On the Relations Between Search and Evolutionary Algorithms

Riccardo Poli and Brian Logan

School of Computer Science
The University of Birmingham
Birmingham B15 2TT
United Kingdom

E-mail: {R.Poli,B.S.Logan}@cs.bham.ac.uk

Technical Report: CSRP-96-7
March 1996

Abstract

Evolutionary algorithms are powerful techniques for optimisation whose operation principles are inspired by natural selection and genetics. In this paper we discuss the relation between evolutionary techniques, numerical and classical search methods and we show that all these methods are instances of a single more general search strategy, which we call the 'evolutionary computation cookbook'. By combining the features of classical and evolutionary methods in different ways new instances of this general strategy can be generated, i.e. new evolutionary (or classical) algorithms can be designed. One such algorithm, GA^*, is described.

1 Introduction

Evolutionary algorithms (EAs) are powerful optimisation techniques inspired by genetics and natural selection [6, 5, 2, 1, 9]. EAs are often referred to as global optimisation methods, to stress the fact that they can effectively explore very large solutions spaces without becoming trapped by local minima. Applying EAs to a new problem is straightforward, the only requirements being: a way of representing the solutions of the problem to be solved, a set of genetic operators, and a fitness function (or at least a partial ordering over the solutions). The effectiveness and simplicity of evolutionary algorithms have lead many people to argue that they are the methods of choice for hard real-life problems, superseding traditional search techniques.

However they are not without their limitations. In particular, the choice of a good problem-specific representation and a good set of problem-specific genetic operators can

make a considerable difference to the efficiency, effectiveness and often even the feasibility of the search [4, 8, 9]. Similar problems arise in classical GOFAI (Good Old Fashioned Artificial Intelligence) state space search, where the selection of appropriate problem representations and operators has long been one of the central concerns. This suggests that it is important to explore the parallels between classical and evolutionary approaches.

In this paper we discuss the relations between evolutionary techniques, numerical methods and classical search methods and argue that the idea that EAs make other techniques obsolete is largely incorrect. We show that EAs are nothing but informed search methods with certain new features which *can* make them superior to other methods for certain problems (in particular the features which make them global search techniques) and lack some important characteristics that make many other classical and numerical search algorithms very effective. A comparative analysis of fourteen evolutionary and non-evolutionary search algorithms shows that they are all constructed from a small number of components which as a whole form what we call 'evolutionary computation cookbook'. These components can be recombined to form new powerful 'recipes', i.e. new search algorithms, which combine the features of classical and evolutionary methods.

The paper is organised as follows. Firstly, in Section 2, we describe the structure of classical search methods, numerical methods and some classes of EAs. Then, in Section 3 we present the 'cookbook' of which all these methods are 'recipes'. In Section 4 we describe a new recipe, the GA^* algorithm, and we report on some experiments in which the algorithm has been used for route planning in complex terrains. We draw some final conclusions in Section 5.

2 Evolutionary and non-evolutionary search algorithms

2.1 Classical search algorithms

Search algorithms are fundamental problem solving methods in artificial intelligence [12, 3, 11]. In order to solve a problem it is necessary: a) to represent its solutions (these are seen as points in what is called the search space for the problem) and b) to design some search operators which given one solution generate new candidate solutions. Points in the search space are also called nodes and the application of the operators to a node is called expansion. Newly generated nodes are called children.

Classical search algorithms maintain a list of the nodes to be expanded and possibly another list to remember the nodes which have already been visited. Most algorithms have the structure outlined in Figure 1.

Different algorithms use different strategies to update the expansion list and therefore to direct the search. They can be divided into two groups, blind search algorithms and informed search algorithms, depending on the strategy used to update the list. Below we outline the strategies used by the most common algorithms.

```
1. Initialise the expansion list with the starting node(s)

2. Repeat until the list is empty

   (a) Remove one node from the expansion list and add it to the list of the
       already-expanded nodes

   (b) Check if a termination criterion is satisfied (e.g. the node
       represents a goal state), if so stop.

   (c) Apply the search operators to the node according to a predefined
       control strategy

   (d) Remove invalid children (e.g. those already seen)

   (e) Update the expansion list with the new nodes
```

Figure 1: Typical classical search algorithm.

2.1.1 Blind search algorithms

Blind search algorithms are characterised by the fact that the only information available to drive the search is a predicate which returns true when a solution for the problem has been found, and false otherwise. Assuming the nodes for expansion are taken from the beginning of the list, children can be inserted in the expansion list according to the following strategies:

Breadth-first search adds the children at the end of the list. This means that ancestors are always expanded before their children and that the search proceeds uniformly in all directions.

Depth-first search adds the children at the beginning of the list. This means that children are expanded before their ancestors.

Non-deterministic search adds the children in random positions in the list.

2.1.2 Informed search algorithms

Informed search algorithms use a problem-specific cost function which evaluates the quality of a candidate solution. Accurate cost information significantly reduces the computational complexity of the search.

The cost function is used to sort the expansion list and nodes are usually taken from the beginning of the list. On the grounds of the cost function adopted, the following classes of algorithms can be identified:

Greedy Search uses as cost measure for a node n an estimate $h(n)$ of the cost of going from n to a goal node.

Branch-and-Bound Search, uses as cost measure the cost $g(n)$ of going from the start node to n.

A^* **Search** uses the sum $g(n) + h(n)$. If h underestimates the real cost of going from n to a goal state, then A^* is guaranteed to find the minimum cost solution while expanding minimum number of nodes.

2.1.3 Incomplete search algorithms

If the search space is finite all the algorithms described above are complete, i.e. they are guaranteed to find a solution if one exists. While theoretically desirable, completeness means that a very large number of nodes (possibly the entire search space) will have to be kept in the expansion list. In practice this is often impossible and the following incomplete search algorithms are often used for real-world problems:

Beam Search is a kind of breadth-first search in which only a finite number of promising nodes is expanded at each stage of the search.

Memory-Bounded A^* forgets less promising nodes when not enough memory is available.

2.2 Numerical search algorithms

There are a number of informed search algorithms which, although normally used for numerical optimisation, can be considered general search strategies. Examples are:

Hill Climbing uses an expansion list containing only a single node. This node is replaced by the best of its children, unless no child is better than the parent, in which case the search stops.

Gradient Descent is similar to hill climbing but uses additional information (the gradient) to generate a single child.

Simulated annealing is also similar to hill climbing (or gradient descent) but sometimes accepts a successor which is worse than its parent in order to escape local minima/maxima.

2.3 Evolutionary algorithms

Evolutionary algorithms work using strategies inspired by nature [6, 5, 2, 1, 9]. As they are usually driven either by a quality (or cost) measure or by a quality comparison predicate, they can be considered informed search algorithms.

Most evolutionary algorithms have the structure outlined in Figure 2, although not all the components described in the figure are present in all cases. In the following subsections we outline the basic features of the main evolutionary algorithms.

```
1. Initialise population
2. Evaluate population
3. Repeat until a stopping criterion is satisfied:
    (a) Select sub-population for reproduction
    (b) Recombine the genes of selected parents
    (c) Mutate the mated population stochastically
    (d) Evaluate the fitness of the new population
    (e) Select the survivors on the basis of their fitness
```

Figure 2: Generic evolutionary algorithm.

2.3.1 Genetic Algorithms

The simplest forms of Genetic Algorithms (GAs) work according to the scheme shown in Figure 2 except that: a) no filtering of the newly generated solutions take place and b) a cloning operator is used to copy some parents in the new population. In some algorithms there is no overlapping between generations and cloning is used to simulate the survival of parents for more than one generation (generational GAs); in others a single new individual is created in each cycle which replaces the current worst individual (steady state GAs). As GAs often use binary representations for the solutions of a problem, crossover and mutation are usually bit-string manipulation operators. However, various alternative representations such as permutation lists or parse trees are used in algorithms derived from the basic GA, e.g. genetic programming [7]. Many other variants of the basic scheme exist.

2.3.2 Evolutionary Strategies

Evolutionary Strategies (ESs) are parameter optimisation techniques. In ESs chromosomes are vectors of real valued parameters. Various forms of ESs exist:

$(1+1)$ **ES** uses a population consisting of only one individual. The parent generates one offspring per generation by applying normally distributed mutations. The standard deviation of mutations changes according to the frequency of successful mutations. If a child performs better than its parent, it replaces it.

$(\mu+1)$ **ES** keeps a population of μ individual and generates the offspring via recombination and mutation .

$(\mu+\lambda)$ **ES** produces λ offspring at each generation via recombination and mutation.

(μ,λ) **ES** replaces the parents with the best μ offspring out of λ.

EAs	*GOFAI*
Individual	Node
Population	Expansion list
Fitness	Cost of a solution
Chromosome	Representation for a solution
Gene	Part of a solution
Crossover/Mutation	Expansion operators
Selection	Expansion strategy

Table 1: Mapping between classical and evolutionary terms.

2.3.3 Evolutionary Programming

Evolutionary Programming (EP) is similar to a $(\mu + \mu)$ ES without recombination. The distinctive features of EP are: a) mutation is performed according to a Gaussian distribution (whose standard deviation depends on the fitness), and b) the mutation operator is controlled by parameters that are also optimised.

3 Evolutionary computation cookbook

Given their natural inspiration, evolutionary algorithms are usually described using a totally different terminology to classical algorithms. However, a clear correspondence can be identified between some of the concepts present in the two worlds which shows that EAs are informed search methods. For example, the concept of population in evolutionary computation clearly corresponds to the expansion list in classical search. Two other key correspondences are: crossover/mutation and expansion operators, and selection and expansion strategy (i.e. the method used to apply the expansion operators). Table 1 summarises the mapping between the AI and the evolutionary vocabularies.

From this established correspondence and the descriptions given in the previous sections, it is possible to start building an 'evolutionary computation cookbook'.

3.1 Ingredients

The first step to build the cookbook is to identify the ingredients present in some or all search algorithms. They are:

A **representation** for the candidate solutions to the problem. In some cases solutions are nodes, in other cases they are paths to a goal node.

A **termination criterion** by which the search is stopped. This can be a procedure which recognises a goal node or any other criterion which determines if the search is no longer productive.

A **quality evaluation** procedure that can give a measure of the quality of each node.

A **quality comparison** predicate that can tell which of two nodes is better.

A **quality gradient** procedure which can tell in which way the current node should be modified in order to get a better node.

A **memory structure** containing the nodes to be expanded (this can be a list, a fixed size population, a single node, etc.)

A **history** containing the nodes already visited in the search.

An **initialisation** procedure which fills the memory structure with one or more nodes.

A **selection procedure** which determines which node(s) are going to be expanded.

A **set of node-expansion operators** which act on one or more nodes to generate new nodes. Several generation strategies are possible: to generate only valid nodes, to generate valid nodes which have not been visited before, to generate partly invalid nodes.

A **control strategy** which controls the application of the operators. The control strategy can be probabilistic or deterministic.

A **repair/filtering procedure** which repairs invalid nodes or filters them out (in some cases no repair is performed and the quality measure of invalid nodes is reduced).

A **memory management procedure** which selects the elements to add to the list, removes elements from the list, and if necessary sorts the list.

3.2 Recipes

Usually a good cookbook does not only describe the ingredients to make a good recipe, it also explains which ingredients to choose and how to combine them. In the evolutionary computation cookbook the strategy is simple:

Take any classical, numerical or evolutionary algorithm
and incorporate some new ingredients.

A good method is to select ingredients that play a key role in other algorithms (although it should not be expected that all combinations of ingredients will be successful). In fact, there already exist several algorithms which can be considered hybrid instances of cookbook recipes (for example, it is not uncommon to see GAs used in conjunction with gradient-descent operators). In addition, good new algorithms can be designed quite easily, as shown in the next section where we consider an example, the GA^* algorithm.

1. Initialise the expansion list with a set of starting nodes and evaluate them.

2. Repeat until the termination criterion is satisfied:

   (a) Select an expansion operator according to a probabilistic or deterministic strategy

   (b) Select the necessary number of nodes using rank selection with probability p.

   (c) Apply the operator and, when appropriate, add the first of the selected nodes to the list of the already-expanded nodes

   (d) Reject all invalid children (e.g. those representing nodes already visited)

   (e) Evaluate the remaining children and update the expansion list keeping it sorted

Figure 3: GA^* search algorithm.

4 A new recipe: the GA^* algorithm

The GA^* algorithm is a hybrid algorithm derived from the recipe of the A^* algorithm with the addition of some evolutionary ingredients. The algorithm, which is shown in Figure 3, is actually a generalisation of the A^* algorithm as can be readily seen by constraining the expansion operators to be unary and using a selection probability $p = 1$. The GA^* is the result of the following considerations.

Firstly, the expansion list in EAs is usually limited and cannot grow. This means that EAs are incomplete search procedures. Although this also happens in classical search, a lot of effort has been devoted to designing algorithms that keep as much information as possible on the past search. In GA^* this information can be used very effectively to drive the future search (via crossover and backtracking) and also to avoid wasting computation by reconsidering the same solutions more than once. This also maintains the diversity in the population.

Secondly, classical and numerical search methods only consider unary expansion operators. One of the major sources of power of EAs derives from their use of binary operators (crossover) which are usually based on the idea of building blocks. Their introduction in GA^* can significantly improve the power of the technique (especially when $h(n)$ is not an underestimate) without requiring major changes in the algorithm (only the selection procedure has to be changed). Operators with arity greater than two could provide additional benefits.

Finally, selection in most classical algorithms is deterministic while in most EAs it is probabilistic. When certain conditions on the quality measure are satisfied deterministic selection can lead to optimal expansion strategies which guarantee, for example, minimum memory requirements, minimum number of expansions, optimum use of the

available memory, etc. On the other hand probabilistic selection also leads to important forms of optimality, like the optimum exploration/exploitation tradeoff, i.e. the optimum compromise between the need to sample the search space to collect information and the need to produce good solutions as soon as possible (e.g. at runtime). This is the reason why we have used probabilistic selection in GA^*.

A simplified version of GA^* (a branch-and-bound version with $h(n) = 0$ and no crossover) has been used in some preliminary experiments in the domain of route planning in complex terrains.

4.1 Route Planning with GA^*

The problem is finding a minimum-cost route between two locations in a digitised map which represents a complex terrain of variable altitude. The problem is complicated by the non-linearity of the cost of going from one pixel to another which varies with the magnitude and the sign of the local gradient (e.g. moving downhill costs much less than moving uphill) as well as the distance travelled.

In order to represent plans as finite-length chromosomes, we have devised a novel representation for plans based on the idea that a complete path between any two points A and B in a map can be considered as the result of applying a set of deformation functions to the straight line segment connecting A and B. If the deformation functions are orthogonal, in which case we call them orthogonal basis plans, then any plan can be obtained as a linear combination of orthogonal plans (the same is true if the basis plans are linearly independent). Given a fixed number of such functions, the coefficients of the linear combination can be represented as finite-length chromosomes.

In our experiments we have used the set of eleven independent triangular deformation functions shown in Figure 4. As a result of this choice GA^* included twenty two search operators which increment or decrement by a fixed amount the deformation coefficients associated with the plan being expanded. A typical plan produced by GA^* is shown in Figure 5(a) superimposed to a terrain map (quantised for display purposes) where brighter grey levels represent bigger altitudes.

In order to have some indications about strengths and weaknesses of the algorithms we have compared it with A^*. Unfortunately, the huge amount of memory and computation required by classical A^* to solve this kind of problems has prevented us from being able to perform the comparison. However, we have been able to use a variant of A^* known as A^*_ϵ (see [10]) which is guaranteed to find solutions that can be worse than optimal by at most ϵ. A plan produced by A^*_ϵ with $\epsilon = 0.1$ is shown in Figure 5(b).

Although no definitive results are available, our experiments seem to suggest that GA^* outperforms A^*_ϵ. Also, we have observed that rank selection prevents GA^* from wasting a lot of effort in local minima, and that the completeness of the algorithm guarantees that even if it is temporarily trapped by one such minimum sooner or later it explores other parts of the search space.

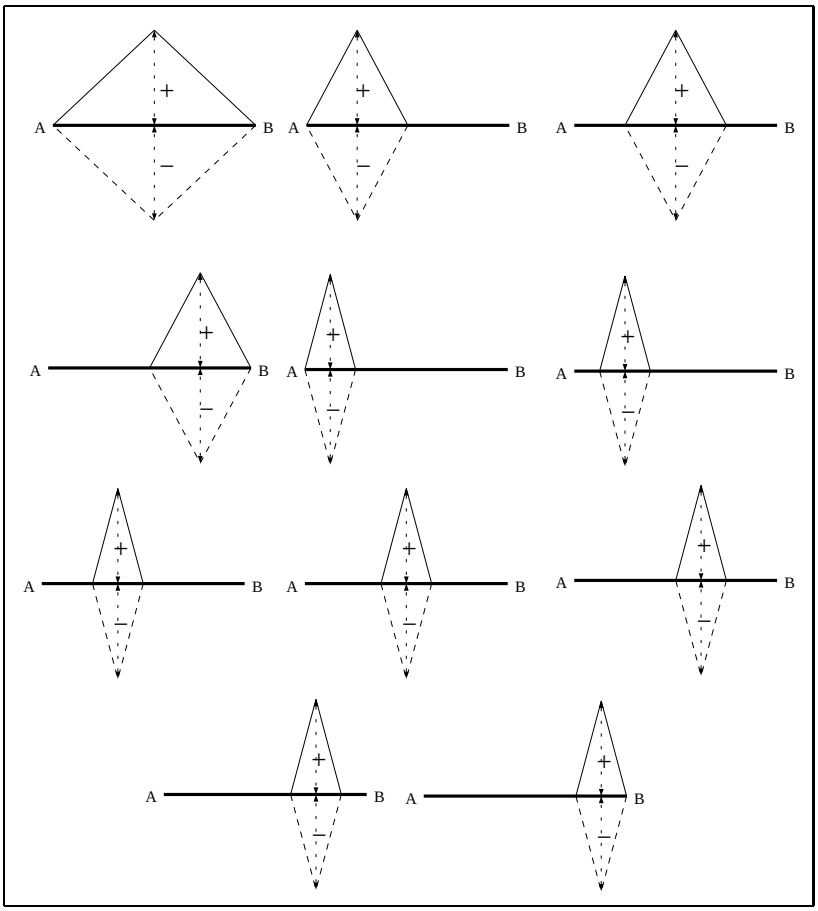
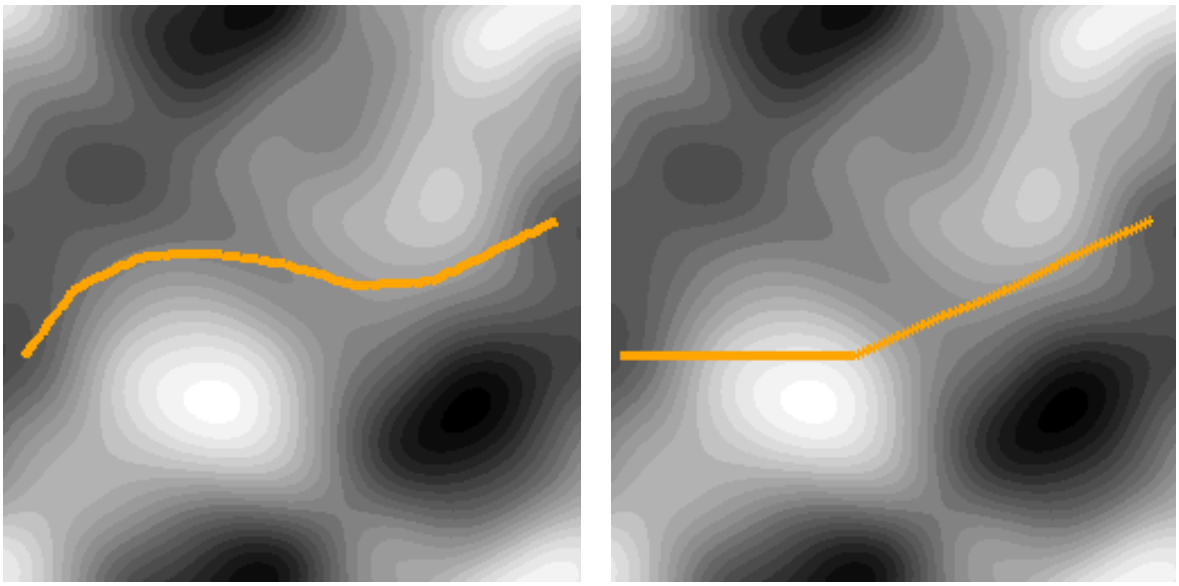


Figure 4: Eleven independent deformation functions used for route planning.



(a) (b)

Figure 5: Route produced by GA^* (a) and the corresponding output of A^*_ϵ.

5 Conclusions

In this paper we have compared many classical, numerical and evolutionary algorithms and identified their similarities and differences. This analysis has allowed us to identify and list the important components of search algorithms in the evolutionary computation cookbook. The cookbook also suggests how to combine these components to generate new algorithms.

We have shown how this approach can be used to produce new algorithms such as GA^* by combining features of classical and evolutionary techniques. We are confident that the framework provided by the cookbook will lead to many other powerful new algorithms, which combine the best of the evolutionary and AI worlds. We also believe that the mapping established in this paper between these worlds will also yield new theoretical advances and ultimately a single unified theory of search.

Acknowledgements

The authors wishes to thank Aaron Sloman and all the members of the EEBIC (Evolutionary and Emergent Behaviour Intelligence and Computation) group for useful discussions and comments. This research is partially supported by a grant under the British Council-MURST/CRUI agreement and a grant from the Defence Research Agency (DRA Malvern).

References

[1] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993.

[2] Lawrence Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[3] Thomas Dean, James Allen, and Yiannis Aloimonos. *Artificial Intelligence: Theory and Practice*. The Benjamin/Cummings Publishing Company, Redwood City, California, 1995.

[4] Marco Dorigo. Genetic and non-genetic operators in ALECSYS. *Evolutionary Computation*, 1(2):151–164, 1993.

[5] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Massachusetts, 1989.

[6] John Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, Massachusetts, second edition, 1992.

[7] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[8] Zbigniew Michalewicz. A hierarchy of evolution programs: An experimental study. *Evolutionary Computation*, 1(1):51–76, 1993.

[9] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin, second edition, 1994.

[10] J. Pearl. a^*_ϵ – an algorithm using search effort estimates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(4):392–399, 1982.

[11] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prendice Hall, Englewood Cliffs, New Jersey, 1995.

[12] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, third edition, 1992.