MSS on Hypergraphs

DM811 Exam Project

October 26, 2008

Professor: Marco Chiarandini

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

Contents

MSS on Hypergraphs

DM811 Exam Project

Sven Simonsen

October 26, 2008

Professor: Marco Chiarandini

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

1 Introduction

This project deals with the uniform and unweighted version of the maximum stable set problem on hypergraphs.
It looks at heuristic methods that aim to find good candidate solutions fast and analyses aspects of the problem in an attempt to improve said methods both in matters of speed and solution quality. Implementations of the methods will be described and competing methods will be compared.

The heuristics discussed are the following:

MSS on Hypergraphs

A construction heuristic that starts with the full set of the hypergraph and removes vertices randomly until the set is stable.

A construction heuristic that starts with an empty set and adds vertices randomly while maintaining a stable set. This is done until the set is maximal.

A local search heuristic that tries to improve a maximal stable set by exploring the neighborhood of (1,x)-swaps.

An improvement of this local search that exploits some observations about the problems structure.

DM811 Exam Project

Sven Simonsen

A random restart stochastic local seach method.

October 26, 2008

An iterated local search method where escape steps are taken by adding vertices to the solution, restabilise and then remaximilise.

The instances used have been renamed to avoid clutter on plots. u-1000-10-1000 will be called T1 (for Type 1), u-100-10-10000 is T2, u-1000-50-1000 is T3 and u-1000-50-10000 is T4. The instance u-1000-50-1000-05.mss would thus be called by the shorter name T3-05 (for Type 3 instance 5).

Professor: Marco Chiarandini

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

2 Construction heuristics

2.1 Some basic observations

Choosing as many vertices of degree zero or one as possible to be part of the solution is never a bad choice, since removing such vertices only enables us to add max 1 other vertice to the solution which is no improvement of the solution and can not lead to one either. Armed with this insight we can make this greedy choice once for any given instance and save it as a basis for later restarts of the heuristic.

If you want to maintain a stable set adding a vertex to the candidate solution can never allow new additions of vertices that were not available before.

MSS on Hypergraphs

DM811 Exam Project

Sven Simonsen

October 26, 2008

2.2 ConstructA

A first cursory look at the problem suggested checking whether a vertex could expand your candidate solution without destabilizing it would be costly, so I tried looking at the problem from the opposite direction. Since a stable set is a set containing no complete edge, why not start with the set of all vertices and remove vertices until stability is reached.

ConstructA does this by randomly choosing an edge fully contained in the candidate solution and removing one of it's vertices from the solution. Since at least one edge is satisfied in every move we are sure that constructA will always reach stability, at which point the algorithm terminates.

To efficiently find the vertices in an edge and the edges a given vertex is part of the algorithm works with an array of edges containing the list of all vertices in that edge for every edge and likewise an array of vertices containing the edges that a vertex is part of. These data structures do not change and are thus only computed once. During every run a list of fully contained edges is maintained and consulted when a new edge has to be chosen randomly.

Professor: Marco Chiarandini

Pseudo code for constructA:

Data: An array of lists: edges
An array of lists: vertices
Result: A list of vertices: solution
Start
Create list of all vertices: solution;
Create list of fully contained edges: fullE;
while *fullE is not empty* **do**
  comp ← a random vertex from a random edge in fullE;
  remove comp from solution;
  for *All edges in vertices[comp]* **do**
    remove the edge from fullE;
  end
end
return solution;

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

This algorithm takes O(|E|) time to complete with the right implementation since it looks at every edge exactly once and subsequently removes it from further consideration.

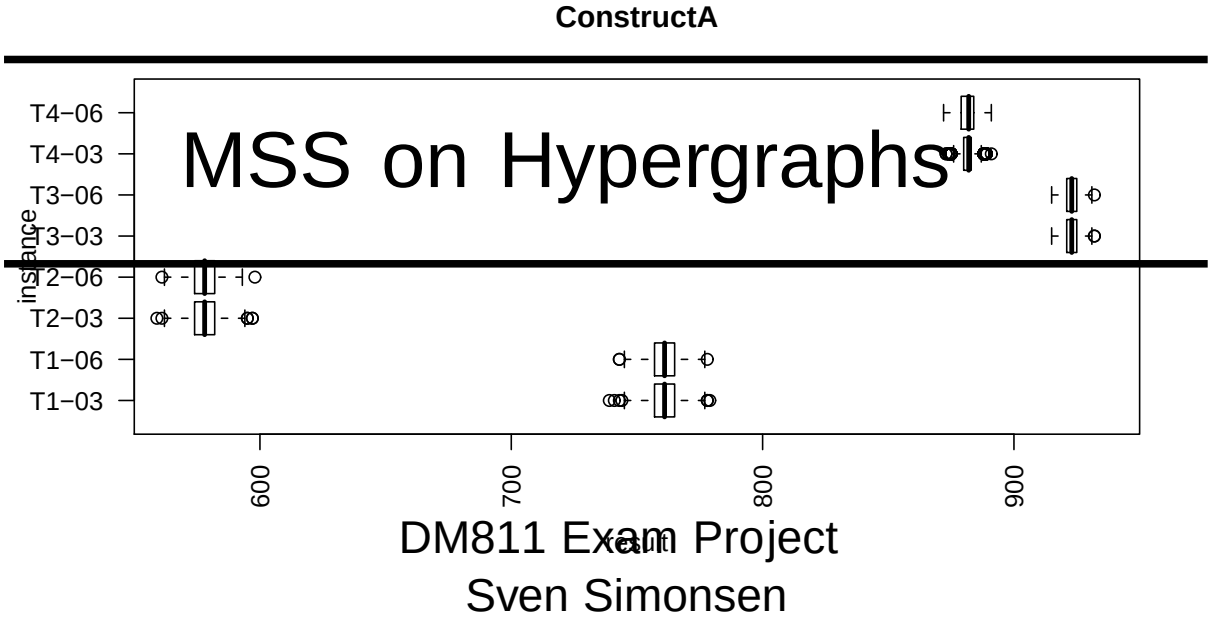

Figure 1: ConstructA's performance on sample instances with 500 iterations per seed and 2 seeds.

October 26, 2008

We see it is an efficient way of getting first useful results, it beats the SCIP on most Types of instances taking under one second running time. But the method is open to improvement since found solution are often not even maximal. It is also noteworthy that constructA has the theoretical possibility to hit every maximal stable set in any given instance.

Professor: Marco Chiarandini

2.3 Construct1

Since constructA did not tend to terminate in maximal stable sets there was definitely room for improvement, so I went back to the initial idea of adding vertices until the cadidate was maximal. Checking from scratch if an edge becomes unstable when a vertex is added would take O(b) time (where b is the common edge size in the current instance) and this would have to be done for every edge that contains the vertex in question.

Instead we store the amount of vertices from the solution that belong to a given edge in an array and update this information every time we add a new vertex to our solution candidate. Now when we try if a vertex can expand our solution we just have to look these values up for every edge it is contained in, if none of these is critical we can safely add the vertex to our solution and update values accordingly. Alternatively one can maintain a list of valid vertices and remove every vertice in an edge from this list as soon as the edge becomes critical, this is what I did in my implementation.

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

Using these structures the algorithm for construct1 would look like this:

Data: An array of lists: edges
An array of lists: vertices
Result: A list of vertices: solution
Start
Greedily add all vertices of deg<2 to solution;
Create an array of values for the edges: valueE;
Create a list of vertices we are free to choose: freeV;
while *there are still vertices in freeV* **do**
  vert ← random vertex from freeV;
  Add vert to solution;
  for *All edges in vertices[vert]* **do**
    increase valueE[edge] by one;
    if *The edge is critical (check valueE)* **then**
      Remove all vertices in edges[edge] from freeV;
    end
  end
  Remove vert from freeV;
end
Return solution;

Here we are looking at running the while loop O(|V|) times checking $b * |E|/|V|$ in average per iteration and removing one vertex. Taking a closer look we see that we will check every edge at most b times or once per vertex in it, so we look up edges O(b|E|) times. In total this results in O(|V|+b|E|) running time for construct1.

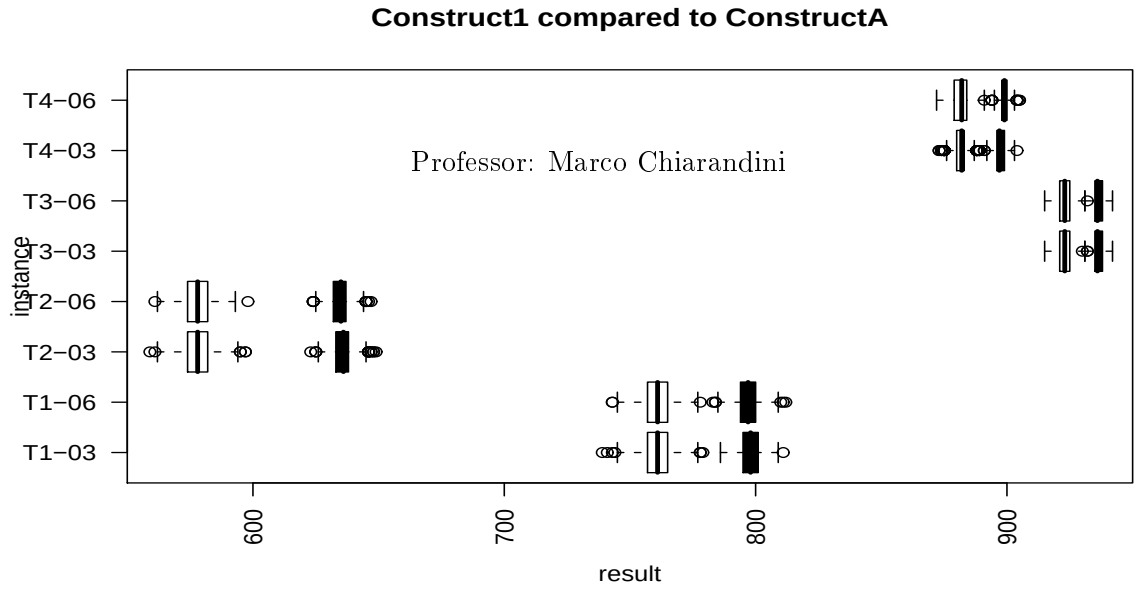

Figure 2: Construct1 (shown in black) compared to ConstructA on the same samples with 1000 iterations per instance.

Results are always maximal which could be the reason that Construct1 beats constructA by a considerable margin. The single runs are a little slower than with constructA (ConstructA can do 500 iterations in under a second even on Type 4 instances while Construct1 can "only" do 100 per second on those) but overall fast enough to be used as a basis for local search. This is especially true since even single runs of Construct1 tend to beat 100 iterations of ConstructA. Like with ConstructA we observe that Construct1 can result in every maximal set in the instance. This is a nice property when one wants to use it as a basis for random restarts of local search, since we avoid always ending up in the same local optima.

3 Local search

3.1 More observations

The neighborhood we will be looking at is that of a (1,x)-swap, meaning you remove one vertex from the solution and then try to add as many of the adjacent vertices to the solution as possible. We will soon see why it is efficient to always maintain a maximal stable set when working with this kind of neighborhood.

3.2 Local1

When pondering a local search heuristic for MSS on hypergraphs I realised that the main difference to normal graphs were the many stages an edge could be in. In normal graphs an edge is either empty or critical, either of which can be checked in constant time allowing quick assessment of different swaps. This is more complicated when dealing with hypergraphs. Assuming we want to look at if removing a vertex from the solution can lead to improvement, what we have to do is count the adjacent vertices that could be added once we remove the vertex we are checking. Bear in mind that depending on instance the list of adjacent vertices could be quite long and to check if their addition would be valid we have to count their adjacent vertices as well.

Since we are ultimately only interested in maximal stable sets, as these are strictly better candidates, we can safely restrict our local search to start with and maintain a maximal stable set. This restricts the possible situations any local search step can start in, to a smaller set with some nice properties that can be exploited to remove the need to look up $\Omega(b^2)$ vertices every time we wish to evaluate a remove. We will again keep track of the amount of vertices from the solution that belong to a given edge.

Since we maintain a maximal stable set we will only have to consider the effect on edges that have exactly one open position prior to our move. We will call these edges critical and since we are keeping track of edge value they are easy to identify. If an edge is not critical we are certain that it is not this edge that prevents the vertices in it from being added to the solution, so since we maintain maximality we don't have to check any vertex from this edge and can safely move on to the next.

Local1 implements this reasoning with a first improvement strategy, the pseudo code looks like this:

MSS on Hypergraphs

DM811 Exam Project

Sven Simonsen

October 26, 2008

Professor: Marco Chiarandini

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

Data: An array of lists: edges
An array of lists: vertices
A list of vertices: solution
Result: A list of vertices: solution
Start
Create an array of edge values from solution: valueE;
while *There are still viable vertices* **do**
  vert ← random viable vertex;
  improve ← 0; **for** *All edges that contain vert* **do**
    if *valueE[edge] = edge.size-1* **then**
      for *All vertices in edge* **do**
        if *vertex can be added to solution once vert is removed* **then**
          improve+1;
        end
      end
    end
  end
  if *improve > 1* **then**
    remove vert from solution;
    maximize solution by adding neighbors of vert;
    update valueE;
    mark all vertices in solution as viable;
  end
  else
    mark vert as not viable;
  end
end
return solution;

A call to Local1 starts by creating the correct edge value array wich takes O(b|E|) time. Afterwards the while loop is run until we reach a local maxima, how often this happens is hard to approximate but we can take a closer look on the runtime of a single improvement consisting of at most O(|V|) iterations. Under these iterations we consider every vertex of the solution as removal candidate once and then have to look at all it's neighbours, but since every edge can only be taken into consideration b times at most O($b^2|E|$) vertices are checked for addition to the solution. Checking if a vertex can be added takes constant time times it's degree O($|E|$), because we use the edge value array. Finding an improvement thus takes O($|V|b^2|E|^2$) time while updating the arrays after the (1,x)-swap takes O($|E|^2$) time. While these bounds are not very tight they show us that one local improvement is done in polytime in other words we have a PLS-problem.

3.3 Local2

It is obvious that local1 does some redundant checking of possible removes so we would expect a speedup if we could identify exactly which vertices need to be rechecked after a (1,x)-swap. The x vertices added to the solution don't open up new possibilities so we concentrate on the vertex that was removed.

Edges that are not critical after the move give us no new opportunities, but the only ones that contain the removed vertex and remain critical are those that where critical before and contain a new vertex added in the move. Vertices in these edges don't change status either as the removed vertex can not be inserted into the solution again without removing one vertex from every critical edge that holds one of the x added vertices. In fact not even the added vertices should be considered for removal as vertices adjacent to them would have been added before if this was possible (with the exception of the recently removed vertex, but that would require to remove all new vertices).

So what is then really to reconsider? Well the situation changed to the better for some vertices adjacent to the removed vertex, vertices that we tried to add but could not without violating stability. These vertices are now contained in fewer critical edges than before the move and could maybe be added in a later move where one of the vertices adjacent to them is removed from the solution. Because of this vertices in the solution that are adjacent to vertices we tried but were unable to add during our move should be reconsidered for removal.

Improving the code for local1 we arrive at the following pseudo code:

DM811 Exam Project
Sven Simonsen

October 26, 2008

Professor: Marco Chiarandini

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

```
Data: An array of lists: edges
An array of lists: vertices
A list of vertices: solution
Result: A list of vertices: solution
Start
Mark all vertices in solution as viable;
while There are still viable vertices do
    vert ← random viable vertex;
    create blank list of vertices: noImp;
    improve ← 0; for All edges that contain vert do
        if valueE[edge] = edge.size-1 then
            for All vertices in edge do
                if vertex can be added to solution once vert is removed then
                    improve+1;
                end
                else
                    add vertex to noImp;
                end
            end
        end
    end
    if improve > 1 then
        remove vert from solution;
        mark vert as not viable;
        maximise solution by adding neighbors of vert;
        for all vertices in noImp do
            mark all vertices adjacent to vertex as viable;
        end
    end
    else
        mark vert as not viable;
    end
end
return solution;
```


Experimental analysis shows the improvements to be considerable given the right instance type(see section 4.2).

4 Stochastic local search methods

4.1 Random restart (RSS)

The easiest stochastic local search strategy at this point is to run Construct1 hand the result to Local2 and after a local maximum is found restart. This random restart is not an inherently bad strategy, but it's problem is that it does not exploit any of the information gathered by previous runs and thus does many redundant calculations. On the other hand the strategy has per definition no problems with getting stuck in local optima.

4.2 Iterated local search (ILS)

We are already generating pretty good results with our random restart strategy. To increase effectiveness even further we look to more elaborate stochastic local search methods. Assuming we want to keep our general framework, specifically a local search that operates on maximal stable sets only, limits the spectrum of methods at our disposal slightly. We have little opportunity to fiddle with the objective function, as worsening moves must be executed very carefully to not break the maximality of a solution, and random steps in the (1,x)-swap neighborhood are unlikely to help us escape the local maximum.

What we will do instead is force a number of vertices into the solution that were not part of it, then restabilise by removing vertices adjacent to the ones added by force (being careful not to remove the vertices just added) and finally using a variant of Construct1 to make the set maximal. So our perturbation mechanism makes a move in a (x,k+y)-swap neighborhood where we specify k (the number of vertices to add) and x and y are chosen by the following cleanup operations. Given the relatively controlled structure of this maneuver we can maintain our list of edge values throughout increasing the speed of the following call to local2. Moreover we are certain that we would need at least k (1,x)-swaps to return to the original situation by removing the new vertices one at a time.

Pseudo code for Forced:

Data: An array of lists: edges
An array of lists: vertices
The number of vertices to add: factor
A list of vertices: solution
A list of edge values: ValueE
Result: A list of vertices: solution
A list of edge values: ValueE
Start
Create list of vertices not in solution: notV;
Create list of vertices: chosen;

Choose max(notV.size, factor) vertices from notV and add to Chosen;
for *All vertices in Chosen* **do**
  add vert to solution;
  for *All edges that contain vert* **do**
    increase valueE for this edge;
  end
end
for *All edges that have value[edge]=edge.size* **do**
  choose a random vertex in edge that is not in Chosen;
  remove the random vertex from solution;
  for *All edges that contain the random vertex* **do**
    increase valueE for this edge;
  end
end
Run a variant of Construct1 that starts with solution and expands to maximality while maintaining valueE;
return solution and valueE;

Of course if the number of vertices forced into the solution was too small we risk for local2 to simply undo the additions in favor of adding those vertices that we just removed and we would end up in the same local maximum. On the other hand adding large quantities of vertices prevents reuse of already optimised parts of the hypergraph and consequently brings us closer to a random restart in terms of computation time and expected solution quality. We return to this matter in our experimental analysis of different settings for this factor.

5 Experiments

5.1 Test1

The first test runs ConstructA and Construct1 a sampling of instances. The goal with this test was to compare average solution quality of the two constructors to decide which of them would be more suited to be the basis of a local search.

Instances tested against where T1-03, T1-06 , T2-03, T2-06, T3-03, T3-06, T4-03 and T4-06. We thus take a sampling of every instance type. Every instance was tested with seeds 1 and 2 running 500 iteration per seed.

Calls looked like this:
"java dm811e/ConstructA -i Instances/u-1000-10-10000-06.mss -o Output1/CA-u-1000-10-10000-06-s1.out -aux Auxput1/CA-u-1000-10-10000-06-s1.aux -t 500 -s 1"
Every candidate solution was saved to the parameter set with "-aux file" when encountered. CPU-time spent on the local searches was also saved for each run to enable time comparisons.

The results show Construct1 clearly outperform ConstructA in matters of solution size while the time beautifully reflects the analysis of computational cost. We recall ConstructA runs in O(|E|) time while Construct1 has a runtime of (|V|+b|E|). Now in the tests ConstructA is only slightly faster on instance types 1 and 2 but significantly faster on Type 1, but when the edges grow bigger it wins ground and at the end is about 5 times as fast as Construct1 on instance type 4 where edges dominate the picture.

5.2 Test2

The second test runs Local1 and Local2 on the same 8 sample instances with seeds 1 and 2 given 1 minute per run. The goal of this test was to find out if the improvements done from Local1 to Local2 would give a noticeable increase in computation speed. To get a starting point to search from both algorithms call Construct1 prior to every search iteration

Calls looked like this:
"java dm811e/Local1 -i Instances/u-1000-10-10000-06.mss -o Output2/L1/u-1000-10-10000-06-s2.out -aux Auxput2/L1/u-1000-10-10000-06-s2.aux -t 60 -s 2"
We are only interested in number of iterations done in the given time period so we save this

information to the aux file, specified with "-aux file", at the end of each run.

The test yielded these results:

Instance	iter Local1	iter Local2	improvement factor
T1-03	4993	11338	2.27
T1-06	4942	11036	2.23
T2-03	1059	2281	2.15
T2-06	1082	2333	2.16
T3-03	5805	5976	1.03
T3-06	5636	5514	0.98
T4-03	718	827	1.15
T4-06	729	842	1.16

The picture is quite clear. For instances with small edges we see considerable improvement yielding twice as many iterations in the same time frame, but on instances with big edges we see only marginal improvement. In fact in instances of type 3 where edge size is highest and edge number is lowest among instance types we deal with, it seams like the inflection point is reached where the extra computations necessary to maintain the list of valid vertices outweigh the gain.

As a result of these tests I chose to use Local2 in further searches in the (1,x)-swap neighborhood since it did perform notably better than Local1 most of the time and pretty equal in the worst case.

5.3 Test3

Now we want to find out what parameter to use in Forced when dealing with particular instance types. The test designed for this purpose runs RSS with Construct1 and Local2 as a base and ILS with parameters (1,2,3,4,7,10,15) for evaluation. The same sampling of instances from the 4 types is used and we run with seeds 1 and 2. Time limit was 5 min.

Calls looked like this:
"java dm811e/Forced -i Instances/u-1000-50-1000-03.mss -o Output3/ILS3/u-1000-50-1000-03-s1.out -aux Auxput3/ILS3/u-1000-50-1000-03-s1.aux -t 300 -s 1 -f 3"
Where "-f k" sets the parameter of forced to k.

One could argue that since the seed directly determines the start point, because Construct1 is only run once per call, more seeds should be tested. Given a general lack of time I decided this was not feasible and the results indicate it not to be absolutely necessary since none of the more intense searches got stuck in low local maxima.

	RSS	ILS1	ILS2	ILS3	ILS4	ILS7	ILS10	ILS15
T1-03-s1	847	870	870	869	866	863	861	859
T1-03-s2	847	870	869	867	867	864	861	858
T1-06-s1	849	874	871	871	869	865	864	860
T1-06-s2	849	870	872	871	869	866	862	861
T2-03-s1	679	707	705	702	699	694	690	688
T2-03-s2	678	707	708	703	699	696	691	688
T2-06-s1	679	706	707	703	698	694	692	692
T2-06-s2	680	709	708	703	700	695	691	689
T3-03-s1	952	956	955	954	954	954	953	953
T3-03-s2	952	956	956	954	954	954	953	953
T3-06-s1	951	956	955	955	955	954	953	953
T3-06-s2	951	957	955	954	954	954	953	953
T4-03-s1	912	916	915	914	915	914	913	914
T4-03-s2	913	916	915	914	914	914	914	913
T4-06-s1	913	916	915	915	914	914	914	914
T4-06-s2	913	916	915	915	914	914	914	914

Table 1: Solutions for Test3

Solutions are generally very close in size and outperform RSS nicely. There seems to be a trend favoring low parameters but this is hard to see so we will take a look at solution ranks to get a better insight.

	RSS	ILS1	ILS2	ILS3	ILS4	ILS7	ILS10	ILS15
T1-03-s1	15	1	1	4	8	10	11	13
T1-03-s2	15	1	4	6	6	9	11	14
T1-06-s1	15	1	3	3	7	10	11	14
T1-06-s2	15	6	2	3	7	9	12	13
T2-03-s1	15	2	4	6	7	10	12	13
T2-03-s2	16	2	1	5	7	9	11	13
T2-06-s1	16	4	3	5	8	10	11	11
T2-06-s2	15	1	2	5	7	9	13	14
T3-03-s1	15	1	4	5	5	5	10	10
T3-03-s2	15	1	1	5	5	10	10	10
T3-06-s1	15	2	3	3	3	7	11	11
T3-06-s2	15	1	3	7	7	7	11	11
T4-03-s1	16	1	3	6	3	6	13	6
T4-03-s2	13	1	3	6	6	6	6	13
T4-06-s1	15	1	3	3	7	7	7	7
T4-06-s2	15	1	3	3	7	7	7	7

Table 2: Solution ranks for Test3 (ties are treated as min value)

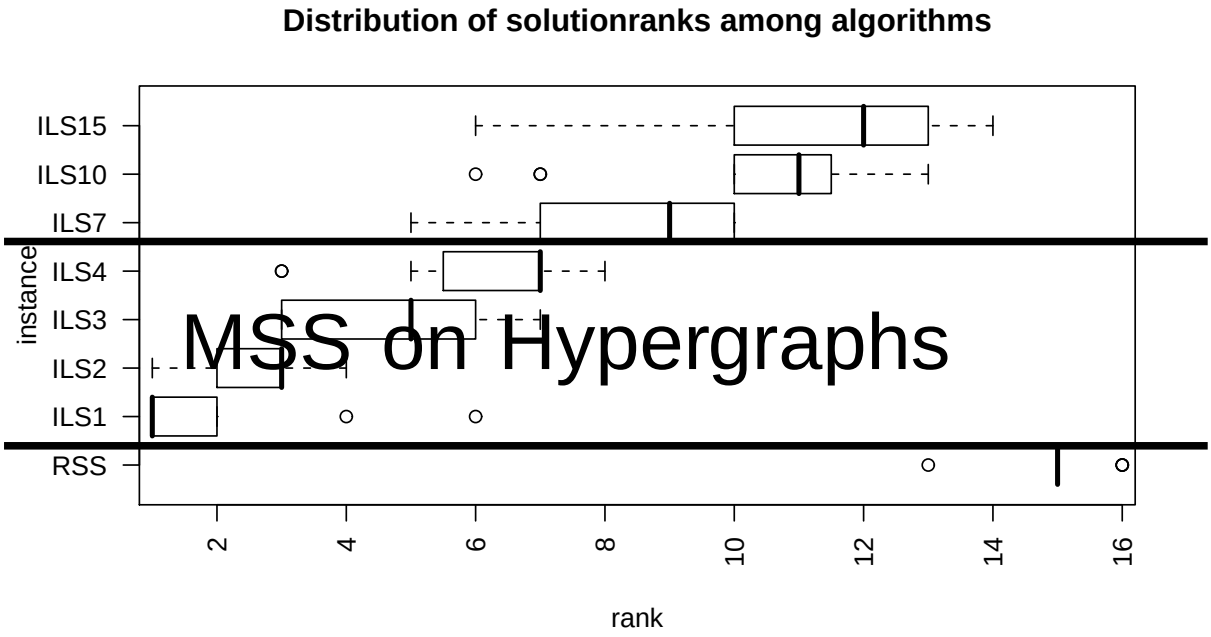

Figure 3: Solutions ranked per instance.

Surprisingly the plot indicates that setting the parameter as low as 1 or 2 would be the optimal choice regardless of type. Apparently even forcing only one new vertex into the solution can be enough to escape from local maxima of the (4,x)-swap-neighborhood. The reason for this slightly odd result can presumably be found in the cleanup following a forced addition of vertices. Because even adding 1 vertex can result in a cascading effect when the vertex is part of many saturated edges and thus many vertices have to be removed. This then can open up many opportunities for maximisation of the set resulting in a (x,k+y)-swap with large y-term.

Given it's structure ILS1 is also the fastest of the flok, since it shuffles working solutions the least, resulting in a strong case in favor of chosing low parameter settings for all 4 types of instances.

6 Conclusion

Since ILS with parameter k=1 came out slightly ahead on instances with big edges and shared the top spot on instances with small edges I chose to calculate my final result table with this heuristic. Seed was set to 1 except where I knew of a better solution from a prior runs with seed 2.

Instance	Instancefile	Runtime sec	Seed	Solution
T1-01	u-1000-10-1000-01.mss	300	1	871
T1-02	u-1000-10-1000-02.mss	300	1	869
T1-03	u-1000-10-1000-03.mss	300	1	870
T1-04	u-1000-10-1000-04.mss	300	1	869
T1-05	u-1000-10-1000-05.mss	300	1	869
T1-06	u-1000-10-1000-06.mss	300	1	874
T1-07	u-1000-10-1000-07.mss	300	1	868
T1-08	u-1000-10-1000-08.mss	300	1	871
T1-09	u-1000-10-1000-09.mss	300	1	868
T1-10	u-1000-10-1000-10.mss	300	1	870
T2-01	u-1000-10-10000-01.mss	300	1	705
T2-02	u-1000-10-10000-02.mss	300	1	709
T2-03	u-1000-10-10000-03.mss	300	1	707
T2-04	u-1000-10-10000-04.mss	300	1	708
T2-05	u-1000-10-10000-05.mss	300	1	709
T2-06	u-1000-10-10000-06.mss	300	1	709
T2-07	u-1000-10-10000-07.mss	300	1	709
T2-08	u-1000-10-10000-08.mss	300	1	709
T2-09	u-1000-10-10000-09.mss	300	1	708
T2-10	u-1000-10-10000-10.mss	300	1	709
T3-01	u-1000-50-1000-01.mss	300	1	955
T3-02	u-1000-50-1000-02.mss	300	1	956
T3-03	u-1000-50-1000-03.mss	300	1	956
T3-04	u-1000-50-1000-04.mss	300	1	955
T3-05	u-1000-50-1000-05.mss	300	1	956
T3-06	u-1000-50-1000-06.mss	300	1	957
T3-07	u-1000-50-1000-07.mss	300	1	955
T3-08	u-1000-50-1000-08.mss	300	1	956
T3-09	u-1000-50-1000-09.mss	300	1	956
T3-10	u-1000-50-1000-10.mss	300	1	955
T4-01	u-1000-50-10000-01.mss	300	1	915
T4-02	u-1000-50-10000-02.mss	300	1	916
T4-03	u-1000-50-10000-03.mss	300	1	916
T4-04	u-1000-50-10000-04.mss	300	1	915
T4-05	u-1000-50-10000-05.mss	300	1	916
T4-06	u-1000-50-10000-06.mss	300	1	916
T4-07	u-1000-50-10000-07.mss	300	1	916
T4-08	u-1000-50-10000-08.mss	300	1	916
T4-09	u-1000-50-10000-09.mss	300	1	916
T4-10	u-1000-50-10000-10.mss	300	1	917

Table 3: Solutions achieved with ILS1

MSS on Hypergraphs

DM811 Exam Project

Sven Simonsen

October 26, 2008

Professor: Marco Chiarandini

Department of Mathematics and Computer Science
University of Southern Denmark, Odense