A title

A sub-title

02715 - Large Scale Optimisation, F04

Albert Einstein – "study number" Niels Bohr – "study number"

March 7, 2011

Teacher: Thomas Stidsen

Technical University of Denmark

Contents

1 Introduction

A short introduction where the algorithm and problems are mentioned/introduced.
A short description of the individual sections, ex:

The remaining of this report is organised as follows. In section 2 the Set Covering Problem is defined. In section 3 the two the basic GRASP algorithm for the SCP, as described by Feo and Resende ([1] and [2]), is introduced. In section three a Lagrangian pricing scheme suggested by Caprara, Fischetti and Toth, (CFT), [4] is integrated in the basic GRASP algorithm of section two. In section four we investigate some features found in PROGRES, [3], and discuss their applications in our GRASP implementation. Results are presented at the end of each section. The performance of the different algorithms is statistically compared in section five. We conclude the report in section six. We have chosen 45 scp problems from Beasley's OR library, [8]. 9 of these are used as the training set and 36 as the test set.

2 Problem Formulation

Description of the problem(s), ex:

The Set Covering Problem (SCP) can be defined as follows. Given an $m \times n$ matrix, A, with elements $a_{ij} \in \{0, 1\}$, let the set of rows be given as $M = \{1, \ldots, m\}$ and the set of columns as $N = \{1, \ldots, n\}$. We say that a column $j \in N$ covers a row $i \in M$ if $a_{ij} = 1$. To each column is associated a cost c_j, $j \in N$. The problem is now to select a minimum cost set of columns $S \subseteq N$ such that each row $i \in M$ is covered. Put formally this can be written as follows:

$$\min \sum_{j \in N} c_j x_j \tag{1}$$

$$\text{s.t.} \sum_{j \in N} a_{ij} x_j \geq 1, \qquad \forall i \in M \tag{2}$$

$$x_j \in \{0, 1\},\ j \in N\ , \tag{3}$$

where $x_j = 1$ if column j is in S and $x_j = 0$ otherwise. Furthermore we will in the following refer to the following sets:

$$J_i = \{j \in N \ \mid a_{ij} = 1\}\ ,$$

which is the set of columns covering row i. We also define:

$$I_j = \{i \in M \ \mid a_{ij} = 1\}\ ,$$

to be the set of rows covered by column j. This will prove its usefulness when we describe the different algorithms.

3 Metaheuristic Algorithm description

> *Introduce the algorithm* **This section is very very important**. *Ex.*

This section will contain a treatment of the overall metaheuristic known as Greedy Randomised Adaptive Search Procedure (GRASP), [1].
The general idea of this algorithm is to extend the well known greedy[1] algorithm with an element of randomness. This enables us to search a larger area of the solution space by including what seems to be poor columns at first. The concept of adaptiveness is introduced such that the choices made by the algorithm depend on the current state of the solution.

3.1 The Algorithm

> *Describing algorithms is difficult. Using pseudo code is often a good idea. Give a brief overall description and then concentrate on the important parts of the algorithm. Be carefull not to include too much description, i.e. I do not need a carefull description of how you got the problem data into the algorithm* GetInputProblem *below. You may briefly describe the standard components and then concentrate on the non-standard parts.*

The framework of the GRASP heuristic is quite simple. Consider the pseudo code presented in Algorithm 1. First we make a greedy randomised solution in line 3. In general, GRASP seeks to improve the result found by *MakeGreedyRandomizedSolution* through a local search, as indicated in line 4. However we will not use local search in our implementation as our primary aim is to achieve a good solution fast. Since the local search can be quite time consuming, and often offers only small improvements to the randomised greedy solution we will not pursue this subject in the rest of this report.

Now looking a little closer at line 3 in Algorithm 1 we can elaborate on the MakeGreedyRandomizedSolution() function. Pseudo code for this function can be seen in Algorithm 2. In lines 2 and 3 we initialise the problem. Lines 4–10 is the actual solution construction.

[1] Using a greedy strategy all columns are assigned a utility, given their prices and number of rows covered by them, at any given iteration. At each iteration the column with the greatest utility is chosen until all rows are covered.

Algorithm 1 GRASP

```
1: input ← GetInputProblem
2: while NotStop do
3:    S ← MakeGreedyRandomizedSolution(input)
4:    S' ← Do LocalSearch(S)
5:    if cost(S') < cost(BestFoundSolution) then
6:       BestFoundSolution ← S'
7:    end if
8: end while
9: return BestFoundSolution
```

First we make a restricted candidate list (RCL) based on the set of possible candidates. The selection of candidates to become members of the restricted candidate list is based on a price–per–row–covered scheme. In this scheme the cost of each column can be expressed as $\gamma_j = c_j/|I_j \cap M^*|$, where M^* is the set of yet uncovered rows. Notice that this pricing scheme complies with the demand for some adaptiveness in the algorithm.

In [1] the candidates are allowed to enter the RCL if their utility satisfies $\gamma_j \geq \alpha \cdot \gamma_{max}, \ \forall j$, for some value of $\alpha < 1$. Alternatively one can choose a fixed number of columns to enter the RCL in each iteration. We have chosen to work with the first strategy since that way we have a guarantee that the elements in RCL are not worse than a given limit decided by α. Another obvious advantage is that we do not need to maintain a sorted list of columns at each iteration so as to choose the best columns to go into the RCL. Another strategy is a combination of having a maximum number of elements in the RCL as well as having an α limit strategy.

After the RCL has been made an element is chosen at random from this list in line 6. All candidates in the RCL are equally likely to be chosen in our implementation of GRASP in the first version. We shall later see that it is possible to give the best candidates a higher probability of being selected. This is investigated in section **??**.

In lines 7 and 8 the solution and the set of candidates are updated. The new solution and set of candidates are used to update the relative cost per row covered for each column. Finally When a feasible solution is found the redundant columns are removed from it by decreasing order of their prices.

3.2 Implementation

A brief implementation description is good but be carefull that it does not become too lengthy. This description is a little too long but otherwise competent.

The implementation of GRASP has been made in the programming lan-

Algorithm 2 MakeGreedyRandomizedSolution()

```
1: init:
2: Solution ← ∅
3: Candidates ← input
4: while Solution not feasible do
5:    RCL ← MakeRestrictedCandidateList(Candidates)
6:    s ← SelectRandomElement(RCL)
7:    Solution ← Solution ∪ s
8:    Candidates ← Candidates\{s}
9:    UpdateRelativeCosts(Candidates)
10: end while
11: RemoveRedundantColumns(Solution)
```

guage C, and the source code can be seen in appendix A.[2] As for the data structures used in our implementation we experimented with representing rows and columns both as linked lists of structures as well as arrays of structures. Using linked lists requires a lot of put and get operations which takes more time than simple array calls by index. On the other hand using arrays makes the code more difficult to read due to subtleties that appear when converting array indices to column or row elements. Using arrays of structures instead of linked lists of structures made our code faster by a factor of 3, which is why we stuck with the arrays of structures for the rest of our implementation.

The main file is `grasp.c` which complies with Algorithm 1. For each test case a number of iterations of the GRASP algorithm are made. There are two parameters which we will be dealing with. The first and most obvious is the value of α. Choosing this value too large will result in a reduction of the total solution space searched. On the other hand choosing it too small will result in solutions generated almost purely at random from all the possible columns. It is noted that setting $\alpha = 1$ is equal to doing a pure greedy search.

Our initial objective function value, for all test runs of the algorithm, is determined by doing one iteration of procedure `greedy` with a value of $\alpha = 0.1$. This has been done to ensure that we are able to see the effect of choosing different values of α.

A feature that can not be read from [1] is the removal of redundant columns. As the solution construction progresses some of the previously included columns may become redundant. Different strategies exist to solve this problem. The simplest of these is to sort the redundant columns after decreasing cost and start removing the most expensive redundant column.

[2]Experiments of Eric Galyon show that the programming language Java (JDK1.2), on the arithmetical operations alone, is about 8 times slower than C/C++, and on other operations the difference is even bigger.

Another strategy much similar to this is suggested in [4], where redundant columns are removed, most expensive column first, until only ten redundant columns exist in the solution. The remaining ten columns are removed based on enumerations of all possibilities. We have chosen to use the simple scheme since again our primary aim is to obtain good solutions fast.

The stopping criterion for the algorithm is simply an upper bound on the number of performed iterations.

4 Results

Good parameter tuning, testing and presentation/interpretation of the results is essential in the report. First describe briefly the test sets used (as described below).

As mentioned in the introduction we use the test sets found in Beasley's OR library, [8]. The different test sets have been summarised in table 1. Each set is identified by a name (e.g. scpb3), a size and a density. The density is the average number of rows covered by a single column relative to the total number of rows.

Name	Size	Density
scp51 ... scp55	(200,2000)	2 %
scp61 ... scp65	(200,1000)	5 %
scpa1 ... scpa5	(300,3000)	2 %
scpb1 ... scpb5	(300,3000)	5 %
scpc1 ... scpc5	(400,4000)	2 %
scpd1 ... scpd5	(400,4000)	5 %
scpnre1 ... scpnre5	(500,5000)	10 %
scpnrf1 ... scpnrf5	(500,5000)	20 %
scpnrh1 ... scpnrh5	(1000,10000)	5 %

Table 1: Test sets and their characteristics.

Then **carefully** *describe* **all** *the parameters which are part of the algorithm. It is a good idea to make a table containing a description of the parameters and the possible values they may take*

Parameter	Possible values	Test values
Start temperature (T_{start})	$T_{start} \geq 0$	{ 25, 50, 100 }
Temperatur decrease (α	$\alpha \in]0, 1[$	{ 0.80, 0.85, 0.90, 0.95}
Finish temperature (T_{finish})	$T_{finish} \geq 0$	-

Usually there are too many parameters to perform exhaustive tests on all combinations. Hence it is necessary to exclude some of the parameters, i.e. assign reasonably variables to the least critical parameters. Describe which parameters to fix, why they are considered least critical and why the value you assign is resonable. For the remaing parameters, exhaustive tests on all combinations should be performed, see the table below where two values (for simulated annealing, start temperature T_{start} (the rows) and temperatur reduction α (the columns) values are given. For each combination of values (ex. $T_{start} = 20$ and $\alpha = 0.80$) average performance and standard deviation should be tested. The tests should be performed on the **parameter tuning set**. *The data set is divided into two different parts: The* **parameter tuning set** *and the* **test set**. *From the set of test data a few samples are taken (ex. 3-4 examples) of different size. Parameter tuning is now performed by testing all combinations of possible parameter values and calculate the average performance on the entire parameter tuning set. Remember that for each parameter choice a number of tests should be performed (e.g. 10). The result of the parameter tuning is a set of parameters with the best average performance and a lowest standard deviation.*

Why is it needed to make the division into a parameter tuning set and a test set ? Because you would otherwise tune the parameters to particular statistical properties and this would give an unrealistic performance of your algorithm.

	0.80	0.85	0.90	0.95
100	(Avg., δ)			
50				
25				

Then the entire **test set** *should be tested, but only for the chosen best set of parameters and the results as given in the table below. The most important result is unfortunately not in the table below: The* **gap** *between the optimal solution/lower bound and the average performance of the metaheuristic. This is calculated as given below:*

$$gap = 100 \cdot \frac{(Mean - Best)}{Best}$$

This is the main result of your report !. *The best way of comparing two algorithms for efficiency is to compare the average gap for all the datasets (bottom line of table 2).*

	Alg. 1.				Alg. 2				
Problem	Best	Mean	Variance	Time	Best	Mean	Variance	Time	BNS
scp51	257	259.60	2.15	5.5	263	263.00	0.00	2.6	253*
scp52	307	313.75	8.41	5.2	321	321.80	0.70	2.6	302*
scp53	228	228.85	0.24	4.8	230	230.00	0.00	2.4	226*
scp54	245	246.70	0.64	5.1	246	246.00	0.00	2.4	242*
scp55	212	212.50	0.47	4.9	212	212.00	0.00	2.6	211*
Avg.									
scp61	142	142.80	0.59	1.9	142	142.00	0.00	0.8	138*
scp62	146	147.65	3.50	1.9	146	150.20	5.70	1.0	146*
scp63	145	147.00	1.89	1.9	148	148.00	0.00	1.0	145*
scp64	132	132.90	0.31	1.9	135	135.00	0.00	0.8	131*
scp65	168	170.20	2.17	1.9	168	170.40	1.80	1.0	161*
Avg.									
scpa1	256	258.25	0.93	7.7	258	258.20	0.20	4.4	253*
scpa2	256	259.65	3.19	8.1	256	256.40	0.80	4.4	252*
scpa3	237	240.00	2.32	7.8	237	238.60	1.30	4.4	232*
scpa4	238	241.85	1.82	7.7	240	240.00	0.00	4.6	234*
scpa5	239	241.05	0.89	7.4	238	239.40	0.80	4.4	236*
Avg.									
scpb1	69	69.90	0.31	6.5	69	69.20	0.20	3.2	69
scpb2	76	77.20	1.01	6.8	76	76.20	0.20	3.6	76*
scpb3	80	80.80	0.17	6.7	80	80.20	0.20	3.4	80*
scpb4	79	80.50	1.00	6.9	81	81.40	0.30	3.4	79*
scpb5	72	72.00	0.00	6.8	72	72.00	0.00	3.4	72*
Avg.									
scpc1	229	233.35	2.03	14.7	234	235.60	0.80	7.0	227*
scpc2	222	224.05	1.31	14.2	221	221.80	0.20	7.0	219*
scpc3	248	252.90	5.46	13.4	246	248.20	3.70	7.4	243*
scpc4	225	227.75	2.30	13.8	231	231.00	0.00	7.0	219*
scpc5	217	218.35	0.45	13.7	218	218.00	0.00	6.6	215*
Avg.									
scpd1	61	62.50	0.58	10.8	62	62.00	0.00	5.2	60*
scpd2	67	67.10	0.09	10.7	67	67.00	0.00	5.6	66*
scpd3	73	73.70	0.33	10.9	73	73.00	0.00	6.0	72*
scpd4	62	62.05	0.05	10.9	62	62.40	0.30	5.6	62*
scpd5	61	61.70	0.43	11.2	61	62.40	0.80	5.4	61*
Avg.									
scpnre1	29	29.00	0.00	15.9	29	29.00	0.00	10.0	29
scpnre2	31	31.50	0.26	17.0	30	30.80	0.20	10.0	30
scpnre3	27	27.95	0.05	16.4	27	27.20	0.20	10.2	27
scpnre4	28	28.90	0.09	16.9	28	28.60	0.30	10.0	28
scpnre5	28	28.00	0.00	16.7	28	28.00	0.00	9.8	28
Avg.									
scpnrf1	14	14.50	0.26	26.8	14	14.00	0.00	16.8	14
scpnrf2	15	15.00	0.00	26.8	15	15.00	0.00	17.6	15
scpnrf3	14	14.95	0.05	27.3	14	14.80	0.20	17.0	14
scpnrf4	14	14.65	0.24	26.9	14	14.00	0.00	17.4	14
scpnrf5	14	14.05	0.05	26.6	14	14.00	0.00	17.4	13
Avg.									
scpnrh1	66	67.30	0.43	61.2	65	66.40	1.30	31.6	63
scpnrh2	66	67.20	0.43	65.3	67	67.00	0.00	34.0	63
scpnrh3	62	63.20	0.38	70.6	62	62.40	0.30	36.8	59
scpnrh4	61	61.95	0.26	58.9	61	61.60	0.30	42.2	58
scpnrh5	57	57.90	0.31	59.4	57	57.40	0.30	41.0	55
Avg.									
Total Avg.									

Table 2: Data from executions

5 Discussion

What happens in the results ? Which algorithm is best ? Is one algorithm better for the small/big data sets ? Which algorithm is most stable ?

6 Conclusion

A brief conclusion

The deterministic greedy algorithm for the SCP is known to give results that on average are 11.9% larger than the best known solutions, [3]. Using randomisation as introduced in GRASP, we obtain results that on average only differ 3% from the best known solutions. To get better solutions we need to run the algorithm for a long time. If we are satisfied with near–optimal solutions the simple GRASP will give us good solutions relatively fast. Furthermore GRASP is easy to implement.

Except for few of the smaller problems the extension of GRASP to use Lagrangian pricing did not provide any significant improvement to the results given by the simple GRASP algorithm. It seems that using Lagrangian pricing alone does not provide extra information on the neighbourhood structure. The fact that we obtained better results for some of the smaller problems[3] indicates that it is worth pursuing the new pricing strategy.

CFT uses a number of speeding and refining strategies. Because of the time limits of this project we have not been able to test these refining strategies on our implementation of GRASP using Lagrangian prices. We have however compared our results from the simple GRASP implementation to the more refined PROGRES algorithm. These comparisons indicate a slight improvement in the quality of the solutions as well as significant reduced running times following from the speeding and refinement strategies.

An important advantage of the new GRASP with Lagrangian pricing is that it provides us with a lower bound enabling us to evaluate the maximal deviation from an optimal solution. However it proved difficult to implement the subgradient procedure efficiently such that optimal or near–optimal solutions could be obtained for all test problems.

The downside of the new pricing strategy was that it turned out that the running times were unreasonably long and that the subgradient procedure accounted for almost 90 % of the time spend. This difficulty calls for a more detailed research on different methods for speeding up the procedure.

6.1 Further Work

CFT ([4]) suggests that we remove the superfluous columns by a decreasing price strategy until there are 10 columns left and then remove columns in an optimal way using enumeration. Implementing this approach might help the algorithms improve the near optimal solutions on the last bit to optimality.

It also remains as future work to investigate the speeding and refining techniques proposed in section **??** and by CFT in [4] on GRASP using lagrangian pricing strategy. Especially different strategies for updating the

[3] Problems scp51 to scp65.

step size in the subgradient procedure, the effect of column fixing and optimal removal of redundant columns should be pursued.

References

[1] Feo, Thomas A. and Resende, Mauricio G.C.
Greedy Randomized Adaptive Search Procedures
Journal of Global Optimization 6: 109-133, 1995.

[2] Feo, Thomas A. and Resende, Mauricio G.C.,
A Probabilistic Heuristic For a Computationally Difficult Set Covering Problem
Operations Research Letters 8 (1989) 67-71.

[3] Haouari, Mohamed and Siala, J. C.[4]
A Probabilistic Greedy Search Algorithm for the Set Covering Problem.

[4] Caprara, A., Fischetti, M. and Toth, P.,
A Heuristic Method for the Set Covering Problem. Published in Operations Research Vol. 47, No. 5, September-October 1999, pp. 730-43.

[5] Cormen, T. H., Leiserson, C. E. and Rivest, R. L.,
Introduction to Algorithms. ISBN 0-262-53091-0.

[6] William J. Cook, William H. Cunningham, William R. Pulleyblank og Alexander Schrijver
Combinatorial Optimization. ISBN 0-471-55894-X.

[7] Grossman, Tal and Wool, Awishai,
Computational Experience with Approximation Algorithms for the Set Covering Problem.
European Journal of Operational Research 101 (1997), 81-92.

[8] Beasley, J. E., *OR Library*,
http://mscmga.ms.ic.ac.uk/jeb/orlib/scpinfo.html

[9] Kourosh Marjani and Tomas Ward,
Special Project on Set Covering Problem
Technical University of Denmark, IMM, 2002 (not published).

[4]Both authors are from Ecole Polytechnique de Tunisie, BP 743 2078, La Marsa, Tunisia. We have no information on when and where the article was published.