DM811 - Finding maximum stable sets on hypergraphs

October 26, 2008

Contents

1 Introduction and definitions

This report is documentation of the project that serves as the exam of the course DM811: "Heuristics and Local Search Algorithms for Combinatorial Optimization", Department of Mathematics and Computer Science, University of Southern Denmark. The topic of the project i finding large cardinality stable set on k-uniform unweighted hypergraphs. It should be noted that this problem is NP-hard meaning that for large instances it would be inappropriate to use exact methods. Therefore three construction heuristics have been designed and implemented. To improve on these results two local search algorithms have been designed, and both of these have been improved with mechanisms to escape local optima.

All construction heuristics have been run on all the given 40 instances for the seeds 0 to 9 (the non-randomized though have only been run once). The two local search algorithms have been run on all three construction heuristics. It turns out that one of the construction heuristics produces far better results then any other, and because of that only that construction heuristic was tested together with the two improved local search algorithms. These tests were limited run for 300 seconds. All tests have been performed on computers having an Intel Core 2 CPU, 1.86 GHz, 2GB of memory and the Ubuntu Linux distribution version 8.04 with kernel version 2.6.24-21-generic. The implementation has been carried out using java. For running the program small shell-scripts have been written (instructions on how to use this can be found at the end of the report). All graphs are made in R.

A note about terminology: An edge and a vertex are said to be incident if the vertex is contained in the edge. I will be using the term "degree" of a vertex, defined as the number of edges incident to a vertex. Two vertices are said to be adjecent if there is at least one edge containing both vertices. These definitions are also consistent with those of Claude Berge in his book "Hypergraphs"(1989). I will throughout the report use the term "the set" referring to the working set of vertices to be included in the stable set (this might or might not be a feasible solution). I will call an edge "full" if it is a subset of the working set. I will call an edge "critical" if all it's vertices but one are included in the working set (the edge is critical in the sense that adding the last vertex will result in a violation). The "violation" of a working set is the number of edges in the graph fully included in the working set. The violation contribution of a vertex v refers to the change of violation by adding/removing v (nonpositive if v is in working set, and nonnegative if v is not in working set). A stable set is said to be "maximal" when it's not possible to add a vertex and yield a new stable set.

2 Data Structures

Through this whole project I am using a datastructure representing a hypergraph. This datastructure contains the incidence matrix and a boolean array "contained" telling whether or not any vertex is in the set. For the sake of efficiency I use and the following static data:

- For each vertex v, all edges containing v

- For each edges e, all vertices in e
- For each vertex v, all vertices adjacent to v

And the following non-static data is also maintained:

- The sets of vertices included and not-included in the set
- For each edge the number vertices in the edge that is in the set
- For each vertex v it's "violation contribution"
- The total number of edges violating the stable set criterion
- For each vertex v all"critical edges" incident to v
- For each vertex v all "full edges" incident to v

Adding a vertex v to the set requires maintaining the redundant data mentioned above, it is done as shown (algorithm 1). The way to remove a vertex is quite similar and has the same complexity.

Algorithm 1: Adds a vertex v to the set

```
1: set contained[v] ← true, add v to the set of contained vertices, and remove
   v from the set of not-contained vertices
2: for e ∈ Edges incident to v do
3:   if e is now full then
4:     Increment the number of violations
5:     for v' ∈ e do
6:       add e to v's list of incident full edges
7:       remove e from v''s list of incident critical edges
8:     end for
9:   else
10:    if e is now critical then
11:      for v' ∈ e do
12:        add e to v's list of incident critical edges
13:      end for
14:    end if
15:  end if
16: end for
```

Computational Analysis Assume that the graph has n vertices and m edges of size k. The for-loop in line 2 will be entered m times, and because all edges has size k the computational complexity of this procedure will be $O(m \cdot k)$. Using aggregate analysis the complexity of adding all vertices to the set can be determined. It is clear that each edge is incident to k vertices, and because of that the content of the for loop will in total be executed $m \cdot k$ times, meaning that adding all vertices to the set has a complexity of $O(m \cdot k^2)$.

3 Construction Heuristics

In this section three different construction heuristics are presented. The main idea is briefly explained and afterwards a pseudocode scetch is given. For all the construction heuristics an analysis of the computational complexity is given.

3.1 Increasing Degree

It seems as a good idea to start by taking the vertices with low degrees. This is coming from the assumption that taking a vertex with low degree rules out the lowest number of possible vertices in the future. The algorithm then works by iteratively adding the vertex with the lowest degree that can be added without violating the stable set constraint. The algorithm then halts when the stable set is maximal. The pseudocode scetch is shown ("algorithm 2").

Algorithm 2: Increasing Degree Construction Heuristic.

```
1: ∀i ∈ 1..n calculate d[i] ← degree(i)
2: minindex ← 1
3: nrVerticesConsidered ← 0
4: while nrVerticesConsidered < nrVertices do
5:   for j ← 1 to nrVertices do
6:     if d[j] < d[minIndex] then
7:       minIndex ← j
8:     end if
9:   end for
10:  if violationcontribution[minIndex] = 0 then
11:    Add vertex minIndex to set S
12:  end if
13:  d[minIndex] ← ∞
14:  nrVerticesConsicered ← nrVerticesConsicered + 1
15: end while
```

Computational Analysis Let n and m be the number of vertices and edges respectively and let all edges have size k. The first line can of course be done in time $O(n)$ (using the incidence lists stored in the graph data structure). The content of the while-loop starting on line 4 will be entered n times. The content of the foor-loop starting on line 5 will be executed $O(n^2)$ times, and an upper bound for the add on line 11 is obtained by considering the complexity of adding all the vertices - this has complexity $O(m \cdot k^2)$. The total time complexity of this algorithm will then be $O(m \cdot k^2 + n^2)$.

3.2 Remove Worst Vertex Heuristic

This algorithm starts by adding all vertices to the set (this will surely generate an infeasible set), and then iteratively choose the *most violating* until the set is feasible. The "most violating" vertex is the vertex incident to the largest number of fully contained edges. The pseudocode is shown (algorithm 3).

Algorithm 3: Remove Worst Vertex Heuristic

```
1: for i ← 1 to n do
2:    Add vertex i to set
3: end for
4: maxViolVert ← 1
5: while There still is violations do
6:    for j ← 1 ∈ set do
7:       if violations[j] > violations[maxViolVert] then
8:          maxViolVert ← j
9:       end if
10:   end for
11:   Remove vertex maxViolVert
12: end while
```

Computational Analysis Since this algorithm will be nore more expansive than first to add all vertices, and after that remove all vertices, it is clear that an upper bound of the time complexity is $O(n^2 + m \cdot k^2)$ (the n^2 comes from the for-loop in line 6).

3.3 Random generation

The random construction heuristic iteratively selects a random vertex, and checks whether or not it can be added. If it can be added without violations this is done. It halts when the found set is maximal. A pseudocode scetch is shown (algorithm 4).

Algorithm 4: Random Selection Construction Heuristic.

```
1: nrVerticesConsidered ← 0
2: while nrVerticesConsidered < nrVertices do
3:    choose a random vertex v not in set
4:    if violationcontribution[v] = 0 then
5:       Add vertex v to set
6:    end if
7:    nrVerticesConsicered = nrVerticesConsicered + 1
8: end while
```

Computational Analysis Using the same reasoning as with the "IncreasingDegree" it is clear that this construction heuristic has the complexity of $O(m \cdot k^2)$.

3.4 Comparison

I have run my implementation of all the construction heuristics on all the 40 instances. For the random generation heuristic i have used the seeds 0 to 9 (both included). Figure 1 shows the quality of the solutions for each class of instances. From these results it is obvious that the implementation of the "removeWorst" heuristic performs way better then the other two since the boxplots are non-overlapping.

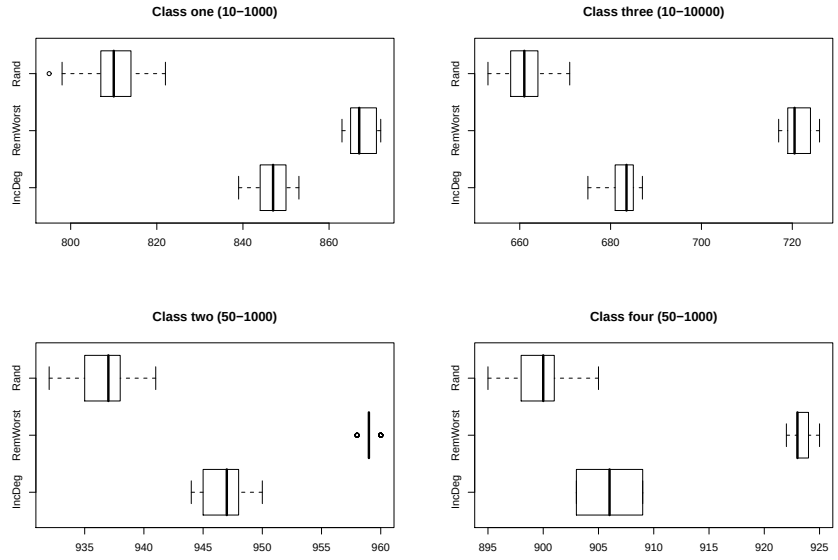

Figure 1: Boxplots showing the qualities of the solutions for the different algorithms for the four different instances

4 Local Search Algorithms

I have devised two local search algorithms, I have called them "addnfix" and "rm1add2".

4.1 Addnfix

This algorithm starts by adding a random vertex (not already included) to the set. This will propably cause one or more edges to be fully included. To try to repair this infeasible solution, I use a local search algorithm, with the "Best improvement" strategy and the "swap neighbourhood" - I use the number of edges fully contained as the evaluation function. This means that the set is a stable set if and only if the evaluation is 0. If the solution could be repaired using only swap-operations a new, and larger, stable set has been found, and this procedure starts again. Otherwise, if at some time no swap operation can reduce the evaluation function, the last feasible set is returned.
A pseudocode scetch of the repairing part is shown (algorithm 5).

Algorithm 5: Addnfix local seach algorithm

```
1: while NumberOfViolations ≠ 0 do
2:   minDelta ← 0
3:   for i ∈ allVerticesInSet do
4:     for j ∈ allVerticesNotInSet do
5:       if delta(j, i) < minDelta then
6:         minAdd ← j
7:         minRemove ← i
8:         minDelta ← delta(j, i)
9:       end if
10:    end for
11:  end for
12:  if minDelta ≥ 0 then
13:    No improvement can be made - halt
14:  else
15:    Add vertex minAdd
16:    Remove vertex minRemove
17:  end if
18: end while
```

Speedup technique Delta evaluation is used so I don't have to calculate all violations in the entire graph, instead I only consider relevant edges. Note that to add vertex v_1 and remove vertex v_2 the gain in violation is the number of critical edges incident to v_1 (to which v_2 is not incident), and the loss of violation is the number of full edges incident to v_2 (to which v_1 is not incident). A pseudocode scetch of the delta evaluation algorithm is shown (algorithm 6).

Algorithm 6: The delta function for algorithm 5
Input: Vertex to add j, and vertex to remove i

```
1: delta ← 0
2: for v ∈ critical edges incident to j do
3:   if i is not incident to v then
4:     delta ← delta + 1
5:   end if
6: end for
7: for v ∈ full edges incident to i do
8:   if j is not incident to v then
9:     delta ← delta − 1
10:  end if
11: end for
12: return delta
```

Computational analysis Let again G be a hyper graph with n vertices and m edges of size k. The size of the neighbourhood is $O(n^2)$ and for all possible swaps the deltaviolation is computed. As mentioned before each vertex will on average be in $\frac{m \cdot k}{n}$ edges. To calculate the delta evaluation for all $i, j = 1..n$ will therefore totally take $O(n^2) \cdot O\left(\frac{m \cdot k}{n}\right) = O(n \cdot m \cdot k)$ (which is clearly dominating the time to insert/remove the two vertices).

4.2 Rm1add2

The idea of this local search algorithm is to check if it's possible to remove one vertex from the set and add two, to obtain a new stable set. The neighbourhood of this function is all possible ways to add two vertices and remove one, meaning that the size of the neighbourhood is $O(n^3)$. This algorithm halts when it is not possible to remove 1 and add 2 and still have a feasible solution. A scetch of the algorithm is shown at algorithm 7.

Algorithm 7: The remove 1 add 2 local search algorithm

```
1:  improvement ← TRUE
2:  while improvement do
3:    improvement ← FALSE
4:    for rmIndex ∈ verticesInSet do
5:      for addIndex1 ∈ vertices not in setadjecent to rmIndex do
6:        for addIndex2 ∈ vertices not in setadjecent to rmIndex do
7:          if feasible(rmIndex, addIndex1, addIndex2) then
8:            add vertices addindex1, addIndex2
9:            remove vertex rmIndex
10:           improvement ← TRUE
11:           Go to line 2
12:         end if
13:       end for
14:     end for
15:   end for
16: end while
```

Speedup techniques: Even though the neighbourhood has size $O(n^3)$ it can in many cases be pruned a lot by using the fact, that if a vertex is removed the only vertices that should be considered for adding are the vertices adjecent to the removed vertex (assuming that the set was already stable). This doesn't alway prune the neighbourhood though - for example in the instances given where there are 10000 edges of size 50 all vertices are adjecent to all vertices meaning that this "reduction" in search space doesn't prune at all.

In algorithm 7 the function "$feasible(rm, add1, add2)$" is used. It returns TRUE if it yields a feasible solution remove vertex rm and insert vertices $add1$ and $add2$. If $add1$ or $add2$ is contained in any critical edge that does not contain rm it will certainly not yield a stable set to insert both of them. Otherwise if $add1$ and $add2$ are incident to the same edge e, and e would be fully contained if $add1$ and $add2$ were added, then if rm is not in e it will yield a non-stable set. If neither of these conditions are satisfied the solution will still be valid. A pseudocode scetch is shown (algorithm 8).

Algorithm 8: Feasible
Input: Vertex to remove rm, and vertices to add $add1$, $add2$.

```
1: for e ∈ critical edges incident to add1 do
2:   if rm ∉ e then
3:     return FALSE
4:   end if
5: end for
6: for e ∈ critical edges incident to add2 do
7:   if rm ∉ e then
8:     return FALSE
9:   end if
10: end for
11: for e ∈ edges incident to add1 and add2 not containing rm do
12:   if e would be fully contained by adding add1 and add2 then
13:     return FALSE
14:   end if
15: end for
16: return TRUE
```

Computational Analysis The neighbourhood has size $O(n^3)$, and the when the "feasible" function is called for all values of $rm, add1, add2$ the average number of edges per vertex will be $\frac{m \cdot k}{n}$, so the total complexity of this algorithm will be $O(n^2 \cdot m \cdot k)$. It is obvious that this dominates the cost to add/remove the vertices.

I should here be noted, that I had some experiments about expanding this algorithm to a "remove2-add3"-strategy, but the neighbourhood (with size $O(n^5)$) seemed too large to make it run in a reasonable time.

4.3 Comparison

For all three construction heuristics I have run the two local search algorithms for the seeds 0 to 9. The boxplots grouped on the four instance classes is shown on figure 2. It can be seen that in most cases "rm1add2" local search algorithm yields slightly better result than the "addnfix", although in all cases the boxplots are very overlapping.

5 Stochastic Local Search Algorithms

Both of my local search algorithms have been extended with escape mechanisms enabling them to find other, and hopefully better, local optima. It should be noted that there is a major difference between type of the two algorithms, the first is working with infeasible solutions, the second doesn't.

5.1 Addnfix Tabu

I have extended the Addnfix local search algorithm in such a way, that the best improvement strategy now makes the best possible swap, even if this swap will increase the value of the evaluation function. When a vertex has been remove

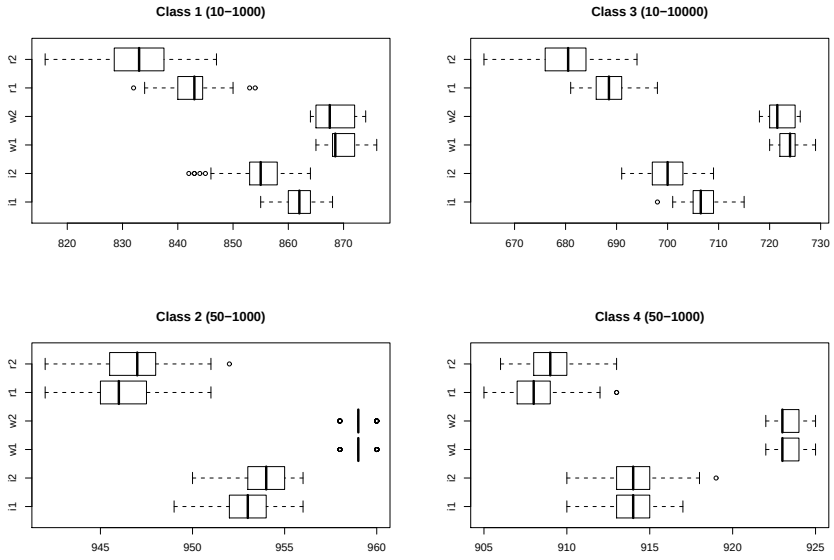

Figure 2: Boxplots of the qualities of the solutions for combinations of the three construction heuristics and the two local searches. i, w and r mean IncreasingDegree, removeWorst and random respectively. 1 and 2 mean rm1add2 and addnfix respectively.

it is a tabu for a number of iterations, this number is selected randomly with a uniform distribution between 0 and 30. The reason why the tabu tenure is chosen in this way is that it seemed to result pretty well in my preliminary tests. During the development I made some experiments to improve the tabu search using the ideas from "reactive tabu search" [1] but it didn't yield any better results in my preliminary tests.

5.2 Rm1Add2 Iterated Local Search

The Rm1Add2 has been improved by the use of Iterated Local Search. When a local optimum is detected diversification is performed by removing 42 random vertices from the set. As with the tabu search I have done some preliminary superficial tests and on that background I have chosen 42.

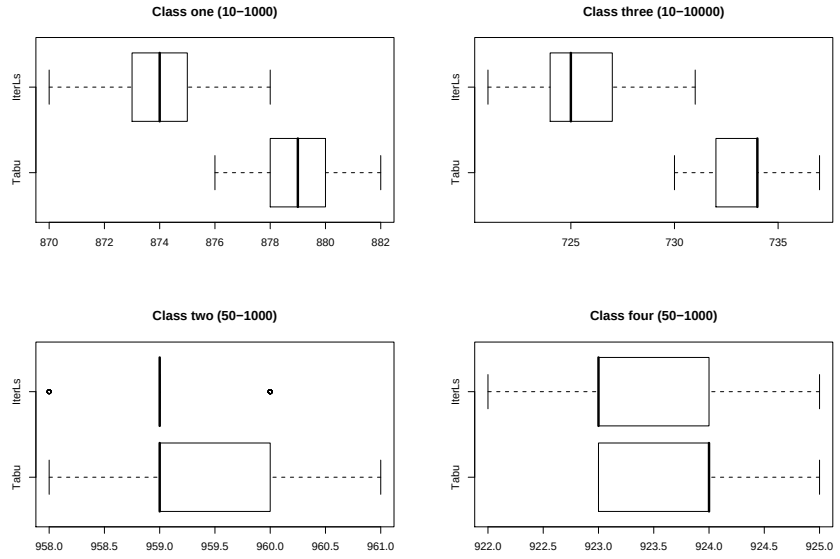

Figure 3: Boxplots showing the quality of the solutions for the implementations of the two algorithms grouped in the four instances

5.3 Results

I have run my implementation of the two previously mentioned algorithms on all instances for 5 minutes for the seeds 0 to 9. Boxplots for the results of the runs grouped by the four instance classes are shown on figure 3. For all the runs I have used the "Remove worst" construction heuristic as it was the heuristic yielding the far best results (see 3.4). For all classes of instances the tabu search performs better than the iterated local search although it should be noted that

[1] Page 11, slide 12 from the course slides

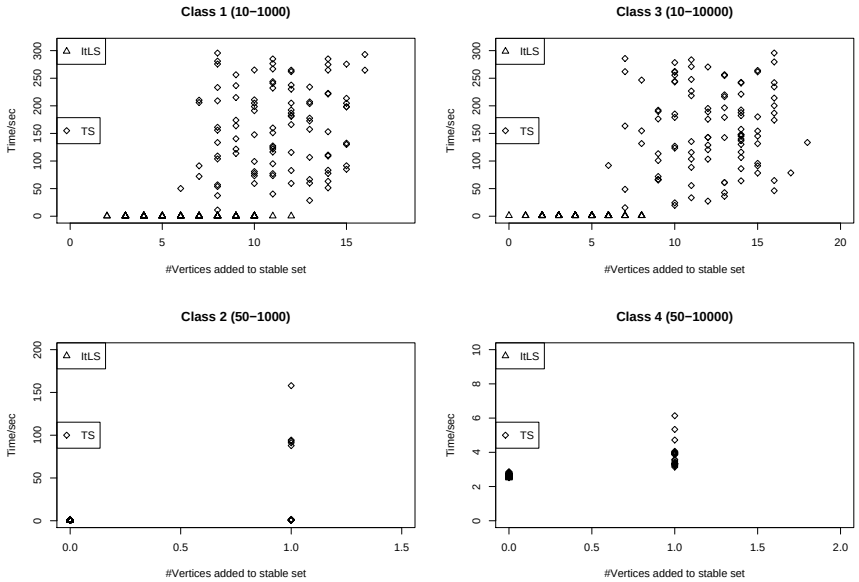

Figure 4: Scatterplot that for each instance shows corresponding values of solution cost and the time it took to reach that solution

there are very little difference between the quality of the results using the two methods for the classes two and four.
I have also plotted corresponding values for solution quality and the time it took to reach that solution on figure 4. It can be seen that for classes 1 and 3 the tabu search continues to improve during the whole run, whereas the iterated local search improves a lot in a very short time and then it is not able to find better solutions after that. For all runs the iterated local search algorithm finds it's best solution after less than 20 seconds.
This suggests that the diversification doesn't work as aimed - maybe because the local search finds the same local optimum again and again.
Generally both of my stochastic local search algorithms behave very poorly on the instance classes 2 and 4. The tabu search is the only of the two search methods making improvements for these instance classes. For instance 4 all improvements are made within 7 seconds and for instance 2 nearly all improvements are made within 100 second. This highly suggests that diversification mechanism doesn't work very well - maybe the tabu search would behave better if the tabu tenure were increased, as it might increase the chance to find solutions "far" from the solution found

Using the runs described above I yield the results showed at table 1, all obtained by the tabu search.

Instance	Best Solution
u-1000-10-1000-01.mss	880
u-1000-10-1000-02.mss	880
u-1000-10-1000-03.mss	881
u-1000-10-1000-04.mss	879
u-1000-10-1000-05.mss	879
u-1000-10-1000-06.mss	882
u-1000-10-1000-07.mss	881
u-1000-10-1000-08.mss	881
u-1000-10-1000-09.mss	879
u-1000-10-1000-10.mss	880
u-1000-10-10000-01.mss	733
u-1000-10-10000-02.mss	735
u-1000-10-10000-03.mss	737
u-1000-10-10000-04.mss	735
u-1000-10-10000-05.mss	736
u-1000-10-10000-06.mss	734
u-1000-10-10000-07.mss	735
u-1000-10-10000-08.mss	735
u-1000-10-10000-09.mss	735
u-1000-10-10000-10.mss	736
u-1000-50-1000-01.mss	960
u-1000-50-1000-02.mss	959
u-1000-50-1000-03.mss	959
u-1000-50-1000-04.mss	960
u-1000-50-1000-05.mss	960
u-1000-50-1000-06.mss	961
u-1000-50-1000-07.mss	960
u-1000-50-1000-08.mss	959
u-1000-50-1000-09.mss	959
u-1000-50-1000-10.mss	959
u-1000-50-10000-01.mss	925
u-1000-50-10000-02.mss	923
u-1000-50-10000-03.mss	924
u-1000-50-10000-04.mss	924
u-1000-50-10000-05.mss	924
u-1000-50-10000-06.mss	923
u-1000-50-10000-07.mss	924
u-1000-50-10000-08.mss	925
u-1000-50-10000-09.mss	923
u-1000-50-10000-10.mss	925

Table 1: For each instance the best result archieved.

6 Concluding remarks

Three construction heuristics have been presented and tested, and it turns out that on all tested instances the "Remove Worst" construction heuristic outperforms the other two in quality. Two local search algorithms have been presented, there are really no big difference in the quality of the solutions, though both of them perform best using the "Remove Worst" construction heuristic. These two local search algorithms have been improved with a tabu search and an iterated local search, respectively. It turns out that for 300 seconds fixed times runs, the tabu search outperforms the iterated local search in terms of solution quality. It should be noted that no of my local search algorithms are able to improve much the solution found by construction heuristic in the cases where the edges have size 50. Maybe a parameter tuning or improvement of the tabu search strategy would be able to improve on these result. Maybe it is because the solutions found really are so close to optimal that one shouldn't expect to find a better solution using nonexact methods.

7 How to run the program

A script "mss" has been written. It works as follows:

```
mss -i infile -t time -s seed -o outfile -tt val -ch chNr -ls lsNr
```

Where `infile` specifies the instance file, `outputfile` specifies the file where the found solution should be written. `tt` sets the tabutenure (this parameter is of course relevant for the tabu search). Ch is the construction heuristic number. They are numbered as follows: 1) increasingDegree, 2) removeWorst, 3) random. lsNr specifies which local search algorithm is to be used. They are numbered as follows: 1) "rm1add2", 2) "addnfix", 3) "addnfix-tabu", 4) "rm1add2IteratedLocalSearch".