M a t h e m a t i c a
B a l k a n i c a

New Series Vol. 25, 2011, Fasc. 5

A Parallel Branch-and-Bound Algorithm with Restarts for Solving the Hierarchical Covering Location Problem

Predrag Stanojević

The Hierarchical Covering Location Problem (HCLP) is an NP-hard problem. In this paper we describe an efficient Brand-and-Bound algorithm for solving HCLP, implemented using openMP to achieve parallelization. This is a general approach to achieving High Performance Computing and can be applied to other optimization problems solvable by Branch-and-Bound algorithms.

MSC2010: 90C57, 68W10

Key Words: optimization, parallelization, high performance computing, branch and bound, hierarchical location

1. Introduction

Covering location problems encompass a range of applications concerning the location of various kinds of facilities which cover the so-called demand areas. A "service distance" determines whether a demand point is covered by an established facility. The hierarchical covering models are a sub-class in which there are facilities that provide different levels of service. Such models are applied to problems of establishing health care facilities, for example. In the two-level hierarchical model, facilities on the first level (for example: clinics) provide only the basic service and cover a small area, while the facilities on the second level (in this case: hospitals) provide both, the basic and the advanced levels of health care. Other applications of such models include education facilities, production-distribution systems etc. The objective of such optimizations is to maximize the area covered by these facilities, while respecting certain preset constraints.

In this paper we consider the 2-level hierarchical maximal covering location problem (HCLP). It is a well-studied problem and it is known to be NP-hard

as a generalization of the p-median problem. HCLP is usually solved by using heuristic methods ([1], [2], [3]). However, an exact method presented herein is equally efficient, since high performance computing techniques were applied to a sophisticated branch-and-bound algorithm, which itself is optimized to solve large scale problem instances.

The parallelization approach presented herein can be applied to any branch-and-bound algorithm with binary variables. Our computational results demonstrate the effectiveness of this approach, sometimes achieving super linear speedups. The results are compared to the exact solutions obtained by using the CPLEX solver.

This paper is organized as follows: in section 2, the mathematical formulation of the problem is presented; in section 3, the pseudo-code for the parallel branch-and-bound algorithm is presented and explained; in section 4 computational results are given and in section 5 we draw some conclusions about our approach.

2. Problem formulation

The integer linear programming model for solving HCLP from [1] is used:

$$max \sum_{j \in J} F_j X_j \tag{1}$$

$$\sum_{i \in I} a_{ij} Y_i + \sum_{i \in I} b_{ij} Z_i \geq X_j, j \in J \tag{2}$$

$$\sum_{i \in I} c_{ij} Z_i \geq X_j, j \in J \tag{3}$$

$$\sum_{i \in I} Y_i = P \tag{4}$$

$$\sum_{i \in I} Z_i = Q \tag{5}$$

$$X_j \in \{0,1\}, j \in J \tag{6}$$

$$Y_i, Z_i \in \{0,1\}, i \in I \tag{7}$$

where the parameters are:

$J =$ $\{0, 1, 2, ...M - 1\}$ is the set of demand areas,

$I =$ $\{0, 1, 2, ...N - 1\}$ is the set of potential facility sites,

F_j is the population of the demand area j,

$a_{ij} =$ 1 if the demand area j can be covered by level 1 service offered at a level 1 facility located at $i \in I$,

$a_{ij} =$ 0 otherwise,

$b_{ij} =$ 1 if the demand area j can be covered by level 1 service offered at a level 2 facility located at $i \in I$,

$b_{ij} =$ 0 otherwise,

$c_{ij} =$ 1 if the demand area j can be covered by level 2 service offered at a level 2 facility located at $i \in I$,

$c_{ij} =$ 0 otherwise,

$P =$ the number of level 1 facilities to be located and

$Q =$ the number of level 2 facilities to be located.

The binary decision variables X_j, Y_i, Z_i have the following meaning:

$X_j = 1$ if the demand area j is covered by both level 1 and level 2 service,

$X_j = 0$ otherwise,

$Y_i = 1$ means that a level 1 facility is established at location $i \in I$,

$Y_i = 0$ otherwise,

$Z_i = 1$ means that a level 2 facility is established at location $i \in I$,

$Z_i = 0$ otherwise.

Therefore, considering the definition of F_j and X_j, the objective function (1) maximizes the population covered by the established facilities. The constraint set (2) ensures that a demand area j is included in the covered population only if either a level 1 facility or a level 2 facility that can cover it is established. Similarly, the constraint set (3) states that a demand area j must be covered by an established level 2 facility. a_{ij}, b_{ij} and c_{ij} parameters used in the constraint sets (2) and (3) are pre-calculated from the input data to determine whether a demand area i is within a predefined service distance radius of

a facility j, for the specified level of service. Finally, the constraints (4) and (5) limit the number of level 1 and level 2 facilities, respectively.

The model contains $2 * M + 2$ constraints and $2N + M$ variables.

3. Parallel Branch-and-Bound algorithm with restarts

In order to solve HCLP, it is necessary to set precisely Q out of N Z-variables to 1. There are $\binom{N}{Q}$ combinations for doing this. Similarly, there are $\binom{N}{P}$ combinations for choosing the Y-variables.

In all BnB algorithms, the running time is extremely dependant on the selection of the order in which the decision variables are processed. If the "crucial" variables are selected at a low depth of the branching, the search tree will be narrowed down, thus reducing the running time. In the case of the HCLP problem, the Z variables are more significant, because they represent the facilities that provide both level 1 and level 2 service and thus completely determine if a demand area is covered (both constraints (2) and (3) are satisfied). The Y variables only partially determine if a constraint (2) is met.

Further, if q and d are such that $(Q-q) < (N-d)/2$, then it follows that $\binom{N-d}{Q-q} < \binom{N-d}{Q-q+1}$. Therefore, when BnB is searching for a solution at $depth = d$, the size of the remaining sub-tree will be smaller if the number of variables already set to 1 was q than if it was $q-1$.

For these reasons, the branch-and-bound algorithm is implemented in two stages. In the first stage, the Z binary variables are determined, and in the second stage, the Y variables. Branching is realized via auto-recursive calls, first by setting the value of a selected binary decision variable to 1, then to 0. A greedy strategy is adopted: variables that add most to the current solution are first selected. This is in line with first trying the value of 1 for variables.

Finally, for the same reasons as discussed above, the size of the search tree would be smallest if we could somehow already know the best solution and first process those Z variables that are set to 1 in the best solution. Because of this, the following strategy is adopted: the BnB algorithm runs until a solution better than the currently best solution is found. When such solution is found, the BnB stops and restarts, this time first processing the Z variables from the previous best solution. This is repeated until the BnB finishes without finding a better solution, which means that the search space is exhausted and no better solution exists.

The "Bound" part of the BnB is implemented by solving the linear programming (LP) relaxation of HCLP in which X, Y and Z variables are real numbers between 0 and 1. Such relaxation of the problem has polynomial complexity and can be solved very efficiently by using standard methods. An external library is used for finding the LP relaxation solution.

Table 1: Encoding of the first 3 Z-variables for each thread

Thread 0			Thread 1			Thread 2			Thread 3			Thread 4			Thread 5			Thread 6			Thread 7		
0	0	0	0	0	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	0	1	1	1
Z(2)	Z(1)	Z(0)	Z(2)	Z(1)	Z(0)	Z(2)	Z(1)	Z(0)	Z(2)	Z(1)	Z(0)	Z(2)	Z(1)	Z(0)	Z(2)	Z(1)	Z(0)	Z(2)	Z(1)	Z(0)	Z(2)	Z(1)	Z(0)

In every node of the search tree, the LP relaxation solution is obtained by passing the currently fixed Z and/or Y variables and calling the LP solver. If such solution cannot be better than the current best binary solution, the search of that sub-tree is abandoned and the BnB algorithm backtracks.

Parallelization is used to speed up the search process. In the initialization phase, depending on the parameter k, such that $threads = 2^k$, first k "best" Z variables are selected. Then, each of these Z variables is set to 0 or 1 (depending on a thread) and the BnB algorithm is called in parallel on every thread. The following table shows the encoding of the first k Z variables for each thread. In this example, $k = 3$, $threads = 8$.

After the same k Z variables were initiated differently for each thread, the BnB algorithm is called in parallel for all threads at the same time. As previously stated, as soon as any thread finds a better solution than the current best solution, all threads are restarted, going through the same initialization process.

The pseudo-code of this algorithm follows.

```
ParallelBnB(k)
{
  //Z variables of the current best solution
  previousBestZ = array of N 1s

  repeat {
    K = findIndexesOfBestZLocations(k)

    //this section is executed in parallel
    in parallel do {
      //every threads reads its ThreadID, from 0 to k-1
      get ThreadID
      empty Z array
      for i = 0 to k-1 {
         Z(K(i)) = binary digit on the position "i" in ThreadID
      }
      //q=the number of so far established Z locations. q<=k<=Q
      q = number of 1s in the binary encoding of ThreadID
      call BnB(ThreadID,q,k)
```

```
      //threads finish when they complete the search or
      //when they receive the restart signal
      wait until all threads finish
    } end parallel
  } until no restart signal received
  //if there was a restart signal then some thread has improved
  //the best solution so we must go back and search again using
  //the new best solution as a start
  //if no restart - the search was completed without improving
} end ParallelBnB

//First stage of BnB, for selecting the Z variables
BnB(ThreadID, q, depth)
{
  stop if restart signal received;

  //Q Z-locations established. Now find Y locations
  if(q==Q)
  {
    empty Y array
    //go to the second stage of BnB, to select Y variables
    selectY(ThreadID, 0, 0);
    return;
  }

  //BOUND.
  //The binary solution we are looking for cannot be better
  //than the real solution with the currently established Zs
  if(LPSolution(Z array) <= bestValue)
    return;

  //find "best" Z candidate location
  z = findIndexOfBestZ();

  Z(z) = 1;
  //BRANCH. 1 more Z-location established.
  BnB(ThreadID,q+1,depth+1);

  //check if there are enough Z locations left
  if(depth+Q-q<n)
  {
    Z(z) = 0;
    //BRANCH. No new Z-locations established
    BnB(ThreadID,q,depth+1);
  }
```

```
} end BnB


//Second stage of BnB, for selecting the Y variables
selectY(ThreadID, p, depth)
{
  stop if restart signal received;

  //BOUND. The binary solution we are looking for cannot be better
  //than the real solution with the currently established Zs and Ys
  if(LPSolution(Z array,Y array) <= bestValue)
    return;

  y = findIndexOfBestY();
  //if this is the last location that can be established
  if(p+1==P) {
    Y(y) = 1;
    X array = 1/0 as in constraints (2) and (3)
    value = objective function
    //if we improved the best result
    if(value > bestValue)
    {
      //set the global best solution
      bestValue = value;
      store current solution as the best solution,
      store Z array into previousBestZ array,
      send restart signal to all threads
    }
    return;
  }
  Y(y) = 1;
  //BRANCH. 1 new Y location established.
  selectY(ThreadID,p+1,depth+1);

  //check if there are enough Y locations left
  if(depth+P-p<N)
  {
    Y(y) = 0;
    //BRANCH. 0 new Y locations established.
    selectY(ThreadID,p,depth+1);
  }
} end selectY


//Finds k "best" Z locations which are used to initialize BnB
```

```
findIndexesOfBestZLocations(k) {
  empty set K
  for i = 0 to k-1
    call findIndexOfBestZ() and add that index to the set K
  return set K
} end findIndexesOfBestZLocations

//Finds one "best" unused Z location
findIndexOfBestZ()
{
  //we first try with the Zs from the previous best solution
  find i, i in previoustBestZ, if Z(i) not defined
  with the maximum value of variable add(i)
    //How much this Z adds to the current population covered.
    //This is the population covered by Z(i),
    //but not already covered by other established Zs
    add(i) = additional population covered by Z(i)
  if found such i, return its index

  //if not found do the same for other Zs
  find i, i not in previoustBestZ, if Z(i) not defined
  with the maximum value of variable add(i)
    add(i) = additional population covered by Z(i)

  return its index
} end findIndexOfBestZ

//Finds one "best" unused Y location
findIndexOfBestY() {
  find i, if Y(i) not defined
  with the maximum value of variable add(i)
    //How much this Y adds to the current population covered.
    //This is the population covered by Y(i),
    //but not already covered by other established Zs and Ys
    add(i) = additional population covered by Y(i)

  return index i of best Y
} end findIndexOfBestY
```

4. Computational Results

The parallel BnB algorithm for solving HCLP was implemented using the C programming language and parallelization was achieved by using the *openMP* library. The same test instances were used as in [1]. One instance, $g5$, which proved to be the most difficult, was tested 3 times, with different values of P and Q. The results obtained by the parallel BnB were compared to the results obtained by the CPLEX 10.1 solver. Both BnB and CPLEX tests were carried out on an Intel Core i7-860 2.8 GHz with 8GB RAM memory under Windows 7 Professional operating system. This processor has 8 cores and each BnB test was run 4 times, using 1, 2, 4 and 8 threads (cores) respectively. CPLEX 10.1 does not take advantage of multi-core processors.

Since both CPLEX and BnB are exact methods, the objective function result was always the same in each of the 5 test runs (1 CPLEX and 4 BnB), for every test instance.

The small test instances, with $N = M \leq 100$ posed no challenge neither to BnB nor to CPLEX, with solving times very close to 0. The solving time for test instances with $N = M = 150$ depended mostly on the value of the Q parameter, again proving that determining the Z variables is more important for the running time than the Y variables.

It is important to note that the BnB algorithm does not necessarily take the same solving path when run with different number of threads, because of the "restart" strategy. The order in which threads are launched is not deterministic; also, the processor (core) usage is allocated by the operating system and therefore it is also not deterministic. When one thread obtains a better solution it sends the restart signal to other threads that at that point still have not finished. For the non-deterministic reasons stated above, when run with various numbers of threads, this causes different order in which currently best solutions are obtained, which further causes different order of restarts. This only affects the time needed to reach the solution that later cannot be further improved. Once such solution is obtained and no more restarts are done, the BnB becomes completely deterministic. This is reflected in the test results.

The columns in the following table are: "name" is the name of a test instance, followed by N, M, Q and P parameters; "obj" shows the objective function result which is always the same for CPLEX and BnB; columns "Time" and "Sol.Time" represent the total solving time and the initial time when the best solution was reached, respectively. In the case of the BnB this is also the time of the final restart. Finally, for BnB, the number of Z and Y nodes traversed is shown, as well as the number of LP relaxation cuts.

Table 2: Computational results, 1 and 2 threads

Instance					CPLEX 10.1		BnB 1 Thread				BnB 2 Threads				
Name	N(=M)	Q	P	obj	Time	Sol.Time	Time	Sol.Time	Z nodes	Y nodes	LP cuts	Time	Sol.Time	Z nodes	Y nodes	LP cuts
hclp20	20	3	3	349	0.011	0.01	0.01	0	7	5	7	0.01	0.01	25	42	9
hclp45	45	3	2	1009	0.334	0.1	0.15	0.07	24	9	22	0.16	0.08	52	26	42
hclp50	50	5	5	1106	0.055	0.03	0.16	0.15	31	63	25	0.25	0.23	129	53	114
g1	100	5	5	2069	0.505	0.377	0.44	0.14	29	28	26	0.39	0.24	50	39	41
g5	150	16	22	10587	45.905	45.836	111.15	3.42	6086	515	6081	101.6	13.24	8131	241	8105
g5b	150	19	25	11366	173.589	138.164	343.95	6.46	19213	400	19211	249.71	7.64	24205	199	24202
g5c	150	20	5	11160	5.818	5.213	44.13	26.48	3594	55	3527	40.33	26.84	5491	100	5406
g7	150	7	7	8141	3.032	2.879	3.71	2.85	162	107	152	2.17	1.36	111	54	109
g8	150	6	6	7453	2.144	1.734	2.56	1.72	105	149	83	1.01	0.11	42	11	43
g9	150	17	23	10873	92.03	41.228	232.74	16.98	12869	959	12827	165.22	25.66	16859	296	16795

Table 3: Computational results, 4 and 8 threads

Instance					CPLEX 10.1		BnB 4 Threads				BnB 8 Threads				
Name	N(=M)	Q	P	obj	Time	Sol.Time	Time	Sol.Time	Z nodes	Y nodes	LP cuts	Time	Sol.Time	Z nodes	Y nodes	LP cuts
hclp20	20	3	3	349	0.011	0.01	0.01	0.01	8	20	8	0.02	0	3	6	7
hclp45	45	3	2	1009	0.334	0.1	0.16	0.08	58	36	44	0.14	0.06	52	34	44
hclp50	50	5	5	1106	0.055	0.03	0.04	0.02	16	28	16	0.07	0.02	16	19	21
g1	100	5	5	2069	0.505	0.377	0.25	0.17	37	18	39	0.33	0.22	87	71	80
g5	150	16	22	10587	45.905	45.836	64.77	6.86	8172	88	8172	42.59	0.19	7219	74	7225
g5b	150	19	25	11366	173.589	138.164	170.05	17.01	25633	355	25554	146.16	16.11	37125	1138	36943
g5c	150	20	5	11160	5.818	5.213	11.22	1.07	2168	74	2156	8.48	0.63	2123	54	2125
g7	150	7	7	8141	3.032	2.879	1.65	0.24	135	43	132	1.43	0.63	167	88	159
g8	150	6	6	7453	2.144	1.734	0.92	0.17	44	22	46	1.13	0.78	124	88	104
g9	150	17	23	10873	92.03	41.228	111.2	20.39	18538	601	18470	83.14	8.51	18060	257	18015

The test results shown above demonstrate that the BnB algorithm reaches optimal solutions extremely quickly, much faster than the CPLEX solver. For the most difficult test instance $g5b$ BnB took 6.46 to reach the optimal solution, while CPLEX solver took 138.164sec. The instances $g5b$ and $g5c$ were created from the original instance $g5$ by changing the parameters P and Q. For the original instance $g5$, the differences are much bigger: BnB took 0.19sec to reach the optimal solution, while CPLEX took 45.836sec.

For those difficult test instances, BnB might take long time to exhaust the search space and thus prove that the solution reached is actually the optimal one. However, when run using 8 processor cores, BnB is still faster than the CPLEX solver. For the test instance $g5b$ this time was 146.16sec while CPLEX took 173.589sec. It is important to note that CPLEX uses various optimization techniques and cuts which this implementation of BnB does not use.

Parallelization resulted in significant speedups. On some instances, parallelization resulted in super linear speedups of the time until optimal solution is reached. However, it is difficult to judge the effects of parallelization because of the non-deterministic reasons stated above. Larger test instances are needed to better understand the benefits of parallelization.

5. Conclusions

In this paper we presented a parallel branch and bound algorithm with restarts, for solving the Hierarchical Covering Location Problem (HCLP). This is a general approach and can be applied to various other problems with binary variables.

The test results prove the effectiveness of this approach and the advantages of this method over another exact method, CPLEX. Unfortunately, larger test instances were not available, but considering the way the BnB algorithm is designed it can be assumed that the time needed to reach the "best" (or at least near-optimal) solution would still be very short. A potential use of this approach is for solving extremely large instances that cannot be solved to optimality. Using hybrid methods, BnB can be used to obtain "very good" solutions very quickly and these solutions can be used as a starting solution for some other methods (for example: an evolutionary algorithm).

Our future research in this area will focus on generalizing the approach presented herein and applying it to other optimization problems.

References

[1] M. Maric et al. *One Genetic Algorithm for Hierarchical Covering Location Problem*, 9th WSEAS International Conference on EVOLUTIONARY COMPUTING (EC'08), Sofia, Bulgaria, May 2–4, 2008.

[2] Espejo, L.G.A. et al. *Dualbased heuristics for a hierarchical covering location problem*, Computers & Operations Research, Vol.30, 2003, pp. 165–180.

[3] Galvao RD et al. *A Lagrangean heuristic for the maximal covering location problem*, European Journal of Operational Research, **88**, 1996; pp. 114–23.

Faculty of Mathematics,
Belgrade University
Studentski Trg 16
Belgrade, 11000 SERBIA
E-MAIL: djapedjape@gmail.com