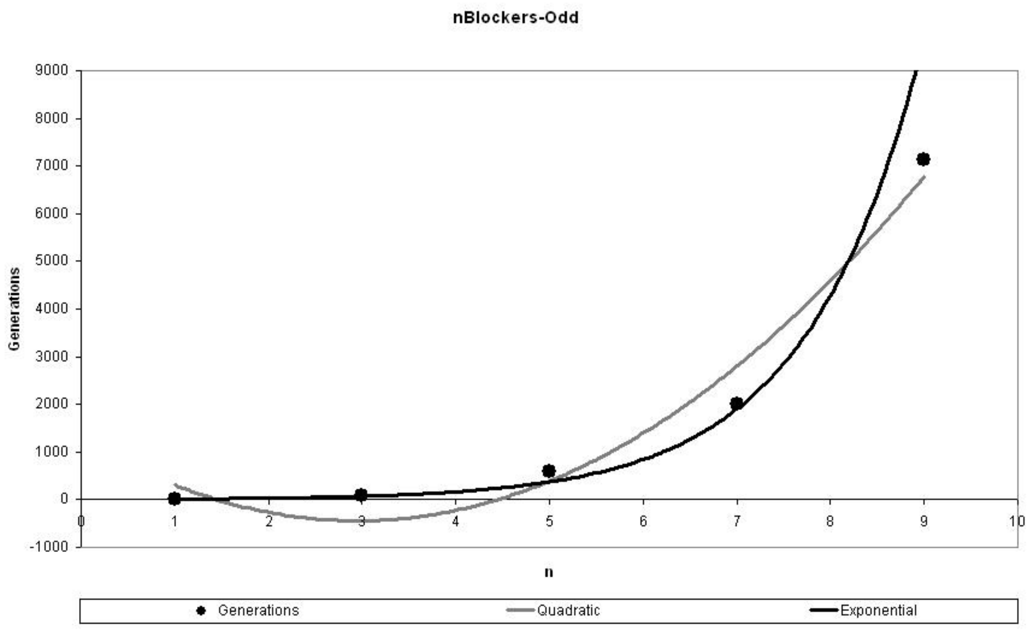


FIGURE 7.2: Generations to solution against instance size for nBlockers (odd).

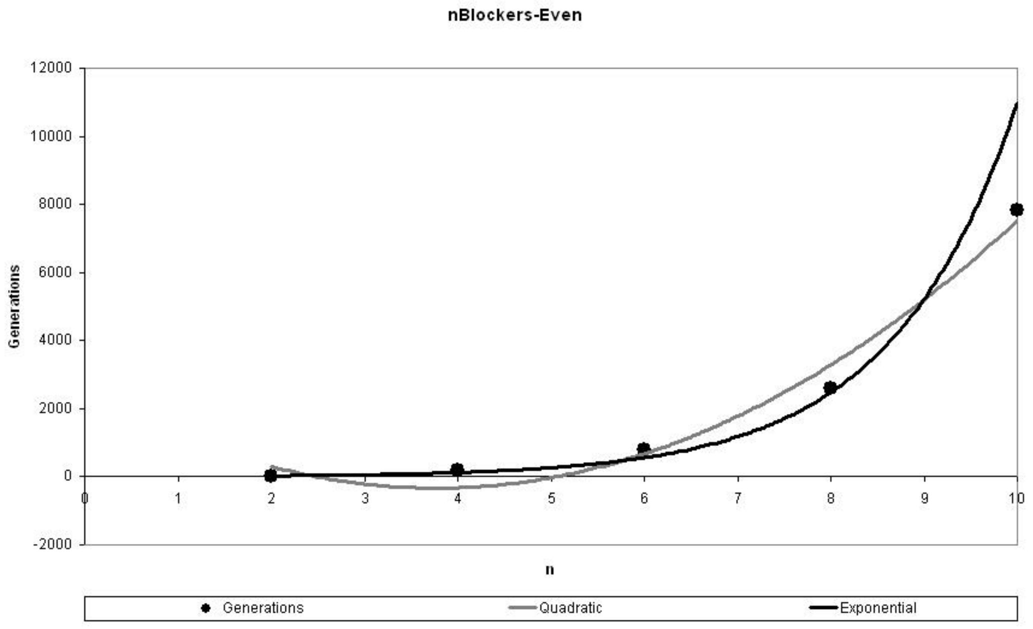


FIGURE 7.3: Generations to solution against instance size for nBlockers (even).

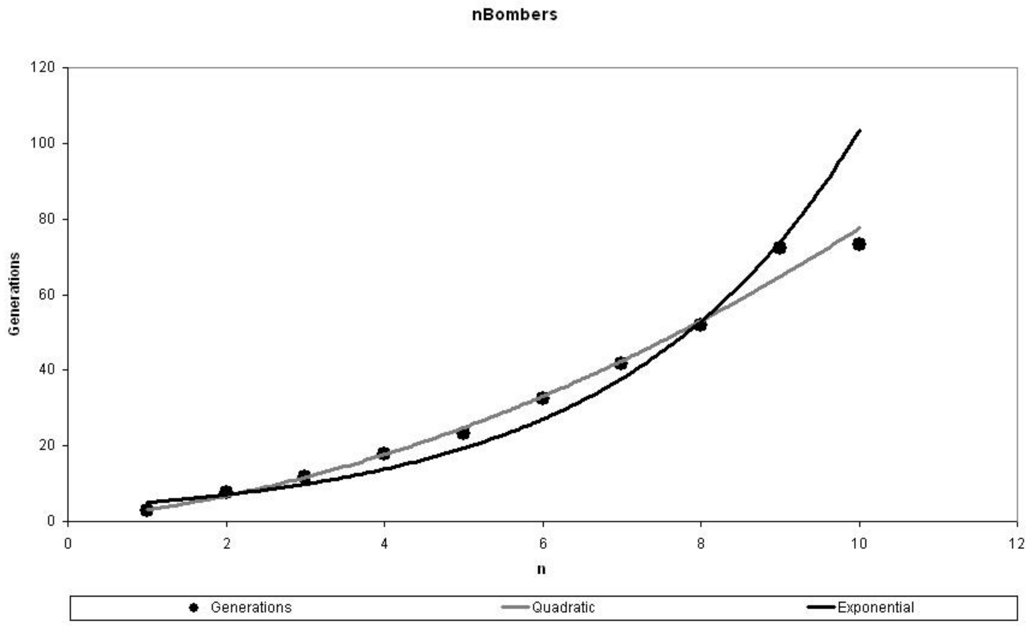


FIGURE 7.4: Generations to solution against instance size for nBombers.

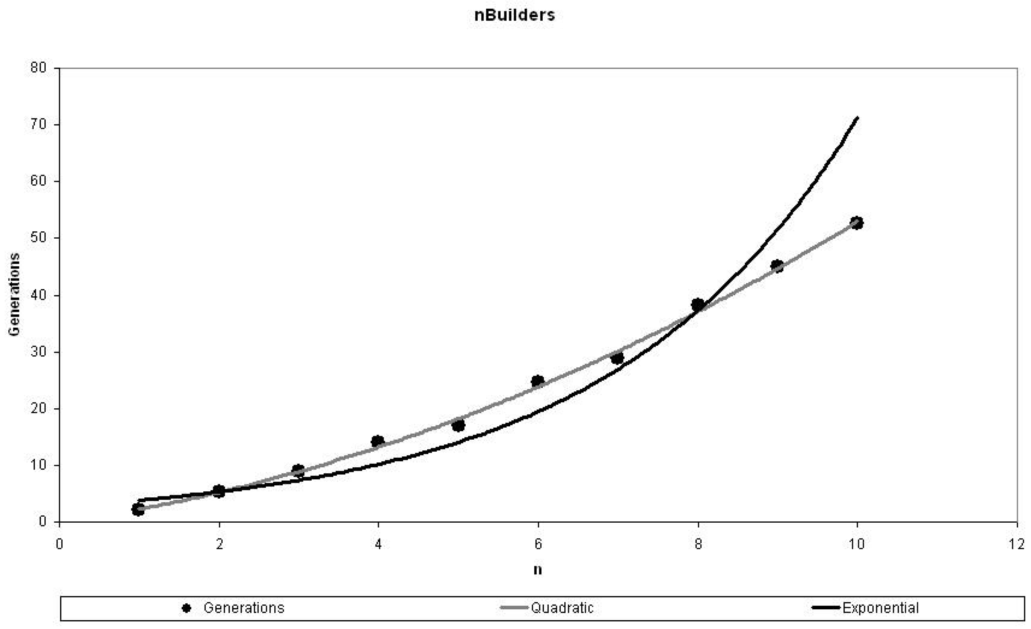


Figure 7.5: Generations to solution against instance size for nBuilders.

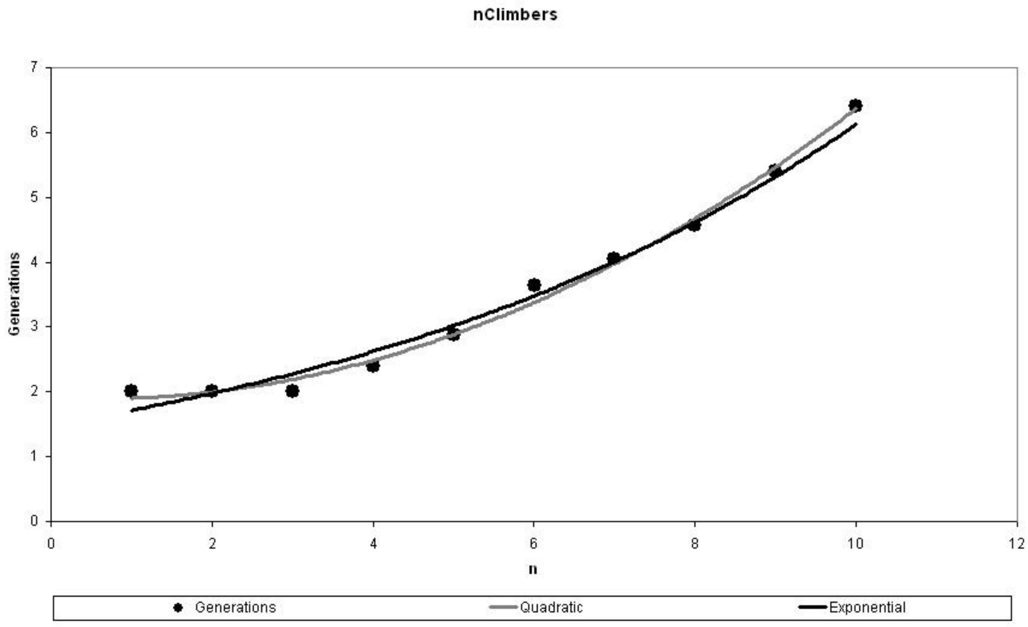


Figure 7.6: Generations to solution against instance size for nClimbers.

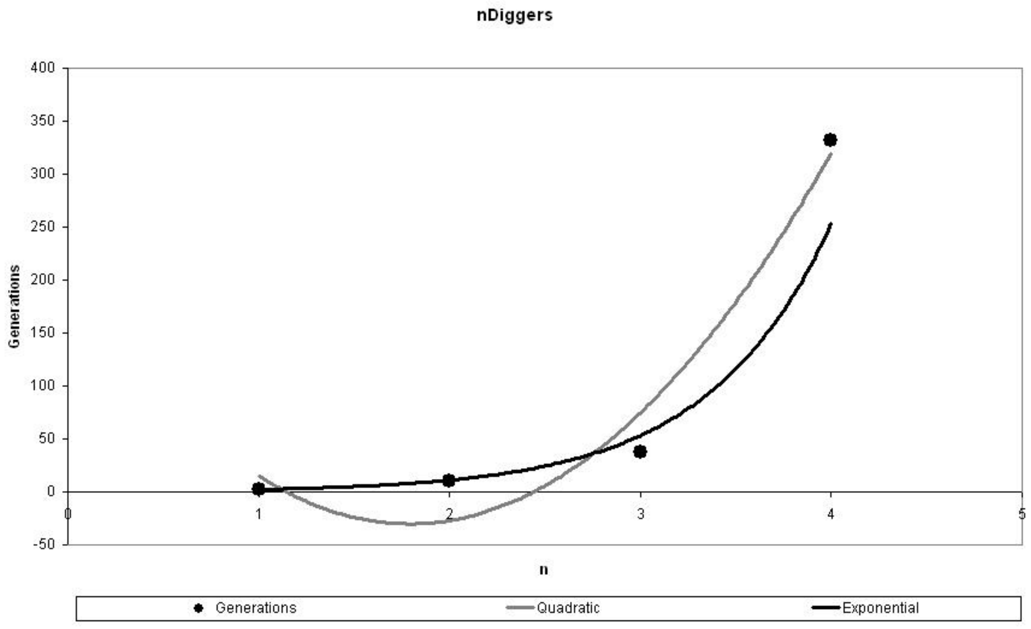


FIGURE 7.7: Generations to solution against instance size for nDiggers.

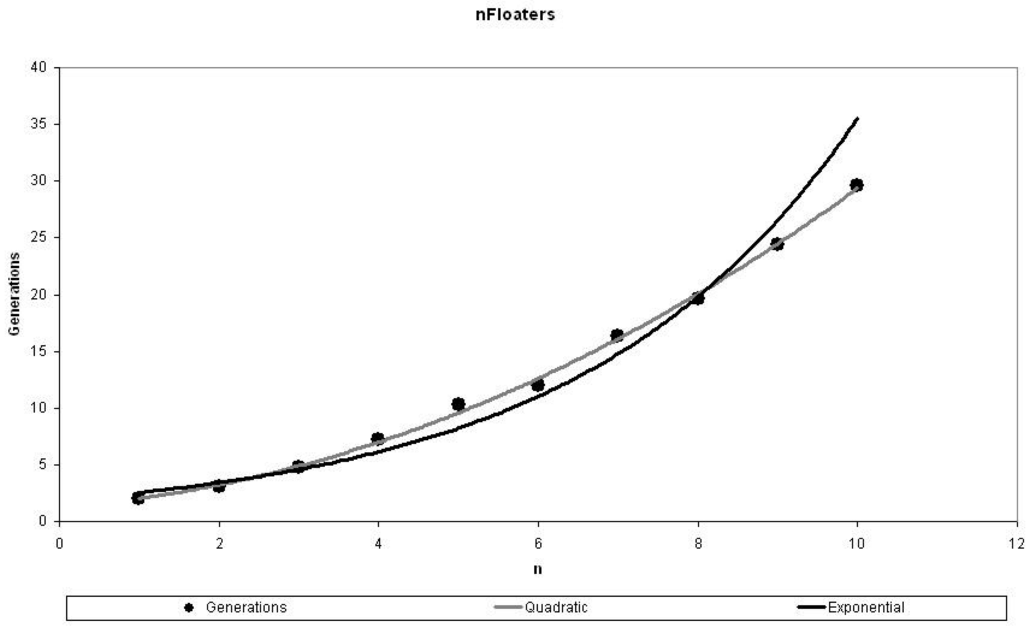


FIGURE 7.8: Generations to solution against instance size for nFloaters.

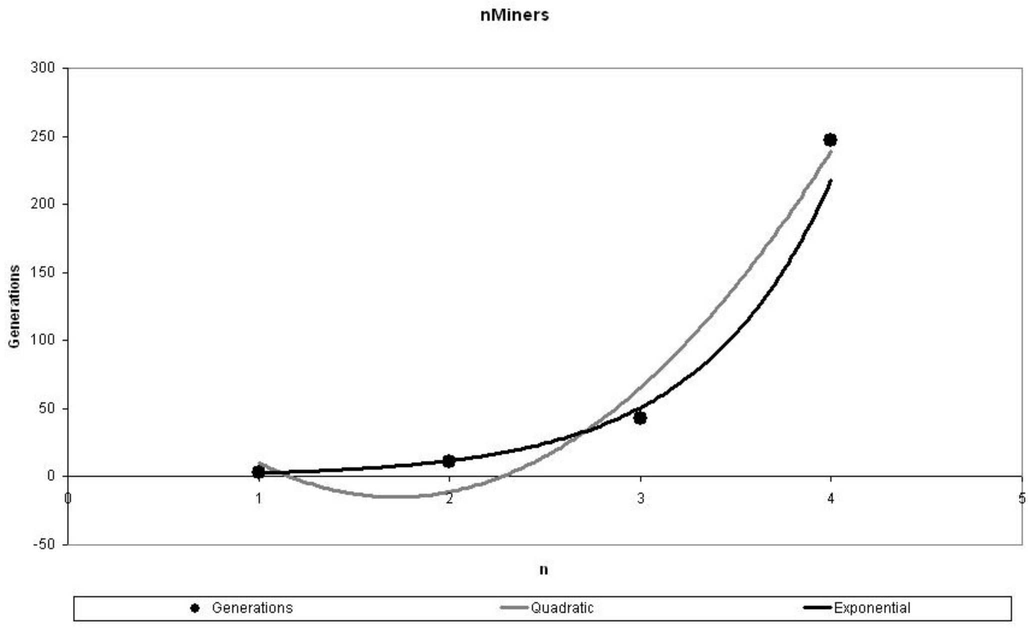


FIGURE 7.9: Generations to solution against instance size for nMiners.

Chapter 8

Conclusions

Concluding remarks of the thesis

"*The distinction between past, present, and future is only a stubbornly persistent illusion*"
Albert Einstein

In this thesis we have described research on a puzzle called LEMMINGS, whereby we have applied a search algorithm in order to test the difficulty of instances.

In chapter 2 we provided a theoretical basis for the thesis. We defined the basic concepts of problem, solution, and algorithm, provided an outline of algorithm and problem analysis, described a background on decision problems and the theory of NP-Completeness, described the P = NP problem, discussed various approaches to tackling a difficult NP-Complete problem, and we discussed some work focussing on the location of difficult instances.

In chapter 3 we described a general constructive search method, detailed some uninformed constructive search algorithms and defined their complexities using algorithm analysis. We also described informed constructive search and detailed a number of algorithms, and described informed local search and detailed some algorithms. We discussed the concepts of state space and search space, outlined issues concerning heuristics, and briefly discussed the No Free Lunch Theorem.

In chapter 4 we provided a survey of NP-Hard puzzles, and expanded Demaine's and Eppstein's surveys by adding extra puzzles. For each puzzle, we provided a brief description of the rules, an indication of its inclusion in various complexity classes, an outline of some of the work towards it, and also some additional sources of information. We identified some puzzles that constitute a research gap.

In chapter 5 we defined a relatively new problem called LEMMINGS that is derived from a popular computer puzzle game called Lemmings. We defined action, instance, solution, feasible solution and outcome with respect to LEMMINGS. We also described some related issues, including a history of Lemmings, and we suggested that LEMMINGS is in NP-Complete and therefore is an interesting and challenging problem for AI.

In chapter 6 we discussed various possible approaches to the LEMMINGS problem, focusing on search. We showed that it is theoretically infeasible to approach LEMMINGS using uninformed search. We described a GA that will successfully approach α-LEMMINGS, and analysed four different initialisation schemes. We were not able to conclude on the best overall initialisation scheme. However, TYPEC and TYPED initialisation schemes, where the solutions are initialised to the result from the previous instance, are beneficial when the instances are similar enough. We also described a successful GA for LEMMINGS, which is the first successful approach for LEMMINGS, and described a set of parameters that seem to work best when a GA is used to approach LEMMINGS.

In chapter 7 we described results of an investigation that aimed to find some difficult LEMMINGS instances. We described partitions of difficult instances as those for which feasible solutions contain non-pass actions that must be played at precisely one time-step and the number of these actions increases. We showed that as the number of these actions increases linearly the time taken to locate a feasible solution grows exponentially for these difficult instance partitions. We showed that the GA for LEMMINGS is limited and does not scale well, has exponential time complexity, and therefore, LEMMINGS cannot be shown to be in P.

8.1 Contributions of the Thesis

Refer to section 1.3 that defines the goals of our thesis. We have achieved all of our goals in this thesis. Therefore, we can identify the following contributions to research

1. In chapter 4 we motivated work on NP-Hard puzzles by highlighting research gaps in the literature. In particular, we motivate work on the game of Lemmings, by classifying it as a self updated puzzle.
2. In chapter 6 we documented the first results for producing feasible solutions to instances of LEMMINGS.
3. In chapter 7 we provided the first evidence on the location of the more difficult LEMMINGS instances. Moreover, we described a method for generating these difficult instances.
4. In chapter 7 we also evaluated the amount of time required by our algorithm for producing feasible solutions to instances of LEMMINGS.

8.2 Discussion of the Contributions

In this section we shall discuss each of the contributions of our thesis.

Motivating Research on NP-Hard Puzzles

It is our hope that research on NP-Hard puzzles will be motivated by our survey in chapter 4. We believe that investigating puzzles will provide interesting insight into areas of AI, for example planning and optimal planning algorithms. The research that has already been performed on some of our surveyed puzzles has yielded some interesting findings. For example, the work by Slaney and Thiebaux [77] on Blocks

World resulted in an appreciation of the structure of hard and easy instances, the work by Culberson and Schaeffer [95] on n-Puzzle resulted in the effective use of pattern databases, and the work by Junghanns and Schaeffer [110, 111] on Sokoban resulted in the relevance cuts enhancement. Research towards these puzzles, and others similar to them, might yield even more interesting results.

We analysed the amount of research that each puzzle had received, and our findings are summerised in Table 4.1. Blocks World, n-Puzzle and Sokoban have received the most research effort. On the other hand, we identified thirteen puzzles that constitute a research gap. This is over half of the surveyed puzzles. It is our hope that, by highlighting these puzzles on the 'research gap', researchers might be motivated to investigate them in particular.

An interesting area for future research might be to perform an in depth analysis of the similarities between the NP-Hard puzzles, or maybe just those contained in NP-Complete. This might be performed by transforming between pairs of these problems. The hidden constraints that emerge only once a solution is attempted are likely to be what makes these problems difficult. It would be interesting to see if there are similarities between the constraints for each puzzle.

These puzzles seem to be similar in that the decision maker is concerned with emptying a set of objects, for example clearing a grid of coloured squares, or being presented with an empty set and attempting to include all of the objects in that set, for example, drawing a line all the way around a set of numbers. There might be a deeper insight to be gained by analysing the various types of goals for each of these puzzles. The reason that some of these puzzles are popular might be because of the satisfaction that is felt by the player as they get closer towards emptying the playing area of clutter.

Possible future research on these puzzles might be concerned with locating difficult instances of various puzzles. If the puzzles can be transformed then maybe the difficult instances of these puzzles will be preserved under the transformation. It might be the case that all of these puzzles are so similar that they can all be transformed to each other, and further that the difficult instances of each are all preserved under these transformations. If this were the case then it might even be possible to develop a single solver that can be applied to all of these puzzles. This would be a sort of universal puzzle solver.

Another interesting direction for future work might be to design a simple p-time puzzle, and then iteratively add more complexity until it can be shown to be in NP-Complete. Through this process we should be able to gain insight into what makes a problem NP-Complete. This might offer an indication as to the similarities between our surveyed NP-Complete puzzles.

The reason why some of these puzzles have received lots of research attention and others have received very little might be because the latter are not in the most easily accessible places. One motivation for chapter 4 is to bring all of the puzzles together in one place so that the reader knows where to look for them, if they wish to research them.

We are particularly interested in self updated puzzles in this thesis. In these puzzles, the state of the problem will update even if the player decides to pass, and these puzzles can solve themselves or become failed, even if the player performs no actual actions. Therefore, pass is modelled as a choice, and the search space is larger and different in nature to player updated puzzles. It might be the case that a different type of search algorithm is required for these self updated puzzles. What is interesting about these puzzles is that they are closer to real life, so insight gained into them might be applicable to real world problems.

Any researcher who is interested in devoting some time to a puzzle might find Table 4.1 helpful. Those problems that have received the most research, up until now, will have a strong starting point from which further research might proceed. On the other hand, for those problems constituting a gap, the only information we have from which to start future work is the problem definition and the inclusion in NP-Hard or NP-Complete.

In chapter 4 we also motivated our research on the Lemmings game. We feel that it offers something different because it is a self updated puzzle, and furthermore we feel that its rules are substantially more complex than the other puzzles, so much so that it is closer to real life problems, whilst retaining the tight constraints and clearly defined goal of the other puzzles. We advocate the use of Lemmings as an AI research model.

Producing Feasible LEMMINGS Solutions

In chapter 6 we produced the first feasible LEMMINGS solutions using a computer. In the process we tackled a sub-problem called α-LEMMINGS. We offered a genetic algorithm that optimised α-LEMMINGS solutions. Thus, future work where we are required to optimise α-LEMMINGS, or even LEMMINGS, solutions has a basis from which to work. There are other problems that might benefit from the approach that we described in section 6.2. For example, there is a real world problem called Automatic Assembly Sequencing, which involves assembling a set of objects to form a sub-assembly, and then inserting that sub-assembly into a larger set of objects, and so on. If the wrong order is chosen at some point, this might not be evident until later on in the assembly, at which point, all actions must be reversed in order correct the mistake and put together the entire assembly. See [1] p68.

We also analysed four initialisation schemes for this approach to α-LEMMINGS. They were equally good over all, but TYPEC and TYPED were found to have some benefit. In these schemes, the solutions are initialised to the result that was found for the previous instance. From our investigation we saw that these two initialisation schemes have some benefit when the instances that are being tackled are similar. Thus, future work might aim to analyse how similar the instances are required to be in order for these schemes to have any benefit. Then, we might attempt to automate the process of judging whether two instances are similar so that we can employ these initialisation schemes at the best time in order to benefit from them.

In chapter 6 we described the first successful approach for producing feasible solutions to LEMMINGS. Thus, future investigations will have a basis from which to work. That is, in the future, time need not be spent in finding an approach for LEMMINGS, but can instead be spent with further investigations. An interesting avenue for further investigation might be to find instances of LEMMINGS that cannot possibly be tackled using our approach. This links to our findings in chapter 7, where we saw that our approach was not able to locate a feasible solution to some

LEMMINGS instances within the time that we gave it.

We have also described a set of parameters that seem to work best when a GA is used to approach LEMMINGS. These parameters agree with those described in [57]. Furthermore, we have shown that it is theoretically infeasible to approach LEMMINGS using uninformed search. This was expected. However, we have discussed it so it does not need to be done in the future.

Difficult LEMMINGS Instances

In chapter 7 we showed that for some partitions of LEMMINGS instances, the time taken to locate a feasible solution grows exponentially. Importantly, these partitions of instances that have exponential growth are the more difficult LEMMINGS instances of those that we have approached. We have attempted to describe these partitions of difficult instances as those for which feasible solutions contain non-pass actions that must be played at precisely one time-step and the number of these actions increases. We have shown that as the number of these actions increases linearly the time taken to locate a feasible solution grows exponentially for these difficult instance partitions.

We provided the first indication of difficult LEMMINGS instances, and have described a method for generating them. Therefore, future work does not need to find them, but can instead concentrate on further studies that use them. This work can either use the ones that are catalogued in Appendix A, or can use our method in order to generate some. The work that might require difficult instances is that for testing the performance of algorithms on benchmark problems.

In chapter 2 we saw how Cheeseman et al. [8] described a phase transition, around which hard instances of NP-Complete problems are located. In this thesis we have described a different approach for locating difficult instances of LEMMINGS. The reason that our approach is different is because our local search algorithm uses random operators and so it is not affected by the phase transition. If our algorithm had used local operators then we could have tried to locate a phase transition for LEMMINGS. Both their approach and our approach that is described here are successful in locating difficult instances. Recall also that Cheeseman et al. said that the instances on the phase transition are more difficult than those hard instances away from the phase transition. Therefore, it might be the case that if we apply an algorithm with local operators to LEMMINGS and locate a phase transition, then we might find even more difficult instances than we have located in this thesis. Finally, recall that Cheeseman et al. showed that in some cases the phase transition remains unchanged when problems are mapped. This suggests that if we located difficult instances of LEMMINGS surrounding a phase transition, then they might map to difficult instances of other problems. This is a further avenue that might be explored in order to locate difficult LEMMINGS instances in the future. That is, if we have a problem for which there exists a phase transition locating difficult instances, and we map that problem to LEMMINGS, then those difficult instances might map to difficult LEMMINGS instances.

We realise that one difficult instance for our approach might actually be a relatively simple instance for a different approach. Equally, our approach might have no trouble in locating a feasible solution to an instance, and this same instance might actually pose real difficulty for some other approach. Moreover, many of the decisions we have made in terms of the representation of our LEMMINGS solution might have been wrong decisions, and there might be better alternatives out there. To which we suggest that future research might attempt to capture a different approach to LEMMINGS and compare the difficult instances for that approach with the difficult instances that we have recognised. It might be that our difficult instances are also difficult for some other approaches.

Disadvantages to the approach are that it does not scale well and is limited over the number of instances that it can solve. However, we argue that we have solved enough instances to gain some insight into a difficulty landscape of LEMMINGS. Thus, we have not solved LEMMINGS because our approach did not solve some of the instances within the time that we allowed. This means that there are still unapproached areas of LEMMINGS. Consequently, our difficult instances are relative to a subset of LEMMINGS, and so there might be other descriptions of generating even more difficult unapproached instances that we do not know about. Also, our measure of difficulty might not be an accurate one because we use an informed method as a means of locating the feasible solutions, and thereby defining our difficult instances. A difficult instance is only difficult for the given informed approach. Our evaluation function might be so incorrect that the difficult instances are not actually difficult at all.

Time Complexity of the Algorithm

We showed that the GA is limited and does not scale well. Importantly, this evolutionary approach to LEMMINGS has exponential time complexity. Thus, future work that attempts to approach LEMMINGS has a measure to compare against so that the best approach can be found. The memory requirements of our GA are very low, and are linear over the size of the LEMMINGS instance. Maybe a better approach for LEMMINGS would have higher memory requirements that would be balanced by more reasonable time requirements.

Since our algorithm has exponential-time complexity, LEMMINGS cannot be shown to be in P. This agrees with Cormode's NP-Completeness proof for Lemmings [22]. In the future we might want to provide evidence for the inclusion of LEMMINGS in NP-Complete, and then attempt to find a polynomial-time algorithm for LEMMINGS. This would then mean that P = NP.

Figure 8.1 shows one of our main achievements for the thesis, that is we have successfully approached LEMMINGS using a genetic algorithm. The majority of LEMMINGS remains unexplored, but at least we have some preliminary knowledge of the difficulty landscape of LEMMINGS instances that we have approached in this thesis. This landscape of instances is catalogued in Appendix A.

Our investigations in this thesis form the first empirical investigations of LEMMINGS. As such they represent early work on the problem. However, we have described first empirical results. We have a basis for future investigations. Therefore, the limits to the approach and results can be addressed in future work. We have highlighted some open questions that arise from the investigations in this thesis, which can

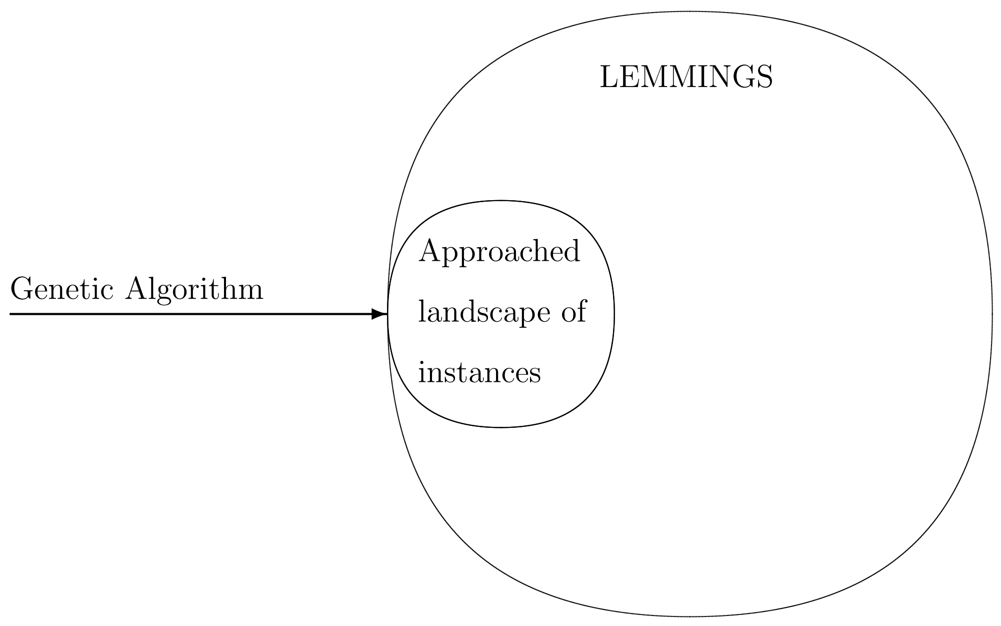
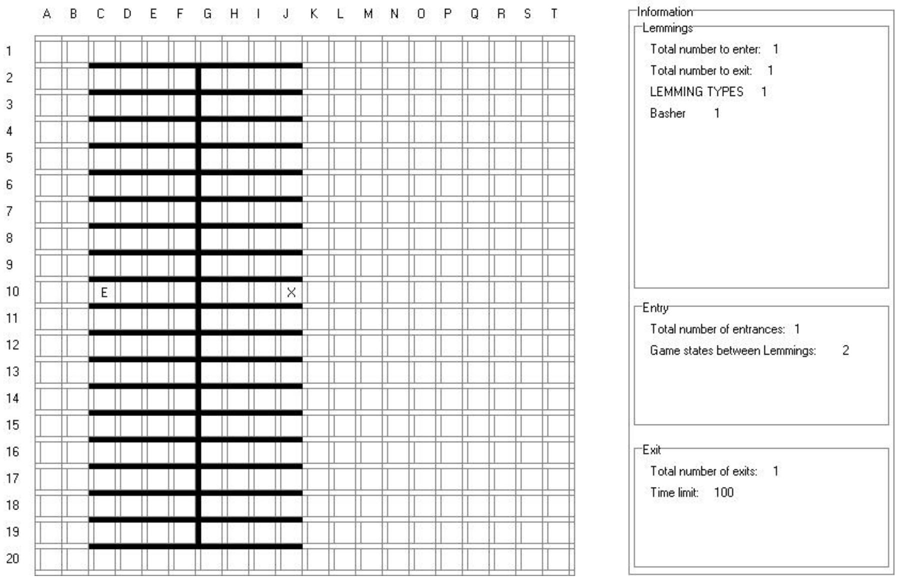


FIGURE 8.1: Approaching the world of LEMMINGS instances.

also be addressed in future work. Importantly, we have performed research on Lemmings (and derivatives), which can be added to the research performed on NP-Hard puzzles that is surveyed in chapter 4.

Appendix A

The Approached LEMMINGS Landscape



Information

Lemmings

Total number to enter: 1

Total number to exit: 1

LEMMING TYPES 1

Basher 1

Entry

Total number of entrances: 1

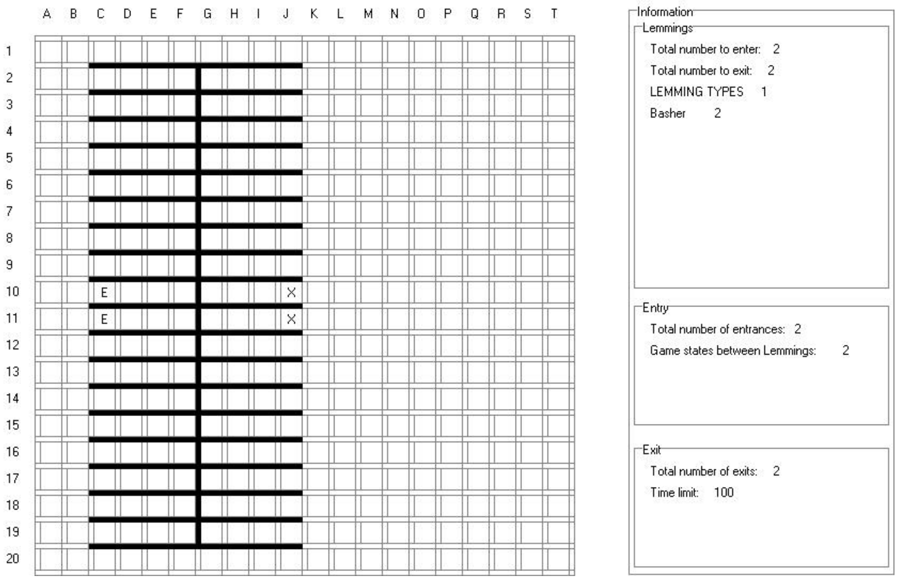
Game states between Lemmings: 2

Exit

Total number of exits: 1

Time limit: 100

Figure A.1: 01Bashers



Information

Lemmings

Total number to enter: 2

Total number to exit: 2

LEMMING TYPES 1

Basher 2

Entry

Total number of entrances: 2

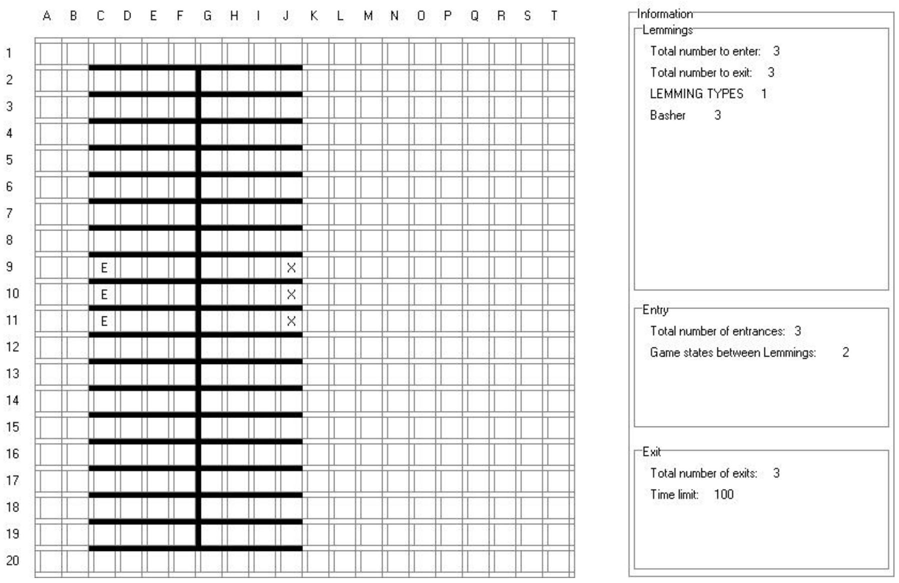
Game states between Lemmings: 2

Exit

Total number of exits: 2

Time limit: 100

Figure A.2: 02Bashers



Information

Lemmings

Total number to enter: 3

Total number to exit: 3

LEMMING TYPES 1

Basher 3

Entry

Total number of entrances: 3

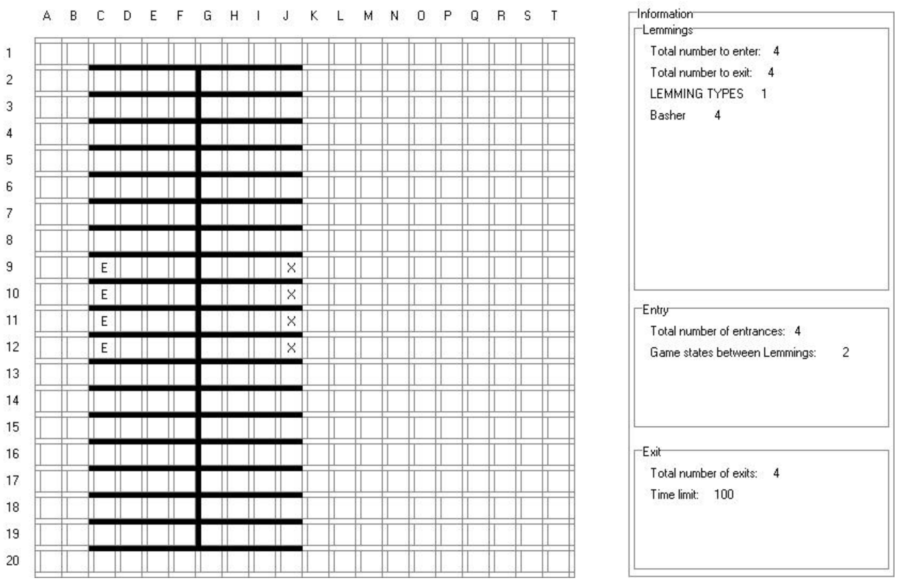
Game states between Lemmings: 2

Exit

Total number of exits: 3

Time limit: 100

Figure A.3: 03Bashers



Information

Lemmings

Total number to enter: 4

Total number to exit: 4

LEMMING TYPES 1

Basher 4

Entry

Total number of entrances: 4

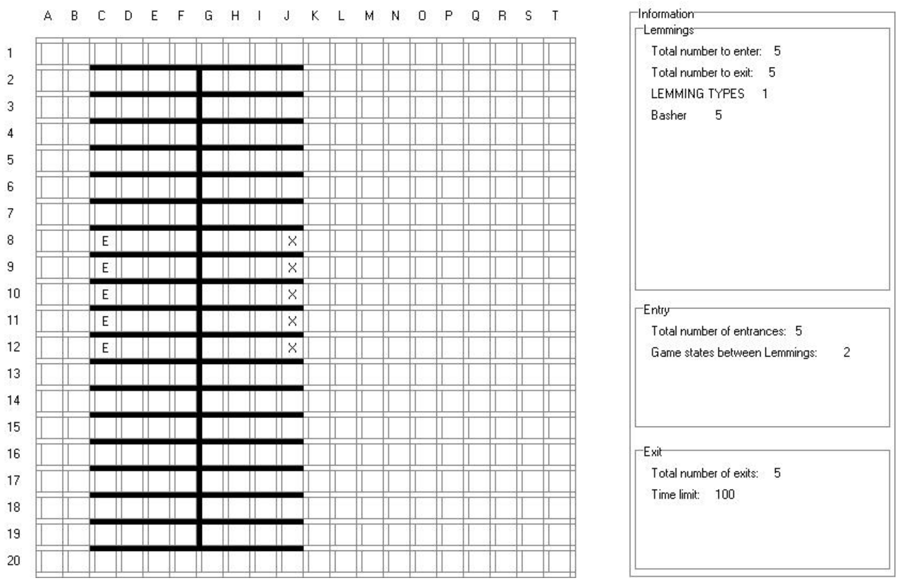
Game states between Lemmings: 2

Exit

Total number of exits: 4

Time limit: 100

Figure A.4: 04Bashers



Information

Lemmings

Total number to enter: 5

Total number to exit: 5

LEMMING TYPES 1

Basher 5

Entry

Total number of entrances: 5

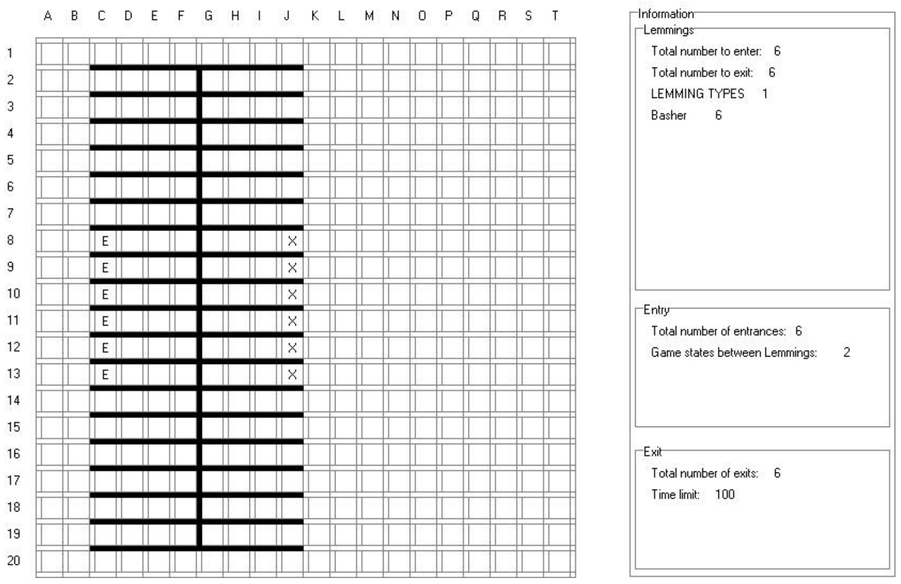
Game states between Lemmings: 2

Exit

Total number of exits: 5

Time limit: 100

Figure A.5: 05Bashers



Information

Lemmings

Total number to enter: 6

Total number to exit: 6

LEMMING TYPES 1

Basher 6

Entry

Total number of entrances: 6

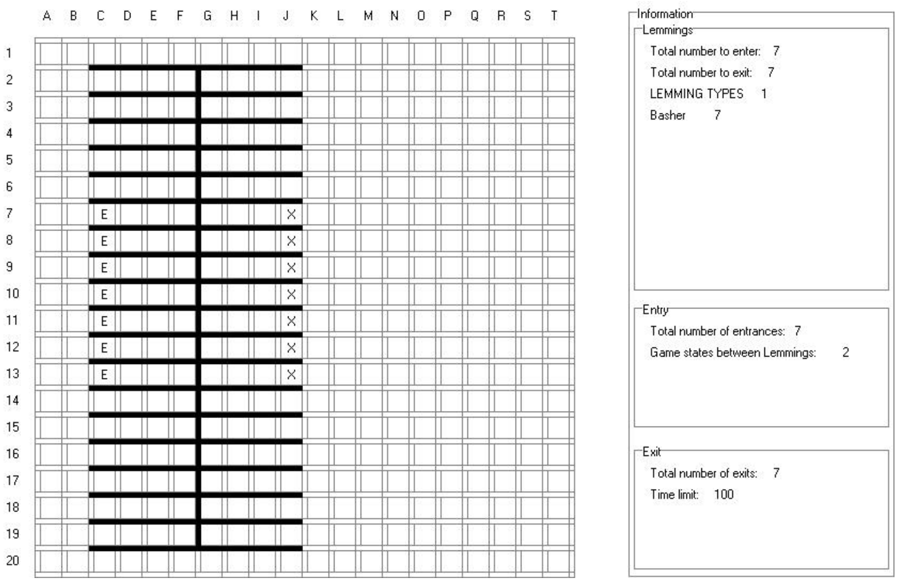
Game states between Lemmings: 2

Exit

Total number of exits: 6

Time limit: 100

Figure A.6: 06Bashers


Information

Lemmings

Total number to enter: 7

Total number to exit: 7

LEMMING TYPES 1

Basher 7

Entry

Total number of entrances: 7

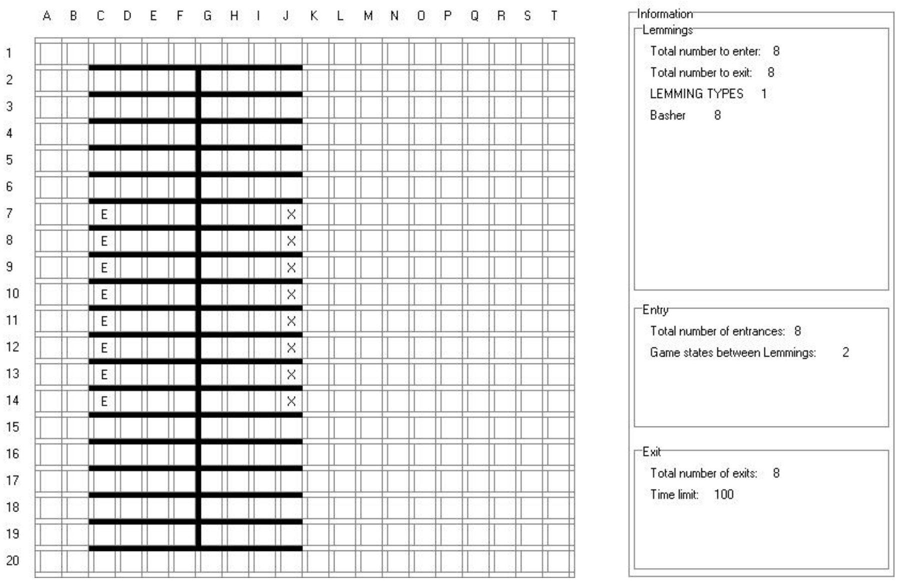
Game states between Lemmings: 2

Exit

Total number of exits: 7

Time limit: 100

Figure A.7: 07Bashers


Information

Lemmings

Total number to enter: 8

Total number to exit: 8

LEMMING TYPES 1

Basher 8

Entry

Total number of entrances: 8

Game states between Lemmings: 2

Exit

Total number of exits: 8

Time limit: 100

Figure A.8: 08Bashers

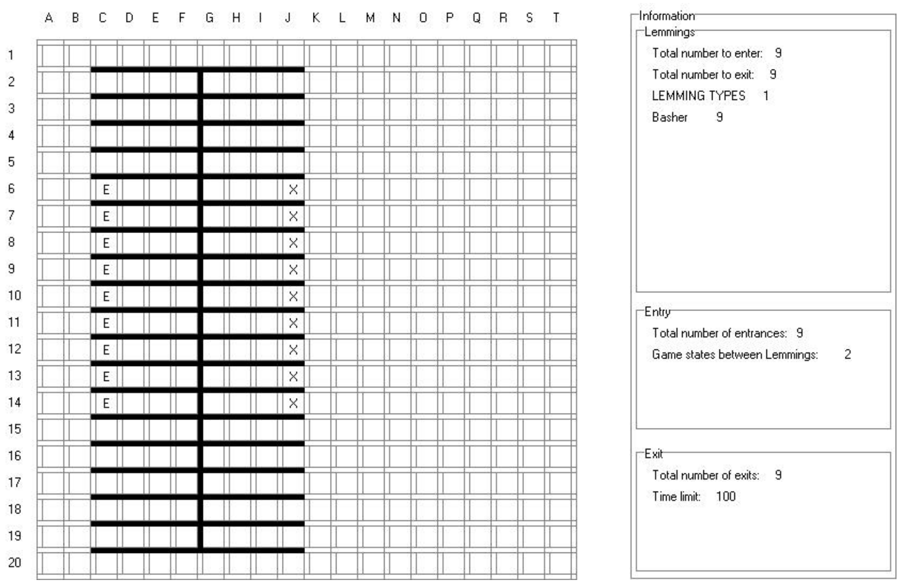

Information
Lemmings
Total number to enter: 9
Total number to exit: 9
LEMMING TYPES 1
Basher 9
Entry
Total number of entrances: 9
Game states between Lemmings: 2
Exit
Total number of exits: 9
Time limit: 100

Figure A.9: 09Bashers

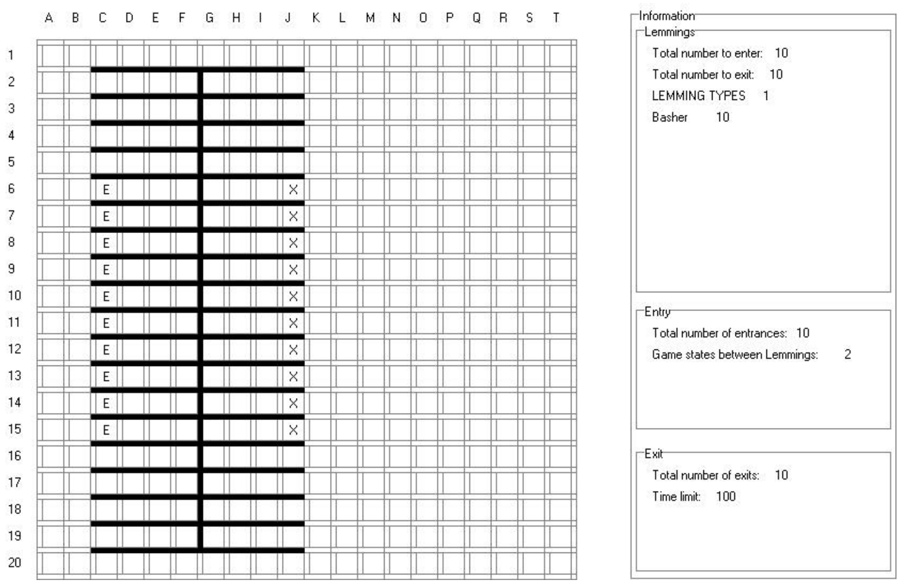
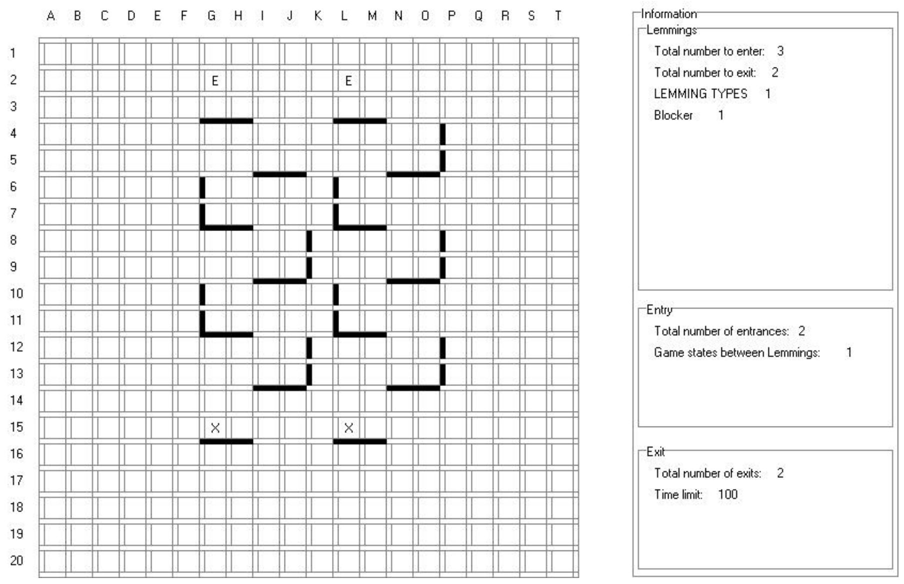

Information
Lemmings
Total number to enter: 10
Total number to exit: 10
LEMMING TYPES 1
Basher 10
Entry
Total number of entrances: 10
Game states between Lemmings: 2
Exit
Total number of exits: 10
Time limit: 100

Figure A.10: 10Bashers



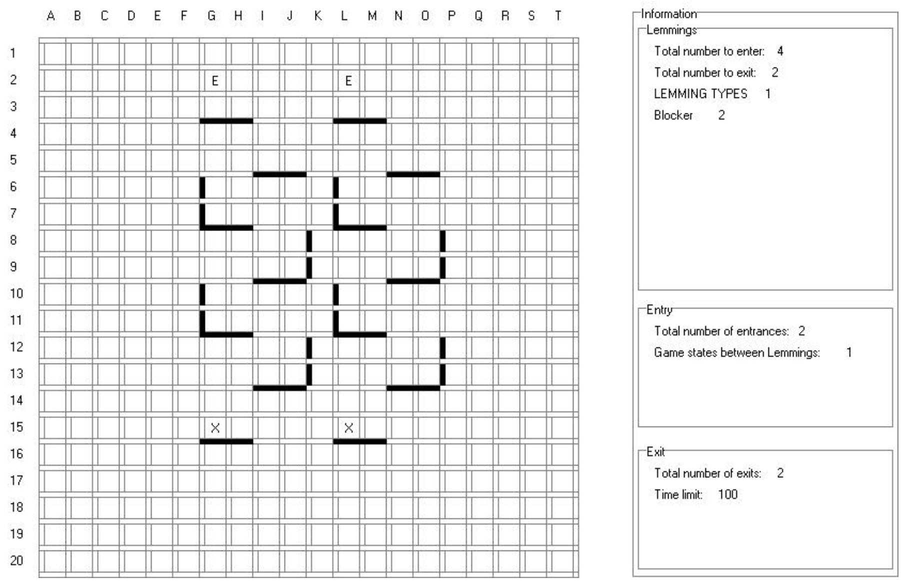
Information

Lemmings
Total number to enter: 3
Total number to exit: 2
LEMMING TYPES 1
Blocker 1

Entry
Total number of entrances: 2
Game states between Lemmings: 1

Exit
Total number of exits: 2
Time limit: 100

Figure A.11: 01Blockers



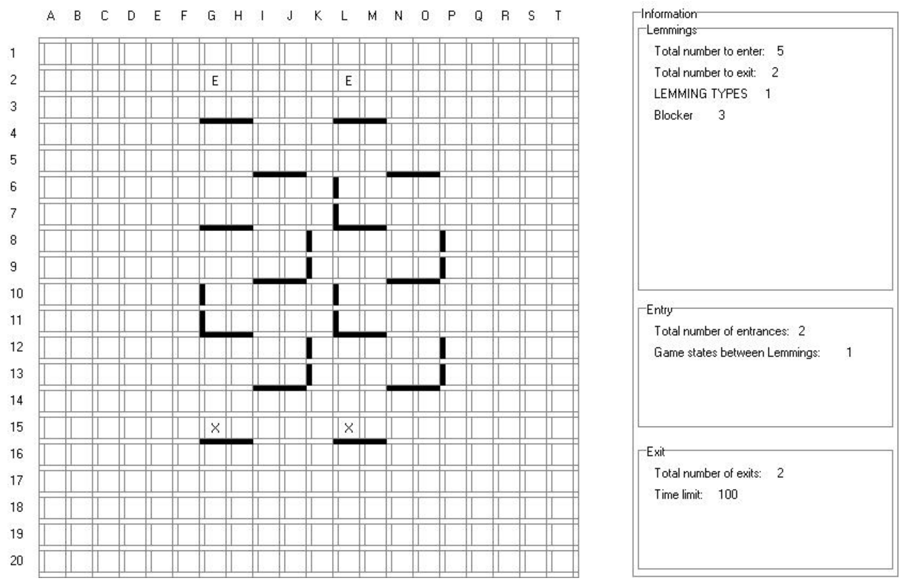
Information

Lemmings
Total number to enter: 4
Total number to exit: 2
LEMMING TYPES 1
Blocker 2

Entry
Total number of entrances: 2
Game states between Lemmings: 1

Exit
Total number of exits: 2
Time limit: 100

Figure A.12: 02Blockers

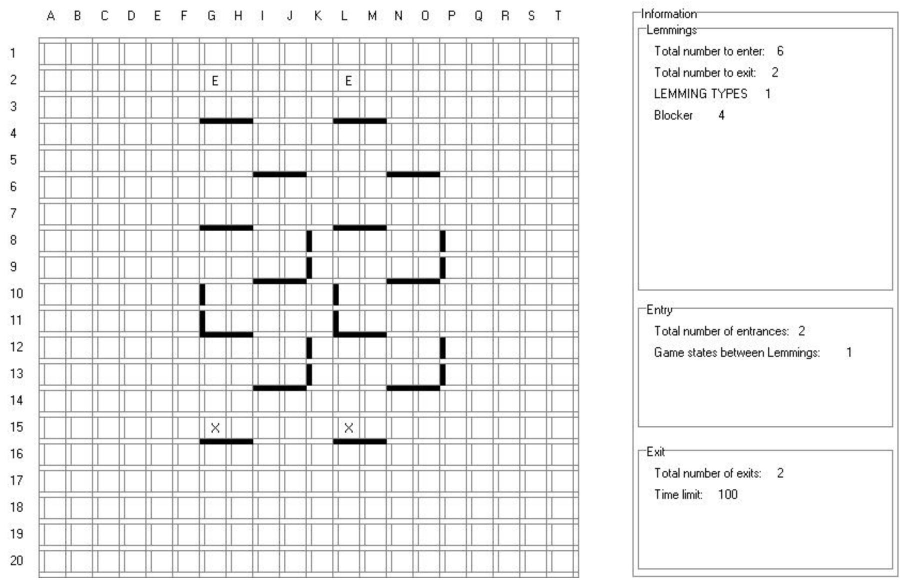


Information
Lemmings
Total number to enter: 5
Total number to exit: 2
LEMMING TYPES 1
Blocker 3

Entry
Total number of entrances: 2
Game states between Lemmings: 1

Exit
Total number of exits: 2
Time limit: 100

FIGURE A.13: 03Blockers

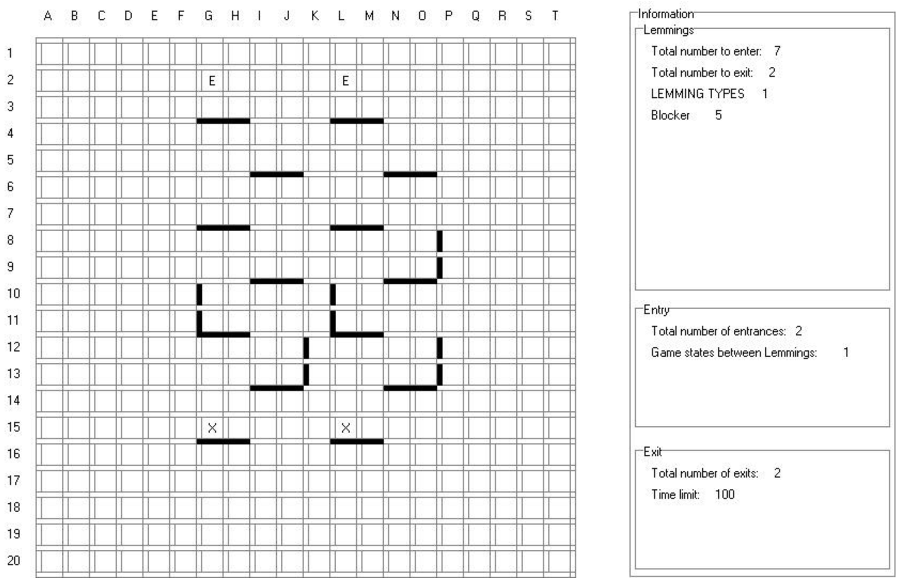


Information
Lemmings
Total number to enter: 6
Total number to exit: 2
LEMMING TYPES 1
Blocker 4

Entry
Total number of entrances: 2
Game states between Lemmings: 1

Exit
Total number of exits: 2
Time limit: 100

FIGURE A.14: 04Blockers



Information

Lemmings

Total number to enter: 7

Total number to exit: 2

LEMMING TYPES 1

Blocker 5

Entry

Total number of entrances: 2

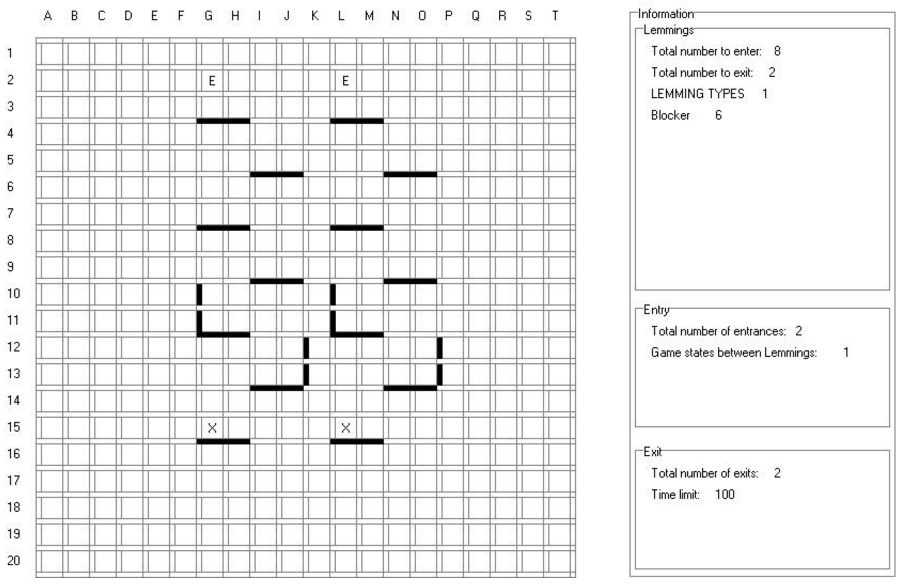
Game states between Lemmings: 1

Exit

Total number of exits: 2

Time limit: 100

Figure A.15: 05Blockers



Information

Lemmings

Total number to enter: 8

Total number to exit: 2

LEMMING TYPES 1

Blocker 6

Entry

Total number of entrances: 2

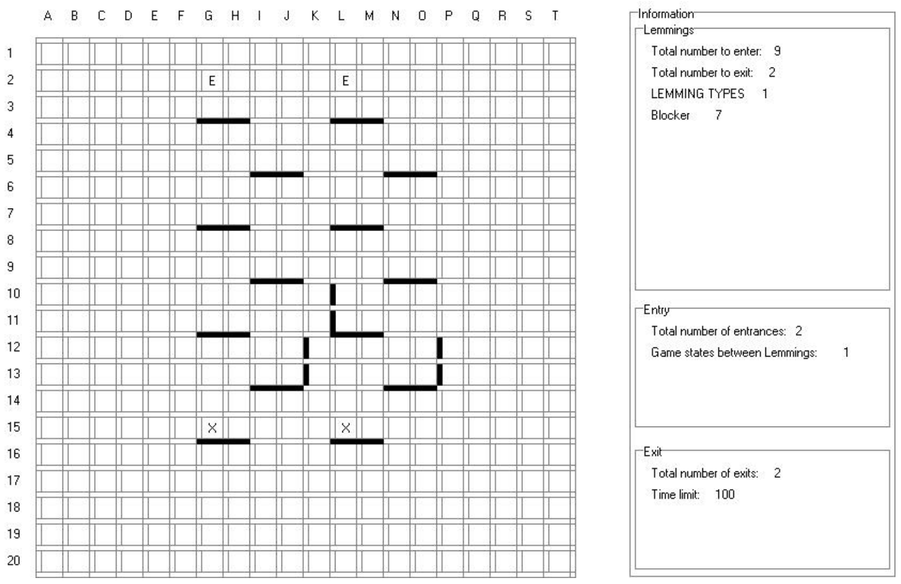
Game states between Lemmings: 1

Exit

Total number of exits: 2

Time limit: 100

Figure A.16: 06Blockers

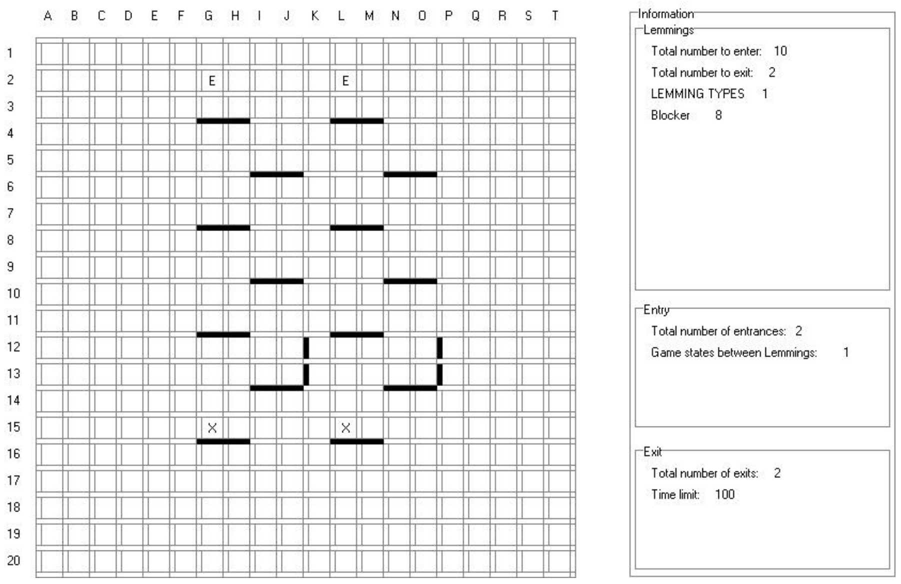


Information
Lemmings
Total number to enter: 9
Total number to exit: 2
LEMMING TYPES 1
Blocker 7

Entry
Total number of entrances: 2
Game states between Lemmings: 1

Exit
Total number of exits: 2
Time limit: 100

Figure A.17: 07Blockers

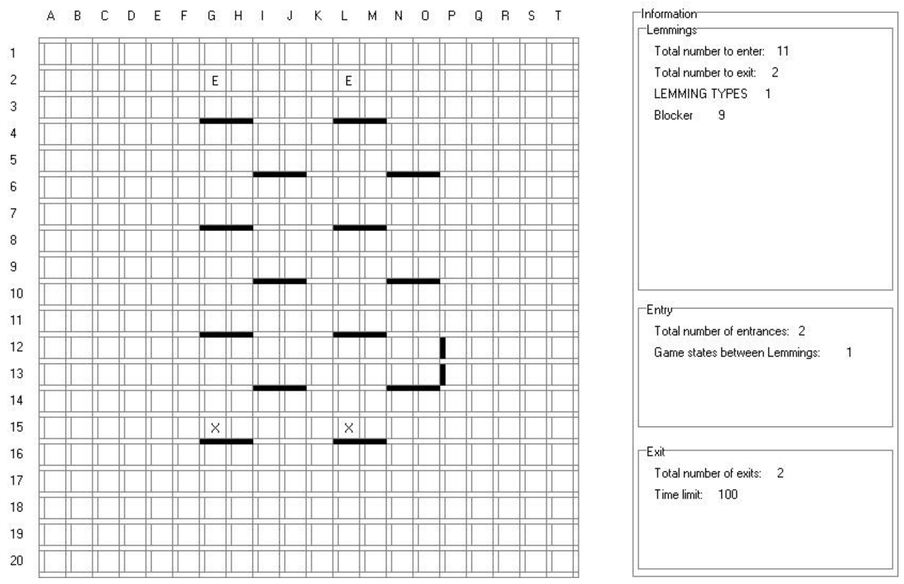


Information
Lemmings
Total number to enter: 10
Total number to exit: 2
LEMMING TYPES 1
Blocker 8

Entry
Total number of entrances: 2
Game states between Lemmings: 1

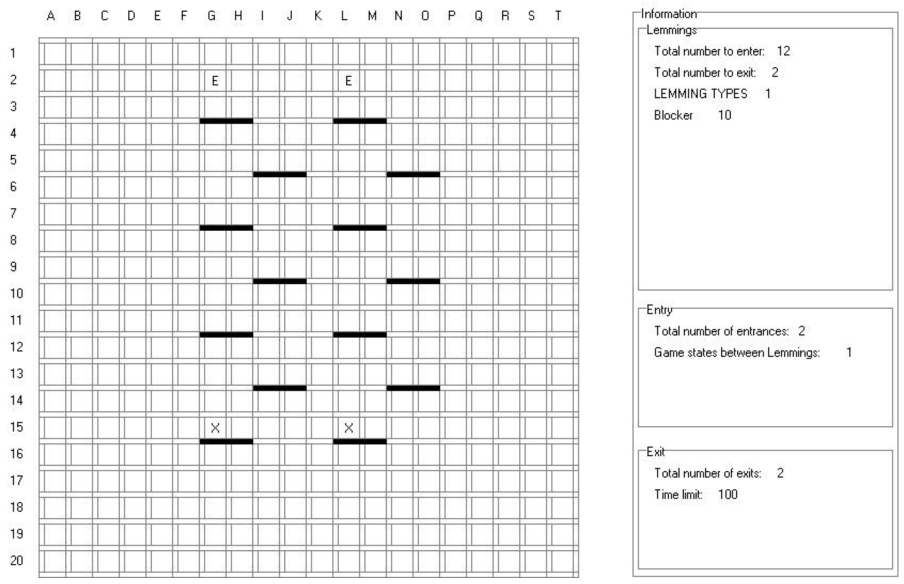
Exit
Total number of exits: 2
Time limit: 100

Figure A.18: 08Blockers



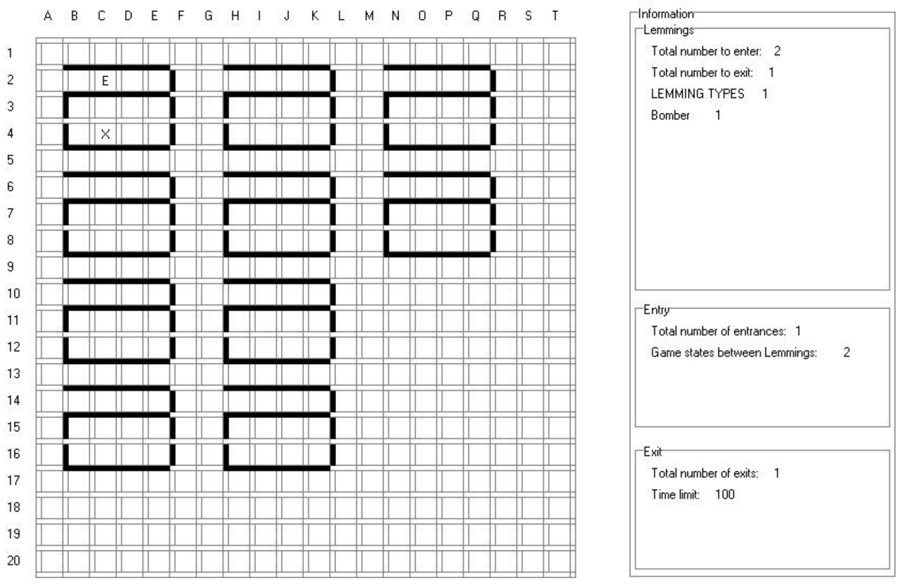
Information
Lemmings
Total number to enter: 11
Total number to exit: 2
LEMMING TYPES 1
Blocker 9
Entry
Total number of entrances: 2
Game states between Lemmings: 1
Exit
Total number of exits: 2
Time limit: 100

Figure A.19: 09Blockers



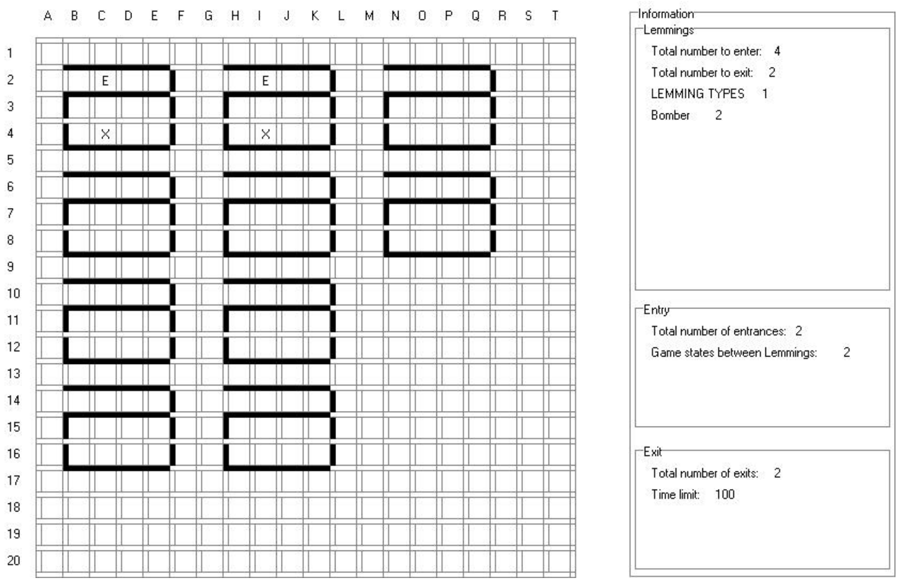
Information
Lemmings
Total number to enter: 12
Total number to exit: 2
LEMMING TYPES 1
Blocker 10
Entry
Total number of entrances: 2
Game states between Lemmings: 1
Exit
Total number of exits: 2
Time limit: 100

Figure A.20: 10Blockers



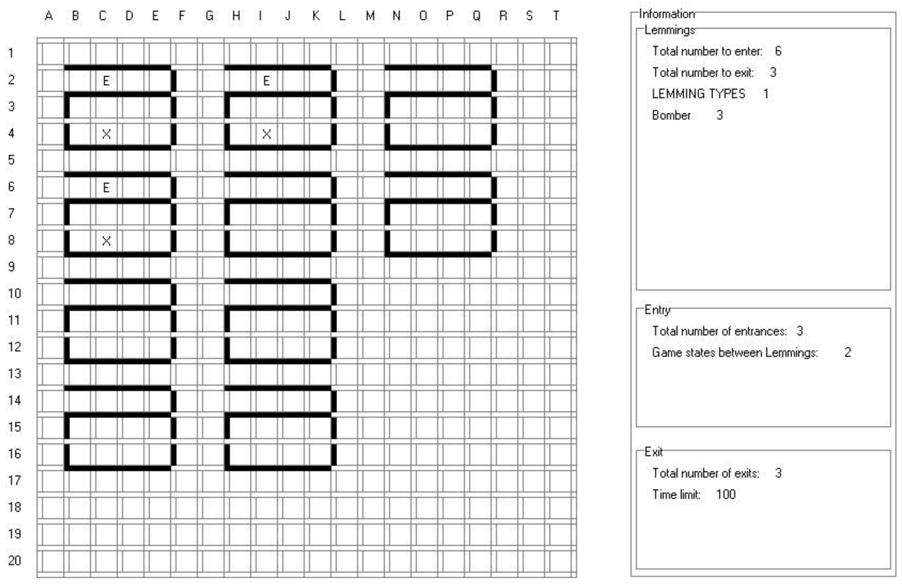
Information
Lemmings
Total number to enter: 2
Total number to exit: 1
LEMMING TYPES 1
Bomber 1
Entry
Total number of entrances: 1
Game states between Lemmings: 2
Exit
Total number of exits: 1
Time limit: 100

Figure A.21: 01Bombers



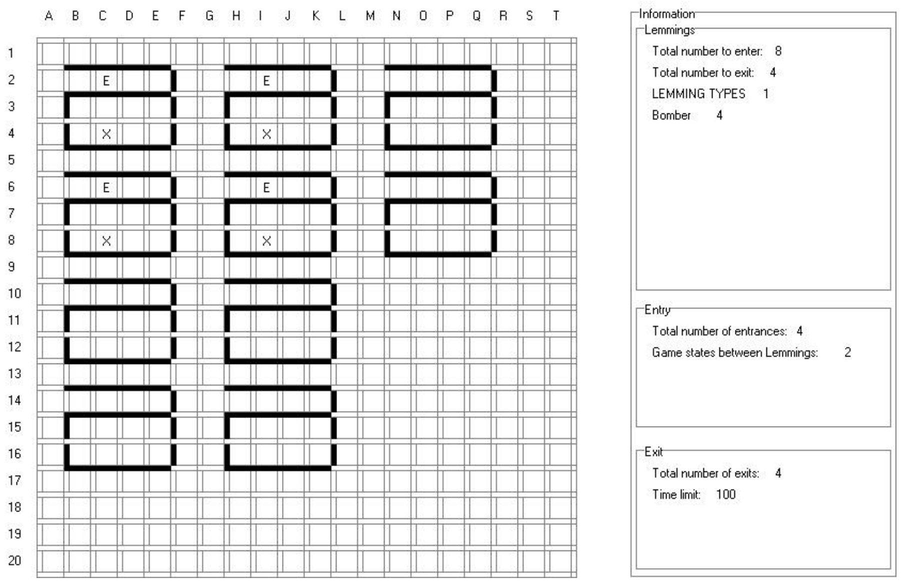
Information
Lemmings
Total number to enter: 4
Total number to exit: 2
LEMMING TYPES 1
Bomber 2
Entry
Total number of entrances: 2
Game states between Lemmings: 2
Exit
Total number of exits: 2
Time limit: 100

Figure A.22: 02Bombers



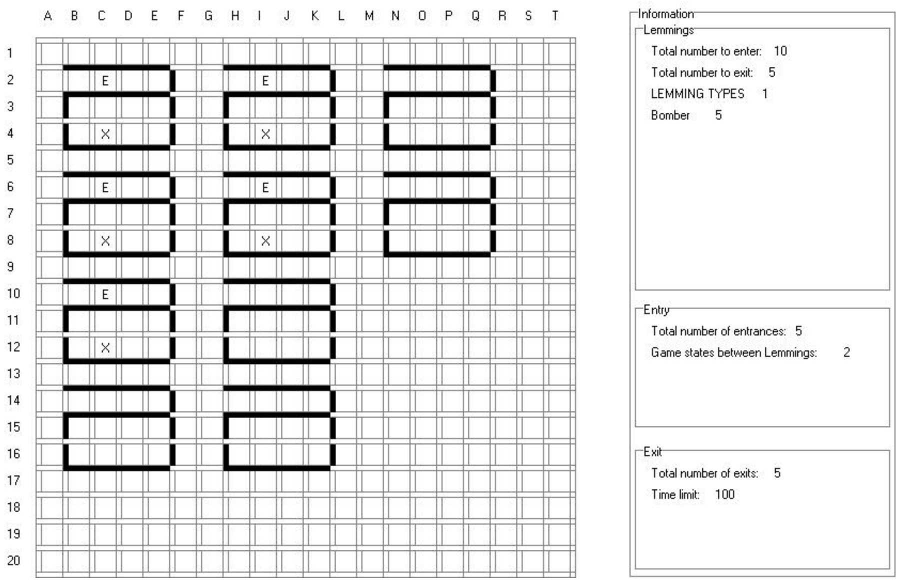
Information
Lemmings
Total number to enter: 6
Total number to exit: 3
LEMMING TYPES 1
Bomber 3
Entry
Total number of entrances: 3
Game states between Lemmings: 2
Exit
Total number of exits: 3
Time limit: 100

Figure A.23: 03Bombers



Information
Lemmings
Total number to enter: 8
Total number to exit: 4
LEMMING TYPES 1
Bomber 4
Entry
Total number of entrances: 4
Game states between Lemmings: 2
Exit
Total number of exits: 4
Time limit: 100

Figure A.24: 04Bombers

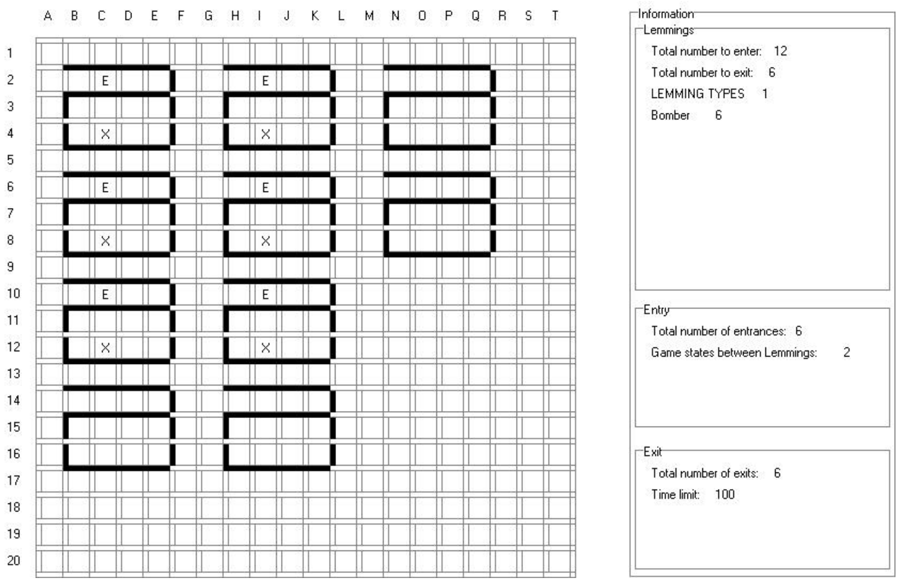


Information
Lemmings
Total number to enter: 10
Total number to exit: 5
LEMMING TYPES 1
Bomber 5

Entry
Total number of entrances: 5
Game states between Lemmings: 2

Exit
Total number of exits: 5
Time limit: 100

Figure A.25: 05Bombers

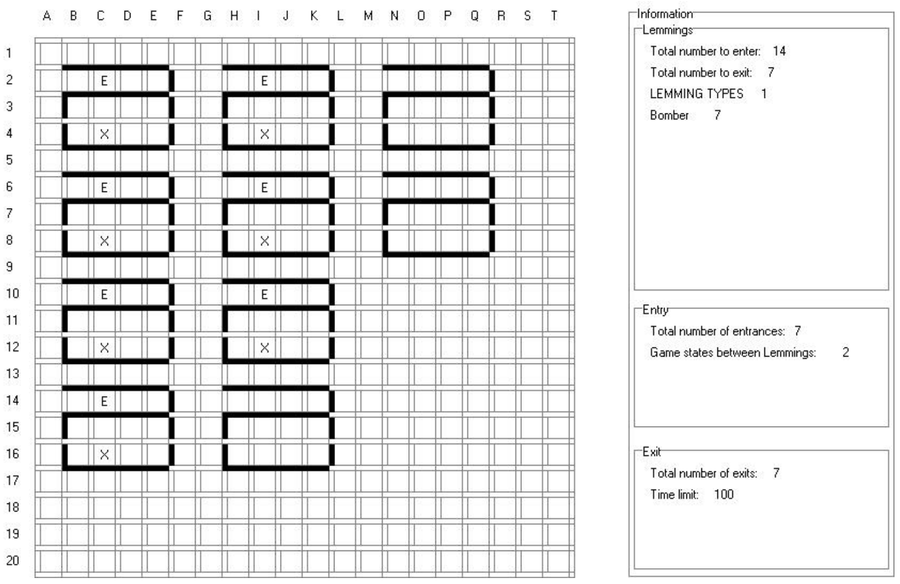


Information
Lemmings
Total number to enter: 12
Total number to exit: 6
LEMMING TYPES 1
Bomber 6

Entry
Total number of entrances: 6
Game states between Lemmings: 2

Exit
Total number of exits: 6
Time limit: 100

Figure A.26: 06Bombers



Information

Lemmings

Total number to enter: 14

Total number to exit: 7

LEMMING TYPES 1

Bomber 7

Entry

Total number of entrances: 7

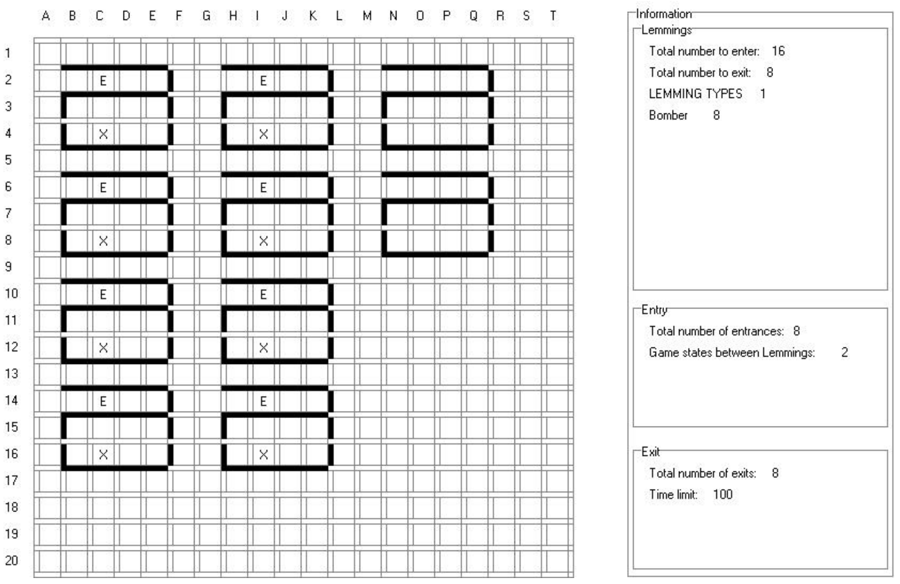
Game states between Lemmings: 2

Exit

Total number of exits: 7

Time limit: 100

Figure A.27: 07Bombers



Information

Lemmings

Total number to enter: 16

Total number to exit: 8

LEMMING TYPES 1

Bomber 8

Entry

Total number of entrances: 8

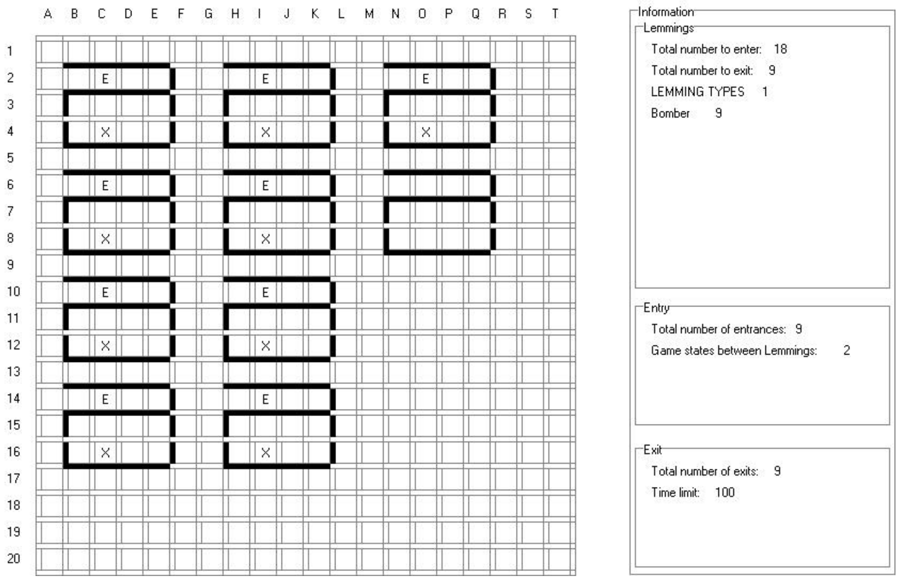
Game states between Lemmings: 2

Exit

Total number of exits: 8

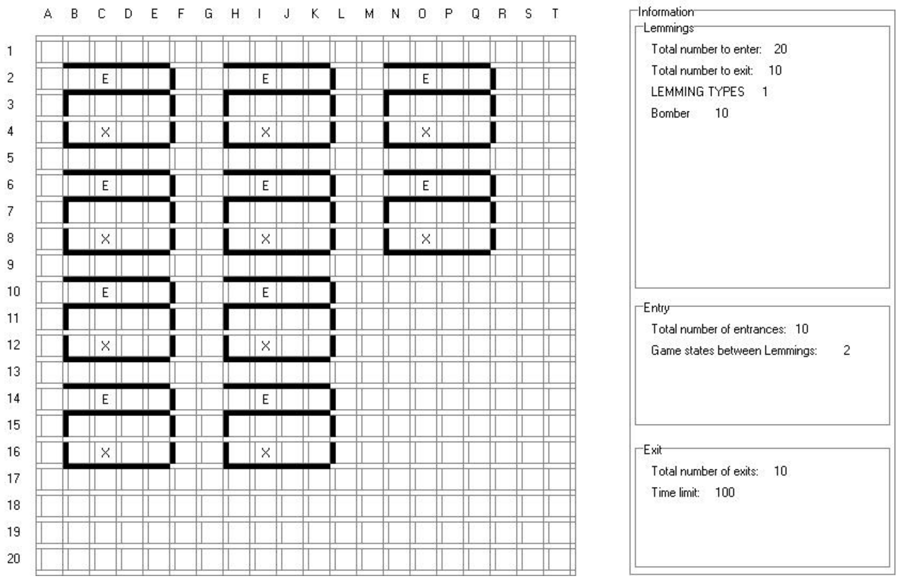
Time limit: 100

Figure A.28: 08Bombers



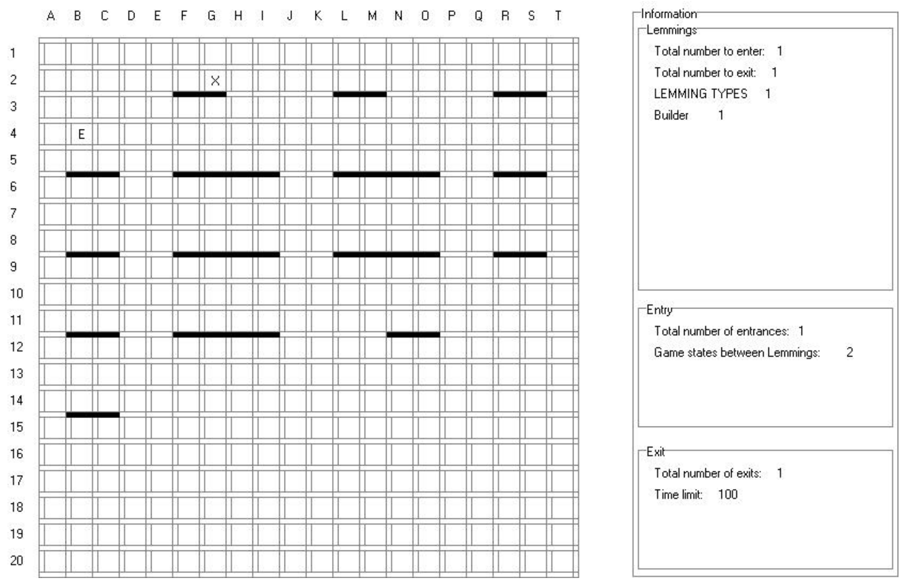
Information
Lemmings
Total number to enter: 18
Total number to exit: 9
LEMMING TYPES 1
Bomber 9
Entry
Total number of entrances: 9
Game states between Lemmings: 2
Exit
Total number of exits: 9
Time limit: 100

Figure A.29: 09Bombers



Information
Lemmings
Total number to enter: 20
Total number to exit: 10
LEMMING TYPES 1
Bomber 10
Entry
Total number of entrances: 10
Game states between Lemmings: 2
Exit
Total number of exits: 10
Time limit: 100

Figure A.30: 10Bombers

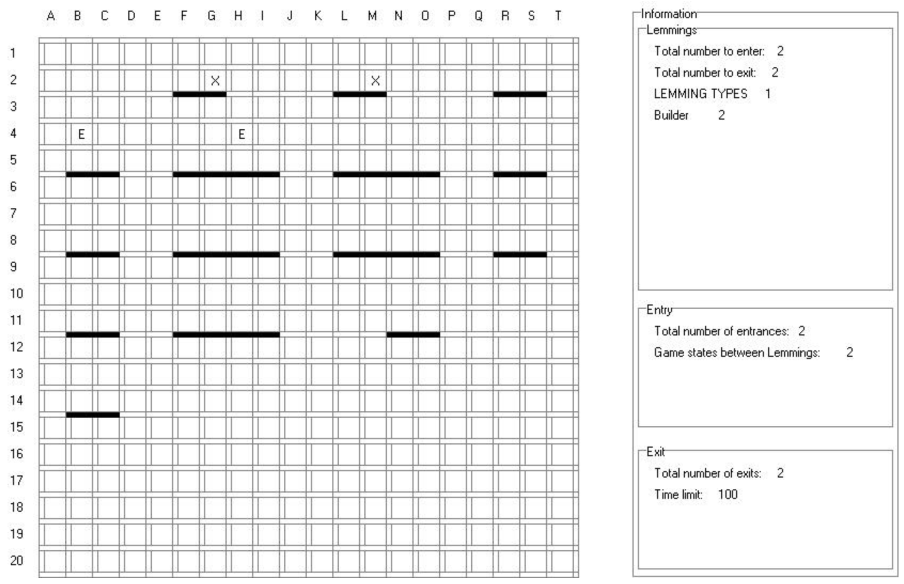


Information
Lemmings
Total number to enter: 1
Total number to exit: 1
LEMMING TYPES 1
Builder 1

Entry
Total number of entrances: 1
Game states between Lemmings: 2

Exit
Total number of exits: 1
Time limit: 100

Figure A.31: 01Builders

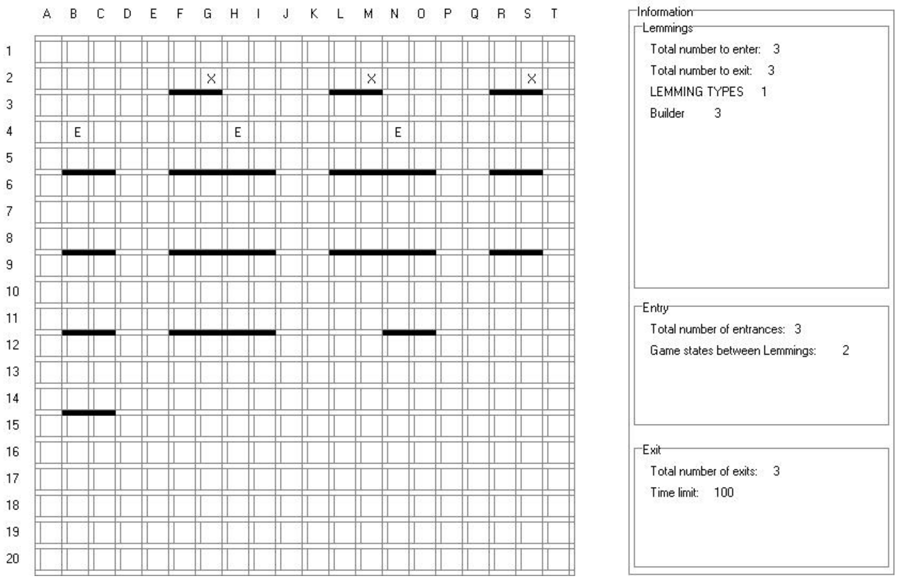


Information
Lemmings
Total number to enter: 2
Total number to exit: 2
LEMMING TYPES 1
Builder 2

Entry
Total number of entrances: 2
Game states between Lemmings: 2

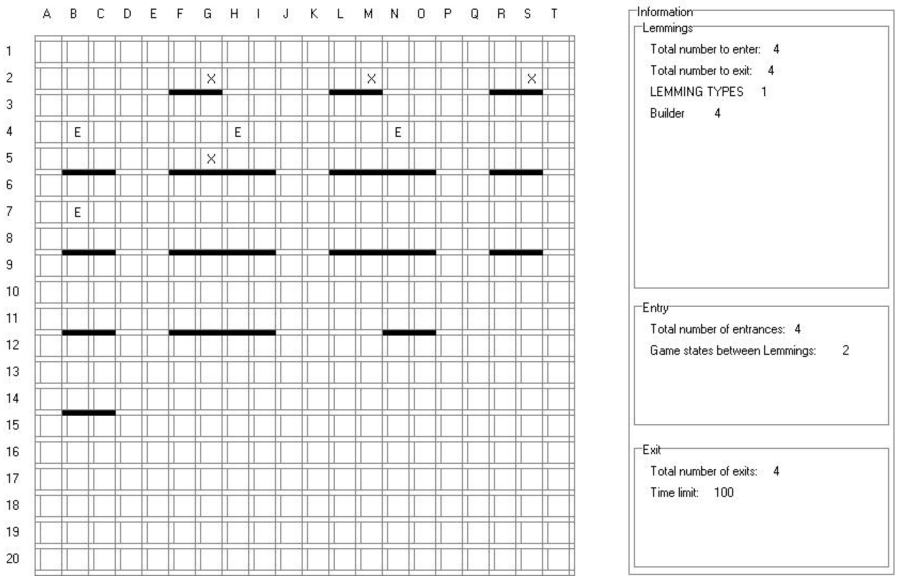
Exit
Total number of exits: 2
Time limit: 100

Figure A.32: 02Builders



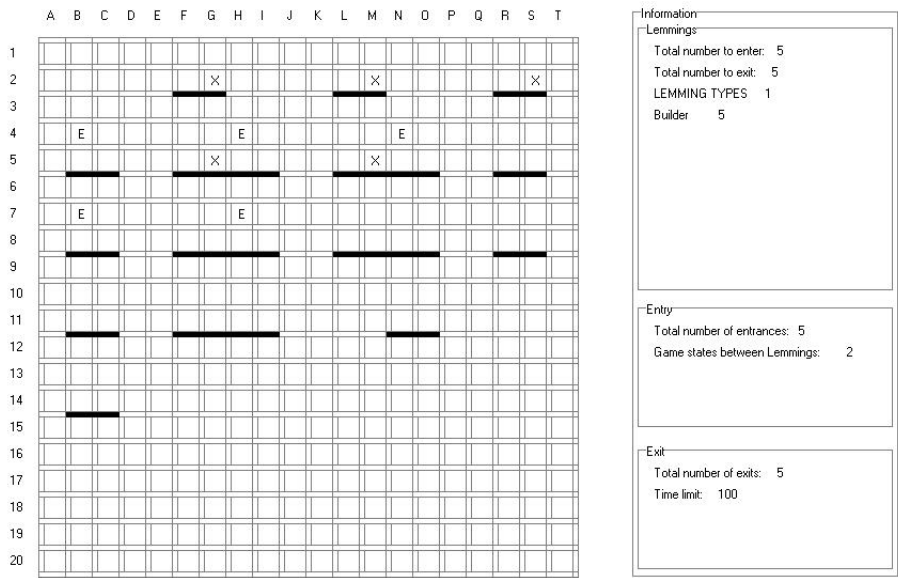
Information
Lemmings
Total number to enter: 3
Total number to exit: 3
LEMMING TYPES 1
Builder 3
Entry
Total number of entrances: 3
Game states between Lemmings: 2
Exit
Total number of exits: 3
Time limit: 100

Figure A.33: 03Builders



Information
Lemmings
Total number to enter: 4
Total number to exit: 4
LEMMING TYPES 1
Builder 4
Entry
Total number of entrances: 4
Game states between Lemmings: 2
Exit
Total number of exits: 4
Time limit: 100

Figure A.34: 04Builders

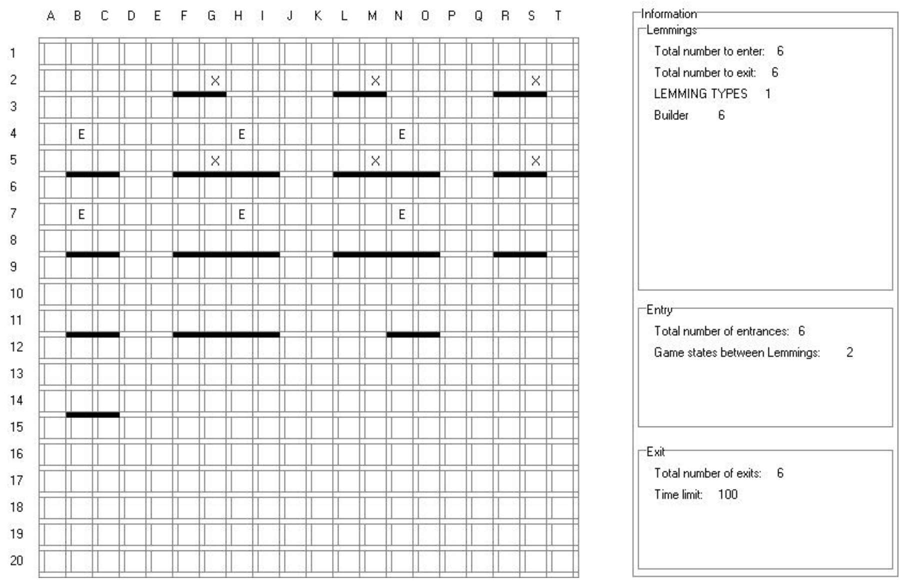


Information
Lemmings
Total number to enter: 5
Total number to exit: 5
LEMMING TYPES 1
Builder 5

Entry
Total number of entrances: 5
Game states between Lemmings: 2

Exit
Total number of exits: 5
Time limit: 100

Figure A.35: 05Builders

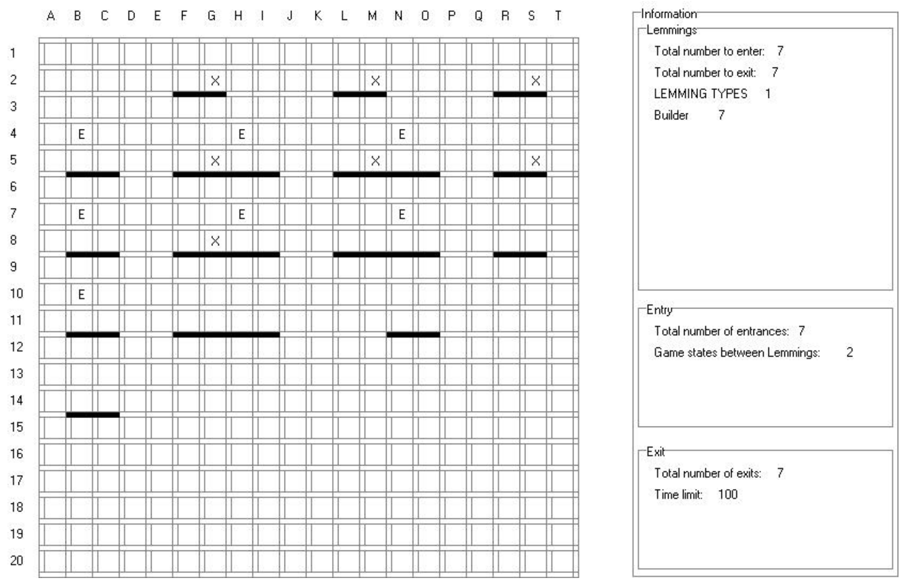


Information
Lemmings
Total number to enter: 6
Total number to exit: 6
LEMMING TYPES 1
Builder 6

Entry
Total number of entrances: 6
Game states between Lemmings: 2

Exit
Total number of exits: 6
Time limit: 100

Figure A.36: 06Builders

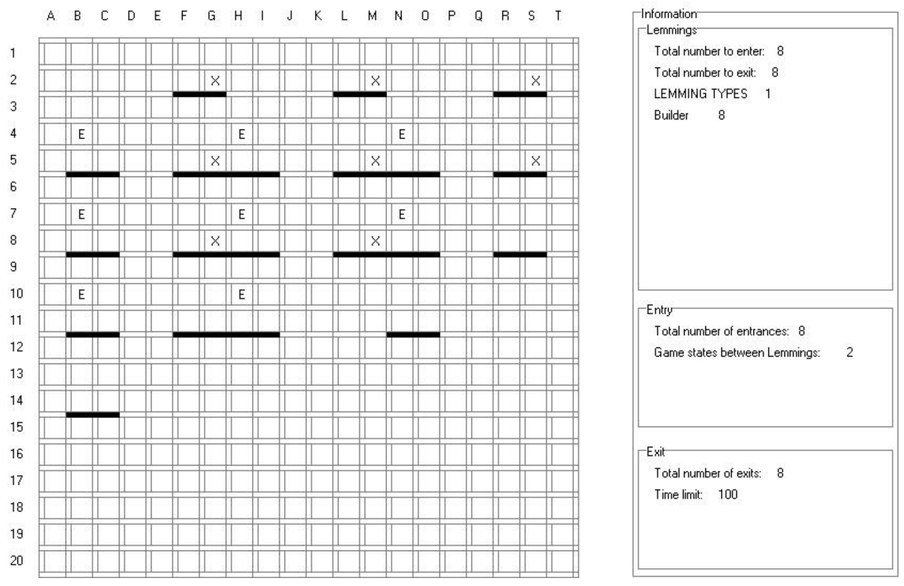


Information
Lemmings
Total number to enter: 7
Total number to exit: 7
LEMMING TYPES 1
Builder 7

Entry
Total number of entrances: 7
Game states between Lemmings: 2

Exit
Total number of exits: 7
Time limit: 100

Figure A.37: 07Builders

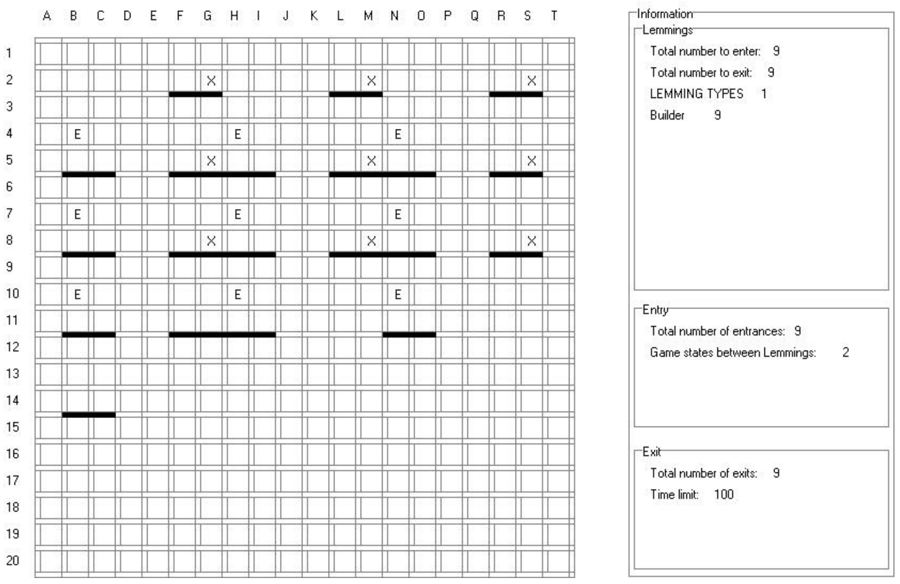


Information
Lemmings
Total number to enter: 8
Total number to exit: 8
LEMMING TYPES 1
Builder 8

Entry
Total number of entrances: 8
Game states between Lemmings: 2

Exit
Total number of exits: 8
Time limit: 100

Figure A.38: 08Builders



Information

Lemmings

Total number to enter: 9

Total number to exit: 9

LEMMING TYPES 1

Builder 9

Entry

Total number of entrances: 9

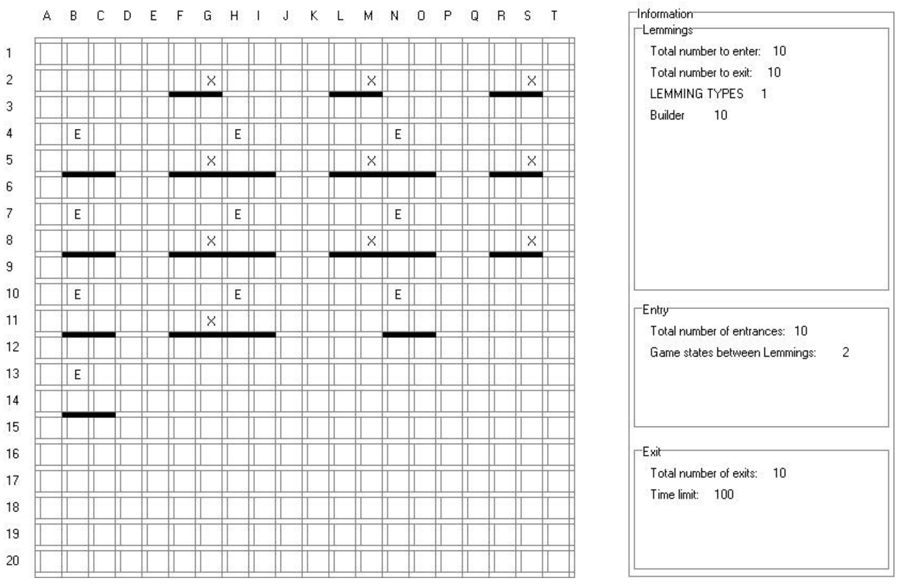
Game states between Lemmings: 2

Exit

Total number of exits: 9

Time limit: 100

Figure A.39: 09Builders



Information

Lemmings

Total number to enter: 10

Total number to exit: 10

LEMMING TYPES 1

Builder 10

Entry

Total number of entrances: 10

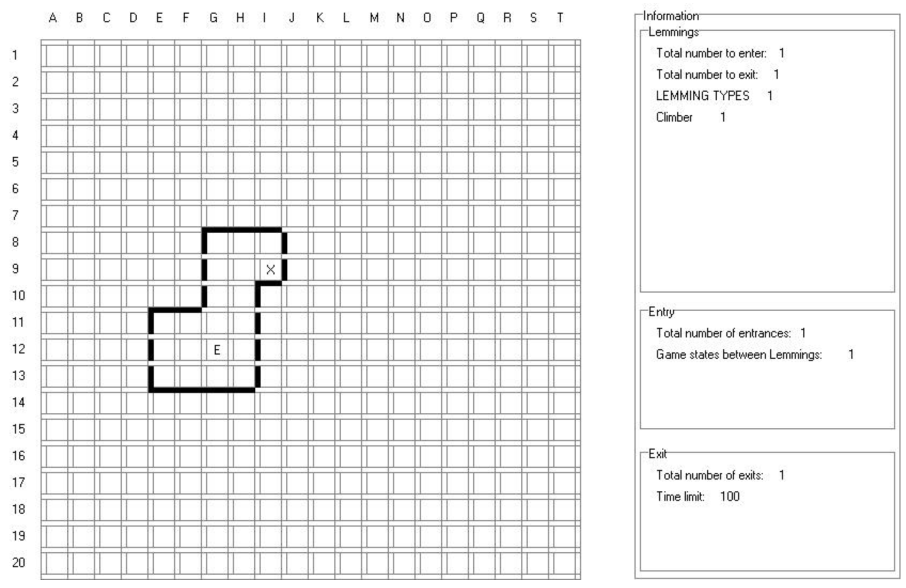
Game states between Lemmings: 2

Exit

Total number of exits: 10

Time limit: 100

Figure A.40: 10Builders

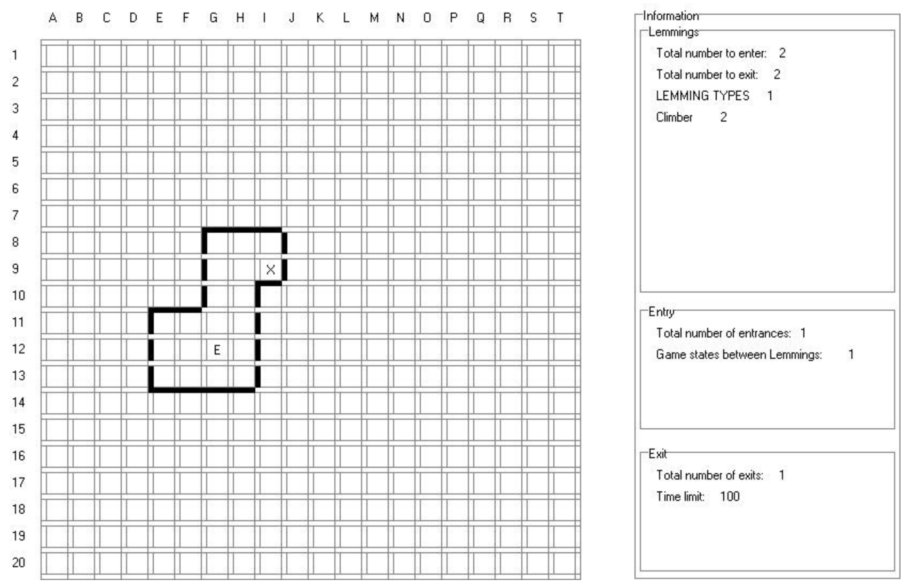


Information
Lemmings
Total number to enter: 1
Total number to exit: 1
LEMMING TYPES 1
Climber 1

Entry
Total number of entrances: 1
Game states between Lemmings: 1

Exit
Total number of exits: 1
Time limit: 100

FIGURE A.41: 01Climbers

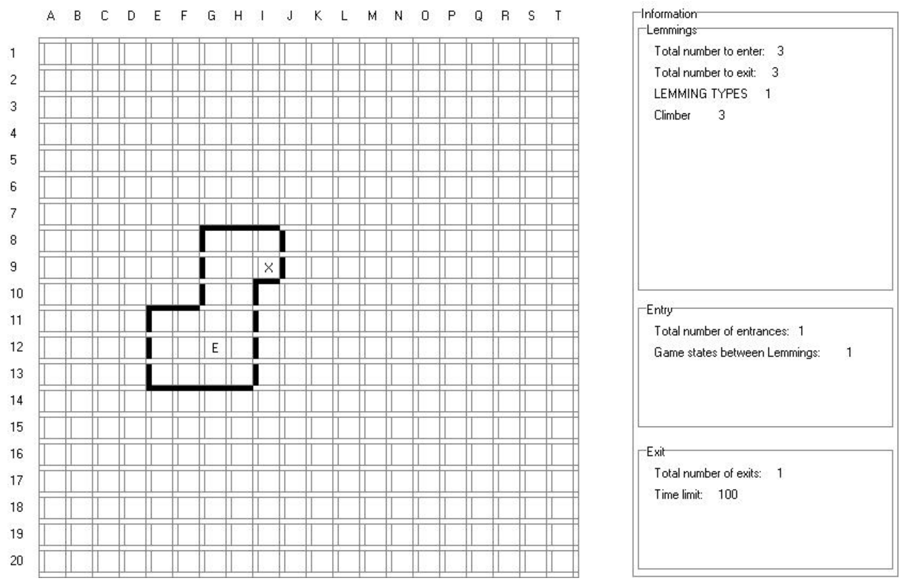


Information
Lemmings
Total number to enter: 2
Total number to exit: 2
LEMMING TYPES 1
Climber 2

Entry
Total number of entrances: 1
Game states between Lemmings: 1

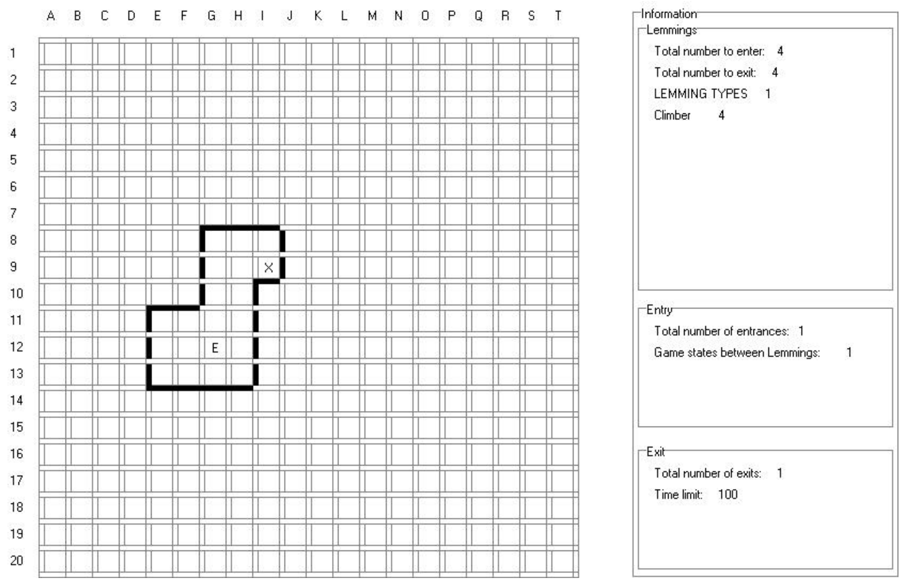
Exit
Total number of exits: 1
Time limit: 100

FIGURE A.42: 02Climbers



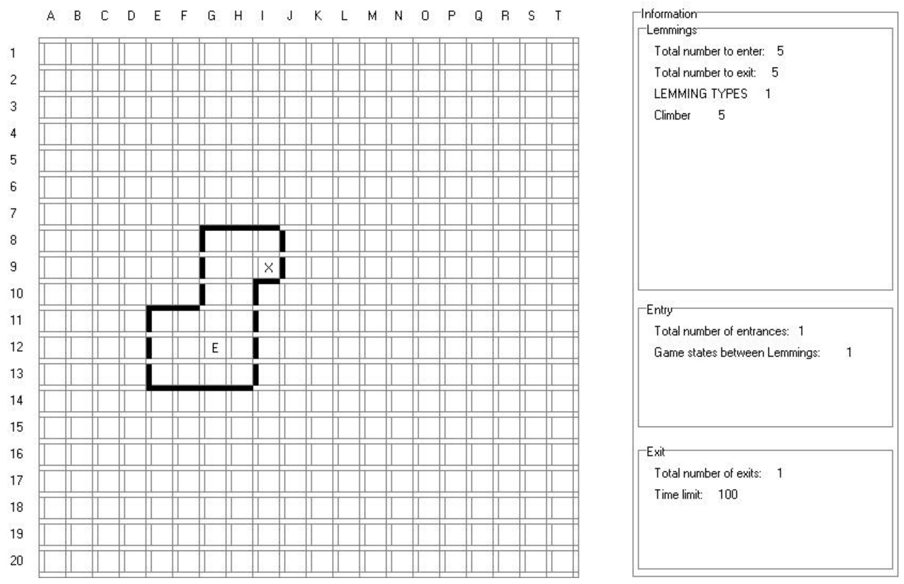
Information
Lemmings
Total number to enter: 3
Total number to exit: 3
LEMMING TYPES 1
Climber 3
Entry
Total number of entrances: 1
Game states between Lemmings: 1
Exit
Total number of exits: 1
Time limit: 100

Figure A.43: 03Climbers



Information
Lemmings
Total number to enter: 4
Total number to exit: 4
LEMMING TYPES 1
Climber 4
Entry
Total number of entrances: 1
Game states between Lemmings: 1
Exit
Total number of exits: 1
Time limit: 100

Figure A.44: 04Climbers

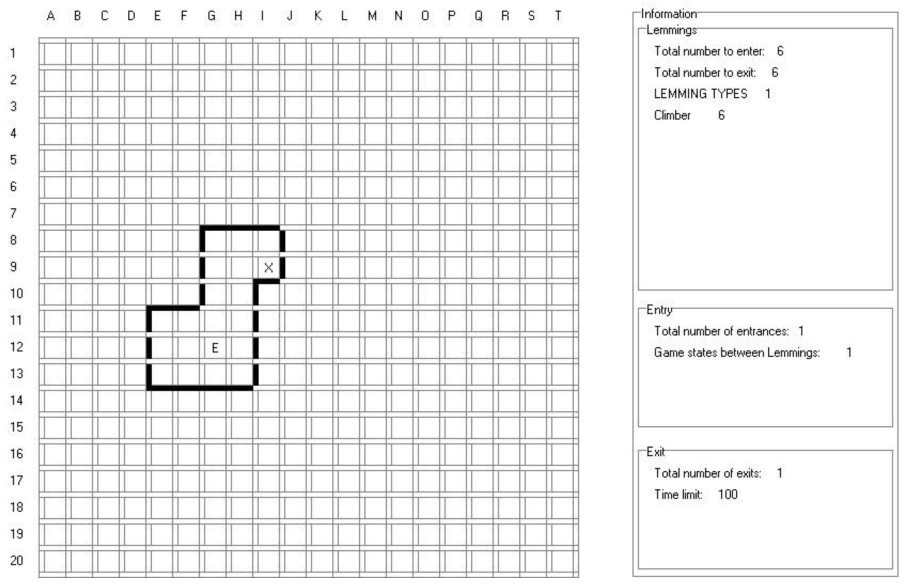


Information
Lemmings
Total number to enter: 5
Total number to exit: 5
LEMMING TYPES 1
Climber 5

Entry
Total number of entrances: 1
Game states between Lemmings: 1

Exit
Total number of exits: 1
Time limit: 100

Figure A.45: 05Climbers

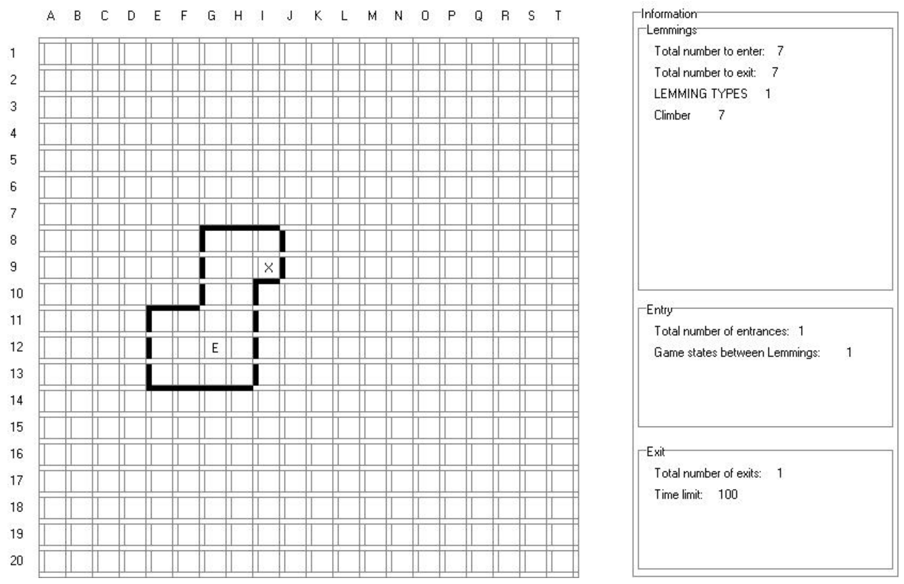


Information
Lemmings
Total number to enter: 6
Total number to exit: 6
LEMMING TYPES 1
Climber 6

Entry
Total number of entrances: 1
Game states between Lemmings: 1

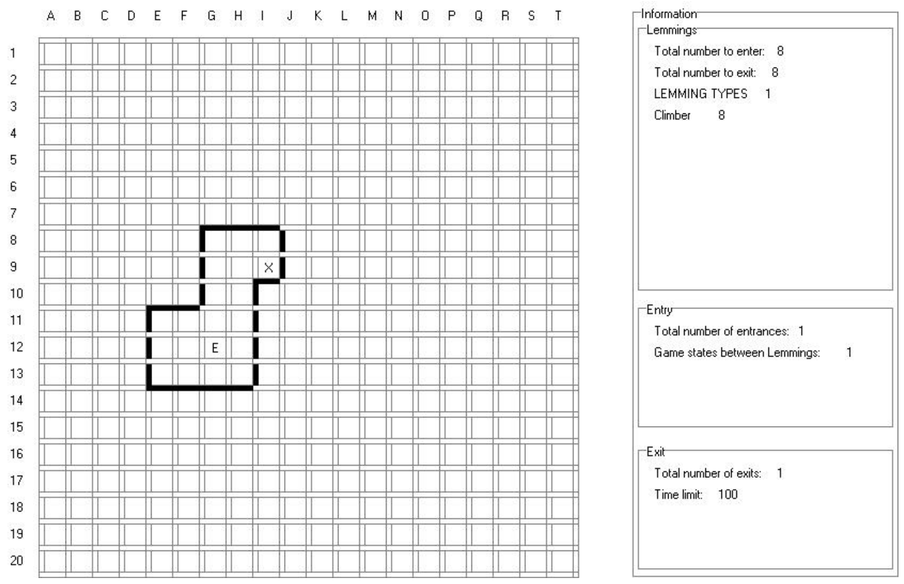
Exit
Total number of exits: 1
Time limit: 100

Figure A.46: 06Climbers



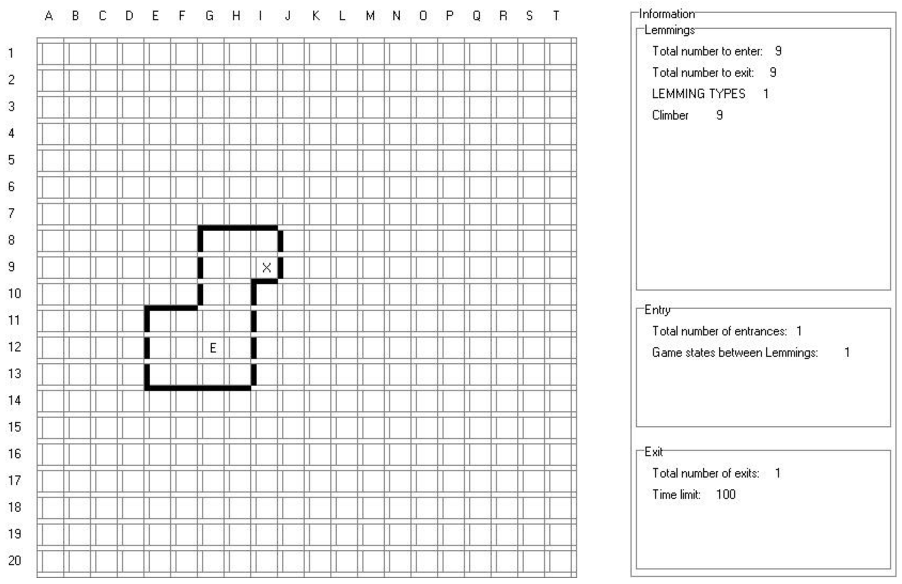
Information
Lemmings
Total number to enter: 7
Total number to exit: 7
LEMMING TYPES 1
Climber 7
Entry
Total number of entrances: 1
Game states between Lemmings: 1
Exit
Total number of exits: 1
Time limit: 100

Figure A.47: 07Climbers



Information
Lemmings
Total number to enter: 8
Total number to exit: 8
LEMMING TYPES 1
Climber 8
Entry
Total number of entrances: 1
Game states between Lemmings: 1
Exit
Total number of exits: 1
Time limit: 100

Figure A.48: 08Climbers

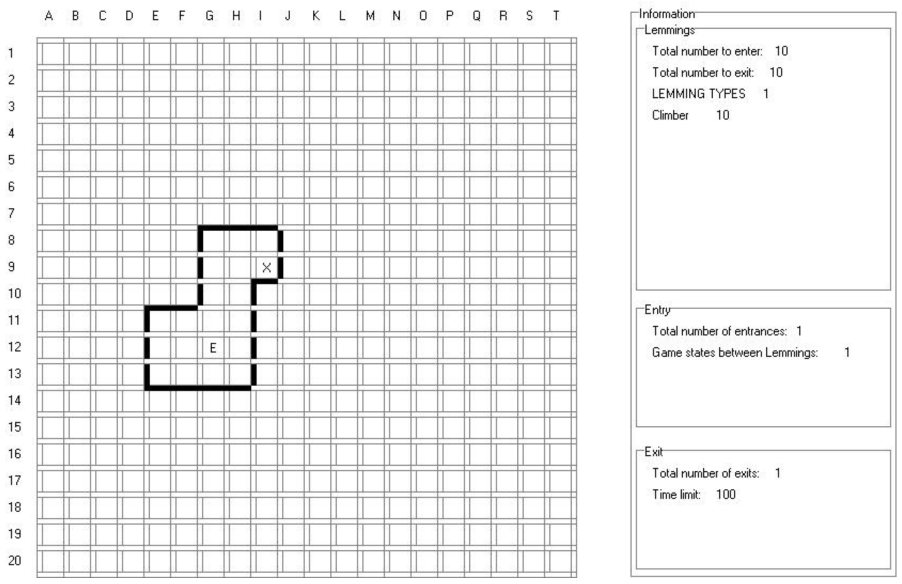


Information
Lemmings
Total number to enter: 9
Total number to exit: 9
LEMMING TYPES 1
Climber 9

Entry
Total number of entrances: 1
Game states between Lemmings: 1

Exit
Total number of exits: 1
Time limit: 100

Figure A.49: 09Climbers

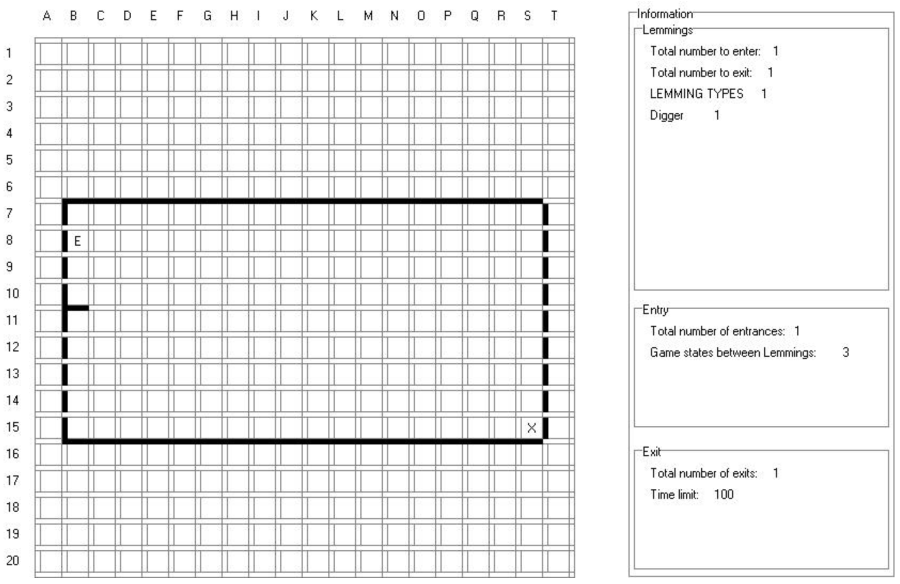


Information
Lemmings
Total number to enter: 10
Total number to exit: 10
LEMMING TYPES 1
Climber 10

Entry
Total number of entrances: 1
Game states between Lemmings: 1

Exit
Total number of exits: 1
Time limit: 100

Figure A.50: 10Climbers

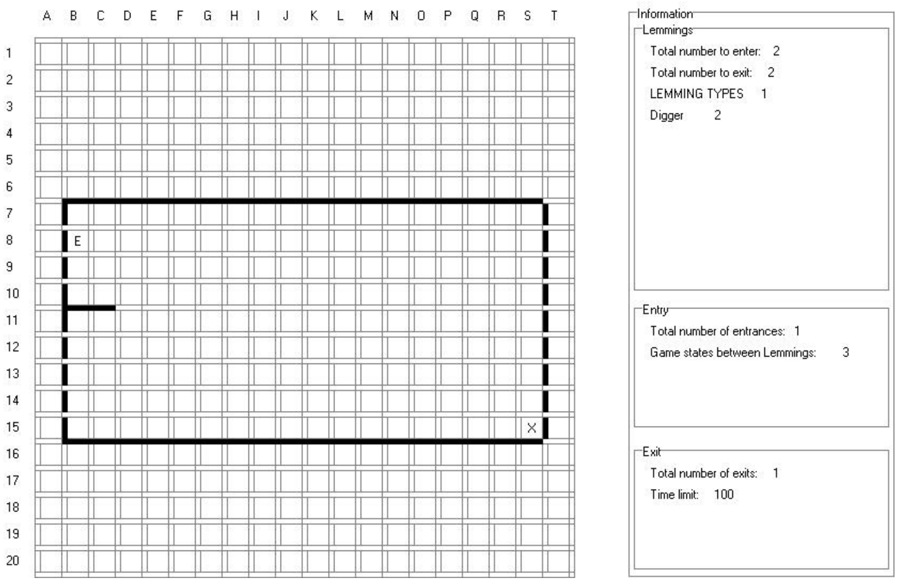


Total number to enter: 1
Total number to exit: 1
LEMMING TYPES 1
Digger 1

Total number of entrances: 1
Game states between Lemmings: 3

Total number of exits: 1
Time limit: 100

Figure A.51: 01Diggers

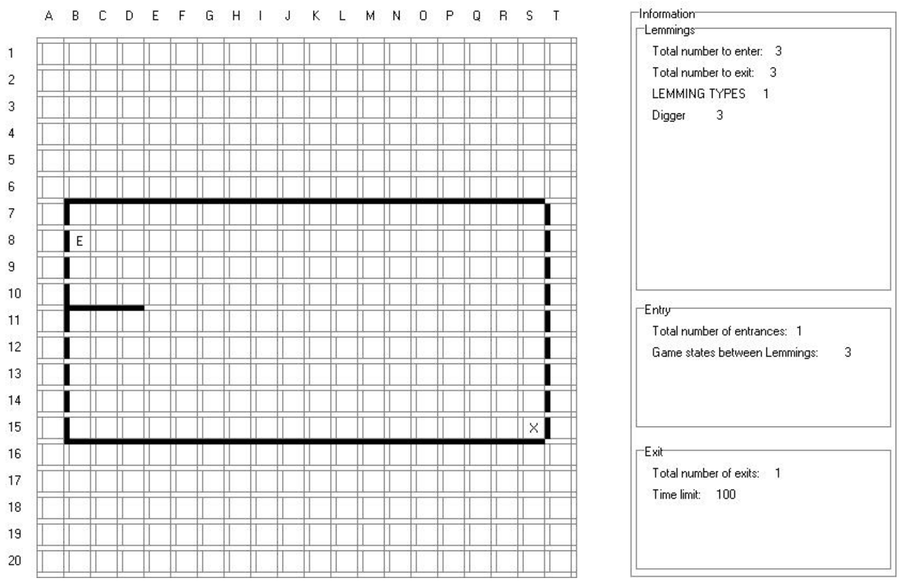


Total number to enter: 2
Total number to exit: 2
LEMMING TYPES 1
Digger 2

Total number of entrances: 1
Game states between Lemmings: 3

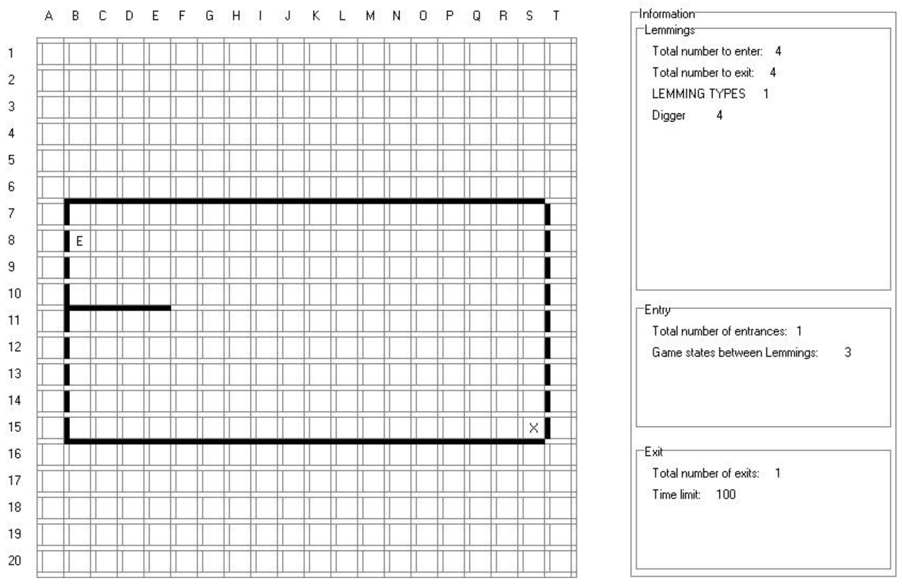
Total number of exits: 1
Time limit: 100

Figure A.52: 02Diggers



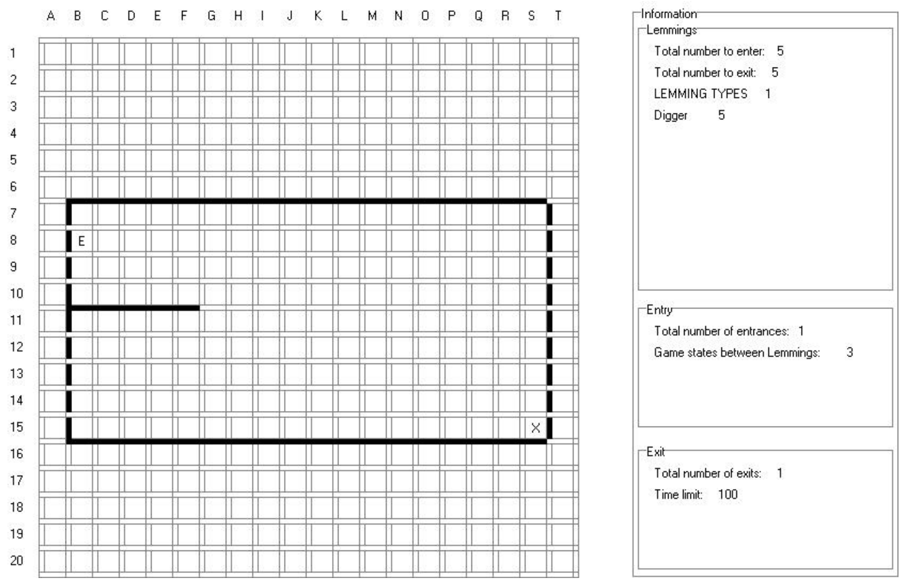
Information
Lemmings
Total number to enter: 3
Total number to exit: 3
LEMMING TYPES 1
Digger 3
Entry
Total number of entrances: 1
Game states between Lemmings: 3
Exit
Total number of exits: 1
Time limit: 100

Figure A.53: 03Diggers



Information
Lemmings
Total number to enter: 4
Total number to exit: 4
LEMMING TYPES 1
Digger 4
Entry
Total number of entrances: 1
Game states between Lemmings: 3
Exit
Total number of exits: 1
Time limit: 100

Figure A.54: 04Diggers

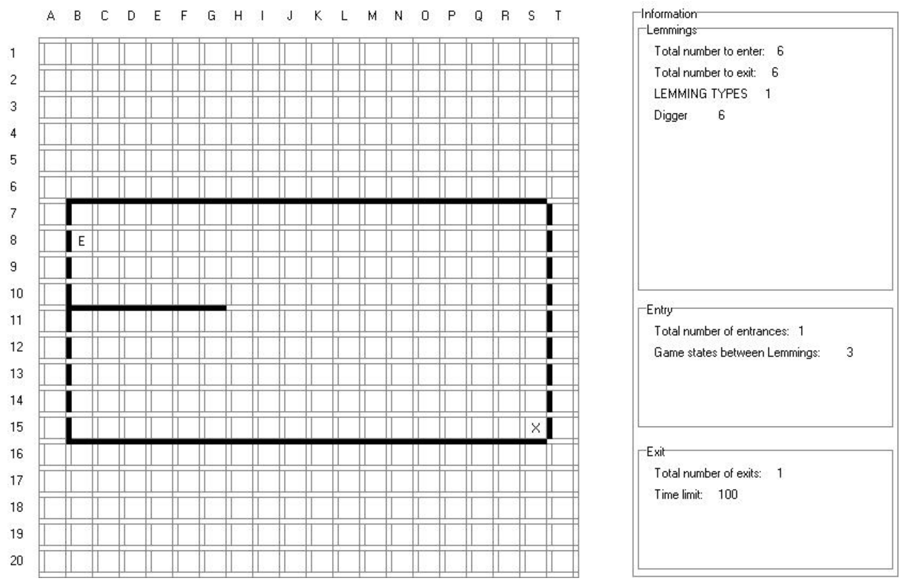


Information
Lemmings
Total number to enter: 5
Total number to exit: 5
LEMMING TYPES 1
Digger 5

Entry
Total number of entrances: 1
Game states between Lemmings: 3

Exit
Total number of exits: 1
Time limit: 100

Figure A.55: 05Diggers

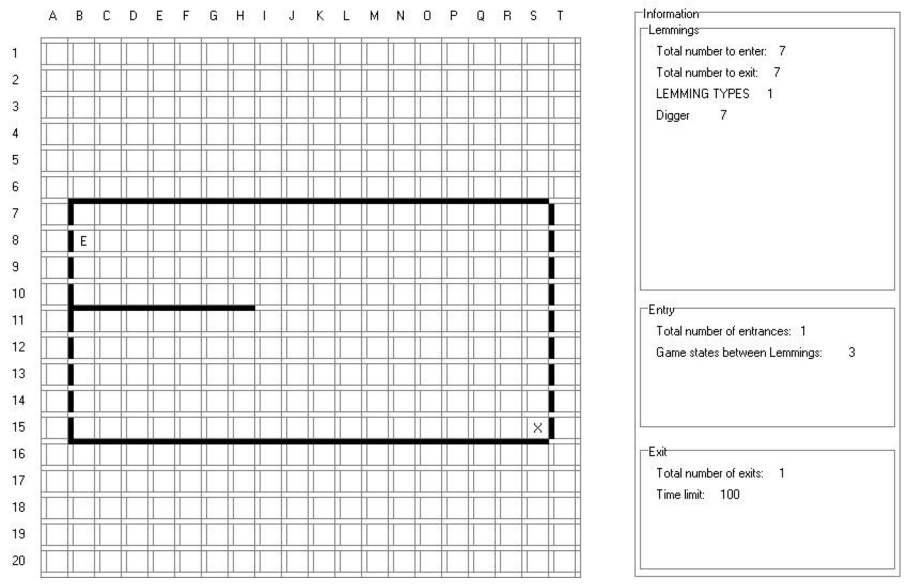


Information
Lemmings
Total number to enter: 6
Total number to exit: 6
LEMMING TYPES 1
Digger 6

Entry
Total number of entrances: 1
Game states between Lemmings: 3

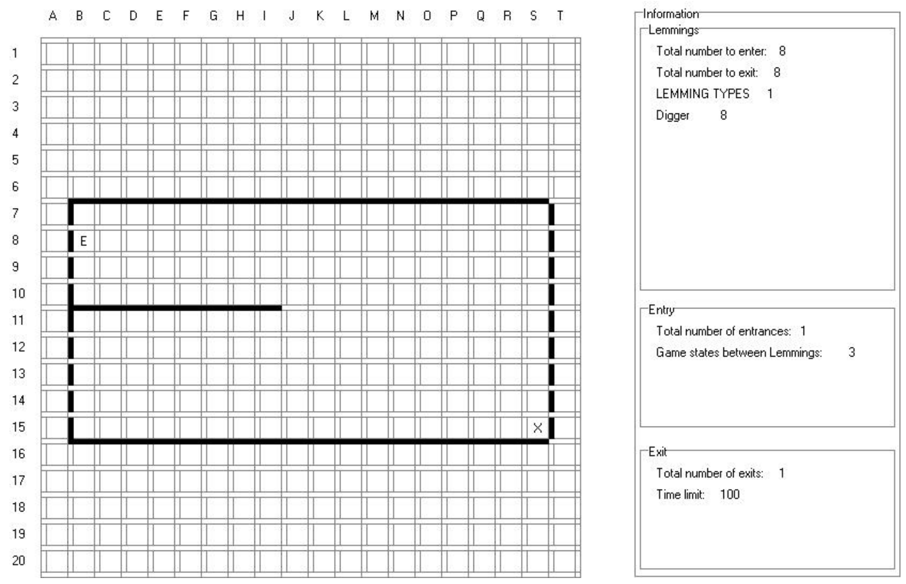
Exit
Total number of exits: 1
Time limit: 100

Figure A.56: 06Diggers



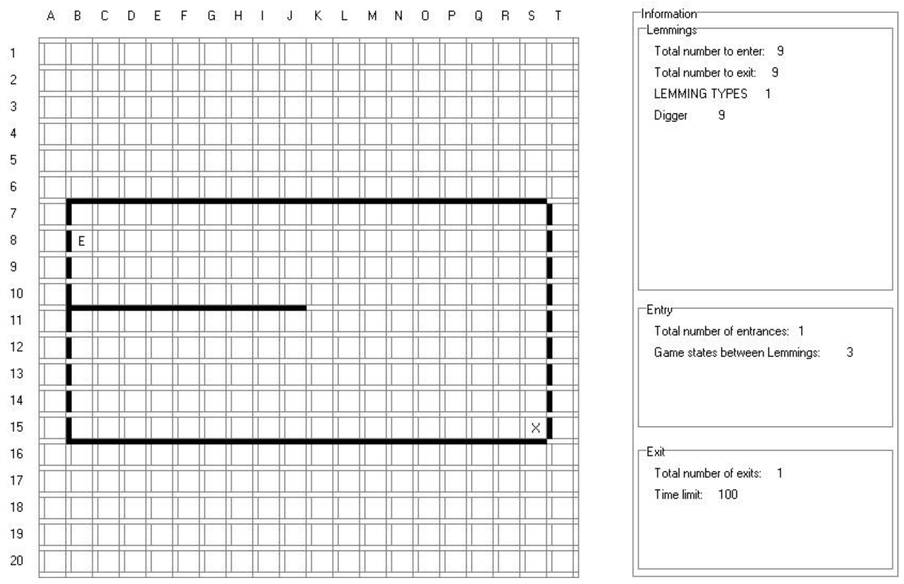
Information
Lemmings
Total number to enter: 7
Total number to exit: 7
LEMMING TYPES 1
Digger 7
Entry
Total number of entrances: 1
Game states between Lemmings: 3
Exit
Total number of exits: 1
Time limit: 100

FIGURE A.57: 07Diggers



Information
Lemmings
Total number to enter: 8
Total number to exit: 8
LEMMING TYPES 1
Digger 8
Entry
Total number of entrances: 1
Game states between Lemmings: 3
Exit
Total number of exits: 1
Time limit: 100

FIGURE A.58: 08Diggers

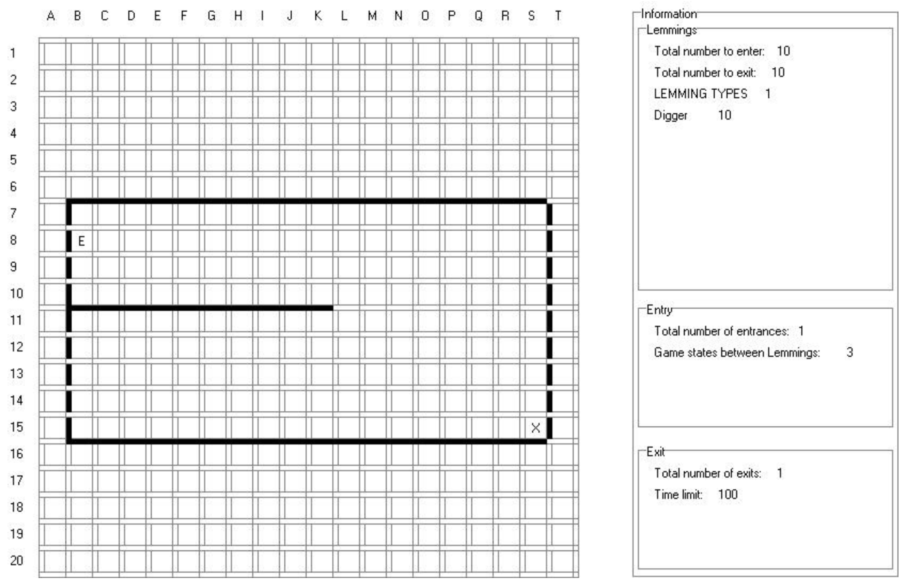


Information
Lemmings
Total number to enter: 9
Total number to exit: 9
LEMMING TYPES 1
Digger 9

Entry
Total number of entrances: 1
Game states between Lemmings: 3

Exit
Total number of exits: 1
Time limit: 100

Figure A.59: 09Diggers

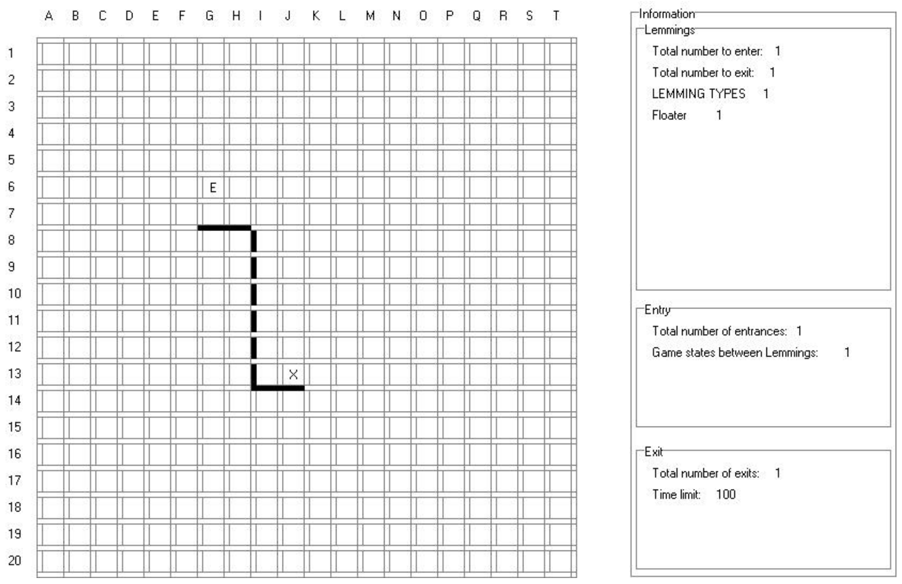


Information
Lemmings
Total number to enter: 10
Total number to exit: 10
LEMMING TYPES 1
Digger 10

Entry
Total number of entrances: 1
Game states between Lemmings: 3

Exit
Total number of exits: 1
Time limit: 100

Figure A.60: 10Diggers



Information

Lemmings

Total number to enter: 1

Total number to exit: 1

LEMMING TYPES 1

Floater 1

Entry

Total number of entrances: 1

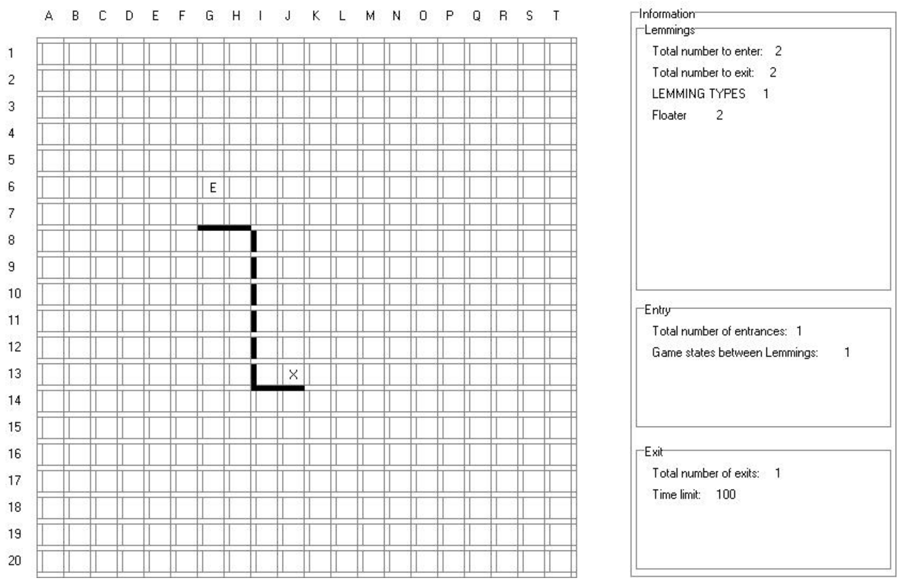
Game states between Lemmings: 1

Exit

Total number of exits: 1

Time limit: 100

Figure A.61: 01Floaters



Information

Lemmings

Total number to enter: 2

Total number to exit: 2

LEMMING TYPES 1

Floater 2

Entry

Total number of entrances: 1

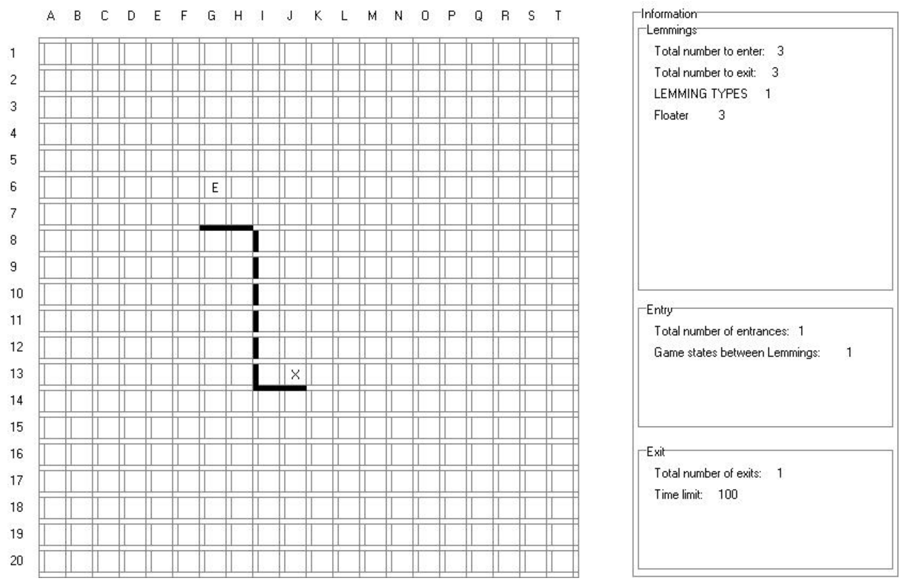
Game states between Lemmings: 1

Exit

Total number of exits: 1

Time limit: 100

Figure A.62: 02Floaters

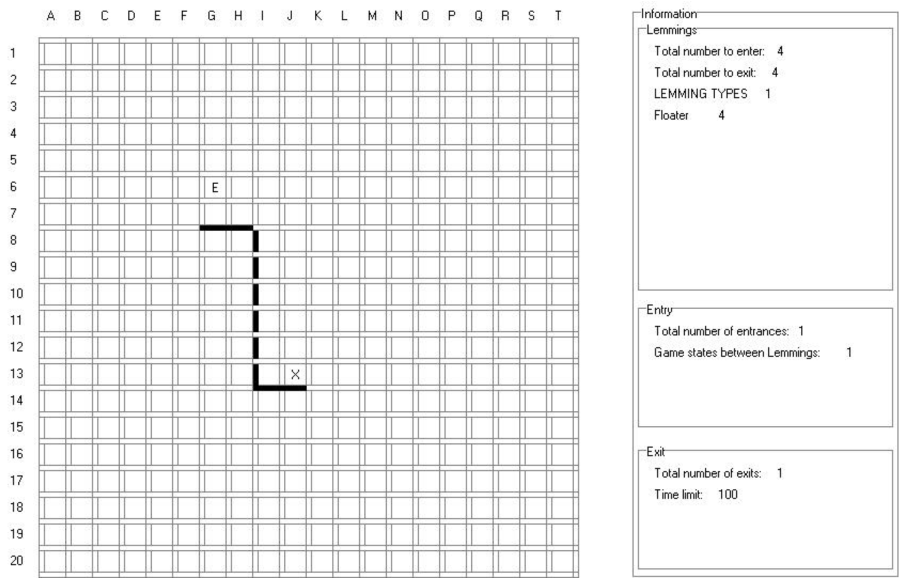


Information
Lemmings
Total number to enter: 3
Total number to exit: 3
LEMMING TYPES 1
Floater 3

Entry
Total number of entrances: 1
Game states between Lemmings: 1

Exit
Total number of exits: 1
Time limit: 100

Figure A.63: 03Floaters

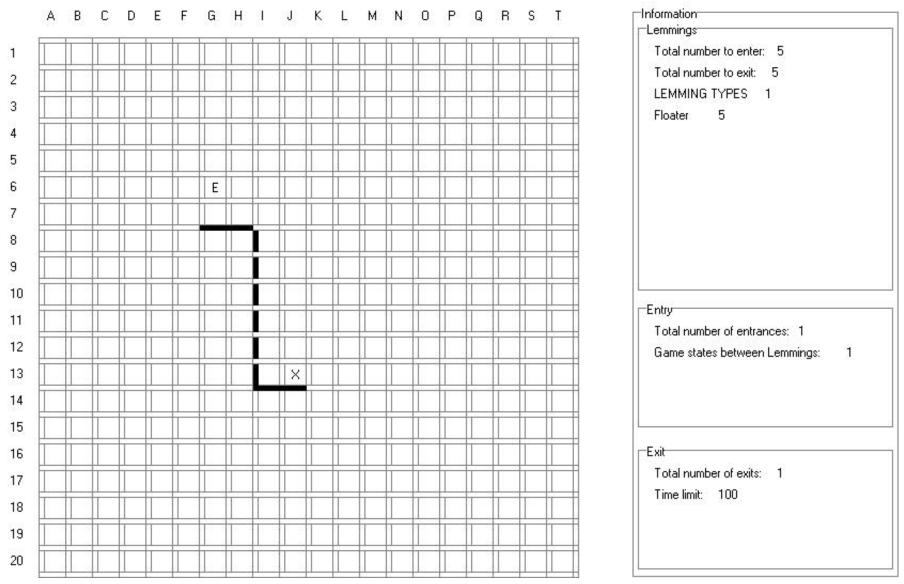


Information
Lemmings
Total number to enter: 4
Total number to exit: 4
LEMMING TYPES 1
Floater 4

Entry
Total number of entrances: 1
Game states between Lemmings: 1

Exit
Total number of exits: 1
Time limit: 100

Figure A.64: 04Floaters

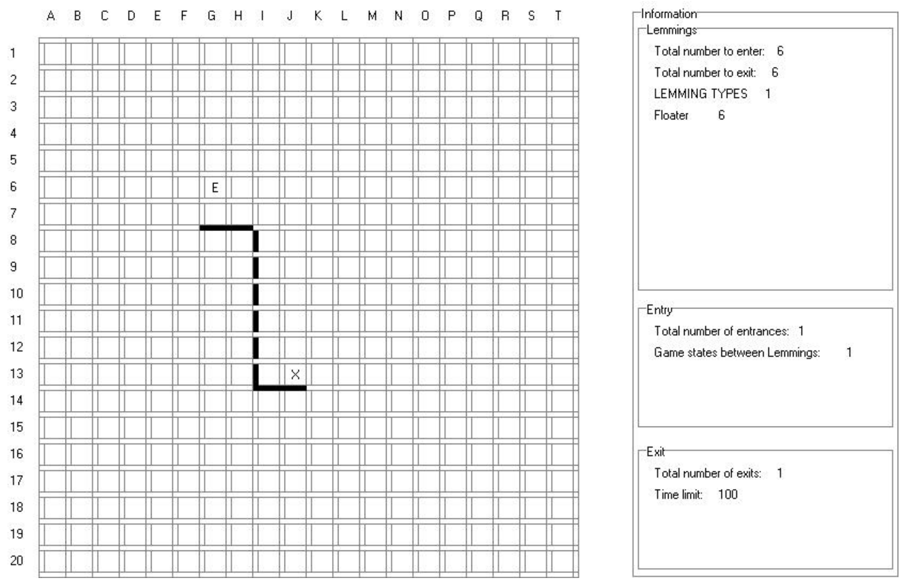


Total number to enter: 5
Total number to exit: 5
LEMMING TYPES 1
Floater 5

Total number of entrances: 1
Game states between Lemmings: 1

Total number of exits: 1
Time limit: 100

Figure A.65: 05Floaters

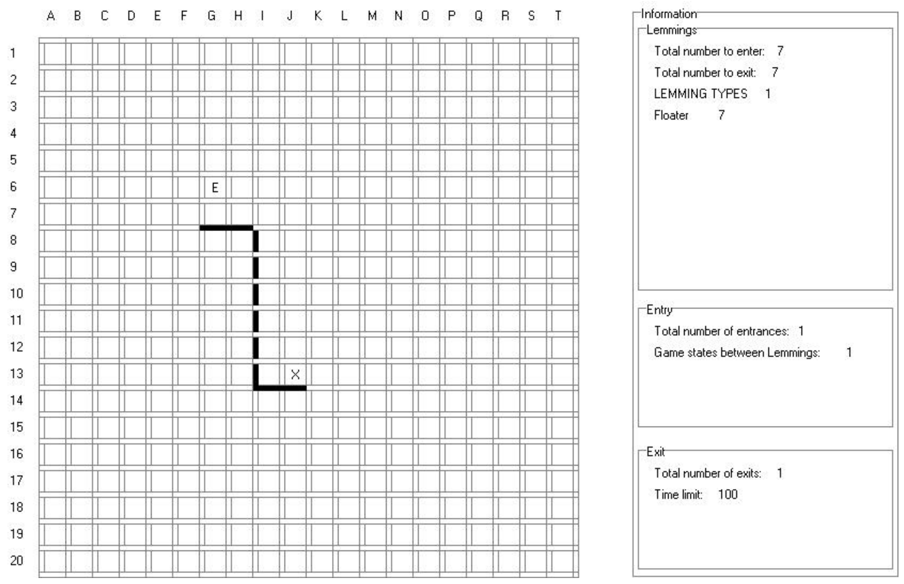


Total number to enter: 6
Total number to exit: 6
LEMMING TYPES 1
Floater 6

Total number of entrances: 1
Game states between Lemmings: 1

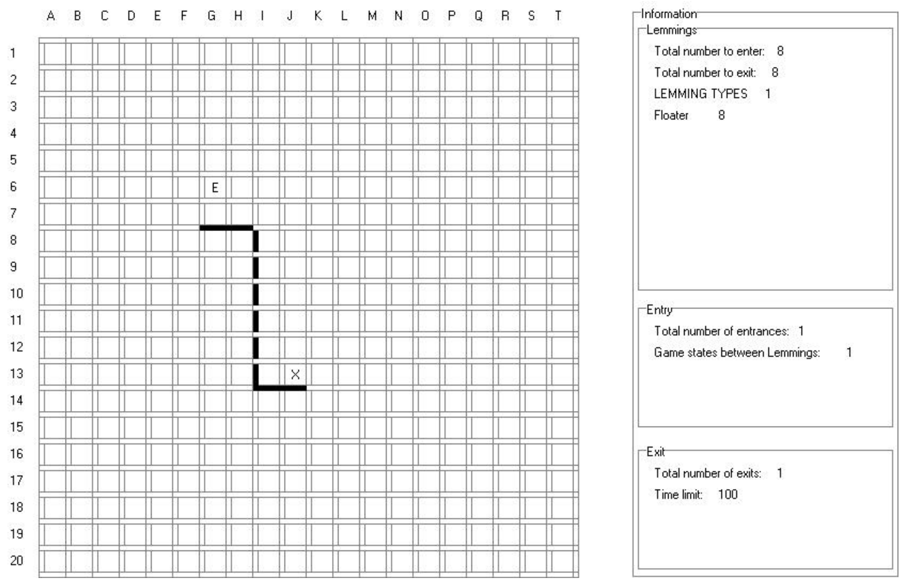
Total number of exits: 1
Time limit: 100

Figure A.66: 06Floaters



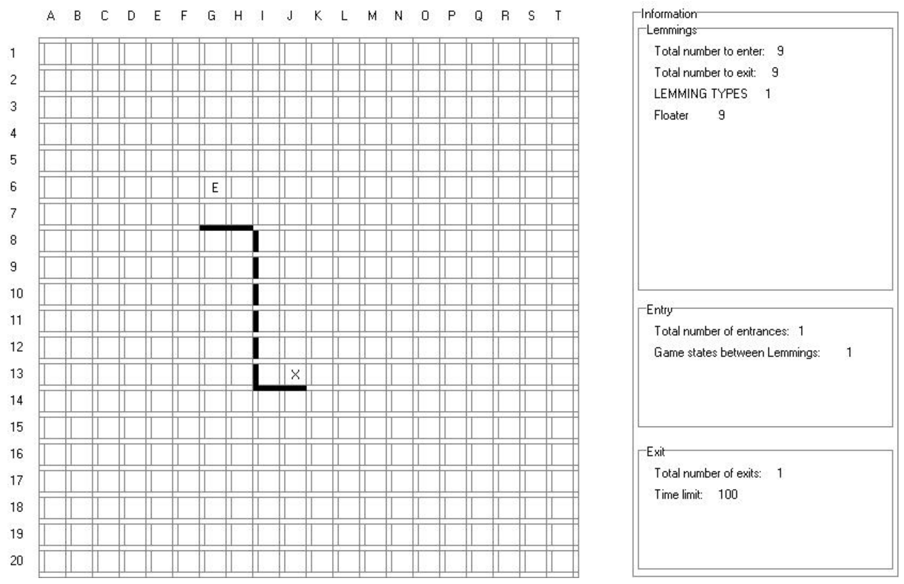
Information
Lemmings
Total number to enter: 7
Total number to exit: 7
LEMMING TYPES 1
Floater 7
Entry
Total number of entrances: 1
Game states between Lemmings: 1
Exit
Total number of exits: 1
Time limit: 100

Figure A.67: 07Floaters



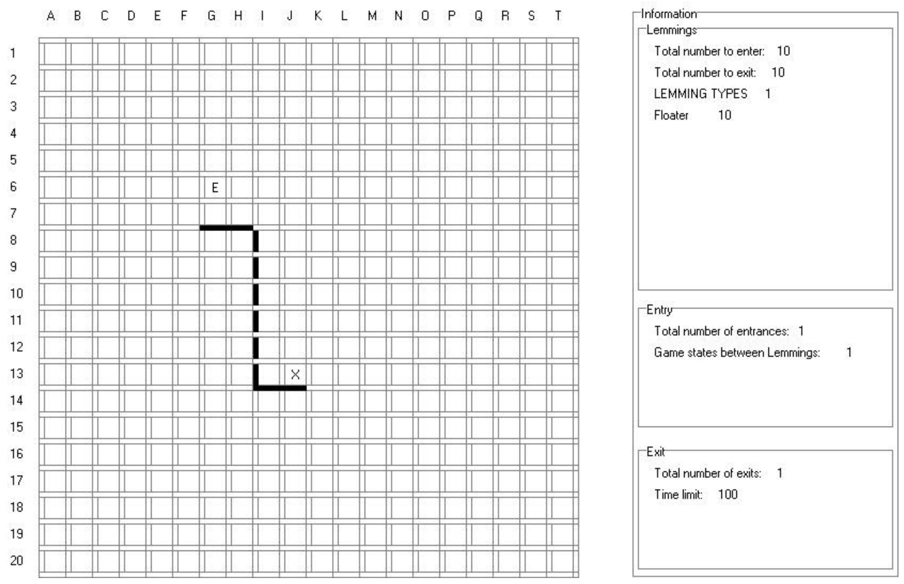
Information
Lemmings
Total number to enter: 8
Total number to exit: 8
LEMMING TYPES 1
Floater 8
Entry
Total number of entrances: 1
Game states between Lemmings: 1
Exit
Total number of exits: 1
Time limit: 100

Figure A.68: 08Floaters



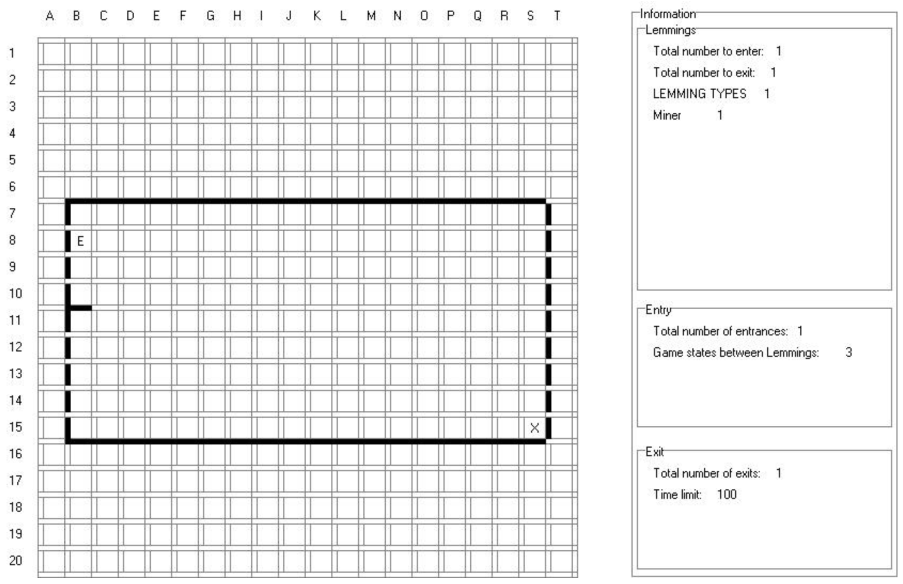
Information
Lemmings
Total number to enter: 9
Total number to exit: 9
LEMMING TYPES 1
Floater 9
Entry
Total number of entrances: 1
Game states between Lemmings: 1
Exit
Total number of exits: 1
Time limit: 100

Figure A.69: 09Floaters



Information
Lemmings
Total number to enter: 10
Total number to exit: 10
LEMMING TYPES 1
Floater 10
Entry
Total number of entrances: 1
Game states between Lemmings: 1
Exit
Total number of exits: 1
Time limit: 100

Figure A.70: 10Floaters

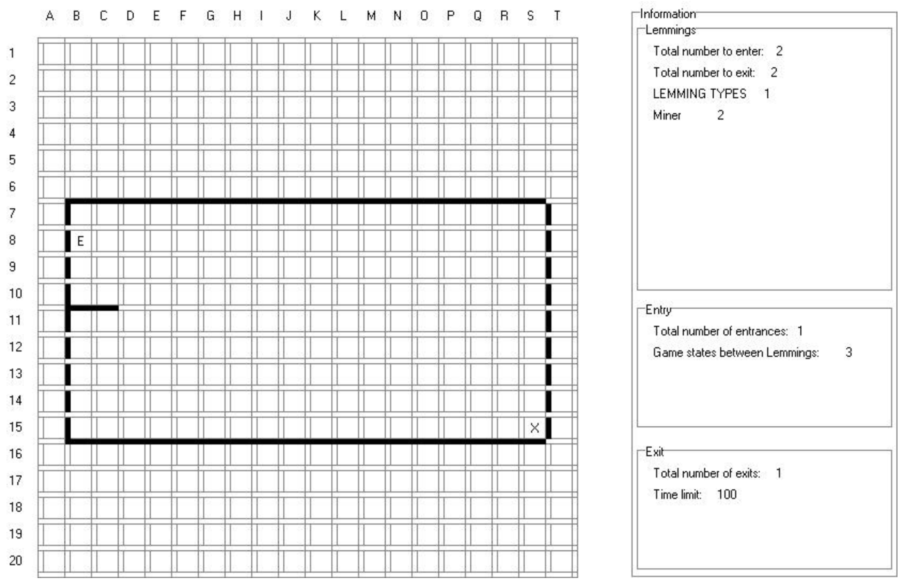


Information
Lemmings
Total number to enter: 1
Total number to exit: 1
LEMMING TYPES 1
Miner 1

Entry
Total number of entrances: 1
Game states between Lemmings: 3

Exit
Total number of exits: 1
Time limit: 100

Figure A.71: 01Miners

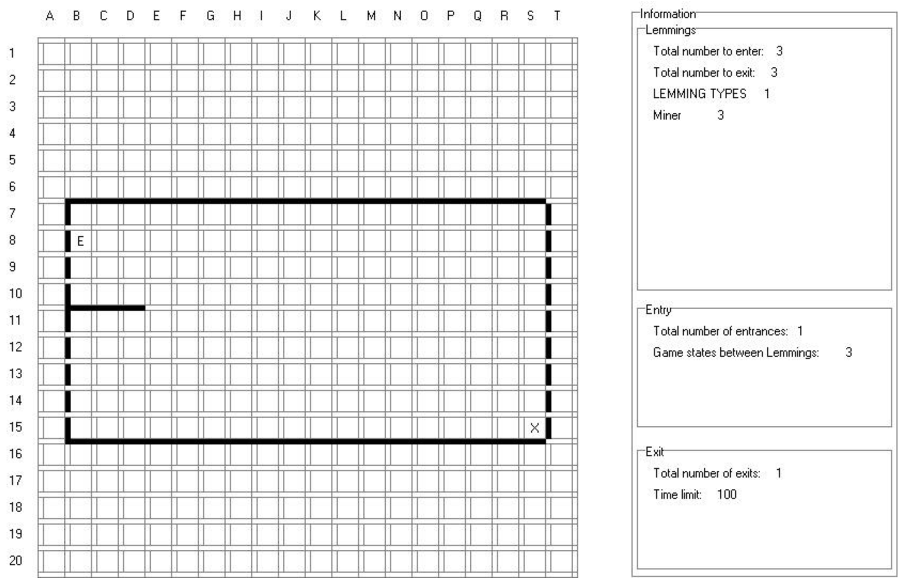


Information
Lemmings
Total number to enter: 2
Total number to exit: 2
LEMMING TYPES 1
Miner 2

Entry
Total number of entrances: 1
Game states between Lemmings: 3

Exit
Total number of exits: 1
Time limit: 100

Figure A.72: 02Miners



Information

Lemmings

Total number to enter: 3

Total number to exit: 3

LEMMING TYPES 1

Miner 3

Entry

Total number of entrances: 1

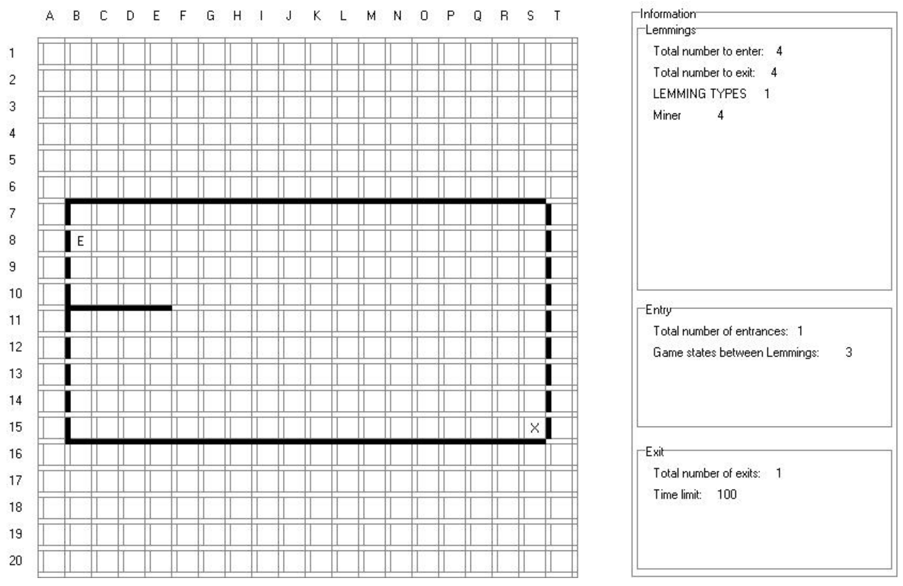
Game states between Lemmings: 3

Exit

Total number of exits: 1

Time limit: 100

FIGURE A.73: 03Miners



Information

Lemmings

Total number to enter: 4

Total number to exit: 4

LEMMING TYPES 1

Miner 4

Entry

Total number of entrances: 1

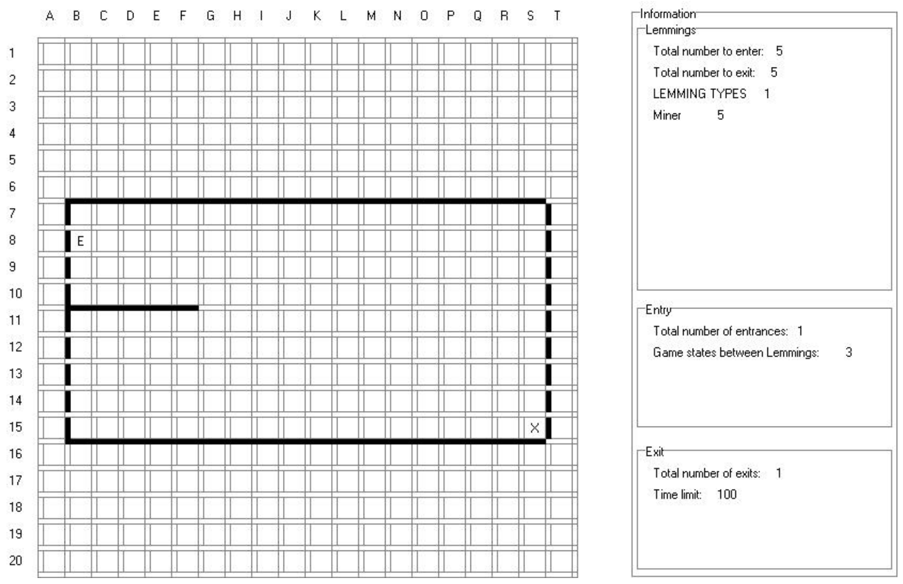
Game states between Lemmings: 3

Exit

Total number of exits: 1

Time limit: 100

FIGURE A.74: 04Miners

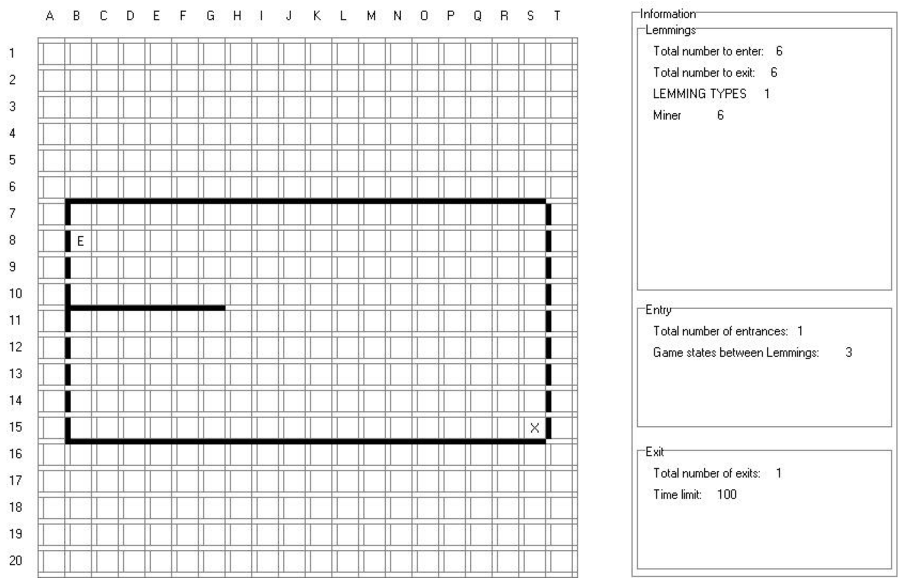


Information
Lemmings
Total number to enter: 5
Total number to exit: 5
LEMMING TYPES 1
Miner 5

Entry
Total number of entrances: 1
Game states between Lemmings: 3

Exit
Total number of exits: 1
Time limit: 100

Figure A.75: 05Miners

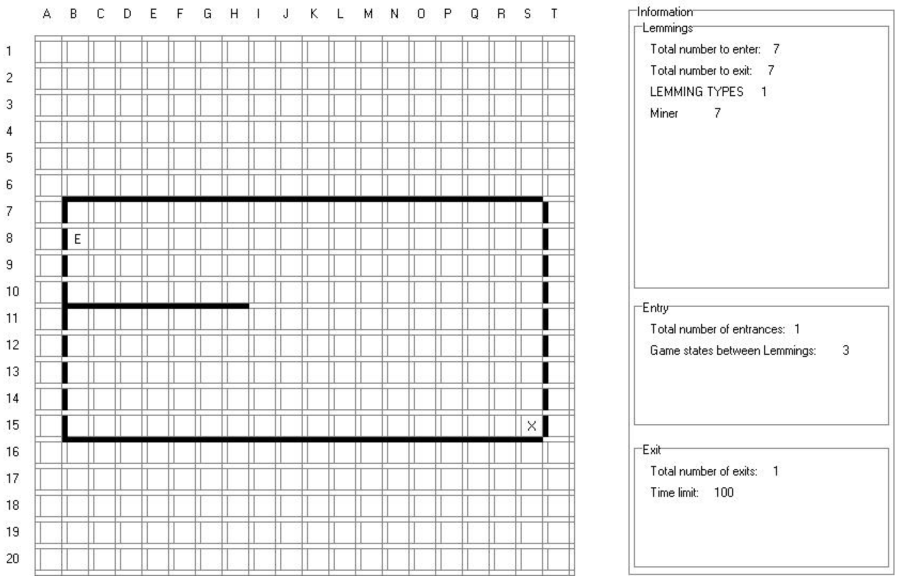


Information
Lemmings
Total number to enter: 6
Total number to exit: 6
LEMMING TYPES 1
Miner 6

Entry
Total number of entrances: 1
Game states between Lemmings: 3

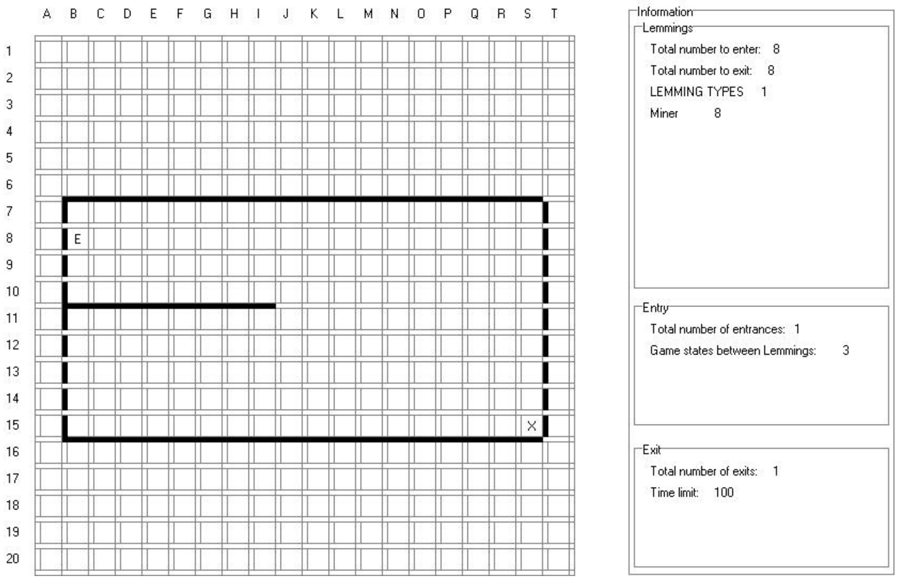
Exit
Total number of exits: 1
Time limit: 100

Figure A.76: 06Miners



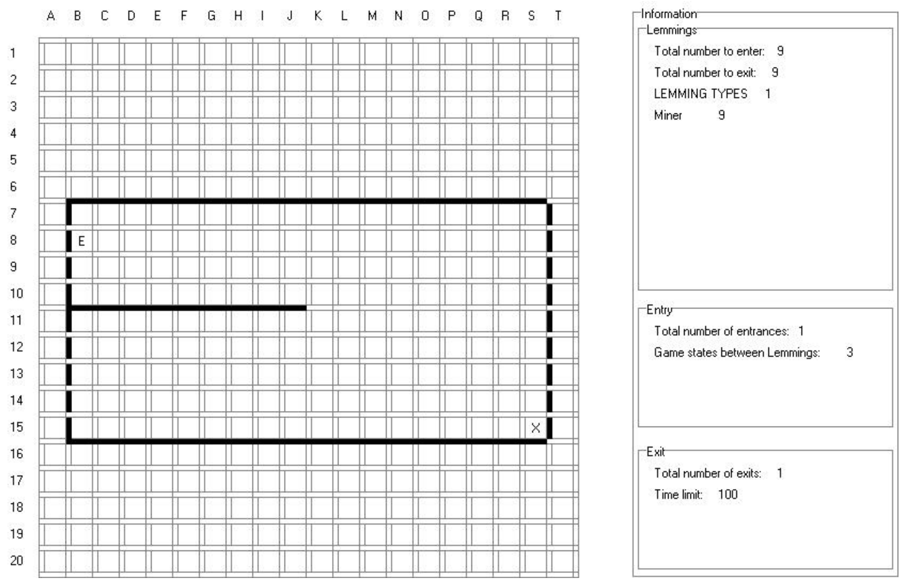
Information
Lemmings
Total number to enter: 7
Total number to exit: 7
LEMMING TYPES 1
Miner 7
Entry
Total number of entrances: 1
Game states between Lemmings: 3
Exit
Total number of exits: 1
Time limit: 100

FIGURE A.77: 07Miners



Information
Lemmings
Total number to enter: 8
Total number to exit: 8
LEMMING TYPES 1
Miner 8
Entry
Total number of entrances: 1
Game states between Lemmings: 3
Exit
Total number of exits: 1
Time limit: 100

FIGURE A.78: 08Miners

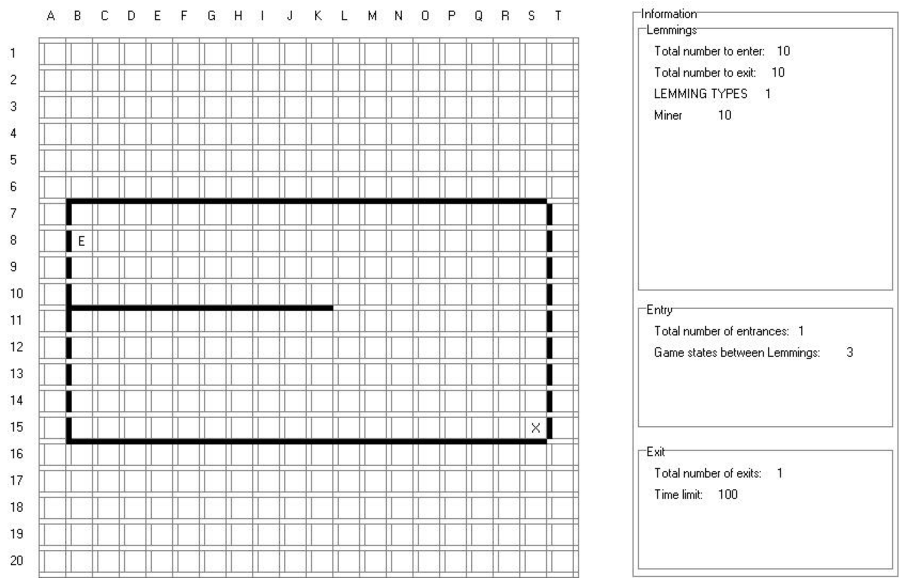


Information
Lemmings
Total number to enter: 9
Total number to exit: 9
LEMMING TYPES 1
Miner 9

Entry
Total number of entrances: 1
Game states between Lemmings: 3

Exit
Total number of exits: 1
Time limit: 100

Figure A.79: 09Miners



Information
Lemmings
Total number to enter: 10
Total number to exit: 10
LEMMING TYPES 1
Miner 10

Entry
Total number of entrances: 1
Game states between Lemmings: 3

Exit
Total number of exits: 1
Time limit: 100

Figure A.80: 10Miners

Appendix B

Publications

The following publications emerged as a result of the work completed during the writing of this thesis.

1. Kendall G and Spoerer K. Scripting the Game of Lemmings with a Genetic Algorithm. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 117-124, 2004.

2. Kendall G and Spoerer K. A Literature Survey of NP-Hard Puzzles. Submitted to *Journal of International Computer Games Association (ICGA).*

3. Kendall G, Parkes A and Spoerer K. Identification of Computationally Difficult Lemmings Instances. Submitted to *Journal of International Computer Games Association (ICGA).*

References

[1] Russel S and Norvig P. *Artificial Intelligence: A Modern Approach.* Pearson Eductation, second edition, 2003.

[2] Luger G. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* Addison Wesley, 2004.

[3] Rich E and Knight K. *Artificial Intelligence.* McGraw Hill Higher Education, 1991.

[4] Newell A, Shaw J, and Simon H. Empirical explorations with the logic theory machine: A case study in heuristics. In Feigenbaum E and Feldman J, editors, *Computers and Thought*, pages 109–133. McGraw-Hill Book Company, Inc., 1963.

[5] Cazenave T. Optimizations of datastructures, heuristics and algorithms for path-finding on maps. In *Proceedings of 2006 IEEE Symposium on Computational Intelligence and Games*, pages 27–33, 2006.

[6] Burke E, Kendall G, Landa Silva J, O'Brien R, and Soubeiga E. An ant algorithm hyperheuristic for the project presentation scheduling problem. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, pages 2263–2270, 2005.

[7] Cook S. The complexity of theorem proving procedures. In *Proceedings of 3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[8] Cheeseman P, Kanefsky B, and Taylor W. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 331–337, 1991.

[9] Shannon C. Programming a computer for playing chess. *Philosophical Magazine*, 41(4):256–275, 1950.

[10] Smith D. Dynamic programming and board games: A survey. *European Journal of Operational Research*, 176:1299–1318, 2007.

[11] Schaeffer J. The games computers (and people) play. *Advances in Computers*, 50:189–266, 2000.

[12] Buro M. Improving heuristic mini-max search by supervised learning. *Artificial Intelligence*, 134:85–99, 2002.

[13] Anshelevich V. A hierarchical approach to computer hex. *Artificial Intelligence*, 134:101–120, 2002.

[14] Iida H, Sakuta M, and Rollason J. Computer shogi. *Artificial Intelligence*, 134:121–144, 2002.

[15] Müller M. Computer go. *Artificial Intelligence*, 134:145–179, 2002.

[16] Tesauro G. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134:181–199, 2002.

[17] Billings D, Davidson A, Schaeffer J, and Szafron D. The challenge of poker. *Artificial Intelligence*, 134:201–240, 2002. Also in Schaeffer J and Jaap van den Herik H, editors, *Chips challenging champions*, pages 243-282. Elsevier Science, 2001.

[18] Sheppard B. World-championship-caliber scrabble. *Artificial Intelligence*, 134:241–275, 2002.

[19] Chellapilla K and Fogel D. Evolving neural networks to play checkers without relying on expert knowledge. *IEEE Trans. Neural Networks*, 10(6):1382–1391, 1999.

[20] Robertson E and Munro I. NP-completeness, puzzles and games. *Utilas Mathematica*, 13:99–116, 1978.

[21] McCarthy J. Partial formalizations and the lemmings game. Technical report, Stanford University, Formal Reasoning Group, 1998. http://www-formal.stanford.edu/jmc/lemmings.pdf, last accessed 22 October 2006.

[22] Cormode G. The hardness of the lemmings game, or oh no, more np-completeness proofs. In *Proceedings of Third International Conference on Fun with Algorithms*, pages 65–76, 2004.

[23] Demaine E. Playing games with algorithms: Algorithmic combinatorial game theory. Technical report, Massachusetts Institute of Technology, 2001. http://arxiv.org/abs/cs.CC/0106019, last accessed 22 October 2006.

[24] Kendall G and Spoerer K. Scripting the game of lemmings with a genetic algorithm. *Proc of the 2004 IEEE Congress on Evolutionary Computation*, pages 117–124, 2004.

[25] Garey M and Johnson D. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman and Company, 1979.

[26] Truss J. *Discrete Mathematics for Computer Scientists*. Addison Wesley, 1999.

[27] Whitely D and Watson J. Complexity theory and the no free lunch theorem. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 317–339. Springer-Verlag New York Inc, 2005.

[28] Darwin C. *The Origin of Species*. Gramercy Books, 1998. Originally published as On the Origin of Species by Means of Natural Selection, Murray J, 1859.

[29] Devlin K. *The Millenium Problems.* Granta Books, 2005.

[30] Turner J. Almost all k-colorable graphs are easy to color. *Journal of Algorithms*, 9:63–82, 1988.

[31] Karp R. Reducibility among combinatorial problems. In Miller R and Thatcher J, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, NY, 1972.

[32] Andreas Junghanns. *Pushing the Limits: New Developments in Single-Agent Search.* PhD thesis, Department of Computing Science, University of Alberta, 1999.

[33] Burke E and Kendall G, editors. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques.* Springer-Verlag New York Inc, 2005.

[34] Moore E. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292, 1959.

[35] Slate D and Atkin L. Chess 4.5 - the northwestern university chess program. In Frey P, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.

[36] Korf R. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[37] Schaeffer J, Culberson J, Treloar N, Knight B, Lu P, and Szafron D. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–290, 1992.

[38] Schaeffer J. *One Jump Ahead: Challenging Human Supremacy in Checkers.* Springer-Verlag, 1997.

[39] Langley P. Systematic and nonsystematic search strategies. In *Proceedings of 1st International Conference AI Planning Systems*, pages 145–152, 1992.

[40] Hart P, Nilsson N, and Raphael B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[41] Pohl I. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1:193–204, 1970.

[42] Felner A, Kraus S, and Korf R. KBFS: K-best-first search. *Ann. Math. Artif. Intell.*, 39(1-2):19–39, 2003.

[43] Zhou R and Hansen E. Breadth-first heuristic search. *Artificial Intelligence*, 170(4-5):385–408, 2006.

[44] Korf R, Zhang W, Thayer I, and Hohwald H. Frontier search. *Journal of the ACM*, 52(5):715–748, 2005.

[45] Glover F. Heuristics for integer programming using surrogate constraints. *Decision Sci.*, 8:156–166, 1977.

[46] Glover F. Tabu search - part i. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[47] Glover F. Tabu search - part ii. *ORSA Journal on Computing*, 2:4–32, 1990.

[48] Gendreau M and Potvin J. Tabu search. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 165–186. Springer-Verlag New York Inc, 2005.

[49] Kendall G and Mohd Hussin N. Tabu search hyperheuristic approach to the examination timetabling problem at university technology MARA. In *Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling*, pages 199–217, 2004.

[50] Kirkpatrick S, Gellat C, and Vecchi M. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[51] Aarts E, Korst J, and Michiels W. Simulated annealing. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 187–210. Springer-Verlag New York Inc, 2005.

[52] Garibaldi J and Ifeachor E. Application of simulated annealing fuzzy model tuning to umbilical cord acid-base interpretation. *IEEE Transactions on Fuzzy Systems*, 7(1):72–84, 1999.

[53] Mladenovic N and Hansen P. Variable neighbourhood search. *Comput. Oper. Res.*, 24:1097–1100, 1997.

[54] Hansen P and Mladenovic N. Variable neighbourhood search. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 211–238. Springer-Verlag New York Inc, 2005.

[55] Abdullah S, Burke E, and McCollum B. An investigation of variable neighbourhood search for university course timetabling. In *Proceedings of the 2nd Multidisciplinary Conference on Scheduling: Theory and Applications (MISTA'05)*, pages 413–427, 2005.

[56] Goldberg D. *Genetic Algorithms in search, Optimization, and Machine Learning.* Addison Wesley, 1989.

[57] Sastry K, Goldberg D, and Kendall G. Genetic algorithms. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 97–125. Springer-Verlag New York Inc, 2005.

[58] Fraser A. Simulation of genetic systems by automatic digital computers. ii: Effects of linkage on rates under selection. *Australian Journal of Biological Sciences*, 10:492–499, 1957.

[59] Holland J. *Adaptation in Natural and Artificial Systems.* MIT Press, 1975.

[60] Chaperot B and Fyfe C. Improving artificial intelligence in a motocross game. In *Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG'06)*, pages 181–186, 2006.

[61] Aickelin U and Dasgupta D. Artificial immune systems. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 375–399. Springer-Verlag New York Inc, 2005.

[62] Cayzer S and Aickelin U. A recommender system based on idiotypic artificial immune networks. *Journal of Mathematical Modelling and Algorithms*, 4(2):181–198, 2005.

[63] Merkle D and Middendorf M. Swarm intelligence. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 401–435. Springer-Verlag New York Inc, 2005.

[64] Dorigo M. *Optimization, learning and natural algorithms.* PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992.

[65] Dorigo M, Maniezzo V, and Colorni A. Positive feedback as a search strategy. Technical Report 91-016, Politecnico di Milano, 1991. `http://iridia.ulb.ac.be/~mdorigo/pub_x_subj.html`, last accessed 22 October 2006.

[66] Kennedy J and Eberhart R. Particle swarm optimization. In *Proc. IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[67] Su Y and Kendall G. A particle swarm optimisation approach in the construction of optimal risky portfolios. In *Proceedings of the 23rd IASTED International Multi-conference Artificial Intelligence and Applications*, pages 140–145, 2005.

[68] Koza J. *Genetic Programming: On the programming of computers by means of Natural Selection*, chapter Four introductory examples of Genetic Programming, pages 147–162. The MIT Press, Cambridge, Massachusetts, 1992.

[69] Koza J and Poli R. Genetic programming. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 127–164. Springer-Verlag New York Inc, 2005.

[70] Yao X and Liu Y. Machine learning. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 341–373. Springer-Verlag New York Inc, 2005.

[71] McCulloch W and Pitts W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–137, 1943.

[72] Ross P. Hyper-heuristics. In Burke E and Kendall G, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 529–556. Springer-Verlag New York Inc, 2005.

[73] Burke E, Kendall G, and Soubeiga E. A tabu search hyperheuristic for timetabling and rostering. *Journal of Heuristics*, 9:451–470, 2003.

[74] Glover F and Kochenberger G, editors. *Handbook of Meta-Heuristics*. Kluwer, 2003.

[75] Wolpert D and Macready W. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995. http://www.santafe.edu/sfi/publications/Working-Papers/95-02-010.ps, last accessed 22 October 2006.

[76] Gupta N and Nau D. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254, 1992.

[77] Slaney J and Thiebaux S. Blocks world revisited. *Artificial Intelligence*, 125:119–153, 2001.

[78] Biedl T, Demaine E, Demaine M, Fleischer R, Jacobsen L, and Munro J. The complexity of clickomania. In Nowakowski R, editor, *More Games of No Chance*. Cambridge University Press, 2002.

[79] Friedman E. Corral puzzles are np-complete. http://www.stetson.edu/ efriedma/papers/corral.pdf, last accessed 22 October 2006.

[80] Takahiro S. The complexities of puzzles, cross sum and their another solution problems (asp), 2001. Thesis for BSc, Department of Information Science, University of Tokyo.

[81] Eppstein D. On the NP-completeness of cryptarithms. *SIGACT News*, 18(3):38–40, 1987.

[82] Friedman E. Cubic is np-complete. Available at http://www.stetson.edu/˜efriedma/papers/cubic.pdf, last accessed 22 October 2006.

[83] Webster F. Instant insanity. Available at http://www.csulb.edu/˜fnewberg/PCTMSummary/FinalPDFs/francine.PDF, last accessed 22 October 2006.

[84] Walsh R and Julstrom B. Generalized instant insanity: A GA-difficult problem. In *Late Breaking Papers at the Genetic Programming 1998 Conference*, 1998.

[85] Král' D, Majerech V, Sgall J, Tichý T, and Woeginger G. It is tough to be a plumber. *TCS: Theoretical Computer Science*, 313(3):473–484, 2004.

[86] McPhail B. Light up is np-complete. Available from http://www.reed.edu/˜mcphailb/lightup.pdf, last accessed 22 October 2006.

[87] Kaye R. Minesweeper is np-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

[88] McPhail B. The complexity of puzzles: Np-completeness results for nurikabe and minesweeper, 2003. Thesis for BA, Division of Mathematics and Natural Sciences, Reed College.

[89] Castillo L and Wrobel S. Learning minesweeper with multirelational learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 533–538, 2003.

[90] Adamatzky A. How cellular automaton plays Minesweeper. *j-APPL-MATH-COMP*, 85(2-3):127–137, 1997.

[91] Quartetti C. Evolving a program to play the game minesweeper. In *Genetic Algorithms and Genetic Programming at Stanford 1998*, pages 137–146. 1998.

[92] Rhee S. Evolving strategies for the minesweeper game using genetic programming. In *Genetic Algorithms and Genetic Programming at Stanford 2000*, pages 312–318. 2000.

[93] Ratner D and Warmuth M. Finding a shortest solution for the N $\times$ N extension of the 15-PUZZLE is intractable. *J. Symbolic Computation*, 10:111–137, 1990.

[94] Reinefeld A. Complete solution of the eight-puzzle and the benefit of node ordering in IDA. In *International Joint Conference on Artificial Intelligence*, pages 248–253, 1993.

[95] Culberson J and Schaeffer J. Searching with pattern databases. In *Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, pages 402–416, 1996.

[96] Korf R and Taylor L. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1202–1207, 1996.

[97] Parberry I. A real-time algorithm for the (n^2-1)-puzzle. *Information Processing Letters*, 56(1):23–28, 1995.

[98] Taylor L and Korf R. Pruning duplicate nodes in depth-first search. In *National Conference on Artificial Intelligence*, pages 756–761, 1993.

[99] Friedman E. Pearl puzzles are np-complete. Available from http://www.stetson.edu/~efriedma/papers/pearl.pdf, last accessed 22 October 2006.

[100] Uehara R and Iwata S. Generalized hi-q is np-complete. *Trans IEICE*, 73:270–273, 1990.

[101] Moore C and Eppstein D. One-dimensional peg solitaire, and duotaire. In Nowakowski R, editor, *More Games of No Chance*, number 42, pages 341–350. Cambridge Univ. Press, 2002.

[102] Ravikumar B. Peg-solitaire, string rewriting systems and finite automata. *Theoretical Computer Science*, 321(2-3):383–394, 2004.

[103] Kiyomi M and Matsui T. Integer programming based algorithms for peg solitaire problems. *Lecture Notes in Computer Science*, 2063:229–240, 2000.

[104] Matos A. Depth-first search solves peg solitaire. Technical Report DCC-98-10, Universidade do Porto, 1998. Available from http://www.dcc.fc.up.pt/Pubs/treports.html, last accessed 22 October 2006.

[105] Jefferson C, Miguel A, Miguel I, and Tarim A. Modelling and solving english peg solitaire. *Computers and Operations Research*, 33(10):2935–2959, 2006.

[106] Kempe D. On the complexity of the reflections game, 2003. Available from http://www-rcf.usc.edu/~dkempe/publications/reflections.pdf, last accessed 22 October 2006.

[107] Yato T. On the NP-Completeness of the slither link puzzle. *IPSJ SIGNotes ALgorithms*, 74:25–32, 2000.

[108] Dor D and Zwick U. SOKOBAN and other motion planning problems. *CGTA: Computational Geometry: Theory and Applications*, 13(4):215–228, 1999.

[109] Junghanns A and Schaeffer J. Single-agent search in the presence of deadlocks. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 419–424, 1998.

[110] Junghanns A and Schaeffer J. Relevance cuts: Localizing the search. In *Computers and Games: Proceedings CG'98. LNCS 1558*, pages 1–14. 1999.

[111] Junghanns A and Schaeffer J. Sokoban: improving the search with relevance cuts. *Theoretical Computer Science*, 252(1-2):151–175, 2001.

[112] Junghanns A and Schaeffer J. Domain-dependent single-agent search enhancements. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 570–575, 1999.

[113] Junghanns A and Schaeffer J. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1–2):218–251, 2001.

[114] Junghanns A and Schaeffer J. Sokoban: A challenging single-agent search problem. In *Proceedings of the Workshop on Computer Games (W31) at IJCAI-97*, pages 27–36, 1997.

[115] Murase Y, Matsubara H, and Hiraga Y. Automatic making of sokoban problems. In *Pacific Rim International Conference on Artificial Intelligence*, pages 592–600, 1996.

[116] O'Rourke J and The Smith Problem Solving Group. Pushpush is NP-hard in 3D, 1999. Available from http://arxiv.org/abs/cs/9911013, last accessed 22 October 2006.

[117] Friedman E. Spiral galaxies puzzles are np-complete. Available from http://www.stetson.edu/~efriedma/papers/spiral.pdf, last accessed 22 October 2006.

[118] Yato T and Seta T. Complexity and completeness of finding another solution and its application to puzzles. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 2003.

[119] Pereira M, Dutra I, and Castro M. Arc-consistency algorithms on a software dsm platform, 2001. Colloquium on Implementation of Cronstraint and Logic Programming Systems - CICLOPS 2001.

[120] Nicolau M and Ryan C. Solving sudoku with the GAuGE system. *Lecture Notes in Computer Science*, 3905:213–224, 2006.

[121] Eppstein D. Nonrepetitive paths and cycles in graphs with application to sudoku, 2005. Available from `http://arxiv.org/PS_cache/cs/pdf/0507/0507053.pdf`, last accessed 22 October 2006.

[122] Lynas A and Stoddart B. Sudoku solver case study: from specification to RVM-forth (part I). In *Proceedings of the 21st EuroForth Conference*, pages 21–34, 2005.

[123] McCarthy J, Minsky M, Rochester N, and Shannon C. Proposal for dartmouth summer research project on artificial intelligence. Technical report, Dartmouth College, 1955. Available at http://www-formal.stanford.edu/jmc/history/dartmouth.html, last accessed 22 October 2006.

[124] Campbell M, Joseph Hoane Jr A, and Hsu F-h. Deep blue. *Artificial Intelligence*, 134:57–83, 2002. Also in Schaeffer J and Jaap van den Herik H, editors, *Chips challenging champions*, pages 97-123. Elsevier Science, 2001.

[125] Fogel D, Hays T, Hahn S, and Quon J. The blondie25 chess program competes against fritz 8.0 and a human chess master. In Louis S and Kendall G, editors, *Proceedings of 2006 IEEE Symposium on Computational Intelligence and Games*, pages 230–235, 2006.

[126] Sadikov A and Bratko I. Learning long-term chess strategies from databases. *Machine Learning*, 63(3):329–340, 2006.

[127] Gallagher M and Ryan A. Learning to play pac-man: An evolutionary, rule-based approach. *Proc of the 2003 IEEE Congress on Evolutionary Computation*, pages 2462–2469, 2003.

[128] Yannakakis G and Hallam J. A generic approach for generating interesting interactive pac-man opponents. In *Proceedings of IEEE 2005 Symposium on Computational Intelligence and Games*, pages 94–101, 2005.

[129] Lucas S. Evolving a neural network location evaluator to play ms. pac-man. In *Proceedings of IEEE 2005 Symposium on Computational Intelligence and Games*, pages 203–210, 2005.

[130] Demaine E, Hohenberger S, and Liben-Nowell D. Tetris is hard, even to approximate. In *Proceedings of 9th Annual International Conference Computing and Combinatorics*, pages 351–363, 2003.

[131] Siegel E and Chaffee A. Genetically Optimizing the Speed of Programs evolved to Play Tetris. In *Advances in Genetic Programming 2*, pages 279–298. 1996.

[132] Yurovitsky M. Playing tetris using genetic programming. In *Genetic Algorithms and Genetic Programming at Stanford*, pages 309–319. 1995.

[133] Germundsson R. A tetris controller – an example of a discrete event dynamic system. Available from citeseer.ist.psu.edu/germundsson91tetri.html, last accessed 22 October 2006.

[134] Beasley D, Bull D, and Martin R. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, 15(2):58–69, 1993.

[135] Leung Y, Gao Y, and Xu Z. Degree of population diversity - a perspective on premature convergence in genetic algorithms and its markov chain analysis. *IEEE Transactions on Neural Networks*, 8(5):1165–1171, 1997.

[136] Jones T and Forrest S. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 184–192, 1995.

[137] Mitchell M, Holland J, and Forrest S. When will a genetic algorithm outperform hill climbing. *Advances in Neural Information Processing Systems*, 6:51–58, 1994.