Construction heuristics for the asymmetric TSP

Fred Glover
School of Business, University of Colorado, Boulder
CO 80309-0419, USA
Fred.Glover@Colorado.EDU

Gregory Gutin
Department of Mathematical Sciences
Brunel University, Uxbridge, Middlesex, UB8 3PH, U.K.
Z.G.Gutin@brunel.ac.uk

Anders Yeo
Department of Mathematics and Statistics
University of Victoria, P.O. Box 3045, Victoria B.C.
Canada V8W 3P4
yeo@Math.UVic.CA

Alexey Zverovich
Department of Mathematic Sciences
Brunel University, Uxbridge, Middlesex, UB8 3PH, U.K.
Alexey.Zverovich@brunel.ac.uk

Abstract

Non-Euclidean TSP construction heuristics, and especially asymmetric TSP construction heuristics, have been neglected in the literature by comparison with the extensive efforts devoted to studying Euclidean TSP construction heuristics. This state of affairs is at odds with the fact that asymmetric models are relevant to a wider range of applications, and indeed are uniformly more general that symmetric models. Moreover, common construction approaches for the Euclidean TSP have been shown to produce poor quality solutions for non-Euclidean instances. Motivation for remedying this gap in the study of construction approaches is increased by the fact that such methods are a great deal faster than other TSP heuristics, which can be important for real time problems requiring continuously updated response. The purpose of this paper is to describe two new construction heuristics for the asymmetric TSP and a third heuristic based on combining the other two. Extensive computational experiments are performed for several different families of TSP instances, disclosing that our combined heuristic clearly outperforms well-known TSP construction methods and proves significantly more robust in obtaining (relatively) high quality solutions over a wide range of problems.

Keywords: Traveling Salesman, Heuristics, Construction Heuristics

1 Introduction

A construction heuristic for the traveling salesman problem (TSP) builds a tour without an attempt to improve the tour once it is constructed. Most of the construction heuristics for the TSP [12, 15] are very fast; they can be used to produce approximate solutions for the TSP when the time is restricted, to provide good initial solutions for tour improvement heuristics, to obtain upper bounds for exact branch-and-bound algorithms, etc.

Extensive research has been devoted to construction heuristics for the Euclidean TSP (see, e.g., [15]). Construction heuristics for the non-Euclidean TSP are much less investigated. Quite often, the greedy algorithm is chosen as a construction heuristic for the non-Euclidean TSP (see, e.g., [12]). Our computational experiments show that this heuristic is far from being the best choice in terms of quality and robustness. Various insertion algorithms [15] which perform very well for the Euclidean TSP produce poor quality solutions for instances that are not (and not close to) Euclidean.

Hence, it is important to study construction heuristics for the asymmetric TSP (we understand by the asymmetric TSP the general TSP which includes both asymmetric and symmetric instances). Our aim is to describe two new construction heuristics for the asymmetric TSP as well as a combined algorithm based on those heuristics. In this paper we also present results of our computational experiments obtained for several different families of TSP instances. These results show that overall the combined algorithm clearly outperforms well-known construction heuristics for the TSP. While other heuristics produce good quality tours for some families of TSP instances and fail for some other families of instances, the combined algorithm appears much more robust. Being a heuristic the combined algorithm is not always a winner among various heuristics. However, we show that it obtains (relatively) poor quality solutions rather seldom.

For the reader interested in solving the TSP on particular families of non-Euclidean instances, this paper may suggest a construction algorithm, which is appropriate for the families under consideration.

2 Terminology and notation

The vertex set of a weighted complete digraph K is denoted by $V(K)$; the weight of an arc xy of K is denoted by $w_K(x,y)$ (we say that K is *complete* if for every pair x,y of distinct vertices of K both xy and yx are arcs of K.) The *length* of a cycle C (path P) is the number of arcs in C (P). The *asymmetric traveling salesman problem* is defined as follows: given a weighted complete digraph K on n vertices, find a Hamiltonian cycle (*tour*) H of K of minimum weight. The *domination number* of a tour T in K is the number of tours in K which are heavier or of the same weight as T. A *cycle factor* of K is a collection of vertex-disjoint cycles in K covering all vertices of K. A cycle factor of K of minimum (total) weight can be found in time $O(n^3)$ using assignment problem (AP) algorithms (for the corresponding weighted complete bipartite graph) [4, 13, 14]. Clearly, the weight of the lightest cycle factor of K provides a lower bound to the solution of the TSP *(AP lower bound).*

We will use the operation of *contraction* of a (directed) path $P = v_1v_2...v_s$ of K. The result of this operation is a weighted complete digraph K/P with vertex set $V(K/P) = V(K) \cup \{p\} - \{v_1, v_2, ..., v_s\}$, where p is a new vertex. The weight of an arc xy of K/P is

$$w_{K/P}(x,y) = \begin{cases} w_K(x,y) & \text{if } x \neq p \text{ and } y \neq p \\ w_K(v_s,y) & \text{if } x = p \text{ and } y \neq p \\ w_K(x,v_1) & \text{if } x \neq p \text{ and } y = p \end{cases} \tag{1}$$

Sometimes, we contract an arc a considering a as a path (of length one).

3 Greedy and Random Insertion heuristics

These two heuristics were used in order to compare our algorithms with well-known ones. Both heuristics are extensively used especially for the Euclidean TSP [12, 15] since they give consistently good results for the Euclidean problem. Despite being the winner among some other construction heuristics for the (general) symmetric TSP, the greedy algorithm produces rather poor solutions for this problem [12].

The greedy algorithm finds the lightest arc a in K and contracts it (updating the weights according to (1)). The same procedure is recursively applied to the contracted digraph $K := K/a$ till K consists of a pair of arcs. The contracted arcs and the pair of remaining arcs form the "greedy" tour in K.

The random insertion heuristic chooses randomly two initial vertices i_1 and i_2 in K and forms the cycle $i_1 i_2 i_1$. Then, in every iteration, it chooses randomly a vertex ℓ of K which is not in the current cycle $i_1 i_2 ... i_s i_1$ and inserts ℓ in the cycle (i.e., replaces an arc $i_m i_{m+1}$ of the cycle with the path $i_m \ell i_{m+1}$) such that the weight of the cycle increases as little as possible. The heuristic stops when all vertices have been included in the current cycle.

4 Modified Karp-Steele patching heuristic

Our first heuristic (denoted by *GKS*) is based on the well-known Karp-Steele patching (*KSP*) heuristic [13, 14]. The algorithm can be outlined as follows:

1. Construct a cycle factor F of minimum weight.
2. Choose a pair of arcs taken from different cycles in F, such that by patching (i.e. removing the chosen arcs and adding two other arcs that join both cycles together) we obtain a cycle factor (with one less cycle) of minimum weight (within the framework of patching).
3. Repeat Step 2 until the current cycle factor is reduced to a single cycle. Use this cycle as an approximate solution for the TSP.

The difference from the original Karp-Steele algorithm is that instead of joining two shortest cycles together it tries all possible pairs, using the best one.

Unfortunately, a straightforward implementation of this algorithm would be very inefficient in terms of execution time. To partly overcome this problem we introduced a pre-calculated $n \times n$ matrix D of patching costs for all possible pairs of arcs. On every iteration we find a smallest element of D and perform corresponding patching; also, the matrix is updated to reflect the patching operation that took place. Having observed that only a relatively small part of D needs to be re-calculated during an iteration, we cache row minima of D in a separate vector B, incrementally updating it whenever possible. If it is impossible to update an element of B incrementally (this happens when the smallest item in a row of D has been changed to a greater value), we re-calculate this element of B by scanning the corresponding row of D in the beginning of the next iteration. Finally, instead of scanning all n^2 elements of D in order to find its minimum, we just scan n elements of B to achieve the same goal.

Although the improved version has the same $O(n^3)$ worst-case complexity as the original algorithm, our experiments show that the aforementioned improvements yield significant reduction of execution time. Pseudo-code for the approach outlined above follows.

Pseudo-code:

$BestCost$ is D, $BestNode$ is B
Arguments:
N - number of nodes
$W = [w(i,j)]$ - $N \times N$ matrix of weights

(* $Next[i] = j$ if the current cycle factor contains the arc (i, j) *)
$Next$: **array**[1..N] **of integer**;
(* $Cost[i, j]$ is cost of removing arcs $(i, Next[i])$ and $(j, Next[j])$, and
  adding $(i, Next[j])$ and $(j, Next[i])$ instead. *)
$Cost$: **array**[1..N, 1..N] **of integer**;
(* $BestCost[i]$ contains a smallest value found in the i-th row of $Cost$ *)
$BestCost$: **array**[1..N] **of integer**;
(* $BestNode[i]$ is column index of corresponding $BestCost[i]$ in $Cost$ *)
$BestNode$: **array**[1..N] **of integer**;

(* Whenever possible, caches row minimum in $BestNode$ and $BestCost$ *)
procedure UpdateCost(r, c, $newCost$: **integer**);
begin
  if ($r < c$) **then** Swap(r, c); (* Exchange r and c values *)

  $Cost[r, c] := newCost$;

  if $BestNode[r] \neq -1$ **and** $newCost < BestCost[r]$ **then**
    (* New value is smaller than current row minimum *)
    $BestNode[r] := c$;
    $BestCost[r] := newCost$;
  else if $BestNode[r] = c$ **and** $newCost > BestCost[r]$ **then**

    (* Current row minimum has been updated to a greater value *)
    $BestNode[r] := -1$; (* stands for "unknown" *)
    $BestCost[r] := +\infty$;
  end if;
end;

function GetPatchingCost(i, j: **integer**): **integer**;
begin
  if vertices i and j belong to the same cycle **then**
    return $+\infty$; (* patching not allowed *)
  else
    return $W[i, Next[j]] + W[j, Next[i]] - W[i, Next[i]] - W[j, Next[j]]$;
  end if;
end;

BEGIN

  Build a cycle factor by solving LAP on the weights matrix D, store result in $Next$ and number of cycles obtained in M;

  (* Initialize $Cost$, $BestCost$ and $BestNode$ *)
  for $i := 1$ **to** N **do**
    $bn := 1$;

    for $j := 1$ **to** i **do**
      $Cost[i, j] := GetPatchingCost(i, j)$;
      if $Cost[i, j] < Cost[i, bn]$ **then** $bn := j$;
    end for;

    $BestNode[i] := bn$;
    $BestCost[i] := Cost[i, bn]$;
  end do;

  repeat $M - 1$ times

    Find the smallest value in $BestCost$ and store its index in i;
    $j := BestNode[i]$;

    update $Cost$:
      1) for each pair of nodes k and l, such as k belongs to the same cycle as i, and l belongs to the same cycle as j:
        $UpdateCost(k, l, +\infty)$;
      2) for each node m, which does not belong to the same cycle as either i or j:
        $UpdateCost(i, m, GetPatchingCost(i, m))$;
        $UpdateCost(m, j, GetPatchingCost(m, j))$;

```
Patch two cycles by removing arcs (i, Next[i]), (j, Next[j]), and
adding (i, Next[j])and(j, Next[i]); update Next to reflect the patching operation;

(* re-calculate BestNode and BestCost if necessary *)
for i := 1 to N do
   if BestNode[i] = −1 then
      (* Needs re-calculating *)
      bn := 1;

      for j := 1 to i do
         if Cost[i, j] < Cost[i, bn] then bn := j;
      end for;

      BestNode[i] := bn;
      BestCost[i] := Cost[i, bn];
   end if;
end for;

end repeat;

END.
```

5 Recursive Path Contraction algorithm

The second heuristic originates from [16]. The main feature of this algorithm is the fact that its solution has a large domination number. Heuristics yielding tours with exponential yet much smaller domination numbers were introduced in the literature on so-called exponential neighbourhoods for the TSP (for a comprehensive survey of the topic, see [5]). Exponential neighbourhood local search [6, 7, 8] has already shown its high computational potential for the TSP (see, e.g., [1, 3]).

The algorithm (denoted by *RPC*) proceeds as follows:

1. Find a minimum weight cycle factor F.
2. Delete a heaviest arc of each cycle of F and contract the obtained paths one by one.
3. If the number of cycles is greater than one, apply this procedure recursively.
4. Finally, we obtain a single cycle C. Replace all vertices of C with the corresponding contracted paths and return the tour obtained as a result of this procedure.

Let c_i be the number of cycles in the i-th cycle factor (the first one is F) and let m be the number of cycle factors derived in RPC. Then one can show that the domination number of the tour constructed by RPC is at least $(n - c_1 - 1)!(c_1 - c_2 - 1)!...(c_{m-1} - c_m - 1)!$ [16].

This number is quite large when the number of cycles in cycle factors is small (which is often the case for pure asymmetric instances of the TSP). By *pure* asymmetric instances we mean instances for which $w(i,j) \neq w(j,i)$ for all or almost all pairs $i \neq j$.

6 Contract-or-Patch heuristic

The third heuristic (denoted by *COP* - contract or patch) is a combination of the last two algorithms. It proceeds as follows:

1. Fix a threshold t.
2. Find a minimum weight cycle factor F.
3. If there is a cycle in F of length at most t, delete a heaviest arc in every short cycle (i.e. of length at most t) and contract the obtained paths (the vertices of the long cycles are not involved in the contraction) and repeat the above procedure. Otherwise, patch all cycles (they are all long) using GKS.

Our computational experiments (see the next section) showed that $t = 5$ yields a quite robust choice of the threshold t. Therefore, this value of t has been used while comparing COP with other heuristics.

7 Computational results

We have implemented all three heuristics along with KSP, the greedy algorithm (*GR*), and the random insertion algorithm (*RI*), and tested them on the following seven families of instances of the TSP:

1. all asymmetric TSP instances from TSPLib;
2. all Euclidean TSP instances from TSPLib with the number of vertices not exceeding 3000;
3. asymmetric TSP instances with weights matrix $W = [w(i,j)]$, with $w(i,j)$ independently and uniformly chosen random numbers from $\{0, 1, 2, ..., 10^5\}$;
4. asymmetric TSP instances with weights matrix $W = [w(i,j)]$, with $w(i,j)$ independently and uniformly chosen random numbers from $\{0, 1, 2, ..., i \times j\}$;
5. symmetric TSP instances with weights matrix $W = [w(i,j)]$, with $w(i,j)$ independently and uniformly chosen random numbers from $\{0, 1, 2, ..., 10^5\}$ $(i < j)$;
6. symmetric TSP instances with weights matrix $W = [w(i,j)]$, with $w(i,j)$ independently and uniformly chosen random numbers from $\{0, 1, 2, ..., i \times j\}$ $(i < j)$;

7. sloped plane instances ([11]). These are defined as follows: for a given pair of vertices p_i and p_j, defined by their planar coordinates $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$, the weight of the corresponding arc is $w(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} - max(0, y_i - y_j) + 2 \times max(0, y_j - y_i)$. We have tested the algorithms on sloped plane instances with independently and uniformly chosen random coordinates from $\{0, 1, 2, ..., 10^5\}$.

For the families 3-7, the number of vertices n was varied from 100 to 3000, in increments of 100. For $100 \leq n \leq 1000$, all results are average over 10 trials each, and for $1000 < n \leq 3000$, the results are average over 3 trials each.

The number and variety of the families used allows us to check robustness of tested algorithms [11]. We use instances produced in both random and deterministic manner. It is perhaps worth recalling that this paper deals only with construction heuristics for the TSP. Thus, we cannot expect near optimal solutions for the majority of instances.

All tests were executed on a Pentium II 333 MHz machine with 128MB of RAM. All results for TSPLib instances are compared to optima. For the asymmetric instance families 3,4 and 7 we used AP lower bound, the weight of the lightest cycle factor. The AP lower bound is known to be of high quality for the pure asymmetric TSP [2]. For the symmetric instance families 5 and 6, we exploited the Held-Karp (HK) lower bound [9, 10], which is known to be very effective for this type of TSP instances [12].

Table 1 provides an overview of the quality of the results obtained. Observe that COP is the only heuristic from the above six that performs well (relatively to the other heuristics) on all tested families. All other heuristics fail on at least one of the families. Compared to the other algorithms, GR is good only for the Euclidean instances. RI, KSP and GKS provide good results for all families apart from the two random symmetric families, where all produce tours of rather low quality. RPC is quite stable, but almost always gives solutions of relatively low quality.

It is worth mentioning that while there are some good construction heuristics for the pure asymmetric TSP, the best example of which is KSP [14, 17], the symmetric non-Euclidean TSP appears much more difficult for the existing construction heuristics. This difficulty is reflected in our computational results. While COP certainly narrows the gap between approximate solutions and the HK lower bound, this gap still remains wide.

All of the tested algorithms except GR and RI start by solving the assignment problem in order to find a minimum weight cycle factor, and then apply various patching techniques to transform the cycle factor into a tour. In our view, the difficulty with the symmetric non-Euclidean TSP for these algorithms stems from the fact that the AP lower bound is far from being sharp for this type of instances. For the vast majority of symmetric instances under consideration, the AP lower bound produces a cycle factor consisting of a large number of short (in the number of vertices) cycles. This makes patching rather ineffective. Two remaining algorithms, GR and RI, also fail on these instances, but perhaps for a different reason. This suggests a need for further study of construction heuristics for the symmetric non-Euclidean TSP.

Tables 2 and 3 show the results obtained for the families 1 and 2. For the family 2, only instances of size between 500 and 3000 vertices are presented due the large total number of

Euclidean instances in TSPLib (note that the values presented in Table 1 reflect all Euclidean TSPLib instances of not more than 3000 vertices).

Figures 1-10 show our results in a more detailed form for all families except for the families 1 and 2. In Figures 1 and 3, we present only the results for KSP, GKS and COP as they are clear winners for the families 3 and 4. For both families the tours produced by COP are almost always better than those by all other tested heuristics. Note that the results of KSP and GKS are very similar for these two families of instances. This suggests that for certain classes of instances the use of GKS instead of KSP is not justified. Notice that GKS requires significantly more involved programming than KSP.

Figures 5 and 7 contain our results on the families 5 and 6 for all heuristics. For the families 5 and 6 COP is again a clear winner. Taking into consideration both families, RPC produces results, which are relatively better or not much worse than those of the others (apart from the tours of COP). Interestingly enough GR behaves worse on the families 4 and 6 than on the families 3 and 5. This justifies partially our use of Families 4 and 6 together with the families 3 and 5.

The quality of results for the family 7 is shown in Figure 9. Being a very good heuristic for the Euclidean TSP, RI is the winner among our heuristics on the family 7, which consists of asymmetric instances somewhat similar to Euclidean instances. While GR is hopeless (and therefore not presented on the chart), and RPC behaves not particularly well on the family 7, KSP, GKS and COP provide tours whose quality is close to that produced by RI, especially when the size of the instance increases.

Apart from being quite effective, COP is also comparable to the greedy algorithm with respect to the execution time, see Figures 2,4,6,8 and 10 for details. RI is clearly the fastest heuristic, but its use should be restricted to Euclidean and close to Euclidean instances as we have seen above.

8 Conclusions

The results of our computational experiments show clearly that our combined algorithm COP can be used for wide variety of the TSP instances as a fast heuristic of relatively good quality. It also demonstrates that theoretical investigation of algorithms that produce solutions of exponential domination number can be used in practice to design effective and efficient construction heuristics for the TSP. Further study of construction heuristics for the symmetric non-Euclidean TSP is suggested.

Acknowledgments We would like to thank the anonymous reviewer and referees for providing useful comments and suggestions on the initial version of the paper. The research of GG was partially supported by a grant from the Nuffield Foundation. The research of AY was partially supported by a grant from DNSRC (Denmark).

References

[1] E. Balas and N. Simonetti, Linear time dynamic programming algorithms for some new classes of restricted TSP's. *Proc. IPCO V*, LNCS 1084, Springer Verlag, 1996, 316-329.

[2] E. Balas and P. Thoth, Branch and bound methods, *The Traveling Salesman Problem*, E.L. Lawler, et al. (eds.), Wiley, 1985, 361-401.

[3] J. Carlier and P. Villon, A new heuristic for the traveling salesman problem. *RAIRO* 24, 245-253 (1990).

[4] W.J. Cook, W.H. Cunninghan, W.R. Pulleyblank and A. Schrijver, Combinatorial Optimization, Wiley, New York, 1998.

[5] V. Deineko and G.J. Woeginger, A study of exponential neighbourhoods for the travelling salesman problem and for the quadratic assignment problem. TR Woe-05, TU of Graz, Graz, Austria, 1997.

[6] F. Glover and A.P. Punnen, The travelling salesman problem: new solvable cases and linkages with the development of approximation algorithms, J. Oper. Res. Soc., 48 (1997) 502–510.

[7] G. Gutin, Exponential neighbourhood local search for the traveling salesman problem. *Computers & Operations Research* 26 (1999) 313-320.

[8] G. Gutin and A. Yeo, Small diameter neighbourhood graphs for the traveling salesman problem: at most four moves from tour to tour. *Computers & Operations Research* 26 (1999) 321-327.

[9] M. Held and R.M. Karp, The traveling-salesman problem and minimum spanning trees. *Oper. Res.* 18 (1970) 1138-1162.

[10] M. Held and R.M. Karp, The traveling-salesman problem and minimum spanning trees: part II. *Math. Prog.* 1 (1971) 6-25.

[11] D.S. Johnson, private communication, 1998.

[12] D.S. Johnson and L.A. McGeoch, The traveling salesman problem: a case study in local optimization. *Local Search in Combinatorial Optimization,* E.H.L. Aarts and J.K. Lenstra (eds.), Wiley, N.Y., 215-310 (1997).

[13] R.M. Karp, A patching algorithm for the nonsymmetric Traveling Salesman Problem. *SIAM J. Comput.* 8 (1979) 561-73.

[14] R.M. Karp and J.M. Steele, Probabilistic analysis of heuristics, in *The Traveling Salesman Problem,* E.L.Lawler, *et al.* (eds.), Wiley, N.Y., 1985, pp. 181-205.

[15] G. Reinelt, *The traveling salesman problem: Computational Solutions for TSP Applications. Springer Lecture Notes in Computer Sci.* 840, Springer-Verlag, Berlin (1994).

[16] A. Yeo, Large exponential neighbourhoods for the TSP, preprint, Dept of Maths and CS, Odense University, Odense, Denmark, 1997.

[17] W. Zhang and R.E. Korf, On the asymmetric traveling salesman problem under subtour elimination and local search, manuscript, Dept. of CS, University of California, LA, 1993.