A Parallel $O(n2^{7n/8})$ Time-memory-processor Tradeoff for Knapsack-like Problems

Li Ken-Li[1,2], Li Ren-Fa[1], Yang Lei[1], Zhou Yan-Tao[1]

[1] School of Computer and Communication, Hunan University,
Changsha, 410082, China
{jt_lrf, jt-yl, jt_zyt}@hnu.cn
[2] Department of Computer Science, University of Illinois at Urbana-Champaign,
Champaign, 61801,USA
kenlili@uiuc.edu

Abstract. A general-purpose parallel three-list four-table algorithm that can solve a number of knapsack-like NP-complete problems is developed in this paper. Running on an EREW PRAM model, The proposed parallel algorithm can solve this kind of problems of size n in $O(n2^{9n/20})$ time, with $O(2^{13n/40})$ shared memory units and $O(2^{n/10})$ processors, and thus its time-space-processor tradeoff is $O(n2^{7n/8})$. The performance analysis and comparisons show that the proposed algorithms are both time and space efficient, and thus is an improved result over the past researches. Since it can break greater variables knapsack-based cryptosystems and watermark, the new algorithm has some cryptanalytic significance.

1 Introduction

Every NP-complete problem can be solved in $O(2^n)$ time by exhaustive search, but this complexity becomes prohibitive when n exceeds 70 or 80. Assuming that $NP \neq P$, we cannot hope to find algorithms whose worst-case complexity is polynomial, but it is both theoretically interesting and practically important to determine whether substantially faster algorithms exist. In this paper we describe a parallel algorithm which can solve the knapsack problem. But owing to the work done by Schoreppel and Shamir [1], our proposed algorithm actually can solve a fair number of NP-complete problems including knapsack, partition, exact satisfiability, set covering, hitting set, disjoint domination in graphs, etc, which can be related by the *composition operator* [1]. Although the proposed algorithm is a versatile algorithm, to make this algorithm more easily be understood, we only take the knapsack problem as the representative to narrate this algorithm.

Given n positive integers $W = (w_1, w_{2, \ldots,} w_n)$ and a positive integer M, the knapsack problem is the decision problem of a binary n-tuple $X = (x_1, x_2, \ldots, x_n)$ that solves the equation: $\sum_{i=1}^{n} w_i x_i = M$. This problem was proved to be NP-complete. Solving the knapsack problem can be seen as a way to study some large problems in number theory and, because of its exponential complexity, some public-key cryptosystem are based on it [2-3]. Branch and Bound algorithms were proposed, but the worst case complexity is still $O(2^n)$ [4]. A major improvement in this area was made by Horowitz and Sahni [4], who drastically reduced the time needed to solve the knapsack problem by conceiving a clear algorithm in $O(n2^{n/2})$ time and $O(2^{n/2})$ space. It is known as the *two-list* algorithm. Based on this algorithm, Schrowppel and Shamir [1] reduced the memory requirements with the *two-list four-table* algorithm which needs $O(2^{n/4})$ memory space to solve the problem in still $O(n2^{n/2})$ time. Using unbalanced four tables, an adaptive algorithm is presented in [5], which can solve the knapsack-like problems according to the available computation source. Although the above algorithm is by far the most efficient algorithm to solve the knapsack problem in sequential, it can not solve any instances where the size n is great.

With the advent of the parallelism, much effort has been done in order to reduce the computation time of problems in all research areas [6-14], most of which are based on CREW (concurrent read exclusive write) PRAM (parallel random access machine) model. Karnin [6] proposed a parallel algorithm that parallelizes the generation routine of the *two-list four-table* algorithm. In his algorithm the knapsack problem could be solved with $O(2^{n/6})$ processors and $O(2^{n/6})$ memory cells in $O(2^{n/2})$ time. The algorithm proposed by Amirazizi and Helman [7] runs in $O(n2^{\alpha n})$ time, $0 \leq \alpha \leq 1/2$, by allowing $O(2^{(1-\alpha)n/2})$ processors to concurrently access a list of this same size. They also present a more feasible *Time-Space-Processor* (*TSP*) model for evaluation of performance of different algorithms for the solution of knapsack-like NP-complete problems [7]. Ferreira [8] proposed a parallel algorithm that solves the knapsack problem of size n in time $T = O(n(2^{n/2})^{\varepsilon})$, $0 \leq \varepsilon \leq 1$, when $P = O((2^{n/2})^{1-\varepsilon})$ processors $S = O(2^{n/2})$ memory units are available. Chang et al. [9] presented another parallel algorithm where the requirement of the sharing memory is $O(2^{n/2})$ by using $O(2^{n/8})$ processors to solve the knapsack problem still in $O(2^{n/2})$ time. Thereafter, based on Chang et al.'s parallel algorithm, Lou and Chang [10] successfully parallelize the second stage of the *two-list* algorithm. Regretfully, it is independently found in [11] and [12] that the analysis of the complexity of the Chang et al.'s algorithm was wrong. In addition to pointing out the wrong in literature [9], we also proposed a CREW-PRAM cost-optimal parallel algorithm [11], and thereafter, a cost-optimal algorithm without memory conflicts was further presented in [13]. It must be pointed out that the space complexity is very important when solving the knapsack-like problems [6,15]. However, because the memories required in both of these two cost-optimal parallel algorithms are still $O(2^{n/2})$, it make the available memory cells a bottleneck when using these algorithms to break practical knapsack based cryptosystem.

Therefore, to further reduce the required memory units for the solution of this kind of NP-complete problems, based on Ferreira's CREW based parallel *three-list* algorithm [14], we proposed a new parallel *three-list four-table* algorithm. The main properties of the proposed algorithm are as follows:

(i) With this algorithm, we can solve knapsack-like problems in $O(n2^{9n/20})$ time, $O(2^{13n/40})$ shared memory units when $O(2^{n/10})$ processors are available. It results in an $O(n2^{7n/8})$ *TSP* trade off, which is considerably better than those of all similar algorithms published so far.

(ii) It can be performed on an EREW (exclusive read exclusive write) PRAM machine model, and thus is a totally without memory conflicts algorithm. Furthermore, the algorithm is completely practical in the sense that it is easy to program and it can handle problems which are almost 1.5 times as big as those handled by previous algorithms.

The rest of this paper is organized as follows. Section 2 explains the parallel *three-list* algorithm, on which the proposed algorithm is based. The proposed parallel algorithm is described in Section 3. Then, in Section 4, the performance comparisons follow. Finally, some concluding remarks are given in Section 5.

2 The parallel three-list algorithm

In 1995, Ferreira presented a parallel *three-list* algorithm, which is based on a CREW PRAM model [14]. The number of processor, time complexity, and space requirements in it are $O(2^{\beta n})$, $O(n2^{(1-\varepsilon/2-\beta)n})$, $O(n2^{\varepsilon n/2})$, $0<\varepsilon<1$, $0\le\beta\le 1-\varepsilon/2$, respectively. It is viewed as an important breakthrough in the research of knapsack-like problems for it can solve the knapsack-like problems in a way of both time and space effective [14]. Because our parallel algorithm is based on this algorithm, we introduce it. To make it easy be understood, let the number of processors be $O(2^{n/10})$.

Algorithm 1: The *Three-list* algorithm
Generation stage

```
1. Divide $W$ into three parts: $W_1 = (w_1,w_2,\ldots,w_{9n/20})$,$W_2 = (w_{9n/20+1},w_{9n/20+2},\ldots,w_{18n/20})$, $W_3 = (w_{18n/20+1}, w_{18n/20+2},\ldots, w_n)$.
2. Form all possible subset sums of $W_1$, $W_2$, then sorted them in an nondecreasing order and store them as $A = [A_1, A_2,\ldots, A_{2^{\frac{9n}{20}}}]$ and $B = [B_1, B_2,\ldots, B_{2^{\frac{9n}{20}}}]$, respectively.
3. Form all possible subset sums of $W_3$, and store them as $C = [C_1,C_2,\ldots, C_{2^{\frac{n}{10}}}]$.
```

Search stage

```
1. For all $C_i$ in $C$ where $1\le i\le 2^{n/10}$
2. $C_i$ execute the binary search over $A + B$:
3. If a solution is found: then stop, output the solution
4. If a solution cannot be found: then stop: output that there is no solution.
```

The time and space complexity of this algorithm are $O(n\times 2^{11n/20})$ and $O(2^{9n/20})$ [14].

Based on its serial algorithm, Ferreira's parallel algorithm is very direct. It runs on a CREW model. The subset sums in list A and B which hold $2^{9n/20}$ subset sums respectively are stored in the shared memory. And each processor P_i $(1\le i\le P)$, which holds the subset sum C_i, execute a "virtual" binary search on the list $A + B$ to make sure whether $A[j] + B[l] = M - C_i$ is satisfied, $1\le j,l\le 2^{9n/20}$ .The parallel *three-list* algorithm consists of the following three main steps [14].

Algorithm 2: parallel *three-list* algorithm

```
for all $P_i$ where $1 \le i \le 2^{n/10}$ do
  1. Generation of the two lists $A$, $B$ and $C$
  2. Sorting of the two lists
  3. Binary search over $A + B$
end
```

The time and space needed in this algorithm are $O(n \times 2^{9n/20})$ and $O(2^{9n/20})$ [14].

3 The proposed parallel algorithm

Although Ferreira's above algorithm is considered as a main breakthrough for the researches on the knapsack problem, it still have an obvious shortcoming, i.e. the *TSP* tradeoff is $O(n \times 2^n)$, which is greater than that of the recent parallel algorithms in [11,14] by a factor n. To overcome this shortcoming, we redesign the two main stages of the parallel *three-list* algorithm. In list generation stage, we introduce four tables to produce two ordered list A and B dynamically. Doing so we can reduce the space complexity from $O(2^{9n/20})$ to $O(2^{13n/40})$. While in list search stage, we replace the matrix search way in [14] with the *two-list* like search algorithm, which is more simply and can reduce the time needed by a factor $O(n)$ in search stage.

In our proposed algorithm, each of the two lists stored in shared memory have a size of $O(2^{9n/20})$, whose elements will be dynamically generated one by one, by using only $O(2^{13n/40})$ shared memory units. Now consider the two stages of the algorithm.

3.1 The generation stage

Using the selection technique [14], Ferreira's parallel search algorithm is subtle. For it reduced the time needed otherwise for direct enumerating on the virtual list $A + B$ from $O(2^{9n/10})$ to $O(n \times 2^{9n/20})$. However, it is a little complicated for it concerns the search of "virtual" matrix [14]. Now we use the simply *two-list* like search to fulfill the list search stage.

Suppose the two ordered list A and B exist before the following algorithm 3 executes. We can use the following *two-list* like search algorithm to make sure that for any $C[k]$, $1 \le k \le 2^{n/10}$ whether exist $A[i]$ and $B[j], 1 \le i, j \le 2^{9n/20}$, such that the formula $A[i] + B[j] + C[k] = M$ can be satisfied.

Algorithm 3: parallel two-list like search algorithm

```
The subset sums in list A and B are sorted in increasing and decreasing order
for all processors $P_k$ where $1 \le k \le 2^{n/10}$ do
  1. $i = 1, j = 1$.
  2. If $A[i] + B[j] = M - C[k]$, then stop: a solution is found, and write the result
     into the shared memory.
  3. If $A[i] + B[j] < M - C[k]$, then $i = i + 1$; else $j = j + 1$.
  4. If $i > 2^{9n/20}$ or $j > 2^{9n/20}$ then stop: there is no solution.
```

```
5. **Goto** Step 2.
End
```

Lemma 1. Let all elements in list *A* and *B* are given, the time needed to perform the algorithm 3 is at most $2 \times 2^{9n/20}$.

Proof. The condition that the loop ends shows that once the variables *i* or *j* is greater than $2^{9n/20}$, the algorithm terminates. While for each computation step, the value of one of the above two variables must increase by 1. So it is obvious that the maximum of the needed time to perform the algorithm 3 is $2 \times 2^{9n/20}$.

Compared with the Ferreira's search algorithm [14], the search time needed here is reduced by a factor $O(n)$. But the space requirements do not increase.

3.2 The search stage

We discuss how to produce all elements of lists *A* and *B* stored in the shared memory. Note that in list search algorithm 3, each processor accesses the elements of the sorted lists *A* and *B* sequentially, and thus there is no need to store all the possible subset sums of *A* and *B* simultaneously in the shared memory—what we need is the ability to generate them quickly (on-line, upon request) in sorted order. So if we generate the two ordered lists dynamically, the needed space will reduced greatly. To implement this key idea, we explore the thoughts in [1] where four tables are used to dynamically produce the two sorted lists. Use *four tables* T_1, T_2, and T_3 , T_4 to produce the two sorted lists *A* and *B*, where T_1 includes all possible subset sums of knapsack entries $(w_1,w_2,\dots,w_{9n/40})$, …, T_4 includes all sums of $(w_{27n/40+1},w_{27n/40+2},\dots, w_{36n/40})$. let $e = 2^{9n/40}$, and mark $T_i = (t_{i1}, t_{i2}, \dots, t_{ie})$, $i = 1,2,3,4$. We first sort all sums in T_1 in an increasing order. Then use a priority queue Q_1 which has a length of $O(2^{9n/40})$. At start, Q_1 stores all pairs of first (T_1) and all elements t_{2i}. It can be updated by two operations *deletion* and *insertion*, which enables arbitrary insertions and deletions to be done in logarithmic time of the length of the queue, and makes the pair with the smallest $t_{1i} + t_{2j}$ sum accessible in constant time. Through the efficient heap implementations of priority queues [1], the following algorithm is designed to dynamically produce all sums of $T_1 + T_2$ in an increasing order. For the processes to generate list *A* and *B* are similar, we focus on the procedures on the process to generate list *A*.

Algorithm 4: algorithm for generating all sums of $T_1 + T_2$ dynamically

```
Tables $T_1 = (t_{11}, t_{12}, \dots, t_{1e})$, $T_2 = (t_{21}, t_{22}, \dots, t_{2e})$ are given
(1) sort $T_1$ into increasing order;
(2) insert into $Q_1$ all the pairs (first $(T_1)$, $t_{2i}$) for all $t_{2i} \in T_2$;
(3) Repeat until $Q_1$ becomes empty.
        $(t_1, t_2) \leftarrow$ pair with smallest $t_1 + t_2$ sum in $Q_1$;
        $S_1 \leftarrow (t_1 + t_2)$
        if $S_1$ is needed and used for the objectivity of computation;
        delete $(t_1, t_2)$ from $Q_1$;
        if the successor $t_1^1$ of $t_1$ in $T_1$ is defined,
        insert $(t_1^1, t_2)$ into $Q_1$;
```

Lemma 2. One element in $T_1 + T_2$ can be produced in $O(9n/40)$ time; while all $2^{9n/20}$ elements can be dynamically generated in $O(n2^{9n/20})$ time with $O(2^{9n/40})$ shared memory units.

Proof. According to the theory of heap [1], one time of deletion and insertion on the heap can be performed with logarithmic time of the size of the heap. Since the heap constructed in algorithm 4 has a size of $2^{9n/40}$ and the combinations of $T_1 + T_2$ have $2^{9n/20}$ elements. It validates the results of lemma 2.

To make the search algorithm perform successfully, we must prepare two queues (heaps) for each processor. As a result, in parallel case, the shared memory must have more memory units than that needed in sequential case.

Combine the above discussions into a whole; we get the final parallel *three-list four-table* algorithm and an overall conclusion on the solution of knapsack-like NP-complete problems.

Algorithm 5: An EREW based parallel three-list four-table algorithm

```
for all processors $P_k$ where $1 \le k \le 2^{n/10}$ do
  1. generate list $C$ and four tables $T_1$, $T_2$ and $T_3$, $T_4$ and sort $T_1$ and $T_3$ in parallel.
  2. construct one min heaps for queue $Q_1$, and one max heaps for queue $Q_2$.
  3. perform algorithm 4.
  4. perform two-list like search algorithm (algorithm 3).
end
```

Theorem 1. n-variable knapsack-like problems can be solved on EREW model in $O(n2^{9n/20})$ time when $O(2^{n/10})$ processors and $O(2^{13n/40})$ memory units are available.

Proof. Producing list C and tables T_2 and T_4 can be finished in n and $2n \times 2^{5n/40}$ time, while tables T_1 and T_3 can be sorted in $4 \times 2^{5n/40}$ time [13]. Each processor will take $2 \times 2^{9n/40}$ time to construct two heaps. Following the lemmas 1 and 2, the total needed time is: $n + 2n \times 2^{5n/40} + 4 \times 2^{5n/40} + 2 \times 2^{9n/20} \times (\frac{9n}{40}) = O(\frac{9n}{40} \times 2^{9n/20})$.

The linear factor has little impact on the time complexity and thus is usually omitted [6-9,14]. So the time complexity of the proposed parallel algorithm is $O(2^{9n/20})$. As for the space complexity, since there are $2^{n/10}$ processors, and each of them need $2 \times 2^{9n/40}$ for the construction of heaps, the total space requirements is $O(2^{13n/40})$. To avoid memory conflicts, at first, we copy the knapsack vector W and scalar M for each processor, which doesn't affect the overall complexity of the proposed algorithm. Thereafter, each processor access and update its own heaps, so it is obvious that all processors have no memory conflicts, and it can be performed on EREW PRAM machine model.

4 Performance comparisons

For the importance of the space complexity [6,15], we adopt the time-space-processor tradeoff (*TSP* tradeoff) [10], as the criterion of evaluation of relevant algorithms.

the *TSP* tradeoff of Karnin's parallel algorithm is $O(2^{5n/6})$ [6]. The number of processor, time complexity, and the *TSP* tradeoff of Ferreira's parallel *three-list* search algorithm in [14] are $O(2^{\beta n})$, $O(n2^{(1-\varepsilon/2-\beta)n})$, $0 \le \beta \le 1-\varepsilon/2$, and $O(n2^n)$, respectively. The parallel algorithm [7] runs in $O(n2^{\alpha n})$ time, $0 \le \alpha \le 1/2$, by allowing $O(2^{(1-\alpha)n/2})$ processors to concurrently access a list of this same size, hence the *TSP* tradeoff of this algorithm is also $O(n2^n)$. Ferreira's parallel *one-list* algorithm [8] bears $O(n2^n)$ *TSP* tradeoff. The performance of Chang et al.'s parallel algorithm [9] is $T = O(2^{n/2})$, $P = O(2^{n/8})$, and $S = O(2^{n/2})$, thus results in a *TSP* tradeoff of $O(2^{9n/8})$. The parallel algorithm Lou and Chang presented had a same performance as Chang et al.'s algorithm. In addition, both of the algorithms in [11] and [13] have a *TSP* tradeoff of $O(2^n)$. From our parallel *three-list four-table* algorithm, one can get a *TSP* tradeoff of $O(9n/40 \times 2^{n/10} \times 2^{13n/40} \times 2^{9n/20}) = O(n2^{7n/8})$.

Among all algorithms that have been published, the *TSP* tradeoff of Karnin's algorithm [6] is the lowest, which is $O(n2^{5n/6})$. However, it has an obvious defect that it can't reduce the execution time even in parallel. In spite of our proposed algorithm is not cost optimal, it go further on the overall time and memory performance than Ferreira's parallel *three-list* algorithm did. Moreover, our algorithm is totally without memory conflicts when different processors access the shared memory.

For the purpose of clarity, the comparisons of the main parallel algorithms published by far for solving the knapsack-like problems are depicted in Table 1. It is obvious that our parallel algorithm outtakes undoubtedly other parallel algorithms in the overall performance.

Table 1. Comparisons of the parallel algorithms for solving the knapsack-like problems

Algorithm	Model	Processor	Time	Memory	*TSP* tradeoff
1 [6]	*CREW*	$O(2^{n/6})$	$O(2^{n/2})$	$O(2^{n/6})$	$O(2^{5n/6})$
2[7]	*CREW*	$O(2^{(1-\alpha)n/2})$	$O(2^{\alpha n})$	$O(2^{(1-\alpha)n/2})$	$O(2^n)$
3[14]	*CREW*	$O(2^{\beta n})$	$O(2^{(1-\varepsilon/2-\beta)n})$	$O(2^{\varepsilon n/2})$	$O(2^n)$
4[8]	*CREW*	$O(2^{(1-\varepsilon)n/2})$	$O(2^{\varepsilon n/2})$	$O(2^{n/2})$	$O(2^n)$
5[9]	*CREW*	$O(2^{n/8})$	$O(2^{n/2})$	$O(2^{n/2})$	$O(2^{9n/8})$
6[10]	*CREW*	$O(2^{n/8})$	$O(2^{n/2})$	$O(2^{n/2})$	$O(2^{9n/8})$
7[11]	*CREW*	$O((2^{n/4})^{1-\varepsilon})$	$O(2^{n/4}(2^{n/4})^{\varepsilon})$	$O(2^{n/2})$	$O(2^n)$
8[13]	*EREW*	$O((2^{n/4})^{1-\varepsilon})$	$O(2^{n/4}(2^{n/4})^{\varepsilon})$	$O(2^{n/2})$	$O(2^n)$
Ours	*EREW*	$O(2^{n/10})$	$O(2^{9n/20})$	$O(2^{n/4})$	$O(2^{7n/8})$

Notation: $0 \le \varepsilon \le 1$, $0 \le \alpha \le 1/2$, $0 \le \beta \le 1-\varepsilon$. The linear factor *n* in algorithms numbered by 1-6 and ours has been ignored for its little impact on the overall performance [6-9,14].

5 Conclusions

A new parallel *three-list four-table* algorithm for solving the knapsack-like problems is presented. Through dynamically producing the elements of the two lists which is to be searched in our *two-list* like search algorithm, we dramatically reduce the space requirements from $O(2^{9n/20})$ in *three-list* algorithm in [14] to $O(2^{13n/40})$. Moreover, the memory conflicts in [14] are also avoided by leave different memory address segment for different processors, permitting the algorithm being able to perform on an EREW machine model. Performance comparisons shows our proposed algorithm greatly outweighs the parallel algorithms presented by far, and thus it is an improved result over the past researches. To our knowledge it is the first time that the knapsack-like problems can be solved without memory conflicts with less than $O(2^{n/2})$ running time when the hardware is also much smaller than $O(2^{n/2})$. Since it can solve problems that are almost 1.5 times as big as those handled by previous algorithms, it has some importance in research of cryptosystem.

References

1. Schroeppel, R., Shamir, A. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. SIAM J. Comput, 1981,10(3):456-464.
2. Chor, B., Rivest, R.L. A knapsack–type public key cryptosystem based on arithmetic in finite fields. IEEE Trans. Inform. Theory, 1988,34(5):901-909.
3. Zhang, B., Wu, H.J., Feng, D.G, Bao, F. Cyptanalysis of a knapsack based two-lock cryptosystem. ACNS 2004, Lecture Notes in Computer Science, Vol. 3089. Springer-Verlag, Berlin Heidelberg New York (2004) 303-309.
4. Horowitz, E., Sahni, S. Computing partitions with applications to the knapsack problem. J. ACM, 1974,21(2): 277-292.
5. Li, K.L, Li,Q.H., Dai, G.M. An adaptive algorithm for the knapsack problem. Journal of Computer Development and Research, 2004,12(7): 1024-1029.
6. Karnin, E.D. A parallel algorithm for the knapsack problem. IEEE Trans, Comput, 1984, 33(5): 404-408.
7. Amirazizi, H.R., Hellman, M.E. Time-Memory-Processor trade-offs, IEEE Transactions on Information Theory, 1988,34(3):505-512.
8. Ferreira, A.G. A parallel time/hardware tradeoff $T \cdot H = O(2^{n/2})$ for the knapsack problem. IEEE Trans. Comput, 1991,40(2):221-225.
9. Chang, H.K.-C., Chen, J.J.-R., Shyu, S.-J. A parallel algorithm for the knapsack problem using a generation and searching technique. Parallel Computing, 1994,20(2):233-243.
10. Lou, D.C., Chang, C.C. A parallel two-list algorithm for the knapsack problem. Parallel Computing, 1997,22(14): 1985-1996.
11. Li, K.L, Li Q.H., Jiang, S.Y. An optimal parallel algorithm for the knapsack problem. Journal of Software, 2003,14(5): 891-896. (in Chinese)
12. Aanches, C.A., Soma, N.Y., Yanasse, H.H. Comments on parallel algorithms for the knapsack problem. Parallel Computing, 2002,28(10): 1501-1505.
13. Li, K.L., Li, Q.H., Li, R.F. Optimal parallel algorithm for the knapsack problem without memory conflicts. Journal of Computer Science and Technology. 2004,19(6): 760-768
14. Ferreira, A.G, Work and memory efficient parallel algorithms for the knapsack problem. International Journal of High Speed Computing, 1995,7(4): 595-606.
15. Woeginger G.J. Space and time complexity of exact algorithms: some open problems. In: R. Downey etc. Proceeding of IWPEC 2004, Lecture Notes in Computer Science, Vol. 3162. Springer-Verlag, Berlin Heidelberg New York (2004) 281–290.