Sub-Exponential Algorithms for 0/1 Knapsack and Bin Packing

Thomas E. O'Neil
Computer Science Department
University of North Dakota
Grand Forks, ND, USA 58202-9015

Abstract - *This paper presents simple algorithms for 0/1 Knapsack and Bin Packing with a fixed number of bins that achieve time complexity* $p(n)\cdot 2^{O(\sqrt{x})}$ *where x is the total bit length of a list of sizes for n objects. The algorithms are adaptations of a method that achieves a similar complexity for the Partition and Subset Sum problems. The method is shown to be general enough to be applied to other optimization or decision problem based on a list of numeric sizes or weights. This establishes that 0/1 Knapsack and Bin Packing have sub-exponential time complexity using input length as the complexity parameter. It also supports the expectation that all NP-complete problems with pseudo-polynomial time algorithms can be solved deterministically in sub-exponential time.*

Keywords: 0/1 Knapsack, dynamic programming, Bin Packing, sub-exponential time, NP-complete problems.

1 Introduction

The comparative complexity of problems within the class NP-Complete has been a recurring theme in computer science research since the problems were defined and cataloged in the early years of the discipline [2]. In 1990, Stearns and Hunt [7] classified a problem to have power index *i* if the fastest algorithm that solves it requires $2^{O(n^i)}$ steps. Assuming that Satisfiability has power index 1, they argued that the Clique and Partition problems have power index one-half. Their analysis is based on two algorithms with time complexity $p(n)\cdot 2^{O(\sqrt{x})}$, where *x* is the length in bits of the input representations and $p(n)$ is a polynomial function of the number of graph edges (for Clique) or the number of integers in the input set (for Partition). These results were interpreted to provide strong evidence that Clique and Partition were easier problems than Satisfiability and most other NP-Complete problems.

In a subsequent study, Impagliazzo, Paturi, and Zane [3] presented another framework for comparison of NP-complete problems. Instead of adopting the power index terminology of Stearns and Hunt, they categorized problems based on weakly exponential ($2^{n^{\Omega(1)}}$) or strongly exponential ($2^{\Omega(n)}$) lower bounds (assuming that Satisfiability will one day be proven to be strongly exponential) and sub-exponential ($2^{o(n)}$) upper bounds. To avoid inconsistencies related to the characterization of input length, they defined a family of reductions (the Sub-Exponential Reduction Family) that would allow the complexity measure to be parameterized. This framework tolerated polynomial differences in the lengths of problem instances, and there was no complexity distinction among Clique, Independent Set, Vertex Cover or *k*-Sat. These conclusions are not consistent with those of Stearns and Hunt, where both Clique and Partition were easier than Satisfiability. It is clear that representations and complexity measures for problem instances play a critical role in complexity analysis.

In classical complexity theory, the complexity measure is the length of the input string. This parameter is formally determined, simply by counting the bits in the string. The advantage of using the formal measure is that it requires no semantic interpretation of the input string, and problems with vastly different semantics can by grouped together in formal complexity classes. Within the class NP-complete, we find that for many problems, the use of simple semantic complexity measures will not clash with detailed analysis based on the formal measure. This is generally true of strong NP-complete problems, where the objects in the input representing variables or nodes or set elements can be numbered (in binary). The numbers are just labels used for identification of the objects. There are other problems in the class, however, where the input contains a list of weights or values, and analysis based on semantic measures such as the number of objects versus the sum (or maximum) of the values can give radically different results: exponential time with one measure, polynomial time with the other. This collection of problems includes Partition, Subset Sum, 0/1 Knapsack, and Bin Packing, which we will refer to as the Subset Sum family. The safest approach to analysis of these problems is to use the formal complexity measure, which incorporates both relevant semantic parameters, and in this paper we show that the Subset Sum family of pseudo-polynomial-time problems is $2^{O(\sqrt{x})}$ (which is sub-exponential).

Stearns and Hunt [7] were apparently the first to demonstrate that an algorithm for the Partition problem exhibits sub-exponential time. The significance of this result was probably obscured by the claim in the same paper that the Clique problem is also sub-exponential, while its dual problem Independent Set remains strongly exponential. This apparent anomaly is a representation-dependent distinction, and it disappears when a symmetric representation for the problem instance is used [5]. The complexity distinction between Partition and Satisfiability, however, appears to have stronger credibility. In [6] it is shown that the sub-exponential upper bound for Partition is also valid for Subset Sum. The algorithm for Subset Sum is a variant of dynamic programming that is much simpler and more general than the backtracking/dynamic programming hybrid that Stearns and Hunt designed for Partition. In this paper, the sub-exponential Subset Sum algorithm is adapted to 0/1 Knapsack and Bin Packing with a fixed number of bins, establishing that these problems are also sub-exponential with respect to the formal complexity measure (total bit-length of input, denoted x). We also abstract from the previous methods a lemma that identifies the property of ordered sets of integers that is exploited to achieve sub-exponential time.

More recent complexity studies in the research literature for problems in the Subset Sum family do not typically use the input length as the complexity parameter. The current upper bound for both Subset Sum and 0/1 Knapsack is apparently $2^{O(n/2)}$ when the number of objects in the list is used as the complexity measure [8]. A lower bound of $\Omega(2^{n/2}/\sqrt{n})$ for Knapsack has also been demonstrated in [1]. The lower bound applies only to algorithms within a model defined generally enough to include most backtracking and dynamic programming approaches. The sub-exponential bounds derived here using the formal complexity measure complement rather than supersede the strongly exponential bounds derived using the number of objects in the input list (denoted n) as the complexity parameter.

2 Generalized Dynamic Programming

The Stearns and Hunt algorithm for Partition [7] combines backtracking with dynamic programming. Such hybrid approaches had been previously described in operations research literature (e.g. [4]). The input set is ordered and divided into a denser and a sparser subset. Backtracking is employed on the sparse subset, while dynamic programming is used for the dense subset. The results are combined to achieve time complexity $2^{O(\sqrt{x})}$, where x is the total length in bits of the input.

In this paper we employ a simpler algorithm that achieves the same goal. The approach was first developed for Subset Sum and Partition [6]. Similar to conventional dynamic programming, it represents a breadth-first enumeration of partial solutions. The problem instance is a list of objects, each of which has a size. The algorithm maintains a pool of partial solutions as it processes each object. The list of objects is ordered by size, and the largest objects are processed first. In contrast with conventional dynamic programming, the pool of solutions is dynamically allocated (hence the acronym DDP, for dynamic dynamic programming). It first grows and then shrinks as more objects are processed. The entire pool of solutions is traversed for each object, updating each solution by possibly subtracting the current object's size from the solution's remaining capacity. Each solution is also evaluated relative to the sum of sizes of the objects yet to be processed. The sum of remaining sizes can be used to prune the pool of solutions depending on problem semantics. This pruning relative to the sum of sizes of the unprocessed objects places a sub-exponential upper bound on the number of partial solutions in the pool.

The time analysis of the DDP method relies on a simple lemma (abstracted from the analysis in [6]) that allows us to bound the k^{th} value in an ordered list as a function of its position in the list and the total bit-length of the entire list (see Lemma 1 below). Bounding the k^{th} value allows us to bound the sum of the first k values as well. This, in turn, leads to a bound on the length of the pool of partial solutions in DDP algorithms.

Lemma 1: Let L represent a list of n positive natural numbers in non-decreasing order, let $L[k]$ represent the k^{th} number in the list, let b_k be the bit length of the k^{th} number, and let b be total number of bits in the entire list: $b = \sum_{i=1}^{n} b_i = \sum_{i=1}^{n} 1 + \lfloor lg\ L[i] \rfloor$. Then $L[k] < 2^{(b-k+1)/(n-k+1)+1}$.

Proof: An upper bound on the value of $L[k]$ for any list with total bit length b is obtained by reserving as few bits as possible for the smaller numbers in the list and as many bits as possible for $L[k]$ and the numbers that follow it. This is accomplished by setting $L[1]$ through $L[k\text{-}1]$ to 1 and distributing the remaining bits equally among the higher $n-k+1$ numbers. In that case, $L[k]$ has no more than $(b-k+1)/(n-k+1)$ bits, establishing $L[k] < 2^{(b-k+1)/(n-k+1)+1}$. □

3 The Knapsack Problem

The 0/1 Knapsack problem is defined as follows: given a set of n objects S with sizes $s[1..n]$ and values $v[1..n]$, find a subset of objects with the highest value whose size is less than or equal to C, the capacity of the knapsack [2]. The problem can also be expressed as a decision problem, where we determine the existence of a subset whose value is greater than or equal to a target value V.

```
//* Given a set of n objects whose sizes are specified
in an array s[1..n] in non-decreasing order and whose
values are stored in an array v[1..n], find the highest
valued subset whose total size is less than or equal to
capacity C.  */

public int Knapsack()
 1)  bestval ← 0;
 2)  sizeofrest ← Σ_{i=1}^{n} s[i]  ;  valueofrest ← Σ_{i=1}^{n} v[i]  ;
 3)  Pool ← {(C, 0)};
 4)  for i ← 1 to n
 5)    size ← s[n – i +1];  value ← v[n – i +1];
 6)    NewList ← { };
 7)    for each sack in Pool
 8)      if (sack.capacity < size )
 9)        continue;
10)      else if (sack.capacity > sizeofrest)
11)        bestval ← max (bestval,
                          sack.value + valueofrest)
12)        remove sack from Pool;
13)      else
14)        bestval ← max (bestval,
                          sack.value + value);
15)        NewList.append ((sack.capacity – size,
                            sack.value + value));
       end for
16)    sizeofrest ← sizeofrest – size;
       valueofrest ← valueofrest – value;
17)    Pool ← merge(Pool, NewList);
     end for
18)  return bestval;
```

Figure 1. The *Knapsack* algorithm.

3.1 The *Knapsack* algorithm

In adapting the DDP method to the *Knapsack* problem, we can iterate either the size or the value array as the control for the outer loop. Here we use the size array. The algorithm keeps a pool of (*capacity*, *value*) pairs representing partially filled knapsacks, initially containing an empty sack represented as (C, 0), where C is the capacity of the empty sack. For each object in S and for each sack currently in the pool, we add a new sack representing the current sack plus the current object. This is accomplished by subtracting the object size from the sack's remaining capacity and adding the object value to the sack's value.

Pseudo-code for the *Knapsack* algorithm is shown in Figure 1. Lines 1-3 initialize the global *Pool*, the *bestval* variable, and variables representing the cumulative size and value of the remaining objects. There is one iteration of the outer *for* loop (lines 4-17) for each object in the set $S = \{y_1, y_2, ..., y_n\}$. The size array s, in which $s[i]$ is the size of object y_i, is assumed to be in non-decreasing order, and the largest numbers are processed first, so object y_{n-i+1} is processed during the i^{th} iteration. The pool of partially filled sacks is updated by the inner *for* loop (lines 7-15). For each sack in the pool, $s[n-i+1]$ is subtracted from its capacity and $v[n-i+1]$ is added to its value, placing the new (*capacity*, *value*) on a second ordered sack list. The pool and the new sack list are merged in the last step of the outer loop (line 17). The best value for a filled sack is updated when appropriate in lines 11 and 14, whenever an updated sack is created. At completion of the outer loop, the best value is returned. The algorithm does not return the contents of the sack with the best value, but this could be accomplished by adding a reference to a subset object to the (*capacity*, *value*) pairs in the pool, increasing the time complexity by no more than a factor of n.

The inner loop has two conditions that moderate the length of the pool. Lines 8 and 9 skip sacks that can't hold the current object. Also, in lines 10-12, sacks with enough capacity to hold all remaining objects are removed from the pool after updating the *bestval* variable. If all remaining objects will fit in a sack, there is no process them one-by-one.

The outer loop also has logic to control the size of the pool. The last step in the outer loop is a sequential merge operation that adds the new partially filled sacks to the pool. If two sacks with the same capacity are encountered during the merge, only the sack with the higher value is added to the pool. Thus the capacities of all sacks in the pool are unique.

3.2 Time Analysis of Knapsack

The time analysis closely follows the method used for the Subset Sum algorithm in [6]. Let $S = \{y_1, y_2, ..., y_n\}$ and assume the sizes are stored in non-decreasing order ($s[i] \leq s[i+1]$). The total number of steps is determined by the size of *Pool*. With each iteration of the outer *for* loop, *Pool* is traversed and possibly extended (requiring 2 passes – one by the inner *for* loop and the other by the sequential merge step). The total amount of work is closely estimated (within a factor of 2) by

$$\sum_{i=1}^{n} |Pool(i)| \qquad (1)$$

where $|Pool(i)|$ is the length of *Pool* at the beginning of outer loop iteration i.

Since the merge operation eliminates duplication of capacities, we can describe length of *Pool*(i) as at most *MaxC*(i), the largest capacity of any sack on the list at the beginning of iteration i. The list is actually smaller than this, since all the capacities between zero and the maximum are not present. We also know that the length of the list can, at most, double with each loop iteration, so regardless

of the maximum value in the list, its length cannot exceed 2^i. This gives us

$$|Pool(i)| \le min(2^i, MaxC(i)). \quad (2)$$

The length of *Pool* will grow rapidly and later possibly shrink as i approaches n. Our goal is to find an upper bound for $MaxC(i)$. Initially $MaxC(1) = C$, which is the capacity of the empty sack. Only smaller-capacity sacks are added to the list, and eventually the larger-capacity sacks are removed when the condition in line 10 becomes true, so

$$MaxC(i) \le \sum_{j=1}^{n-i+1} s[j] \le (n-i+1)\cdot s[n-i+1]. \quad (3)$$

Bounding $MaxC(i)$ thus reduces to finding an upper bound for $s[n - i + 1]$, and Lemma 1 is invoked for this purpose. To complete the analysis, we bound the step counts as a function of b, the total bit length of the size array s. We consider two cases.

Case 1. $n \le \sqrt{b}$. Here we have

$$\sum_{i=1}^{n} |Pool(i)| \le \sum_{i=1}^{n} min(2^i, MaxC(i)) \le n\cdot 2^{\sqrt{(b)}}. \quad (4)$$

Case 2. $n > \sqrt{b}$. In this case we split the summation at $i = \sqrt{b}$.

$$\sum_{i=1}^{n} |Pool(i)| \le \sum_{i=1}^{n} min(2^i, MaxC(i)) \quad (5)$$

$$\le \sum_{i=1}^{\sqrt{b}-1} min(2^i, MaxC(i)) + \sum_{i=\sqrt{b}}^{n} min(2^i, MaxC(i)) \quad (6)$$

$$\le (\sqrt{b}-1)\cdot 2^{\sqrt{b}-1} + \sum_{i=\sqrt{b}}^{n} min(2^i, MaxC(i)) \quad (7)$$

$$\le (\sqrt{b}-1)\cdot 2^{\sqrt{b}-1} + (n-\sqrt{b}+1)\cdot MaxC(\sqrt{b}) \quad (8)$$

$$\le (\sqrt{b}-1)\cdot 2^{\sqrt{b}-1} + (n-\sqrt{b}+1)\cdot \sum_{j=1}^{n-\sqrt{b}+1} s[j]. \quad (9)$$

$$\le (\sqrt{b}-1)\cdot 2^{\sqrt{b}-1} + (n-\sqrt{b}+1)^2\cdot s[n-\sqrt{b}+1] \quad (10)$$

At this point, we employ Lemma 1 to compute the bound for $s[k]$ where $k = n-\sqrt{b}+1$, and we continue by replacing $s[n-\sqrt{b}+1]$ with $2^{\sqrt{b}+1}$:

$$< (\sqrt{b}-1)\cdot 2^{\sqrt{b}-1} + (n-\sqrt{b}+1)^2\cdot 2^{\sqrt{b}+1} \quad (11)$$

$$< (\sqrt{b}+(n+1)^2)\cdot 2^{\sqrt{b}+1} \quad (12)$$

$$< (2n^2+6n+2)\cdot 2^{\sqrt{b}}. \quad (13)$$

This establishes that the time complexity of *Knapsack* is $O(p(n)2^{\sqrt{b}})$ for a polynomial function $p(n)$. The argument b is the total bit length of the list of sizes. The entire input for the problem also includes the capacity C and a list of n values. We can't make any specific assumptions about the relative magnitudes of the sizes and values, but we are certain that if x is the total input length, then b will be smaller than x, and the $O(p(n)2^{\sqrt{b}})$ step count will also be $O(p(n)2^{\sqrt{x}})$.

4 The Bin Packing Problem

The Bin Packing problem is defined as follows: given a set of n objects S with sizes $s[1..n]$, determine whether the objects will fit into a fixed number of k bins, each with a capacity of B. The problem can also be expressed as an optimization problem in which the smallest B is determined [2]. When B is equal to the sum of all sizes divided by k, the problem represents a generalization of the Partition problem.

```
/* Given a set of n objects whose sizes are specified
in an array s[1..n] in non-decreasing order, determine
whether all objects can be stored in k bins, each with
capacity B.
*/

public boolean BinPack()

 1)  sizeofrest = Σ_{i=1}^{n} s[i] ;
 2)  Pool = {(B, B, …, B)};
 3)  for i ← 1 to n
 4)     nextsize ← s[n – i +1];
 5)     NewList ← { };
 6)     for each bintuple in Pool
 7)        if (bintuple.capacity[1] < nextsize )
 8)              continue;
 9)        else if (bintuple.capacity[1] > sizeofrest)
10)              return true;
11)        else
12)           for j ← 1 to k
13)              newtuple ← update(bintuple, j, nextsize);
14)              if (newtuple != null)
15)                 NewList.insert (newtuple);
              end for
           end for
16)     Pool ← NewList;
17)     sizeofrest ← sizeofrest – nextsize;
      end for
18)  return false;
```

Figure 2. The *BinPack* algorithm.

4.1 The *BinPack* Algorithm

When we adapt the DDP strategy to Bin Packing, we find a few significant differences from the Knapsack version. The *BinPack* algorithm is shown in Figure 2. The pool of partial solutions must be a list of k-tuples, where each component of a tuple is the remaining capacity of one of the bins (see line 2). Also, we are not searching for a subset. All the objects in the original set S must be included in the solution. This has implications for the logic in the nested loops of the algorithm. Any partial solution in the inner loop that cannot accommodate the next object can be

discarded (lines 7-8), and the pool of updated partial solutions created by the inner loop replaces the pool from the previous iteration of the outer loop (rather than merging with the previous pool; see line 16). We also find that the test enforcing the upper limit on the size of the pool (relative to the sum of the remaining object sizes) triggers early termination (lines 9-10). This version of the algorithm does not specify what objects are placed in what bins, but this information could be included by associating a reference to a size n object to each partial solution. This would increase the time complexity by no more than a factor of n.

4.2 Time Analysis of *BinPack*

The time analysis of *BinPack* follows the same general logic as the analysis for *Knapsack*. The major difference is the growth rate of the pool of partial solutions. While the pool can double in length with each iteration of the inner loop in *Knapsack*, it can increase in length by a factor of k in *BinPack*. Another significant difference is the cost of suppressing duplicates in the pool of partial solutions. We make the conservative assumption that the insertion of an updated partial solution in the pool takes linear time in the current length of the pool. We demonstrate below that in spite of these significant differences, the time complexity of the algorithm remains sub-exponential.

To proceed with the analysis, let $S = \{y_1, y_2, ..., y_n\}$, and assume the sizes are stored in non-decreasing order ($s[i] \leq s[i+1]$). As with *Knapsack*, The total number of steps is closely related to the size of *Pool*. With each iteration of the outer *for* loop, *Pool* is traversed and replaced with an updated version (called *NewList*). Each insertion into *NewList* requires linear time. The total amount of work is therefore estimated as

$$(\sum_{i=1}^{n} |Pool(i)|^2) \tag{14}$$

where $|Pool(i)|$ is the length of *Pool* at the beginning of outer loop iteration i.

Since the insert operation of line 15 eliminates duplication of capacities, we can describe length of *Pool*(i) as at most $MaxC(i)^k$. If *MaxC*(i) is the largest capacity of any bin in any tuple on the list at the beginning of iteration i, the number of distinct tuples cannot exceed this quantity raised to the power k. This grossly overestimates the number of tuples, since the capacities within each tuple are in non-increasing order and since all the tuples have the same sum. It is an interesting counting problem to determine a tight upper bound for the number of tuples, but the loose bound is sufficient to establish the desired complexity result. We also know that the length of the list can, at most, grow by a factor of k with each loop iteration, so regardless of the maximum value in the list, its length cannot exceed k^i. This gives us

$$|Pool(i)| \leq min(k^i, MaxC(i)^k). \tag{15}$$

Lines 9 and 10 assure us that the algorithm terminates if $MaxC(i)$ exceeds the sum of the remaining object sizes, so we have

$$MaxC(i) \leq \sum_{j=1}^{n-i+1} s[j] \leq (n-i+1)\cdot s[n-i+1]. \tag{16}$$

To complete the analysis, we bound the step counts as a function of x, the total bit length of the size array s. As before, we consider two cases.

Case 1. $n \leq \sqrt{x}$. Here we have

$$\sum_{i=1}^{n} |Pool(i)|^2 \leq \sum_{i=1}^{n} min(k^i, MaxC(i)^k)^2 \tag{17}$$

$$\leq n\cdot(k^n)^2 \leq n\cdot k^{2\sqrt{(x)}} \leq n\cdot 2^{2lg\,k\sqrt{(x)}}. \tag{18}$$

Case 2. $n > \sqrt{x}$. In this case we split the summation at $i = \sqrt{x}$.

$$\sum_{i=1}^{n} |Pool(i)|^2 \leq \sum_{i=1}^{n} min(k^i, MaxC(i)^k)^2 \tag{19}$$

$$\leq \sum_{i=1}^{\sqrt{x}-1} min(k^i, MaxC(i)^k)^2 + \sum_{i=\sqrt{x}}^{n} min(k^i, MaxC(i)^k)^2 \tag{20}$$

$$\leq (\sqrt{x}-1)\cdot k^{\sqrt{x}-1} + (n-\sqrt{x}+1)\cdot(MaxC(\sqrt{x})^k)^2 \tag{21}$$

Then by Lemma 1:

$$< (\sqrt{x}-1)\cdot k^{\sqrt{x}-1} + (n-\sqrt{x}+1)((n-\sqrt{x}+1)2^{\sqrt{x}+1})^{2k} \tag{22}$$

and by algebraic simplification:

$$< (n + n^{2k+1})\,2^{2k(\sqrt{x}+1)} \tag{23}$$

Since k is a constant, this establishes that the time complexity of *BinPack* is $p(n)\cdot 2^{O(\sqrt{x})}$ for a polynomial function $p(n)$.

5 Conclusion

The algorithms in the previous sections demonstrate that dynamic programming with dynamic allocation (DDP) can be used to prove that 0/1 Knapsack and Bin Packing with a fixed number of bins have time complexity $p(n)\cdot 2^{O(\sqrt{x})}$ where x is the total bit length of n input numbers. This places these problems with Partition and Subset Sum in the subclass of NP-complete problems that have sub-exponential upper bounds on running time, when input length is used as the complexity parameter.

The Knapsack problem was formulated as an optimization problem above, while Bin Packing was presented as a decision problem. It is apparent that the *Knapsack* algorithm can be modified to solve the decision version of the problem without changing its time complexity. It is also possible to modify *BinPack* to find the smallest bin capacity needed to store all objects in k bins, as long as k is constant, without changing its time complexity. Given the simplicity and generality of Lemma

1, which provides the foundation for the time analyses, we expect that the DDP method can be applied to any NP-complete problem involving a list of weighted objects that has pseudo-polynomial time complexity.

References

[1] M. Alekhnovich, A. Borodin, J. Buresh-Oppenheim, R. Impagliazzo, A. Magen, and T. Pitassi, "Toward a Model for Backtracking and Dynamic Programming," *Proceedings of the 20th Annual IEEE Conference on Computational Complexity*, pp. 308-322 (2005).

[2] M. Garey and D. Johnson, C*omputers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman Press, San Francisco, CA (1979).

[3] R. Impagliazzo, R. Paturi, and F. Zane, "Which Problems Have Strongly Exponential Complexity?," Journal of Computer and System Sciences 63, pp. 512-530, Elsevier Science (2001).

[4] S. Martello and P. Toth, "A mixture of dynamic programming and branch-and-bound for the subset sum problem," *Management Science* 30(6), pp. 765-771 (1984).

[5] T. E. O'Neil, "The Importance of Symmetric Representation," *Proceedings of the 2009 International Conference on Foundations of Computer Science (FCS 2009)*, pp. 115-119, CSREA Press (2009).

[6] T. E. O'Neil and S. Kerlin, "A Simple $2^{O(\sqrt{x})}$ Algorithm for Partition and Subset Sum," *Proceedings of the 2010 International Conference on Foundations of Computer Science (FCS 2010)*, pp. 55-58, CSREA Press (2010).

[7] R. Stearns and H. Hunt, "Power Indices and Easier Hard Problems", *Mathematical Systems Theory* 23 (1990), pp. 209-225.

[8] G. J. Woeginger, "Exact Algorithms for NP-Hard Problems: A Survey," *Lecture Notes in Computer Science* 2570, pp. 185-207, Springer-Verlaug, Berlin (2003).