Complete Local Search with Memory

Diptesh Ghosh[*]
Gerard Sierksma[†]

SOM-theme A Primary Processes within Firms

Abstract

Neighborhood search heuristics like local search and its variants are some of the most popular approaches to solve discrete optimization problems of moderate to large size. Apart from tabu search, most of these heuristics are memoryless. In this paper we introduce a new neighborhood search heuristic that makes effective use of memory structures in a way that is different from tabu search. We report computational experiments with this heuristic on the traveling salesperson problem and the subset sum problem.

Keywords: Discrete Optimization, Neighborhood, Local Search, Tabu Search

[*] Corresponding Author. Faculty of Economic Sciences, University of Groningen, The Netherlands. Email: D.Ghosh@eco.rug.nl

[†] Faculty of Economic Sciences, University of Groningen, The Netherlands. Email: G.Sierksma@eco.rug.nl

1. Introduction

Discrete optimization problems arise when a decision maker is constrained to make discrete choices. i.e. choices of whether to include or exclude objects from a solutions, or decisions of how many objects to include in the solution. A large number of such problems are NP-hard (for example, the traveling salesperson problem, the knapsack problem, job scheduling problems, etc.) which means that it is extremely unlikely that we will formulate exact algorithms that solve all instances of these problems efficiently. Neighborhood search techniques are often used to obtain good approximate solutions in such situations (refer to Aarts and Lenstra [2] for a comprehensive introduction to these techniques).

Neighborhood search heuristics depend on the concept of a neighborhood defined on the set of all solutions. Loosely described, a *neighborhood* of a solution is a set of solutions that are close to it in some sense. Consider, for example, the traveling salesperson problem (TSP). The 2-opt neighborhood of a TSP tour is the set of all tours that can be obtained by removing two edges from the original tour and re-connecting the two paths thus obtained into a tour different from the original one. The process of reaching a neighbor from a solution is called a *move*. A move is said to be *improving* if the neighbor reached on executing a move has a better objective value than that of the current solution.

Neighborhood search techniques can be classified into those that store attributes of the solutions they visit during their execution, and those that do not. The former type includes heuristics like tabu search (see, for example, Glover [5]) while the latter type includes heuristics like simulated annealing (see, for example, Aarts and Korst [1]). In a wide variety of applications, tabu search and its hybrids have proved superior to memoryless heuristics like simulated annealing. In this paper, we introduce a heuristic called “complete local search with memory”. It makes use of memory structures to store solutions encountered by the heuristic during its execution, and the results of searching their neighborhoods. Therefore, it uses memory structures, but uses them in a way very different from tabu search. In addition, the heuristic described here has a backtracking mechanism that allows it to search solution spaces more thoroughly.

The remainder of this paper is organized as follows. In Section 2, we present the ”complete local search with memory” heuristic. We illustrate its working in Section 3 using an instance of a symmetric non-Euclidean TSP. In Section 4, we report results of computational experiments on two problem domains, the traveling salesperson problem and the subset sum problem. The reason for choosing these two domains in particular is that local search variants are known to be very effective in the former, while they perform poorly in the latter. We summarize the paper in Section 5.

2. Description of the heuristic

An important feature of the adjacency graph imposed on the solution space by common neighborhood structures is that there are, almost always, more than one directed paths from the initial solution to a given solution. None of the existing neighborhood search methods utilize this feature in their design. Complete local search with memory (CLM) makes use of it by keeping track of the solutions visited by the heuristic and preventing it from searching their neighborhoods again at later stages during its execution. The term *memory* in CLM refers to storage space set aside specifically for storing solutions generated by the heuristic. The *size* of the memory refers to the number of solutions that can be stored. This memory is used to maintain three lists of solutions. The first one, called LIVE, stores solutions that are available to the heuristic for future consideration (called exploration). A second list, called DEAD, contains solutions that were in LIVE at some stage, and have already been explored. The third list, called NEWGEN is a temporary store for new solutions being generated by the heuristic during the current iteration.

CLM starts with a given solution as input and puts it in LIVE. DEAD and NEWGEN are initially empty. It then performs iterations described below until an user-defined terminating condition is reached or the heuristic runs out of memory. At the beginning of each iteration, the heuristic picks k solutions from LIVE. Each one of these is explored, i.e. it is transferred from LIVE to DEAD, and all its neighbors with objectives better than a threshold value τ are generated. Each one of these neighbors is then checked for membership in LIVE, DEAD, and NEWGEN. If it is a member of LIVE, then the solution has already been obtained by the heuristic and has not been explored yet (being less promising that the ones that are currently under consideration). If it is a member of DEAD, then the solution has already been generated and explored and we already know the solutions that are obtained by exploring it. If it is in NEWGEN, then it has already been generated in the present iteration. However if the solution is not yet in any of the lists, it is a new solution which merits exploration, and is put in NEWGEN. When all the solutions that were picked have been explored, the solutions in NEWGEN are transferred to LIVE and the iteration is complete. If LIVE is not empty when the stopping rule is reached or the heuristic runs out of memory, a post-processing operation is carried out, in which generic local search is applied to the each of the members of LIVE and the locally optimal solutions thus obtained are added to DEAD. The best solution present in DEAD at the end of the post-processing operation is returned by the heuristic. The pseudocode of the heuristic is provided in Figure 2.1. Note that it is a template heuristic. Apart from the choice of neighborhood structures, one has to decide on the memory size, the stopping rule, the criteria for picking solutions from LIVE, and the values of k and τ.

The following are some plausible stopping rules.

1. Stop when the LIVE list is empty at the beginning of an iteration.
2. Stop when a predefined number of iterations are over.
3. Stop when there has been no improvement in the best solution for a predefined number of iterations.
4. Stop when a local optimum has been reached.

The first condition is perhaps the most natural — it is not possible to continue CLM after this condition is reached. When this condition is satisfied, the heuristic has explored all solutions that can be reached from the initial solution. Consequently this stopping rule results in the longest execution times, and returns the best solution possible. The last condition is one used in

```
Algorithm CLM
Input: Problem instance $I$, initial solution $s_0$, $k$, $\tau$.
Output: A solution to $I$.
begin
    $LIVE \leftarrow s_0$;
    $DEAD \leftarrow \emptyset$;
    while stopping rule is not satisfied
    begin /* Beginning of iteration */
        $NEWGEN \leftarrow \emptyset$;
        $chosen \leftarrow 0$;
        while $chosen < k$ and $LIVE \neq \emptyset$
        begin
            choose a solution $s$ from $LIVE$;
            $chosen \leftarrow chosen + 1$;
            transfer $s$ from $LIVE$ to $DEAD$;
            generate neighbors of $s$ with objectives better
            than a threshold $\tau$;
            for each neighbor $n_s$ of $s$
            begin
                if $n_s$ is not in $LIVE$, $DEAD$ or $NEWGEN$
                begin
                    if sufficient memory is available
                        add $n_s$ to $NEWGEN$;
                    else
                    begin
                        transfer all nodes
                        from $NEWGEN$ to $LIVE$;
                        go to **postproc**;
                    end
                end
            end
        end
        for each $s$ in $NEWGEN$
            transfer $s$ from $NEWGEN$ to $LIVE$;
    end /* End of iteration */
    **postproc**: for each solution $s$ in $LIVE$
    begin
        remove $n$ from $LIVE$;
        obtain a locally optimal solution $n_l$ from $n$ using local search;
        add $n_l$ to $DEAD$;
    end
    return the best solution in $DEAD$;
end.
```

Figure 2.1: Pseudocode for CLM: Complete local search with memory

generic local search. It is likely to be reached far more quickly than the others, and will generally not return very good quality solutions. The second and third stopping rules are commonly used in enhancements of local search, like simulated annealing and tabu search. These are likely to take time intermediate between rules 1 and 4, and generate intermediate quality results. In Figure 2.2 we present the length of the best TSP tour obtained by CLM on a randomly chosen 50-city TSP instance as a function of the number of iterations executed. Table 2.1 depicts the effect of various stopping rules on the solution quality and execution time of CLM for this instance. Note that stopping rule 1 does indeed take the longest time (111 iterations) and returns the best solution (tour length 1198), while stopping rule 4 takes the least time 4 iterations) and returns the worst solution (tour length 1425).

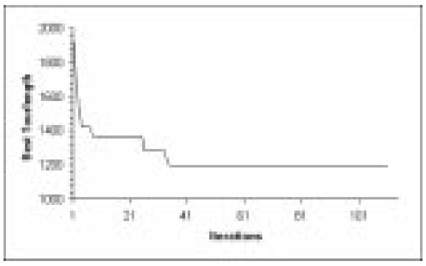

Figure 2.2: Solution quality as a function of iteration number for the CLM heuristic

Table 2.1: Effect of various stopping rules on CLM

	Stopping rule	Stops after Iteration #	Length of best tour found
1.	Stop when LIVE is empty at the beginning of an iteration	111	1198
2.	Stop when 15 iterations are over	16	1368
3.	Stop when there has been no improvement in 15 iterations	24	1368
4.	Stop when a local optimum is reached	4	1425

There are several ways of choosing solutions to explore. Some of these exploration rules are the following.

1. Choose solutions having the best objective values.
2. Choose solutions that allow the best improvements on exploration.
3. Choose solutions at random.

The first condition allows the exploration of the search neighborhood in a manner that explores a few neighbors completely before considering others. The second one takes the search along a path which appears to bring about the best results at that stage. The third rule is perhaps the fastest rule to apply in most circumstances. However the results obtained by applying such a rule are unpredictable. Note that these rules (as well as the stopping criteria mentioned earlier) are representative in nature, individual problem domains may have rules that function in a much superior manner.

The parameter k determines the aggressiveness of the search. A small value of k combined with exploration rule 1 and any of the stopping rules 1, 2 or 3 causes the heuristic to track a small number of neighbors of the initial solution to local optima reachable from them before examining other neighbors. A large value of k allows more neighbors of the initial solution to be examined simultaneously. Therefore a large value of k helps the heuristic to reach good quality solutions in fewer iterations. However, the time taken per iteration, and the size of the $LIVE$ and $DEAD$ lists also increase faster for large values of k. Figure 2.3 illustrates the effect of changing the value of k on the execution of CLM. Notice that the memory is used twice as fast for $k = 2$ than it is for $k = 1$.

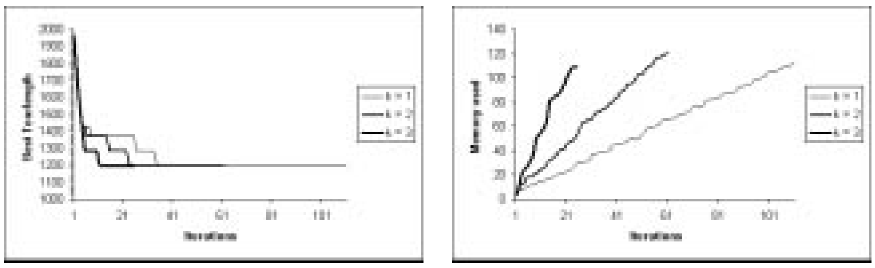

Figure 2.3: Effect of changing the value of k on CLM execution

The parameter τ controls the way in which CLM accepts non-improving moves. If, for example, it is set to a very low value, then all the neighbors of all the solutions get accepted. If it is set to the objective value of the solution being explored, only those solutions that are better than the original solution are explored. In this case however, some promising solutions may not be reached. τ may also be a function of the iteration number. It may initially allow worsening moves (to help the search jump out of a locally optimal initial solution), and later restrict the search to only those moves that lead to better solutions.

A judicious choice of parameters and rules reduces CLM to several common heuristics. For example, in a minimization problem,

1. using stopping rule 1, and setting $\tau = \infty$ results in exhaustive search, assuming that the adjacency graph is connected;
2. using stopping rule 4, exploration rule 1, $k = 1$, and setting τ as the objective value of the solution being explored results in generic local search with steepest descent;
3. using stopping rule 4, exploration rule 3, $k = 1$, and setting τ as the objective value of the solution being explored results in generic local search with random descent;

4. using stopping rules 2 or 3, $k = 1$, setting τ as a suitable function of the iteration number and the objective value of the best solution at that stage, and using an exploration rule that selects solutions randomly among ones generated in the last iteration, results in a simulated annealing heuristic.

3. An example

In this section we illustrate the working of CLM using an instance of a symmetric non-Euclidean TSP. The instance is small, with five cities and one local optimum. It has the following distance matrix.

$\mathbf{d}(.)$	0	1	2	3	4
0	–	34	45	88	92
1	34	–	99	97	5
2	45	99	–	61	90
3	88	97	61	–	52
4	92	5	90	52	–

A five-city symmetric TSP on a complete graph has 12 possible tours (solutions). In our example we label them A through L. The details of these solutions are presented in Table 3.1.

Table 3.1: Feasible tours in the example

Solution Code	Tour representing Solution	Tour Length	Solution Code	Tour representing Solution	Tour Length
A	0-1-2-3-4-0	338	G	0-2-1-3-4-0	385
B	0-1-2-4-3-0	363	H	0-2-1-4-3-0	289
C	0-1-3-2-4-0	374	I	0-2-4-1-3-0	325
D	0-1-3-4-2-0	318	J	0-2-3-1-4-0	300
E	0-1-4-2-3-0	278	K	0-3-1-2-4-0	466
F	0-1-4-3-2-0	197	L	0-3-2-1-4-0	345

We consider an implementation of the CLM heuristic with the following properties:

Neighborhood	A *2-exchange neighborhood*, i.e. one in which a neighbor is obtained from a tour by changing the position of two cities on the tour.
Exploration rule	1, i.e. choose k-best solutions from $LIVE$.
Stopping rule	1, i.e. stop when $LIVE$ is empty at the beginning of an iteration.
Parameters	$k = 2$, τ is chosen to be the objective value of the solution being explored currently.

The neighborhood and τ value induce the adjacency graph shown in Figure 3.1 on the set of solutions. An arc in the digraph represents a feasible move.

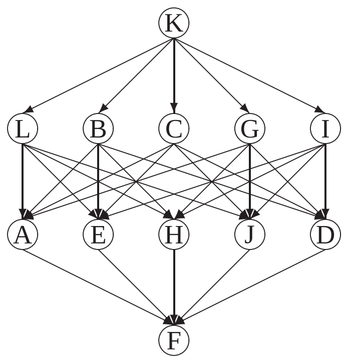

Figure 3.1: Feasible moves between solutions in the example

Let us assume that the initial solution is K (i.e. the tour 0-3-1-2-4-0). This solution is added to the LIVE list and we start Iteration 1. The DEAD and NEWGEN lists are empty at this stage. We choose to explore K in this iteration. It is transferred to DEAD and its neighbors, i.e. solutions B, C, G, I, and L are generated. Since none of these solutions are members of LIVE, DEAD or NEWGEN, they are all added to NEWGEN. At the end of Iteration 1, these solutions are transferred to LIVE. Thus at the beginning of the second iteration, LIVE contains B, C, G, I, and L, and DEAD contains K. In this iteration we explore two solutions from LIVE with the lowest tour lengths; i.e. I (with tour length 325) and L (with tour length 345). I, when explored, adds solutions D, E, H and J to NEWGEN. When L is explored, it generates A, E, H and J. However, since E, H and J are already in NEWGEN, only A is added to it. The other three are ignored. This process continues, CLM generates the optimum solution F in Iteration 3, and finds out that it is locally optimum in Iteration 4. The other solution that is explored in Iteration 4 is J. When J is explored, it generates the solution F. Now F, at this point is in DEAD, i.e. CLM knows the outcome of exploring F. Hence it does not add F to NEWGEN. Using similar arguments, no iteration after the fourth adds any nodes to LIVE. The heuristic terminates at the beginning of iteration 8 when the LIVE list is empty. Table 3.2 shows the various lists at each iteration of CLM.

4. Computational experiments

In this section we report the results of computational tests carried out to compare the solutions returned by CLM with those returned by a more established neighborhood search technique,

Table 3.2: The contents of various lists during the execution of CLM on the example problem

Iteration	Beginning of Iteration ({LIVE}{DEAD})	End of Iteration ({LIVE}{DEAD}{NEWGEN})
1	({K}{})	({}{K}{B, C, G, I, L})
2	({B, C, G, I, L}{K})	({B, C, G}{I, K, L}{A, D, E, H, J})
3	({A, B, C, D, E, G, H, J}{I, K, L})	({A, B, C, D, G, J}{E, H, I, K, L}{F})
4	({A, B, C, D, F, G, J}{E, H, I, K, L})	({A, B, C, D, G}{E, F, H, I, J, K, L}{})
5	({A, B, C, D, G}{E, F, H, I, J, K, L})	({B, C, G}{A, D, E, F, H, I, J, K, L}{})
6	({B, C, G}{A, D, E, F, H, I, J, K, L})	({G}{A, B, C, D, E, F, H, I, J, K, L}{})
7	({G}{A, B, C, D, E, F, H, I, J, K, L})	({}{A, B, C, D, E, F, G, H, I, J, K, L}{})
8	({}{A, B, C, D, E, F, G, H, I, J, K, L})	

tabu search. We used the 2-opt neighborhood structure for both heuristics. This neighborhood structure does not lead to very good quality solutions for most problems, especially when the problems are large, but it is fast and satisfies our aim to make a fair comparison between the two heuristics. We chose problem instances of the traveling salesperson problem (TSP) and the subset sum problem (SSP) for our experiments. For TSP, we used randomly generated problems as well as benchmark problems from TSPLIB. For SSP, we only used randomly generated problem instances. All experiments were carried out on a 450MHz Pentium machine running Linux.

4.1 Computations with TSP instances

We consider a TSP instance of size n as a complete graph on n vertices, in which each edge has a cost associated with it, and the aim is to find a minimum cost Hamiltonian tour in the graph. For our experiments we used 75 random instances as well as 21 benchmark instances of symmetric TSP. The random instances were divided into five data sets containing 15 instances each. These data sets were named T50, T75, T100, T125, and T150, where all instances in set Tx were of size x. The edge costs for all the randomly generated instances were drawn from the interval $[1, 1000]$. We also chose 21 benchmark symmetric TSP instances. These instances were chosen from the resource called TSPLIB [8]. We chose problems with size not more than 250 so that execution times were not too long.

In our tabu search implementation, following the guidelines of Knox [7], we set the tabu length as $3n$ where n is the problem size. We allow the search to continue for $30n$ iterations.

In our CLM implementation, we chose to stop the search when LIVE is empty at the beginning of an iteration. In order to determine the parameters to use in CLM, we carried out experiments with 75 TSP instances of size ranging from 25 to 75. (Figure 4.1 presents the average of results from 25 problems of size 50.) We observed that setting $k = 1$ resulted in slightly better solutions in most cases. For all the instances considered, the solution quality improved rapidly with increasing memory size when there was very little memory available, but this improvement was much less marked when the memory size exceeded around 100 solutions. Therefore, in our CLM implementation, we chose to have a 100 solution memory. We carried out experiments with various threshold factors and found out that we achieved the best results with an iteration dependent threshold. Therefore, we set the threshold parameter τ as

$$\tau = (1 - \alpha) \times (current\_solution\_value), \text{ where } \alpha = \left(\frac{\alpha_0}{1+\beta}\right)^{iteration}.$$

Our experiments with various values of α_0 (ranging from 0.0 to -0.1) and β (ranging from 0.001 to 0.1) showed that the solution quality improved when the value of α_0 was decreased, but the execution time also increased. However, neither the solution quality, nor the execution time, was sensitive to changes in β. In our implementation, we therefore chose $\alpha_0 = -0.1$ and $\beta = 0.1$.

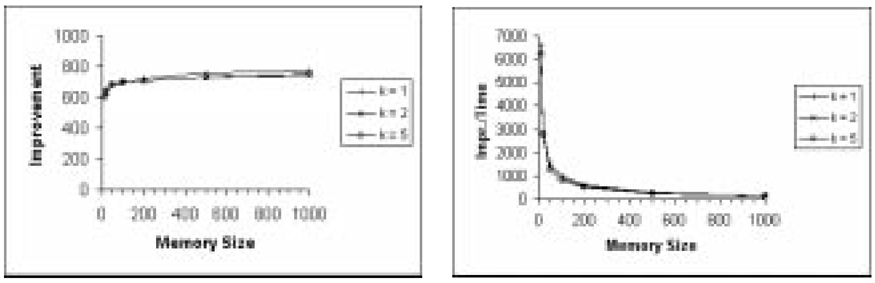

Figure 4.1: Setting parameters for CLM on TSP instances

The results of our experiments with TSP instances are summarized in Table 4.1. For each heuristic, “Tour_cost” refers to the average of the costs of Hamiltonian tours returned, and “Time” refers to the average of the time taken in CPU seconds. The averages are taken over all 15 problem instances in the data set. The results show that CLM returned better quality solutions than tabu search and took less time.

Table 4.1: Results from computations on randomly generated TSP instances

Data Set	Tabu Search		CLM	
	Tour_cost	Time	Tour_cost	Time
T50	1697.9333	23.1307	1342.0667	13.936
T75	1517.0667	122.6907	1313.1333	52.6427
T100	1816.2667	385.562	1394.3333	125.5553
T125	1796.3333	962.2113	1553.1333	262.8807
T150	1818.4	2009.0673	1597.1333	473.2187

Table 4.2: Results from computations on TSP instances from TSPLIB

Problem	Optimal	CLM		Suboptimality
	Tour_cost	Tour_cost	Time	
berlin52	7542	7944	9.86	5.33%
bier127	118282	121230	275.02	2.49%
ch130	6110	6346	274.41	3.86%
ch150	6528	6619	685.15	1.39%
kroA100	21282	22201	155.86	4.32%
kroA150	26524	28360	761.65	6.92%
kroA200	29368	30102	2206.35	2.50%
kroB100	22141	22391	136.05	1.13%
kroB150	26130	27146	497.95	3.89%
kroB200	29437	31928	1650.85	8.46%
kroC100	20749	21315	117.49	2.73%
kroD100	21294	22088	142.53	3.73%
kroE100	22068	23347	122.19	5.80%
lin105	14379	14962	207.93	4.05%
pr76	108159	110806	57.44	2.45%
pr107	44303	45622	77.40	2.98%
pr124	59030	60602	198.34	2.66%
pr136	96772	105112	412.62	8.62%
pr144	58537	61161	152.93	4.48%
pr152	73682	75065	492.15	1.88%
pr226	80369	82573	1838.17	2.74%

Table 4.2 summarizes our results with the 21 instances from TSPLIB. We did not run tabu search on these instances since the optimal solutions were already known. On an average, the solutions returned by CLM for these instances were 4.2% suboptimal.

4.2 Computations with SSP instances

A SSP instance of size n contains a set of integers $W = \{w_1, \ldots, w_n\}$, and an integer C. The objective is to find a subset S of W such that $\sum_{w_j \in S} w_j$ is the largest possible, without exceeding C. Extensive computational experiments show that SSP instances are more difficult to solve when the interval from which the problem data is chosen is wide compared to the size of the problem and C is close to $0.5 \times \sum_{w_j \in W} w_j$ (refer, for example, Ghosh [4]). We used 12 randomly generated data sets (S100-0.3, S100-0.5, S100-0.7, S150-0.3, S150-0.5, S150-0.7, S200-0.3, S200-0.5, S200-0.7, S300-0.3, S300-0.5, and S300-0.7) to compare the performances of CLM and tabu search. The names of the data sets are of the form Sx-y where x refers to the problem size, and y refers to the ratio of C and $\sum_{w_j \in W} w_j$. Each of these data sets contained 25 problem instances. The numbers were chosen from the interval $[100, 1000000]$. The quality of solution is measured in terms of unused capacity (UC) defined as

$$\mathrm{UC} = C - \sum_{w_j \in S} w_j,$$

and the execution time is measured in CPU seconds.

We implemented tabu search with a tabulength of $\sqrt{n}$ and an aspiration criterion that allowed aspiration if the result of a tabu move bettered the best solution thus far.

As for TSP, CLM was implemented with stopping rule 1, (i.e. "stop when $LIVE$ was empty at the beginning of an iteration") and an exploration rule that chose the best (i.e. with lowest UC) solutions for exploration. The parameters of memory size available to the heuristic and k, and the threshold τ were determined experimentally. 100 problem instances were chosen, 25 each of sizes 20, 50, 75, and 100. CLM was implemented with memory sizes ranging from 10 to 1000 nodes, and k values of 1, 2, and 5. Our observations were similar for all problem sizes. (Figure 4.2 graphically depicts the results for $n = 50$.) We observed that there were no appreciable difference between the k values, and that the improvement to execution time ratio dropped sharply at a memory size limit of approximately 100 nodes. We therefore decided to fix $k = 2$ and a memory size of 100 nodes. The threshold parameter τ was identical in form to that used for the TSP instances. After experimenting with various values of α_0 (ranging from 0.0 to -0.1) and β (ranging from 0.001 to 0.1), we found out that here too, β had no effect on the solution quality, but changing α_0 from 0.0 to a slightly negative value dramatically improved the solution quality. Hence α_0 was set to -0.001 and β to 0.1.

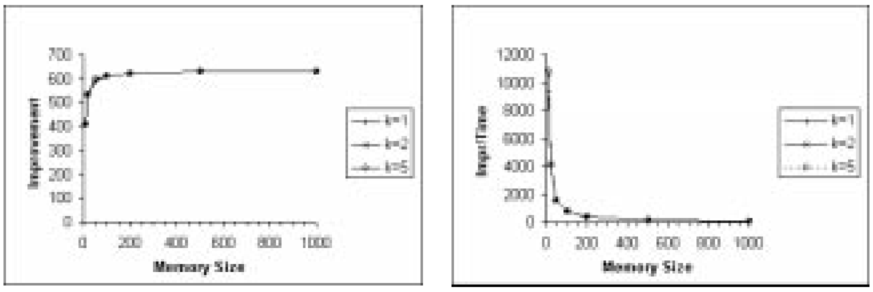

Figure 4.2: Setting parameters for CLM on SSP instances

Our computational experiences on SSP instances are presented in Table 4.3. For each heuristic compared, we report the quality of solutions returned and the execution time. These values are averages taken over all instances in the particular data set. The last two columns in the table count the number of instances in the data set where tabu search and CLM (respectively) provided a unique best result. We observed that for SSP, tabu search took slightly more time than CLM but output marginally better results on the average. Notice that the average UC of solutions decreased as n increased. Since both tabu search and CLM almost always returned solutions with UC$= 0$, when $n = 300$, we did not experiment on larger problems.

Table 4.3: Results from computations on SSP instances

Data Set	Tabu Search		CLM		Tabu Search	CLM
	UC	Time	UC	Time	better in	better in
S100-0.3	1.44	5.2964	3.6	4.5956	12	7
S100-0.5	1.48	6.7824	2.4	5.4716	15	5
S100-0.7	2.6	8.2572	3.32	4.6384	9	10
S150-0.3	0.32	43.3332	1.2	19.714	10	3
S150-0.5	0.2	56.2624	0.56	23.3772	7	3
S150-0.7	0.2	69.1936	0.52	19.2804	7	2
S200-0.3	0.04	42.4408	0.44	39.292	7	1
S200-0.5	0.24	55.52	0.2	44.79	3	4
S200-0.7	0.08	68.5748	0.2	38.8764	4	1
S300-0.3	0.00	137.1176	0.12	122.7964	2	0
S300-0.5	0.04	176.162	0.08	140.7128	2	1
S300-0.7	0.00	215.0836	0.04	122.5128	1	0

5. Conclusions

In this paper we presented a neighborhood search based heuristic called “complete local search with memory” (CLM). We also presented extensive computational experiments on instances from the traveling salesperson problem and the subset sum problem in which we compared the solutions returned by our heuristic to those returned by a simple implementation of tabu search. The comparison was carried out both in terms of the solution quality and the execution time. For traveling salesperson problems, our heuristic out-performed tabu search, while for subset sum problems the results it returned were comparable. In all the problems, our heuristic took less execution time than tabu search.

The CLM heuristic differs from other neighborhood search heuristics in that it takes advantage of the fact that for almost all neighborhood structures it is possible to reach a solution from the initial solution in more than one way. It exploits this by maintaining lists of solutions visited by the heuristic. This use of memory makes CLM more efficient than memoryless search procedures like multiple start local search or simulated annealing. The only other commonly used class of neighborhood search heuristics that uses memory is tabu search. But there are important differences between the way in which these two heuristics use memory. Unless special long-term memory structures are used, tabu search only remembers the last few solutions it visited. CLM is more intensive — it keeps track of all the solutions visited. In addition, the aggressiveness with which it searches the search space can be controlled (by adjusting k), as also the way of dealing with moves that lead to worse solutions (by manipulating the threshold τ). The heuristic is interesting from a theoretical point of view too. It provides a unifying approach to many of the common neighborhood search heuristics used to solve discrete optimization problems.

The memory structures used in CLM are amenable to common memory management schemes that, in principle, should allow the heuristic to effectively solve much larger problems than those attempted in this paper. Such schemes have often been used to produce excellent results in graph search algorithms. This is a direction in which we will be concentrating our future efforts. Other interesting areas of research would involve testing out different exploration strategies and stopping rules that would enable CLM to reach very good solutions in the minimum possible time.

References

[1] E.H.L. Aarts and J.H.M. Korst (1989). Simulated annealing and Boltzmann Machines. Wiley, Chichester.

[2] E.H.L. Aarts and J.K. Lenstra (1997). Local search and combinatorial optimization. Wiley Interscience Publications.

[3] M.R. Garey and D.S. Johnson (1979). Computers and intractability: A guide to the theory of NP-completeness. W.H. Freeman & Co. San Francisco.

[4] D. Ghosh (1998). “Heuristics for knapsack problems: Comparative survey and sensitivity analysis”. Fellowship Dissertation, IIM Calcutta.

[5] F. Glover (1990). “Tabu search—A tutorial”. Interfaces 20, 74-94.

[6] D. S. Johnson, C. S. Aragon, L. A. McGeoch and C. Schevon (1991). “Optimization by simulated annealing: An experimental evaluation. Part II, graph coloring and number partitioning”. Operations Research 39, 378-406.

[7] J. Knox (1994).“Tabu search performance on the symmetric traveling salesman problem”. Computers & Operations Research 21, 867-876.

[8] G. Reinelt (1995). “TSPLIB 95” `http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html`.