Greedy Big Steps as a Meta-Heuristic for Combinatorial Search

Haiou Shen and Hantao Zhang
Computer Science Department
The University of Iowa
Iowa City, IA 52242
{hshen, hzhang}@cs.uiowa.edu

Abstract
We present a meta-heuristic called “greedy big steps” for combinatorial search. This meta-heuristic can be used for both complete backtrack search and incomplete local search. We illustrate this meta-heuristic by presenting a greedy big steps branch-and-bound algorithm which combines local search, branch-and-bound and constraint satisfaction together. We apply this algorithm to the traveling tournament problem (TTP) to show the power of the algorithm. The computational results show that our algorithm can produce good results for highly complex TTP instances. Our results for three TTP instances, CIRC10, CIRC12, and CIRC16, are still the best overall results.

Introduction
Many problems in AI and computer science involve combinatorial search and almost every NP-complete problem can be solved by combinatorial search. In such problems, we try to assign a value to every variable so that all the constraints are met or a utility function reaches optimal. To conduct a complete combinatorial search, the classical depth-first, backtrack search procedure is often used. If the problem involves combinatorial optimization, the classical branch-and-bound procedure is often used. In such procedures, a partial solution, initially empty and guided by a heuristic, is extended one variable at a time along the current search path, until a complete solution or an optimal solution is found.

Heuristics play an important role in the performance of these search algorithms. However, in the beginning of the backtrack search, the heuristic function often makes poor choices because of lack of information. For complex combinatorial problems, these poor choices may prevent the search from finding a solution in limited time. To solve this problem, we propose the so-called “greedy big steps” meta-heuristic to increase the power of combinatorial search. A meta-heuristic is a general method based on very general principles and applicable to a large number of problems (Re-sende & de Sousa 2003). Meta-heuristics are known today to be among the most powerful methods for tackling hard and large combinatorial problems. Target applications of meta-heuristics concern those highly combinatorial and strongly constrained problems which cannot be solved other ways. Typical examples of such applications include resource assignment, planning, scheduling and time tabling and inventory.

Given a partial solution $A$ and a positive integer $K$, let $N(A, K)$ be the set of all partial solutions extended from $A$ by assigning values to $K$ more variables. For conventional backtrack procedures, a heuristic function picks the best candidate from $N(A, 1)$; using the greedy big steps meta-heuristic, a heuristic function picks a candidate from $N(A, K)$, where $K > 1$. Because candidates in $N(A, K)$ contain more information than those in $N(A, 1)$, we can expect that the heuristic function can make better choices.

Of course, now the search tree has a much larger branching factor than before. We can do nothing to reduce the branching factor if we want to maintain the completeness of our algorithm. The whole solution space is there and we cannot change this. If we reduce the branch level (i.e., the depth of the search tree), the branching factor will increase. For many complex combinatorial problems, finding an optimal solution is not practical using today’s computers. Since the search space for these problems is so huge, it is unlikely we can find an optimal solution within a reasonable time. For this reason, we employ the greedy method to select $m$ best children among all possible candidates to explore. By doing this, the algorithm is no longer a complete search algorithm. The search space can be greatly reduced because of reduced branch levels and limited branching factor, and we have the flexibility to change the branching factor depend on problems. Despite of reduced search space, the big steps in many cases provide enough information and allow us to choose good candidates in such reduced search space.

Another important approach to combinatorial search is local search or greedy methods. The greedy big steps meta-heuristic can also help local search methods. In this case, a partial solution is a complete assignment of all variables and we let $N(A, K)$ be the set of all partial solutions which differ with $A$ on $K$ variables. The conventional local search method chooses the best candidate from $N(A, 1)$; using the greedy big steps meta-heuristic, a candidate is chosen from $N(A, K)$, where $K > 1$. A serious problem with the local search method is that the search often stuck at a local optimum. Because $N(A, K)$ provides a wider neighborhood than $N(A, 1)$, we can expect that the former will be more
likely useful to avoid local optimums than the latter.

Because of space limitation, we will focus on using the greedy big steps meta-heuristic in a branch-and-bound search procedure. To illustrate the strength of this new procedure, we will use it to solve the traveling tournament problem (TTP), proposed in (Easton, Nemhauser, & Trick 2001).

In recent years, the scheduling of sport leagues has become an important class of combinatorial optimization applications. Schaerf (Schaerf 1996) used constraint logic programming to solve Dutch “top League” and Italian “Series A” of football. Henz (Henz 2001) employed finite-domain constraint programming to solve an ACC basketball schedule. Hamiez and Hao (Hamiez & Hao 2001) used tabu search to solve a specific sports league scheduling problem. Zhang (Zhang 2002) uses a SAT solver to solve a BigTen basketball schedule problem within one second. Their methods can solve sport scheduling problems with a variety of success. For the optimization problem, integer programming methods do better according to a review (Easton, Nemhauser, & Trick 2001). When constraints and optimization are mixed in one problem, it is very difficult for both constraint programming and integer programming methods. TTP is such a problem with mix of constraints and optimization.

The goal of TTP is to find a feasible solution satisfying all constraints for home and road games with the minimal total traveling distance. TTP is an unusual hard optimization problem and gains a lot of research interest (Anagnostopoulos et al. 2003; Benoist, Laburthe, & Rottemboung 2001; Cardemil 2002; Leong 2003). There is a web site, http://mat.gsia.cmu.edu/TOURN/, maintained by Michael Trick, which contains the latest results. Our method produced the best solutions for nine instances of TTP. Some of these solutions are improved by other researchers team (Anagnostopoulos et al. 2003). At the time this paper was written, we still keep the best records for three instances.

This paper is organized as follows: In the next section we will present the branch-and-bound algorithm with the greedy big steps meta-heuristic. In section 3 we define the TTP problem and show its computational complexity. In section 4 we show how we apply the greedy big steps branch-and-bound algorithm for TTP. Section 5 gives some computational results for some instances of TTP.

**Greedy Big Steps Branch-and-Bound Algorithm**

In this section, we will present a general branch-and-bound algorithm with the greedy big steps (GBS) meta-heuristic to find a minimal-cost solution satisfying a set \( C \) of constraints over a set of variables. The algorithm GBS-BB is shown in Figure 1. In the algorithm, \( C \) is the set of constraints which must be satisfied. We use Unsatisfiable(\( C \)) to test if \( C \) is unsatisfiable or not. Depending on the problem, \( C \) could be a set of propositional clauses, or a set of first-order formulas over a finite domain. Because of this, the implementation of Unsatisfiable() is problem-specific.

\( A \) is the current partial solution. It will be extended from an empty assignment to a complete assignment. Cost(\( A \)) is the current partial solution. It will be extended from an empty assignment to a complete assignment. Cost(\( A \))

![Figure 1: A greedy big steps branch-and-bound algorithm.](image)

```
function GBSValue (A, K)
    Tabu := ∅
    while (TRUE) do
        (C′, A′) := BestMove (C, A, K, Tabu)
        if (Unsatisfiable (C′)) then return MaxValue end if
        if (A′ is a full solution) then return Cost (A′) end if
        Tabu := Tabu ∪ \{A′\}
        if (LowerBound (A′) < BestCost) then
            x := GBS-BB (C′, A′, K)
            if (x < BestCost) then BestCost := x end if
        end if
    end do

end function
```

return the cost for solution \( A, K \), a positive integer number, denotes step size. \( Tabu \) is a list of partial solutions to be avoided. BestMove() function returns a new constraint set and a new partial solution \( A′ ∈ N(A, K) \). That is, \( A′ \) is an extension of \( A \) by assigning values to \( K \) more variables. If there is no more move at this stage, it will return FALSE.

The function BestMove() is also problem-specific, where heuristics are used. If \( Tabu \) already has \( n \) elements, where \( n \) is defined by the user, then BestMove returns FALSE as the next constraints. If \( Tabu \) is empty, then the heuristic function for selecting variables is called to select \( K \) unsigned variables (if there are less than \( K \) variables, then all the remaining variables are selected). If \( Tabu \) is not empty, then the previously selected \( K \) variables are collected into \( V \). Then BestValues(\( C, V, Tabu \)) is called to select the best variables for these \( K \) variables in \( V \), excluding those already used in \( Tabu \) and those which violate \( C \). The heuristic function for selecting values is called in BestValues(). The function Propagation() will simplify \( C \) using the new partial assignment \( B \). Finally, the simplified constraints and a new partial solution are returned.

At first, \( A \) is an empty assignment, \( C \) is the set of original constraints for the problem. After calling BestMove(\( C, A, K, Tabu \)), we obtain a partial solution \( A′ \), which is \( K \) steps further than \( A \), and we get a new constraint set \( C′ \), which defines next available candidates. For the new partial solution \( A′ \), we check the lower bound of \( A′ \). If it is less than the current best cost, we will recursively call GBS-
BB for this branch.

GBS-BB is an exhaustive search function. We can limit the search space by limiting the branching factor. To do this, we limit the length of the Tabu list. If the Tabu list already contains \( n \) partial solutions, the BestMove function will return FALSE, which will lead to an early exit. By doing this, it is no longer a complete search method.

**Example** Suppose we want to solve the maximum satisfiability problem (MAX-SAT) using GBS-BB (Hansen & Jaumard 1990). The constraint set \( C \) is a set of propositional clauses and the cost function returns the number of false clauses in \( C \) under the current partial assignment. The goal is to find a truth assignment that satisfies the maximum number of clauses for MAX-SAT, or equivalently, has the minimum number of false clauses. If we let \( K = 10 \), then we choose ten variables to assign at one node and the node may have up to \( 2^{10} \) children but the height of the search tree is reduced 10 times. If we allow only the best five of the \( 2^{10} \) children to be explored, then the search is incomplete.

**Traveling Tournament Problem**

Creating fair sports schedules for professional leagues and colleges is a very difficult and important task and many researchers used computer methods to create them (Nemhauser & Trick 1998; Russell & Leung 1994; Henz 2001; Zhang 2002). For some sport schedule problems, the constraints about fairness are not enough, and we need to consider the total traveling distance of every team to save money and time. For example, teams in the Major League Baseball (MLB) locate in 17 states of United States and 2 states of Canada. The states stretch from the American east coast to the west coast and from Canada to Florida and California. The total travel distance becomes an important issue for MLB teams. The traveling tournament problem (TTP) tries to consider fair constraints and total travel distance at the same time.

Suppose we have \( n \) teams, \( t_i \) \((i = 1, 2, ..., n)\), \( n \) is a even number. For every two teams \( t_i \) and \( t_j \), \( i \neq j \), \( t_i \) needs to play against \( t_j \) once at \( t_i \)’s home and once at \( t_j \)’s home. So every team needs to play \( 2(n - 1) \) games for one game season. Teams are assumed to be initially at their home and they need to come home after the tournament. Beside this, we have two more rules.

- No more than 3 consecutive road games or 3 consecutive home games are allowed.
- No two teams can play each other in consecutive games.

The optimal solution of TTP is a feasible schedule satisfying all constraints and the total travel distance is minimized.

Easton, Trick and Nemhauser gave two instance classes in (Easton, Nemhauser, & Trick 2001): the national league (NL) instances and the circle (CIRC) instances. The original national league problem has 16 teams. To generate smaller instances, they simply take subsets of the teams. The circle instances are introduced to further simplify the problem. The circle instances assume all cities are on a circle and the distance between two neighbors is 1. A circle instance has a trivial symmetric TSP solution for each team. So a circle instance should be easier than a national league instance. These problem instances can be found at: http://mat.gsia.cmu.edu/TOURN/. On this web site, they provide 9 circle instances: CIRC4, CIRC6, CIRC8, CIRC10, CIRC12, CIRC14, CIRC16, CIRC18, and CIRC20, and 7 National League instances: NL4, NL6, NL8, NL10, NL12, NL14, and NL16.

TTP is a mix of constraint problem and optimization problem. These instances have a huge and complex solution space and are not as simple as they appear to be. Instances of size 4 are easy. A good algorithm can find the optimal solution within 1 second. Instances of size 6 are much harder than size 4. Instances of size greater than 6 are still open today.

Let us use a table to denote a solution. A feasible solution for CIRC4 is illustrated in Table 1. In this table, we can see that there are six time slots for every team. If team T1 plays T2 at T1’ s home at time slot 3, we write “T2” in row 3 column T1 and “@T1” in row 3 column T2. We have some choices for every cell of the table. The total space is decided by the number of choices for every cell and the table size. For size 4 there are less than 2000 feasible solutions by enumeration. For size 6, it is hard to estimate the size of its solution space. Suppose we have only two choices for every available cell in the table. Since the table has 60 cells, it has about \( 2^{60} \) feasible solutions. For size 8, the solution space is much larger than \( 2^{60} \). It is too huge for a search algorithm to find the optimal solution. The constraints make thing ever worse. With all the constraints, it is not trivial to find a feasible solution for large instances and we need to find the best one in all feasible solutions.

Easton, Nemhauser and Trick attempted to solve the problem using both constraint programming and integer programming (Easton, Nemhauser, & Trick 2001). Benoist, Laburthe and Rottembourg (Benoist, Laburthe, & Rottembourg 2001) combined Lagrangean relaxation and constraint programming for this problem. Cardemil (Cardemil 2002) has tried tabu search. According to the TTP web site Xingwen Zhang obtained quite a number of good results, using a combination of constraint programming, simulated annealing and hill-climbing method (Leong 2003). Hentenryck et al. (Anagnostopoulos et al. 2003) employed simulated annealing to attack this problem and obtained by far the best results for several instances.

<table>
<thead>
<tr>
<th>Table 1: CIRC4 Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

GBS-BB for TTP

Let us first use the size four circle instance example in Table 1 to illustrate how the classic branch-and-bound algorithm would solve this problem. The branch-and-bound algorithm
branches and checks bound at every cell in the table, walking one step down every time, as illustrated in Figure 2. At every branching point, the algorithm assigns a possible value to the next available cell and uses the lower bound function to cut branches. For TTP, the branch-and-bound algorithm is not effective enough because the search is deep and the branch factor is high. For size 8, the search tree has about 56 levels. For size 16, it is 240. TTP also has many constraints, and it is not easy to find a tight lower bound function to cut branches. We can see that the search space is too huge and complex for the branch-and-bound algorithm to find optimal solution.

To overcome this problem, without walking one small step at each branching point, we can use the greedy big steps meta-heuristic to jump a big step in the search process. Let $K$ in the meta-heuristic be equal to the number of teams in TTP, we then assign one whole row at each branching point. Figure 3 shows the idea. At first, the table is empty. To go one big step down, we choose the first empty row and assign the value to each cell in the row so that the total distance is minimal. For the next step, we select the next empty row with values compatible with the current partial solution and the total distance is minimal. If there are no more feasible values for the next row, we backtrack and choose the next best values for the previous row. If there is one, we go one big step further, until the whole table is filled out. When we find a feasible solution which is better than the previous solution, we save it and backtrack to find more. If the lower bound of the current partial solution is no better than the current best solution, we backtrack, too. This is an exhaustive and complete search procedure.

By doing this, the search tree has only $2(n-1)$ levels. Of course, now we have a much larger branching factor than before. The degree of the root of the search tree is 12 for size 4, 120 for size 6, 1680 for size 8 and 30240 for size 10. Since the search space is so huge for TTP, we employ the greedy method to select $m$ best branches among all possible branches to explore.

The algorithm GBS-BB-TTP in Figure 4 is following the GBS-BB algorithm in Figure 1. In this algorithm, we need to find the $m$ best branches at each node and it is not a trivial job. Based on the partial solution we already have, we need to generate constraints for the next step. Using those constraints we can generate all feasible rows for this level. If we have all feasible rows, we can pick the $m$ best rows. This is not practical when the problem’s size is big. To speed up this procedure, we employ another branch-and-bound algorithm and a lower bound function to find the $m$ best rows. The algorithm maintains a list, which is empty initially. When we find a feasible row, we insert it into the list until the list has $m$ candidates. After the list is full, i.e., it has $m$ candidates, we use the maximal cost of the $m$ candidates as the bound to search for new candidates. If a better candidate is found, we replace one of the maximal cost with the new one.

It is well-known that tight lower bound is the key for the efficiency of a branch-and-bound algorithm. (Easton, Nemhauser, & Trick 2001) use Independent Bound (IB), which is computed as a traveling salesman problem (TSP). IB does not consider the “no repeaters” rule and the “no more than three consecutive home or three consecutive road games” rule. We have used a tighter lower bound with those additional rules. Since the size of the TSP problem is relatively small, we use another branch-and-bound algorithm to compute IB.

**Computational Results**

To test the algorithm GBS-BB-TTP’s strength, we ran the program to solve NL instances and CIRC instances. The test bed for Table 2 is a cluster of 24 Pentium 4 2.4GHz machines with 1G memory. From Table 2, we can see the program can find some very good results for many instances of TTP even using a branching factor between 2 and 4. For small size instances such as NL4, NL6, CIRC4, CIRC6 and CIRC8, it finds the current best results like other competitive algorithms.

The best results found by our algorithm for CIRC10, CIRC12, and CIRC16 remain the best overall results. Tables 3, 4, 5 show the detailed solutions for CIRC10, CIRC12 and CIRC16. The best results for CIRC14, CIRC18, and CIRC20 found by our algorithms were late surpassed by (Anagnostopoulos et al. 2003); these results are still better...
Figure 4: A greedy big steps branch-and-bound algorithm for TTP.

IntegerMatrix D; // Distance matrix, used in  
BestMoves(A, i, M), which returns the M row candidates to be filled at the i-th row of the table.

procedure GBS-BB-TTP (integer N, M)
  // N is the number of teams.
  // M is the branching factor.
  Row-List Best_A; // Best assignment.
  integer Best_T = MAXINT;
  // Best total travel distance.
  integer level = 1;
  Row-List RowList[2*(N-1)];
  RowList[level] = BestMoves(A, level, M);
  while (TRUE) do
    if (RowList[level].size == 0) then
      if (level == 1) then return ;
      else
        level--; A.popup();
        continue;
    end if
    end if
    A.push(RowList[level].popup());
    level++;
    if (level == 2*(N-1)) then
      RowList[level] = BestMoves(A, level, 1);
      if (RowList[level].size == 0) then
        level--; continue;
      end if
      A.push(RowList[level].popup());
      Dist = TravelDistance(A, D);
      if (Dist < Best_T) then
        Best_A = A; Best_T = Dist;
      end if
      A.popup(); level--; continue;
    else
      integer LB = LowerBound(A, level);
      if (LB < Best_T) then
        RowList[level] = BestMoves(A, level, M);
      end if
      else
        A.popup(); level--; continue;
      end if
    end do
end procedure

Table 2: The experimental results on the TTP instances. The data for Cardemil, Zhang, Van Hentenryck and the current best results come from the TTP web site. The best results for GBS-BB-TTP are collected on a P4, 2.4G cluster. The results for GBS-BB-TTP with different branch factors are collected on the same machine with one hour limit. If the program does not stop in one hour, we stop the search and use the current best result the program has found so far.

<table>
<thead>
<tr>
<th>Instance</th>
<th>GBS-BB</th>
<th>GBS-BB</th>
<th>Cardemil</th>
<th>Zhang</th>
<th>Van Hentenryck</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BF</td>
<td>Cost</td>
<td>Best</td>
<td></td>
<td></td>
<td>Best</td>
</tr>
<tr>
<td>NL4</td>
<td>2</td>
<td>8276</td>
<td>4</td>
<td>8276</td>
<td></td>
<td>8276</td>
</tr>
<tr>
<td>NL6</td>
<td>4</td>
<td>20844</td>
<td>4</td>
<td>23916</td>
<td></td>
<td>23916</td>
</tr>
<tr>
<td>NL8</td>
<td>2</td>
<td>45009</td>
<td>4</td>
<td>44166</td>
<td></td>
<td>39712</td>
</tr>
<tr>
<td>NL10</td>
<td>3</td>
<td>60602</td>
<td>5</td>
<td>61602</td>
<td></td>
<td>59984</td>
</tr>
<tr>
<td>NL12</td>
<td>4</td>
<td>120661</td>
<td>4</td>
<td>117608</td>
<td></td>
<td>119042</td>
</tr>
<tr>
<td>NL14</td>
<td>6</td>
<td>40960</td>
<td>4</td>
<td>40416</td>
<td></td>
<td>40416</td>
</tr>
<tr>
<td>NL16</td>
<td>2</td>
<td>903338</td>
<td>4</td>
<td>89809</td>
<td></td>
<td>278092</td>
</tr>
<tr>
<td>CIR4</td>
<td>2</td>
<td>20</td>
<td>2</td>
<td>20</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>CIR6</td>
<td>7</td>
<td>74</td>
<td>1</td>
<td>64</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>CIR8</td>
<td>2</td>
<td>148</td>
<td>1</td>
<td>132</td>
<td>136</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>141</td>
<td>1</td>
<td>141</td>
<td>139</td>
<td>139</td>
</tr>
<tr>
<td>CIR10</td>
<td>5</td>
<td>274</td>
<td>3</td>
<td>254</td>
<td>282</td>
<td>254</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>262</td>
<td>3</td>
<td>262</td>
<td>290</td>
<td>290</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>552</td>
<td>4</td>
<td>552</td>
<td>582</td>
<td>582</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1184</td>
<td>5</td>
<td>1184</td>
<td>1218</td>
<td>1218</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>1508</td>
<td>5</td>
<td>1508</td>
<td>1544</td>
<td>1544</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>2048</td>
<td>5</td>
<td>2048</td>
<td>2188</td>
<td>2188</td>
</tr>
</tbody>
</table>

Table 3: CIR10 Solution, Cost: 254

<table>
<thead>
<tr>
<th>MIL</th>
<th>NYM</th>
<th>FLA</th>
<th>CIN</th>
<th>PHI</th>
<th>CHI</th>
<th>STL</th>
<th>ATL</th>
<th>MON</th>
<th>TL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NYM</td>
<td>FLA</td>
<td>CIN</td>
<td>PHI</td>
<td>CHI</td>
<td>STL</td>
<td>ATL</td>
<td>MON</td>
<td>TL</td>
<td>MIL</td>
</tr>
<tr>
<td>FLA</td>
<td>CIN</td>
<td>PHI</td>
<td>CHI</td>
<td>STL</td>
<td>ATL</td>
<td>MON</td>
<td>TL</td>
<td>MIL</td>
<td>NYM</td>
</tr>
<tr>
<td>CIN</td>
<td>PHI</td>
<td>CHI</td>
<td>STL</td>
<td>ATL</td>
<td>MON</td>
<td>TL</td>
<td>MIL</td>
<td>NYM</td>
<td>FLA</td>
</tr>
</tbody>
</table>

These results are high competitive. For instance, for instance NL16, since Easton reported the first result in January 2002, there are six updates: Cardemil updated on July 2, 2002; Zhang updated on August 6, 2002 and then on August 28, 2002; Shen updated on October 16, 2002, then on January 6, 2003. Finally, Van Hentenryck updated it on January 14, 2003.

We obtained the best result for CIR12 with a branching factor 3 and for CIR20 with a branching factor of only 2. These experimental results show that our algorithm can make good decisions at very early stage. This algorithm can be easily parallelized on a cluster by letting different machines search on different branches. We obtained best results for large size instances through this kind of parallelization.
Conclusions

We have presented a general meta-heuristic called "greedy big steps" for combinatorial search. We illustrated this meta-heuristic by presenting a greedy big steps branch-and-bound algorithm which combines greedy method, local search, branch-and-bound and constraint programming method together. To show the power of this method, we used this algorithm to attack TTP. Because TTP is a highly complex combinatorial optimization problem, we restricted the branching factor to a small number and still obtained very competitive results. The computational results show that GBS-BB-TTP can solve TTP problems very efficiently. Our results for three TTP instances, CIRC10, CIRC12, and CIRC16, are still the best overall results.

The main advantages of the greedy big steps meta-heuristic are:

- It can be applied to both complete backtrack search and incomplete local search methods.

- The big step made at each branching point in the search provides more global information than a small step can do. This allows heuristic functions to be more effective and leads to a good global solution.

- It is flexible for a user to control the search space by changing the branching factor. When the search space is reduced greatly, the results remain to be of good quality.

- Using this meta-heuristic does not prevent the search procedures from being parallelized.

As future work, we will study properties of GBS and implement other types of search procedures using GBS, such as backtrack procedures for constraint satisfaction problems (CSP) and local search procedures. For TTP, we will study how to combine Van Hentenryck's simulated annealing and our GBS-BB-TTP algorithm. We will also study if a dynamic branching factor will help to improve the performance of GBS-BB-TTP.

References


