Iterative Patching and the Asymmetric Traveling Salesman Problem

Turkensteen, Marcel; Ghosh, Diptesh; Goldengorin, Boris; Sierksma, Gerard

Iterative Patching and the Asymmetric Traveling Salesman Problem

Marcel Turkensteen[1*] Diptesh Ghosh[2] Boris Goldengorin[1]
Gerard Sierksma[1]
1*: Faculty of Economic Sciences, University of Groningen, The Netherlands*
2*: P&QM Area, Indian Institute of Management, Ahmedabad, India*

SOM theme A **The human and technical side of production: the management of interdependencies**

Abstract

Although Branch and Bound (BnB) methods are among the most widely used techniques for solving hard problems, it is still a challenge to make these methods smarter. In this paper, we investigate *iterative patching*, a technique in which a fixed patching procedure is applied at each node of the BnB search tree for the Asymmetric Traveling Salesman Problem. Computational experiments show that iterative patching results in general in search trees that are smaller than the usual classical BnB trees, and that solution times are lower for usual random and sparse instances. Furthermore, it turns out that, on average, iterative patching with the Contract-or-Patch procedure of Glover, Gutin, Yeo and Zverovich (2001) and the Karp-Steele procedure are the fastest, and that 'iterative' Modified Karp-Steele patching generates the smallest search trees.

[*]Corresponding author: Faculty of Economic Sciences, University of Groningen, P.O. Box 800, 9700 AV Groningen,The Netherlands. FAX: +31-50-363-3720, Email: m.turkensteen@eco.rug.nl

(also downloadable in electronic version: http://som.rug.nl/)

Keywords: Asymmetric Traveling Salesman Problem, Branch-and-Bound, upper bounding, patching

1 Introduction

The Asymmetric Traveling Salesman (ATSP) is usually solved exactly by means of Branch-and-Bound (BnB) algorithms and Branch-and-Cut (BnC) algorithms, see Fischetti *et al.* [9]. In BnB type algorithms, an Assignment Problem (AP) is solved at every node of this tree, and the value of the optimal AP solution serves as a lower bound of the ATSP solution. A part of the search tree can be discarded when its lower bound exceeds an upper bound. This upper bound is usually the value of a shortest complete tour found so far. A class of heuristics applied to construct such a tour is *patching*. The question is: at which nodes of the search tree should such a tour be constructed? Patching at a node may reduce the search tree and the solution time, but if the reduction is too small, the overall solution time is increased due to the time invested in patching.

In the literature, the most effective BnB methods do not patch at each node; see for example, Miller and Pekny [13], and Carpaneto *et al.* [1]. These methods use a best first search strategy, i.e., the subproblem with the smallest lower bound is solved. According to these studies, patching at every node is too time-consuming.

In this paper, we consider a BnB algorithm that applies depth first search, which means that the most recently generated subproblem is solved first. This strategy requires algorithms to use much less computer memory than do best first strategies. Hence, it is useful for solving large problems. We apply *iterative patching*, in which a fixed patching procedure is applied at every node of the BnB depth first search tree. Four iterative patching procedures are considered in our computational experiments. These procedures are described in Glover *et al.* [6].

Given a set of locations and the distance between any pair of locations, the ATSP is the problem of finding a shortest Hamiltonian tour; i.e., a shortest round trip visiting each location exactly once. Figure 1 is an example of an underlying graph that defines an instance of an ATSP. The nodes of the graph represent locations, and the arcs the connections between the locations. A number next to an arrowhead denotes the cost of traveling along that arc.

General instances of the ATSP are often solved to optimality by means of enumer-

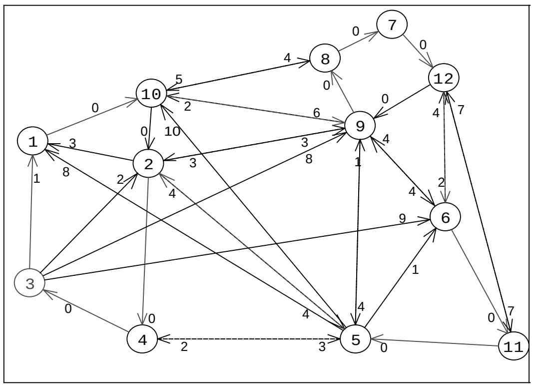

Figure 1: ATSP instance

ation algorithms, in which a fraction of all feasible solutions is checked. BnB methods explore the solution space by using a search tree. We discuss BnB algorithms that solve an Assignment Problem (AP) at each node of the corresponding search tree. After solving the AP a minimum cycle cover F is obtained, say, consisting of k cycles ($k \geq 1$). In the example of Figure 2, three cycles are generated. If $k > 1$, the subcycles in F can be *patched* into a complete tour. BnB algorithms use the value of a patching solution as an upper bound by which nodes of the search tree are fathomed.

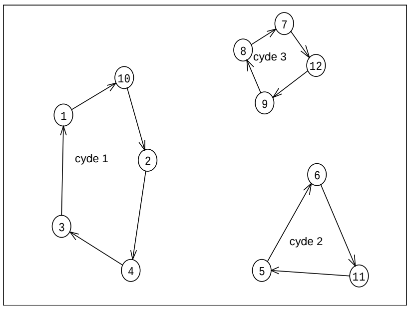

Figure 2: Minimum cycle cover

A *patching operation* is the simultaneous deletion of two arcs from a cycle cover and the insertion of two other arcs, such that the number of cycles is reduced by one. In our example, two patching operations are needed for the generation of a complete tour (see Figure 3), namely first arcs (2,4), (5,6) are deleted and (2,6) and (5,4) are inserted, and then we delete (12,9) and (2,6) and insert (2,9) and (12,6). The resulting tour is generally feasible but not optimal.

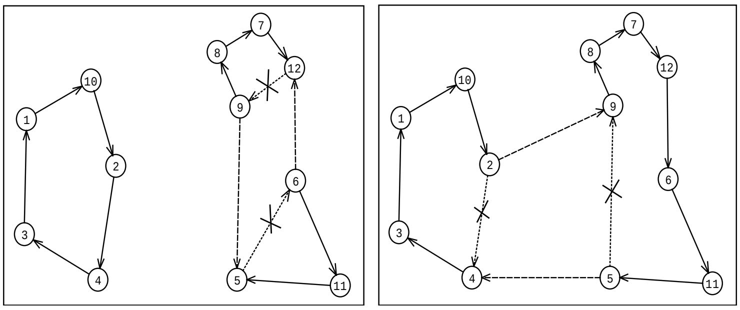

Figure 3: Obtaining a tour by means of two patching operations

In Karp [11], patching is defined as a sequence of $k-1$ patching operations on a cycle cover of k cycles, $k \geq 1$. Recall that even a best possible patching procedure consisting of $k-1$ patching operations does not always yield a shortest complete tour. For example, consider the sparse network in Figure 4. The minimum cycle cover consists of the $k = 2$ cycles $(1,2,3,4,5,1)$ and $(6,7,8,9,6)$ with total length 29. The unique shortest complete tour is $(1,2,8,9,6,7,4,3,5,1)$ with length 31. Since four arcs need to be inserted and deleted, this tour cannot be constructed from the cycle cover by means of one patching operation. Different patching procedures are introduced in the literature; see [6, 11, 12, 15]. These patching procedures are discussed in Section 3.

Most heuristics for the ATSP apply patching procedures only once, such as to obtain approximations to optimal solutions; see e.g. [6–8, 17]. BnB algorithms apply patching procedures in order to obtain good feasible solutions with which parts of the search tree can be discarded. Any heuristic may be used to generate such solutions, but patching procedures are the most natural choices, since they use the structure of the already constructed minimum cycle cover. If a fixed patching procedure is applied at every

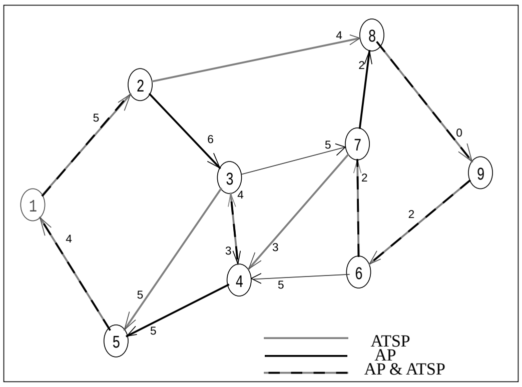

Figure 4: Best patching solution is not a shortest tour

node in a BnB algorithm, we call it *iterative patching*.

The currently best BnB algorithms for the ATSP are introduced in Carpaneto *et al.* [1] and in Miller and Pekny [13]. We call these the CDT algorithm and the MP algorithm, respectively. The CDT algorithm uses the patching procedure from Karp & Steele [12] at the top node of the search tree. Only if the number of zeroes in the reduced matrix at the top node exceeds a threshold value β, then a subtour-merging procedure is carried out at each node of the search tree.

The subtour-merging procedure constructs first an admissible graph of zero-cost elements in the reduced matrix and then tries to find a complete tour in the admissible graph. The subtour-merging procedure patches cycles together, but only when a zero-cost patching operation is available. It usually does not return a complete tour. In Carpaneto *et al.* [1], it is found that a threshold that if β is set to $2.5n$, the solution times are the shortest, where n is the dimension of the instance.

The MP algorithm applies the Karp-Steele patching procedure, but not at every node of the search tree. Nodes close to the top node are patched more often than nodes deep in the tree. This algorithm also applies a subtour-merging procedure at each node.

The CDT and the MP algorithm both use a best first search (BFS) strategy, which means that a node with the smallest lower bound value is expanded next. BFS is the fastest search strategy, but requires exponential memory space. As a consequence, BFS algorithms are generally restricted to small or easily solvable problems [16]. In depth first search (DFS), the most recently generated subproblem is solved first, and it requires polynomial memory space. This makes it suitable for solving large and difficult instances. However, the search trees and solution times of DFS algorithms are usually large.

Miller and Pekny [13] report that iterative patching is too time-consuming. This may be true for BFS algorithms, but our algorithms use DFS. DFS algorithms search through deep nodes of the search tree already at an early stage; lower bounds of such nodes are generally high. A tight upper bound obtained early enables the algorithm to discard a large fraction of these nodes. Therefore, a DFS algorithm is more likely to benefit from a good upper bounding procedure, such as iterative patching, than a BFS algorithm.

The computational experiments in Section 4 compare the search tree sizes and the running times of BnB algorithms that apply iterative patching with a DFS implementation of the CDT algorithm. We apply four patching procedures, namely the ones discussed in Glover *et al.* [6]. The main questions that we answer on iterative patching in this paper are as follows. Is iterative patching effective for DFS algorithms? Is it true that if a patching procedure returns on average shorter tours than some other one, then, again on average, the search tree sizes are smaller and the running times are shorter? Hence, does better patching lead to the smaller search trees and shorter running times?

2 The quality of patching procedures

Let $G(V,A)$ be a graph with vertex set V and arc set A. A minimum cycle cover $F \subset A$ can be determined in $O(n^3)$ time by means of the Hungarian algorithm; see for example [10]. The speed of the Hungarian algorithm can be increased in successor nodes j to $O(n^2)$ by starting from the optimal solution in the parent node, i.e., the node in which subproblem j is generated; see for example Fischetti *et al.* [9].

Patching procedures delete pairs of arcs from F and insert pairs of arcs from $A \setminus F$

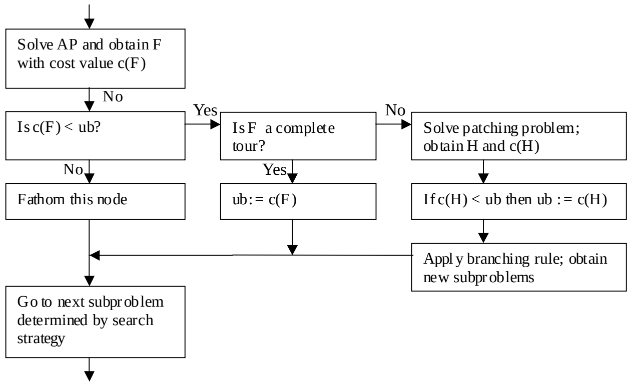

Figure 5: Flowchart of a BnB algorithm with iterative patching

in such a way that a Hamiltonian cycle $H \subset A$ is obtained. The patching cost of any patching procedure P is then denoted by $c_P(F)$ and defined as

$$c_P(F) = \sum_{a \in H \setminus F} c(a) - \sum_{b \in F \setminus H} c(b), \tag{1}$$

where $c(a)$ denotes the cost of arc $a \in A$. The first term of (1) indicates the cost of the new arcs introduced by P, and the second term represents the cost of the arcs removed from the cycle cover. For any subset $Q \subset A$, $c(Q)$ denotes the sum of the cost of the arcs in Q.

Section 3 presents four heuristics for *Karp's Patching Problem (KPP)*, which is the problem of finding an optimal patching from a given cycle cover. The fact that this problem is $\mathcal{NP}$-hard (see [3]) can be seen as follows. Consider a minimum cycle cover consisting of n cycles of length 1. Then any arbitrary Hamiltonian tour can be constructed by means of $n-1$ patching operations. So the ATSP reduces to KPP, and hence, the KPP is $\mathcal{NP}$-hard. So we have to rely on heuristics to solve KPPs.

Let $F_j \subset A$ denote a minimum cycle cover at node j of the BnB search tree in progress. By $BnB(Br, S, UBS)$ we denote a BnB algorithm for the ATSP that applies branching rule *Br*, search strategy *S*, and upper bounding strategy *UBS*. A branching rule *Br* partitions the current feasible regions into subsets. We consider branching rules that only depend on the current minimum cycles cover. The search strategy S in this paper is DFS. The upper bounding strategy UBS consists of two components: the first component prescribes at which nodes an upper bounding procedure should be applied, and the second component specifies the upper bounding procedure to be used. Clearly, iterative patching is an upper bounding strategy, where a tour is generated at every node of the search tree by means of a fixed patching procedure. If no confusion is likely, we simply write $BnB(UBS)$, since S and Br are fixed in this study.

Note that, in case of DFS, the order of node expansion is independent of the bounds used at each subproblem. For instance, if both algorithms $BnB(P_1)$ and $BnB(P_2)$ explore two subproblems S_1 and S_2, and $BnB(P_1)$ explores S_1 before S_2, then $BnB(P_2)$ will explore S_1 before S_2 as well.

Let $ub_j(UBS)$ be the current *upper bound*, i.e. the shortest complete tour obtained until node j using upper bounding strategy UBS. Recall that, when the UBS is iterative patching, we obtain at each node of the search tree a complete tour, i.e. a candidate for the value of $ub_j(UBS)$.

Node k is called a *successor* of j in a search tree if j is an intermediate node of the shortest path between k and the top node of the search tree; we use the notation $k \propto j$. Since the feasible region of the AP at node k is a subset of the feasible region of the AP at node j, we have of course that $c(F_k) \geq c(F_j)$ if $k \propto j$; see e.g. [16].

In case of iterative patching, one may expect that if patching costs are low, then upper bounds are tighter and a larger number of subproblems can be fathomed. Theorem 1 formalizes this assertion: if for each instance patching procedure P_1 is cheaper than patching procedure P_2, then the search tree of $BnB(P_1)$ will be smaller than the search tree of $BnB(P_2)$.

For any iterative patching procedure P, let $BnB(P)$ be the algorithm that uses P iteratively. Define $\#BnB(P)$ as the size of the solution tree of $BnB(P)$, i.e. the number of nodes in this tree. We assume in Theorem 1 that $BnB(P_1)$ and $BnB(P_2)$ use the same AP-solver implementation.

Theorem 1 *Let $\mathcal{F}$ be the set of minimum cycle covers of a given instance of the ATSP,* *and let P_1 and P_2 be two patching procedures such that their respective patching costs satisfy $c_1(F) \leq c_2(F)$ for each $F \in \mathcal{F}$. It then follows that $\#BnB(P_1) \leq \#BnB(P_2)$.*

Proof. For any given instance of the ATSP, let $T(Br)$ be the complete search tree based only on branching rule Br, i.e. the search tree in which all possible solutions are enumerated. Usual BnB procedures apply the following pruning operations:

1. If at a certain node of $T(Br)$ F is a complete tour, then all successor nodes are deleted from $T(Br)$.

2. If at a certain node of $T(Br)$, say j, it holds that $c(F_j) \geq ub_j(P)$, then this node and all its successors are fathomed.

For any patching procedure P, $BnB(P)$ deletes nodes from the complete search tree $T(Br)$ until the usual BnB tree remains, which we denote by $T(P)$. Clearly, pruning operation (1) is independent of the patching procedure used, since the AP solver implementation is taken fixed. Actually, at each node the same minimum cycle cover is found.

We now show that $T(P_1) \subseteq T(P_2)$ by showing that if node j is fathomed under P_2, then it is also fathomed under P_1. This is the case, if for each node j, it holds that $c(F_j) \geq ub_j(P_2) \Longrightarrow c(F_j) \geq ub_j(P_1)$. So we need to show that $ub_j(P_1) \leq ub_j(P_2)$ for each node j on the path obtained by the search strategy S. Thus, $BnB(P_2)$ is only able to discard nodes if $BnB(P_1)$ discards them, which implies that $\#BnB(P_1) \leq \#BnB(P_2)$.

Obviously, for the first node $j = 0$, it holds that $ub_0(P_1) \leq ub_0(P_2)$. Now assume that $ub_j(P_1) \leq ub_j(P_2)$ at node j. Let k be the next unsolved subproblem after node j according to the search strategy S. We show that $ub_k(P_1) \leq ub_k(P_2)$. Let $H_P(F)$ be the patching solution of procedure P given minimum cycle cover F.

After solving the AP at node j, both algorithms compare $c(F_j)$ with their current upper bounds. Three scenarios are possible:

1. If $ub_j(P_1) \leq ub_j(P_2) \leq c(F_j)$, then both algorithms fathom node j and both procedures proceed to node k. Clearly, $ub_k(P_1) = ub_j(P_1) \leq ub_j(P_2) = ub_k(P_2)$.

2. If $c(F_j) < ub_j(P_1) \leq ub_j(P_2)$, then both algorithms execute patching at node j. Since $c_1(F_j) \leq c_2(F_j)$, it follows that $c(H_1(F_j)) = c(F_j) + c_1(F_j) \leq c(F_j) + c_2(F_j) = c(H_2(F_j))$. Since $ub_k(P_i) = \min\{ub_j(P_i), c(H_i(F_j))\}$ for $i = 1,2$, we have that $ub_k(P_1) \leq ub_k(P_2)$.

3. If $ub_j(P_1) \leq c(F_j) < ub_j(P_2)$, then $BnB(P_1)$ fathoms node j, and $ub_k(P_1) := ub_j(P_1)$. $BnB(P_2)$ solves an additional patching problem at node j and possibly at the successor nodes of j. Let q be the successor node of j in which the best patching solution is obtained, i.e. $q = \arg\min_l\{c(H_2(F_l)); l \propto j, l = j\}$. After searching through all successors of j, or after discarding them, $BnB(P_2)$ arrives at node k with $ub_k(P_2) \geq \min\{ub_j(P_2), c(H_2(F_q))\}$. Clearly, $ub_k(P_1) = ub_j(P_1)$. Furthermore, it holds that $ub_j(P_2) \geq ub_j(P_1) = ub_k(P_1)$, and that $c(H_2(F_q)) \geq c(F_q) \geq c(F_j) \geq ub_j(P_1) = ub_k(P_1)$. Hence, $ub_k(P_2) \geq ub_k(P_1)$.

Hence, for all nodes j on the path according to S through $T(Br)$, we have that $ub_j(P_1) \leq ub_j(P_2)$. Therefore, $\#BnB(P_1) \leq \#BnB(P_2)$. ■

Theorem 1 can be extended to upper bounding strategies UBS for which the upper bound generated at node j is at least $c(F_j)$. In that case, upper bounds are only obtained at nodes at which a complete tour is constructed; elsewhere, the patching costs are infinite. For example, consider a BnB algorithm $BnB(P;ni)$ that applies patching procedure P not iteratively. It follows from Theorem 1 that its search tree is always at least the size of the search tree of the algorithm $BnB(P)$ that applies P iteratively.

In general, there are few iterative patching procedures that always return better patching solutions than some other one. Therefore, it makes more sense to consider the average performance of patching procedures. To this end, we conduct computational experiments in Section 4.

The most important measure of the quality of algorithms are solution times. Actually, high quality patching solutions may lead to long solution times of subproblems.

So usually, a trade-off is made between the quality of the patching and time invested in patching. For instance, if patching procedure P is only applied at the top node, the search tree is larger than the tree with iterative patching procedure P. However, the average solution time at the nodes is smaller. In Section 4, solution times are taken into account more explicitly.

The following observation allows to increase the speed of iterative patching without losing quality. Recall, that if a cycle cover F consists of k cycles, patching is a sequence of $k-1$ patching operations. Call the cycle cover after the i-th patching operation F_i, and denote its cost by $c(F_i)$, $i = 1,\ldots,k-1$. If $c(F_i)$ exceeds the cost of the current best solution *ub*, the patching procedure will certainly not lead to a better solution, since the cost of each patching operation is nonnegative. Hence, we can abort the patching after i steps and save running time.

3 Patching Procedures

We now compare the performance of four iterative patching procedures based on the four most well-known patching algorithms. We start with a short description of these four patching procedures. All these procedures have a worst-case time complexity of $O(n^3)$, see Glover *et al.* [6].

Karp-Steele patching (KSP) was introduced in Karp and Steele [12]. Starting with the minimum cycle cover F, KSP patches the two longest subcycles successively by using a cheapest patching operation. In our example, KSP patches cycles 1 and 3 by deleting (10,2) and (9,8), and adding (10,8) and (9,2); see Figure 6. The new cycle is then patched with cycle 2 by removing (12,9) and (5,6), and inserting (5,9) and (12,6).

Modified Karp-Steele patching (MKS), also called *Greedy Karp-Steele patching*, see Glover *et al.* [6], performs the cheapest patching operation among all pairs of cycles in the current cycle cover. The patching costs are then updated and the procedure is repeated until a complete tour is obtained. Since it compares in general more patching operations than KSP, MKS is more time-consuming. In our example, MKS joins cycles 2 and 3 by deleting arcs (5,6) and (12,9), and inserting (5,9) and (12,6). Cycle 1 is

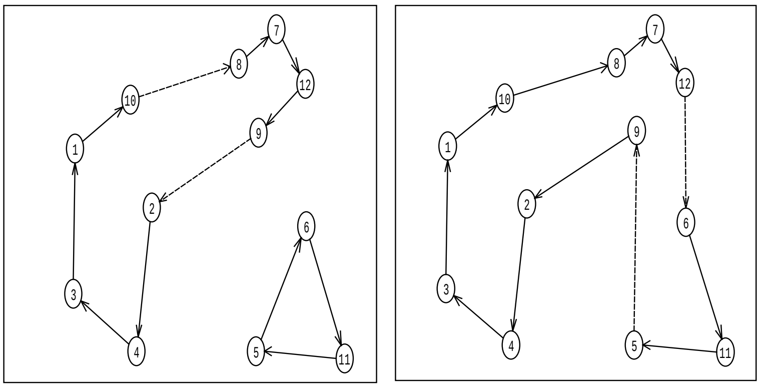

Figure 6: Karp-Steele patching in action

included by inserting (2,9) and (5,4) and removing (2,4) and (5,9); see Figure 3.

Recursive Path Contraction (RPC) was introduced in Yeo [15]. From all, say k, cycles a most expensive arc is deleted and the remaining paths are contracted, so transformed into single nodes. On these k nodes an AP is solved. So every contracted path is connected to another contracted path. The procedure is carried out recursively until one cycle is obtained. The calculations of Section 4 use the implementation from Glover *et al.* [6]. In our example, the most expensive arc from every cycle is deleted, namely (3,1), (5,6) and (12,9). The end nodes 3, 5 and, 12 are assigned to nodes 9, 1, and 6, respectively. Finally, the tour depicted in Figure 7 is obtained.

Contract-or-Patch (COP) is a two-stage procedure consisting of RPC in the first stage and, either MKS or KSP in the second stage; see [6] and [7]. All cycles with length less than a user-defined threshold value t are patched using RPC. In Gutin *et al.* [7], it is shown that the threshold value $t = 5$ is the most robust choice for different types of instances. Given the cycle cover from Figure 2, cycles 2 and 3 are patched using the RPC procedure. The long cycles in the current cycle cover are patched with either KSP or MKS. In Section 4, the faster procedure KSP is selected, since in Johnson *et al.* [8] it is asserted that there is no significant difference in the patching cost of COP using either KSP or MKS.

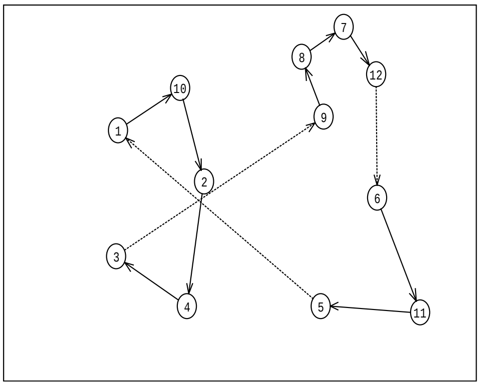

Figure 7: RPC patching solution

4 Computational experiments

In this section, we compare both the tree sizes and the running times of the algorithms presented in Table 1. Recall that the size of a BnB tree is the number of subproblems solved before the first optimal solution is determined, i.e. the number of nodes visited on the path followed through $T(Br)$ according to search strategy S. The results of iterative patching procedures are compared with the results of the DFS implementation of the CDT algorithm. The DFS implementation is of practical use, because it solves ATSPLIB and symmetric instances which a BFS approach cannot solve; see for example Carpaneto *et al.* [1] and Miller and Pekny [13].

Table 1: Patching strategies tested

Name	Patching strategy
$BnB(KSP)$	Iterative KSP
$BnB(MKS)$	Iterative MKS
$BnB(RPC)$	Iterative RPC
$BnB(COP)$	Iterative COP
$BnB(CDT)$	CDT algorithm

The experiments are performed on a Pentium 4 computer with speed 2 GHZ and 256 MB RAM under Windows 2000. The programming language is C and the compiler is GNU with speed -o2. Our branching rule branches by a largest cost arc in the shortest subcycle of a minimum cycle cover. In a forthcoming study we will apply tolerance-based branching rules, where branching is performed on an arc with the smallest tolerance value (the amount at which the cost can be changed without changing the solution at hand). The iterative patching procedures are tested for the following types of instances:

1. Asymmetric TSPLIB instances (see [14]);
2. Randomly generated instances with varying degree of symmetry;
3. Randomly generated instances with varying degree of sparsity.

From all asymmetric TSPLIB instances we have selected 16 instances that are solvable within reasonable time limits. The random instances have degree of symmetry 0, 0.33, 0.66, and 1, where the *degree of symmetry* is defined as the fraction of off-diagonal entries in the cost matrix $\{c_{ij}\}$ that satisfy $c_{ij} = c_{ji}$. The third class of instances consists of instances with varying *degree of sparsity*, being defined as the fraction of the total possible number of arcs that are missing. We study instances with degree of sparsity of 0, 0.25, 0.5, and 0.75. The usual random instances have problem size 60, 70, 80, 100, 200, 300, 400, and 500, except for the random instances with degree of symmetry larger than 0; they have problem size 60, 70, and 80. Only these samples of (quasi-)symmetric instances are considered, since computation times for larger symmetric instances tend to be extremely long. The instances with varying degree of sparsity have problem size 100, 200, and 400. The arc costs are drawn from a discrete uniform distribution supported on $\{1, 2, ..., 10^4\}$; for each problem set and for all problem sizes, 10 instances are generated. In comparison with other studies, namely, [1], and [13], our random instances are relatively small, whereas our symmetric instances are relatively large. For example, the MP algorithm by Miller and Pekny [13] solves random instances of size 500000, but solves symmetric instances of size less than 30 only.

The average *size of the search tree* of the algorithms is shown in Table 2. In order to make the results more comparable, we have used *normalized* results, i.e., we have fixed

Table 2: Normalized size of search tree for usual BnB (CDT = 100)

	CDT	KSP	MKS	RPC	COP
ATSPLIB	100.00	95.03	94.27	101.40	95.37
Usual random	100.00	47.27	43.97	129.98	47.27
Degree of symmetry 0.33	100.00	50.81	50.65	106.75	51.16
Degree of symmetry 0.66	100.00	74.52	73.66	101.45	75.44
Full symmetry	100.00	99.79	99.77	99.97	99.80
Degree of sparsity 0.25	100.00	51.66	51.20	113.26	51.66
Degree of sparsity 0.50	100.00	56.13	56.13	126.68	56.13
Degree of sparsity 0.75	100.00	56.43	56.35	129.98	56.43

the results of $BnB(CDT)$ at 100. The number '50.65' in the MKS-column means that the $BnB(MKS)$ generates on average about half the number of subproblems of $BnB(CDT)$ for instances with degree of symmetry 0.33.

Table 2 shows that, except for the RPC procedure, iterative patching leads to smaller search trees. The search tree reductions of iterative patching are large for usual random and sparse instances; the sizes of the trees of $BnB(KSP)$, $BnB(MKS)$ and $BnB(COP)$ are half the size of the search tree of $BnB(CDT)$. The reductions of iterative patching are smaller for symmetric and ATSPLIB instances. On average, the search trees generated by $BnB(MKS)$ are the smallest, whereas $BnB(RPC)$ only generates reasonably small search trees for symmetric instances.

Table 3: Normalized running times

	CDT	KSP	MKS	RPC	COP
ATSPLIB	100.00	114.81	139.56	114.44	116.01
Usual random	100.00	55.81	60.24	140.44	54.45
Degree of symmetry 0.33	100.00	72.22	72.22	170.83	55.56
Degree of symmetry 0.66	100.00	93.33	103.70	132.22	85.00
Full symmetry	100.00	108.24	126.76	114.98	111.54
Degree of sparsity 0.25	100.00	62.64	73.57	125.13	62.33
Degree of sparsity 0.50	100.00	69.05	90.16	144.79	77.44
Degree of sparsity 0.75	100.00	73.79	85.33	153.29	73.88

In Table 3, we present the normalized *running times*. For usual random and sparse

Table 4: Search tree sizes and solution times (seconds) of ATSPLIB instances

	CDT		KSP		MKS		RPC		COP	
Instance	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time
ft53	21189	2.20	20111	2.31	20111	2.64	21189	2.36	20111	2.42
ft70	26025	3.57	25831	3.85	25831	4.40	26025	4.01	25831	4.07
ftv33	7455	0.16	7065	0.22	7061	0.27	7307	0.22	7065	0.22
ftv35	7305	0.16	6945	0.16	6939	0.22	8267	0.22	6951	0.22
ftv38	7325	0.22	6195	0.22	6195	0.27	10101	0.38	6195	0.16
ftv44	3753	0.11	619	0.01	619	0.05	3753	0.16	3083	0.16
ftv47	29539	1.10	29025	1.26	29017	1.76	29539	1.32	29031	1.37
ftv55	114403	4.73	92447	4.51	92447	5.82	114785	5.44	103839	5.55
ftv64	252755	11.87	43441	3.19	43441	4.18	252755	15.93	43441	3.52
ftv70	326827	23.41	253873	24.95	206195	27.36	410545	35.60	261199	24.73
ftv170	1796439	1073.63	1796149	1300.88	1796159	1614.56	1796459	1198.96	1796149	1276.87
rbg323	3	0.05	3	0.05	1	0.05	9	0.05	3	0.01
rbg358	3	0.05	3	0.05	1	0.16	7	0.11	5	0.05
rbg403	3	0.05	3	0.05	1	0.11	7	0.11	3	0.05
rbg443	3	0.05	3	0.05	1	0.11	3	0.11	3	0.05
br17	3674829	16.59	3674829	24.23	3674829	32.69	3674829	24.51	3674829	24.40

Table 5: Search tree sizes and solution times (seconds) of usual random instances

	CDT		KSP		MKS		RPC		COP	
n	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time
60	6508	0.60	3808	0.38	3808	0.44	12880	1.10	3808	0.33
70	10828	1.21	4528	0.44	4528	0.71	18522	2.14	4528	0.55
80	21834	2.75	9014	1.26	8622	1.48	27822	4.1	9014	1.26
100	13454	2.42	9002	1.92	6814	1.81	17424	3.73	9002	1.98
200	138522	114.	36390	33.	36390	40.	172054	151.	36390	33.
300	412930	798.	178498	481.	178498	551.	500100	1081.	178498	424.
400	525088	2142.	284994	1410.	284982	1746.	640440	2825.	284994	1349.
500	951188	6428.	434576	3687.	432000	5284.	1456440	10868.	434576	3889.

instances, iterative patching is clearly more effective; the search tree reduction outweighs the time invested in patching at nodes. Although $BnB(MKS)$ often requires the smallest search trees, $BnB(COP)$ and $BnB(KSP)$ mostly display smaller running times. This indicates that the speed of solving patching problems is relevant. Solution times of iterative patching are longer for instances from the ATSPLIB and for symmetric instances than of $BnB(CDT)$, although in both cases the differences are small.

The following tables show the absolute search tree sizes and solution times in more detail. For most ATSPLIB instances, the search tree reductions of iterative patching are minor, and the solution times increase; see Table 4. For the usual random instances, the iterative patching procedures $BnB(KSP)$, $BnB(MKS)$, and $BnB(COP)$ have clearly smaller search tree sizes and solution times than $BnB(CDT)$; see Table 5. These benefits appear to be independent of the instance size. Finally, Table 6 presents the absolute tree sizes and solution times of sparse and symmetric instances.

Symmetric and ATSPLIB instances can be considered ‘hard’, i.e., even small in-

Table 6: Search tree sizes and solution times (seconds) of symmetric and sparse instances

	CDT		KSP		MKS		RPC		COP	
Instance	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time
Degree of symmetry 0.33	122520	13	58878	8	58724	8	129914	19	59458	7
Degree of symmetry 0.66	259626	33	202894	33	200630	38	264444	45	204470	32
Full symmetry	114984046	17584	114912026	19182	114908592	22521	109843207	19271	114915850	19972
Degree of sparsity 0.25	637872	1935	362188	1451	354610	1801	732500	2386	362188	1434
Degree of sparsity 0.50	653016	1797	368736	1341	368736	1746	801526	2350	368736	1345
Degree of sparsity 0.75	704832	1998	386468	1467	386392	1909	883026	2857	386468	1472

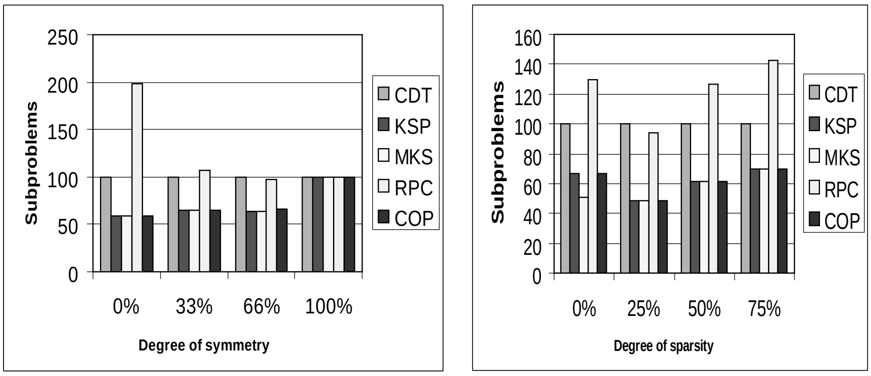

Figure 8: Normalized search tree sizes of instances with varying degree of symmetry (n = 60) and sparsity (n = 100), CDT = 100

stances have large search trees and running times. For these instances, cycle covers often consist of many short cycles. Hence, tours obtained by patching are long, and only minor parts of the search tree can be discarded, so the small reductions of the search tree do not compensate for the time invested in patching at each node. This explains the special behavior of symmetric and ATSPLIB instances.

Table 8 and Figure 8 show that, as the degree of symmetry increases, the search trees of $BnB(CDT)$ and $BnB(RPC)$ converge to the size of the other trees. Hence, applying iterative patching makes no sense for symmetric instances. On the other hand, the degree of sparsity does not influence the relative search tree sizes of the algorithms; see Figure 8. So sparsity does not influence the usefulness of iterative patching.

In Glover *et al.* [6], the performance of patching heuristics on solution quality is studied. The results show that MKS returns the best patching solutions for ATSPLIB

Table 7: Ordering of the top node solution quality and the number of iterations

	Average relative excess over AP lower bound		Normalized search tree size (CDT = 100)	
ATSPLIB	MKS	3.36%	MKS	86.15
	KSP	4.29%	KSP	87.99
	COP	4.77%	COP	88.81
	RPC	18.02%	RPC	103.38
Usual random	COP	1.88%	MKS	43.97
	MKS	3.36%	COP	47.27
	KSP	3.11%	KSP	47.27
	RPC	106.65%	RPC	129.98
Full symmetry	COP	79.87%	MKS	99.77
	RPC	183.57%	KSP	99.79
	MKS	586.92%	COP	99.80
	KSP	744.22%	RPC	99.97

instances, and COP for random instances, both symmetric and asymmetric. In Table 4, the solution quality results from Glover *et al.* [6] are compared with our search tree sizes. The results show that the ordering with respect to solution quality of patching procedures differs from the ordering with respect to search tree sizes of the corresponding iterative patching procedure. This phenomenon may be caused by the following effect. Recall that, when iterative patching is applied, patching solutions are constructed at each node of the search tree. It may be misleading to take into consideration the patching quality only at the top node of the search tree, and expect that for all nodes in the search tree on average the same quality holds. Actually, it is more likely that good upper bounds are found deep in the search tree and that the average patching solution quality deep into the tree differs from the average top node patching quality. In fact, top node cycle covers may consist of many short cycles, whereas subcycles tend to become longer as the BnB algorithm proceeds deeper into the search tree, because our branching rule attempts to break short cycles. This may explain the differences in the orderings according to the average patching quality and to the average search tree size of the iterative patching procedures.

Consider for example the iterative patching procedures *RPC* and *COP*. $BnB(RPC)$ needs long running times and large search trees for random instances, because *RPC* deletes an arc from every cycle without calculating patching costs. Therefore, if cycles are long, bad patching operations are likely. COP, on the other hand, patches long cycles carefully, leading to smaller search trees.

5 Conclusion

We studied the performance of four iterative patching procedures, being fixed patching procedures at every node of the search tree, which we compared with the performance of a depth first search implementation of the CDT algorithm by Carpaneto *et al.* [1]. Our performance measures are the size of the search tree and the running times of the algorithms. Clearly, there is a trade-off between the quality of patching, leading to smaller search trees, and the speed of solving each patching problem. We conclude with an answer to the main questions.

Is it worthwhile to use iterative patching procedures? At least, search trees are always smaller. However, only for 'practical' instances the solution times are shorter when $BnB(CDT)$ is applied. A side effect of iterative patching is that if calculations are finished prematurely, a satisfactory solution is often at hand; see Zhang [16].

Which iterative patching procedure is the most efficient one? On the whole, the algorithm using MKS generates the smallest solution trees, and our COP and KSP implementations achieve the best solution times.

References

[1] G. Carpaneto, M. Dell'Amico, P. Toth, *Exact Solution of Large-Scale, Asymmetric Traveling Salesman Problems,* ACM Transactions on Mathematical Software 21 4, 1995, pp. 394–409.

[2] J. Cirasella, D.S. Johnson, L.A. McGeoch, W. Zhang, *The Asymmetric Traveling Salesman Problem: Algorithms, Instances Generators, and Tests*, In A.L. Buchsbaum, J. Snoeyink, *Algorithm Engineering and Experimentation, Third International Workshop, Alenex, 2001*, Lecture Notes in Computer Science, Vol. 2153, 32–59. Springer Verlag, Berlin, 2001.

[3] V.G. Deineko, personal communication, 2004.

[4] M.E. Dyer, A.M. Frieze, *On Patching Algorithms for Random Asymmetric Travelling Salesman Problems*, Mathematical Programming 46, 1990, 361–378.

[5] A.M. Frieze, G.B. Sorkin, *The Probabilistic Relationship between the Assignment Problem and Asymmetric Traveling Salesman Problem*, 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Washington DC, 2001, 652–660.

[6] F. Glover, G. Gutin, A. Yeo, A. Zverovich, *Construction Heuristics and Domination Analysis for the Asymmetric TSP*, European Journal of Operational Research, 2001, 129:555–568.

[7] G. Gutin, A. Zverovich, *Evaluation of the Contract-or-Patch Heuristic for the Asymmetric Traveling Salesman Problem*, Submitted, 2000.

[8] D.S. Johnson, G. Gutin, L.A. McGeoch, A. Yeo, W. Zhang, A. Zverovich, *Experimental Analysis of Heuristics for the ATSP*, in: The Traveling Salesman Problem and its Variations, G. Gutin, A.P. Punnen (eds.), Kluwer Academic Publishers, 2002, 445–487.

[9] M. Fischetti, A. Lodi, P. Toth. *Exact Methods for the Asymmetric Traveling Salesman Problem*, in: The Traveling Salesman Problem and its Variations. G. Gutin and A.P. Punnen (eds.). Kluwer Academic Publishers, 2002, 169–194.

[10] R. Jonker, A. Volgenant, *Improving the Hungarian Assignment Algorithm*, Operations Research Letters 5, 1986, 171–175.

[11] R.M. Karp, *A Patching Algorithm for the Nonsymmetric Traveling Salesman Problem*, SIAM Journal of Computing 8, 1979, 561–73.

[12] R.M. Karp, J.M. Steele, *Probabilistic Analysis of Heuristics*, in: The Traveling Salesman Problem, E.L. Lawler *et al.* (eds.), Wiley, New York, 1990, 181–205.

[13] D.L. Miller, J.F. Pekny, *Exact Solution of Large Asymmetric Traveling Salesman Problems*, Science 251, 1991, 754–761.

[14] G. Reinelt, *TSPLIB*, `http://www.informatik.uni-heidelberg.de/groups/-comopt/software/TSPLIB95/`, 1995.

[15] A. Yeo, *Large Exponential Neighbourhoods for the TSP*, preprint, Department of Mathematics and Computer Science, Odense University, Odense, Denmark, 1997.

[16] W. Zhang, *Truncated Branch-and-Bound: A Case Study on the Asymmetric TSP*, Proc. AAAI-93 Spring Symposium on AI and NP-Hard Problems, 160–166, Stanford, 1993.

[17] W. Zhang, R.E. Korf, *On the ATSP under Subtour Elimination and Local Search*, Research Report, University of Los Angeles, Computer Science Department, February 27, 1993.