Integration of Structural Constraints into TSP Models

Nicolas Isoart, Jean-Charles Régin

Integration of Structural Constraints into TSP Models

Nicolas Isoart and Jean-Charles Régin

Université Côte d'Azur, CNRS, I3S, France
{nicolas.isoart,jcregin}@gmail.com

Abstract. Several models based on constraint programming have been proposed to solve the traveling salesman problem (TSP). The most efficient ones, such as the weighted circuit constraint (WCC), mainly rely on the Lagrangian relaxation of the TSP, based on the search for spanning tree or more precisely "1-tree". The weakness of these approaches is that they do not include enough structural constraints and are based almost exclusively on edge costs. The purpose of this paper is to correct this drawback by introducing the Hamiltonian cycle constraint associated with propagators. We propose some properties preventing the existence of a Hamiltonian cycle in a graph or, conversely, properties requiring that certain edges be in the TSP solution set. Notably, we design a propagator based on the research of k-cutsets. The combination of this constraint with the WCC constraint allows us to obtain, for the resolution of the TSP, gains of an order of magnitude for the number of backtracks as well as a strong reduction of the computation time.

Keywords: Global constraint, TSP, propagator

1 Introduction

The traveling salesman problem (TSP) is an NP-Hard problem. It has many applications and has been motivated by concrete problems, such as school bus routes, logistics, routing, etc. Almost all types of resolution methods (MIP, SAT, CP, evolutionary algorithms, etc.) have been used to solve it. When the graph is Euclidean, the most efficient program is the Concorde software [1]. Unfortunately, it cannot deal with additional constraints that are very present in real-world problems such as Pickup & Delivery, Dial-a-Ride, automatic harvesting, etc.

Solving the TSP is difficult since it involves finding a single cycle passing through all the vertices of a graph such that the sum of the costs of the edges it contains is minimal. It is quite easy to model the fact that each vertex belongs to a cycle. Indeed, it is sufficient that each vertex has at least two distinct neighbors, in other words, each vertex must be the end of at least two edges. Such a result can be obtained by modeling the problem as an assignment problem, which is solved in polynomial time. However, this model is not sufficient to obtain a single cycle in the graph, because the assignment corresponds to a coverage of the vertices by a set of disjoint cycles. From this model, we obtain solutions where each vertex belongs to a cycle, but not to a unique cycle. The covering by a unique cycle can be achieved by imposing that the subgraph generated by the selected edges is connected. The combination of these two aspects is what makes the TSP so difficult.

Unlike the previous approach, a model can be built based on the notion of a connected subgraph. It was exactly the idea of Held and Karp [7, 8] who represented this notion by a 1-tree that is formed by a node x, two adjacent edges of x and a spanning tree of the graph without x. A 1-tree such that each node has a degree 2 is a Hamiltonian cycle, and a minimum 1-tree with these constraints is an optimal solution of the TSP. The use of a 1-tree is interesting because a minimum 1-tree is a good lower bound of the TSP. In addition, its computation is strongly related to the computation of a minimum spanning tree.

Held and Karp proposed to relax the degree constraints with a Lagrangian relaxation. More precisely, the cost of the edges are modified in order to integrate the violation of these degree constraints. If a node has a degree strictly greater than 2, then the cost of its adjacent edges are decreased, and if the degree is strictly less than 2 then the cost of its adjacent edges are increased. A convergence towards the optimal solution is obtained by computing a succession of minimum 1-tree based on the Lagrangian relaxation.

The *weighted circuit constraint* (WCC) [2] implements the approach in constraint programming. This constraint can be considered as the state of the art in CP as mentioned by Ducomman et al. [4]: "The best approach regarding the number of instances solved and quality of the bound is the Held and Karp's filtering".

In this paper, we propose to improve the WCC by adding methods for solving Hamiltonian cycles (i.e. TSP without costs). To do this, we consider the work of Cohen and Coudert [3] on the structure of the Hamiltonian cycles carried out for the FHCP Challenge [6]. Fig. 1 shows an example in which the structure of the graph is important for the Hamiltonian cycle search. There is no Hamiltonian cycle in this graph, because it is impossible to find a cycle that visits all the vertices that pass only once through node C. Such a graph is said to be 1-connected: there is a vertex in the graph such that its removal disconnects it. We can therefore define a new structural constraint: if a graph is 1-connected, then it does not contain a Hamiltonian cycle.

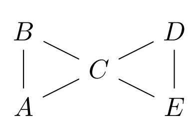

Fig. 1. Butterfly graph.

This idea can be extended to edges. For instance, consider two edges a_1 and a_2 whose deletion disconnects the graph (i.e. it is 2-edge-connected). If there exists an Hamiltonian cycle then it necessarily contains a_1 and a_2. We propose to study k-edge-connected graphs for $k > 1$, and in particular values $k = 2$ and $k = 3$, which are common in practice. From this study, we defined a general filtering algorithm named k-cutset propagator.

This article is organized as follows: first, we recall some concepts of graph theory. Then, we formally define the structural constraints used in our method of solving the TSP. Next, we define a new data structure called cycled spanning tree, which is used to define a new algorithm to exploit structural constraints. The last part experimentally shows the advantages of our method. Finally, we conclude.

2 Preliminaries

2.1 Definitions

The definitions about graph theory are taken from [12].

A **directed graph** or **digraph** $G = (X, U)$ consists of a **vertex set** X and an **arc set** U, where every arc (u, v) is an ordered pair of distinct vertices. We will denote by $X(G)$ the vertex set of G and by $U(G)$ the arc set of G. The **cost** of an arc is a value associated with the arc. An **undirected graph** is a digraph such that for each arc $(u, v) \in U$, $(u, v) = (v, u)$. If $G_1 = (X_1, U_1)$ and $G_2 = (X_2, U_2)$ are graphs, both undirected or both directed, G_1 is a **subgraph** of G_2 if $V_1 \subseteq V_2$ and $U_1 \subseteq U_2$. A **path** from node v_1 to node v_k in G is a list of nodes $[v_1, ..., v_k]$ such that (v_i, v_{i+1}) is an arc for $i \in [1..k-1]$. The path **contains** node v_i for $i \in [1..k]$ and arc (v_i, v_{i+1}) for $i \in [1..k-1]$. The path is **simple** if all its nodes are distinct. The path is a **cycle** if $k > 1$ and $v_1 = v_k$. A cycle is **Hamiltonian** if $[v_1, ..., v_{k-1}]$ is a simple path and contains every vertex of X. The **length** of a path p, denoted by $length(p)$, is the sum of the costs of the arcs contained in p. For a graph G, a solution to the **traveling salesman problem (TSP)** in G is a Hamiltonian cycle $HC \in G$ minimizing $length(HC)$. An undirected graph G is **connected** if there is a path between each pair of vertices, otherwise it is **disconnected**. The maximum connected subgraphs of G are its **connected components**. A k-edge-connected graph is a graph in which there is no edge set of cardinality strictly less than k disconnecting the graph. A **tree** is a connected graph without a cycle. A tree $T = (X', U')$ is a **spanning tree** of $G = (X, U)$ if $X' = X$ and $U' \subseteq U$. The U' edges are the **tree edges** T and the $U - U'$ edges are the **non-tree edges** T. A **bridge** is an edge such that its removal increases the number of connected components. A partition (S, T) of the vertices of $G = (X, U)$ such that $S \subseteq X$ and $T = X - S$ is a **cut**. The set of edges $(u, v) \in U$ having $u \in S$ and $v \in T$ is the **cutset** of the (S, T) cut. A k**-cutset** is a cutset of cardinality k. A k-cutset is minimum iff there is no subset of the k-cutset that disconnects the graph.

2.2 HCWME : Hamiltonian cycle with mandatory edges

CP-based algorithms solving the TSP tend to:

- Eliminate edges that cannot be part of the optimal solution.
- Define edges belonging to any optimal solution, called mandatory edges.

Since each optimal solution of the TSP is a Hamiltonian cycle, the TSP solution set is a subset of the solutions of the Hamiltonian cycle problem (HCP).

Property 1. Given $G = (X, U)$. If $a \in U$ belongs to all HCP(G) solutions, then a necessarily belongs to all TSP(G) solutions.

Property 2. Given $G = (X, U)$. If HCP(G) has no solution, then TSP(G) has no solution.

As the concept of mandatory arc is introduced, we formulate the Hamiltonian cycle with mandatory edges problem (HCWMEP) :
INSTANCE : A graph $G = (X, U)$ and a set of mandatory edges $M \subseteq U$.
QUESTION : Is there a Hamiltonian cycle in G containing all the edges of M ?

Since the HCP is an NP-Complete problem, HCWMEP(G, M) is NP-Complete.

3 Structural constraints

We will use the following notations $G = (X, U)$, $n = |X|$, $m = |U|$, $M \subseteq U$ the set of mandatory edges of G, $\mathcal{P} =$ HCMWEP(G, M). When not specified we will assume that G is symmetrical, connected and that a k-cutset is minimum.

Proposition 1. *Let K be a k-cutset, then any Hamiltonian cycle C contains an even and strictly positive number of edges from K.*

Proof. Consider a k-cutset of G and C a hamiltonian cycle. The k-cutset partition G into two sets of vertices X_1 and X_2. Let u be our starting vertex in X_1, by definition C visits all the vertices of G and ends up visiting u (its starting vertex). Thus, visiting the vertices of X_2 involves taking one edge of the k-cutset and taking a different one to come back into X_1, at that moment: either all the vertices of X_2 have been visited and we end up joining u without using other edges of the k-cutset, or we have to visit X_2 again and return to X_1, every time we visit X_2 from X_1 we need an edge to go in, and another to go back: this means an even number of edges and the proposition holds. □

From Proposition 1, we define Properties 3, 4, 5, 6 and 7.

Property 3. If there is $\{a_1, a_2\}$, a 2-cutset in G, then a_1 and a_2 become mandatory: $M \leftarrow M + \{a_1, a_2\}$.

Property 4. If there is a k-cutset with k odd containing $k-1$ mandatory edges in G, then the non-mandatory edge a is deleted because it cannot be part of a Hamiltonian cycle: $E \leftarrow E - \{a\}$.

Property 5. If there is a k-cutset with k even containing $k-1$ mandatory edges in G, then the non-mandatory edge a becomes mandatory: $M \leftarrow M + \{a\}$.

Property 6. If there is a 1-cutset in G, then $\mathcal{P}$ has no solution.

Property 7. If there is a k-cutset with k odd containing k mandatory edges in G, then $\mathcal{P}$ has no solution.

Definition 1. *It is said that two problems $\mathcal{P}$ and $\mathcal{P}'$ are equivalent if their solution sets are in bijection. We then note that $\mathcal{P} = \mathcal{P}'$.*

Corollary 1. *Given $\mathcal{P}' = \mathcal{P}$. If one or more of Properties 3, 4, 5, 6 or 7 are applied to $\mathcal{P}$, then $\mathcal{P}$ and $\mathcal{P}'$ remain equivalent.*

Proof. Immediate from Proposition 1. □

We write $a^* \in U$ a mandatory edge.

Example 1:

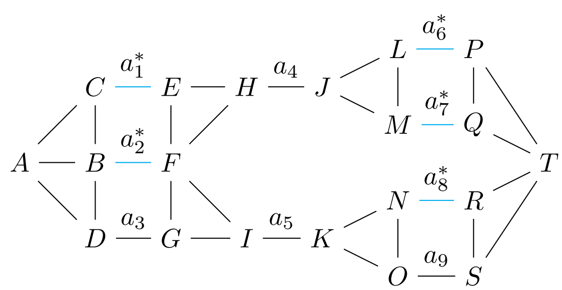

Fig. 2. Graph G_1.

From Properties 3, 4 and 5, we can remove some edges from Fig. 2 and make them mandatory:

• $\{a_4, a_5\}$ is a 2-cutset: if we want to connect the "left" part of H and I to the "right" part of J and K by a cycle we must take (H,J) and (I,K) so a_4 and a_5 become mandatory.

• $\{a_1^*, a_2^*, a_3\}$ is a 3-cutset and with $\{a_1^*, a_2^*\}$ mandatory: it is a cutset with an odd cardinality with an even number of mandatory edges. Then we can delete a_3, because by choosing it the cutset would become a mandatory set of edges with an odd cardinality.

• $\{a_6^*, a_7^*, a_8^*, a_9\}$ is a 4-cutset and with $\{a_6^*, a_7^*, a_8^*\}$ mandatory: it is a cutset with an even cardinality with 3 mandatory edges, so a_9 must be mandatory. Fig. 3 shows how G_1 is modified when Properties 3, 4 and 5 are applied.

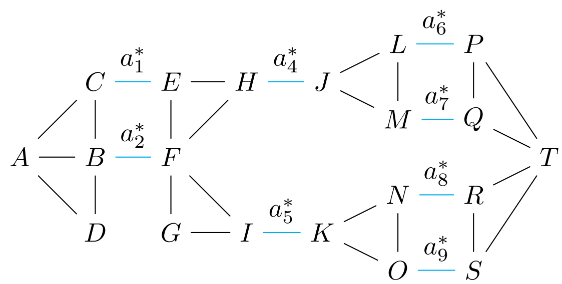

Fig. 3. Application of Properties 3, 4 and 5 on G_1.

Example 2:

Now, consider Fig. 4. From Property 7, there is no Hamiltonian cycle:

- $\{a_4\}$ is a 1-cutset: there is no Hamiltonian cycle connecting $\{I, J, K\}$ to the other part of the graph.
- $\{a_1^*, a_2^*, a_3^*\}$ forms a 3-cutset with three mandatory edges.

Properties 3, 4, 5, 6 or 7 are based on the cardinality of the cutsets, so it is reasonable to ask how many cutsets a graph can have.

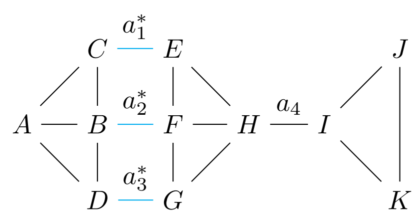

Fig. 4. Graph G_2.

Property 8. The number of cutset of a graph of order n is 2^n.

Proof. Any part $S \subseteq X$ forms an (S, T) cut. The cardinality of the powerset of $S \subseteq X$ is 2^n, so there are 2^n cutsets. □

In the case of the undirected graph, an (S, T) cut has the same cutset as the (T, S) cut. Hence, the number of distinct cutsets is $2^n/2 = 2^{n-1}$.

In the case of a very dense graph, there is a low probability of satisfying Properties 3, 4, 5, 6 or 7 for a small value of k. Nor does it seem very reasonable to apply these properties with a high value of k for at least two reasons:

• The complexity of the algorithms of k-cutset increases with k because they are enumeration algorithms [14].

• The relationship between the cardinality of the cutset and the number of mandatory edges is strong. The more k increases and the less chance we have of satisfying one of Properties 3, 4, 5, 6 or 7.

Consequently, we propose to study k =1, 2 and 3 with the following behaviors:

• 1-cutsets: raise a fail.

• 2-cutsets: make the edges of the 2-cutset mandatory.

• 3-cutsets: consider only the 3-cutsets with at least 2 mandatory edges. If it contains a non-mandatory edge, then it must be removed, otherwise a fail is raised.

4 k-cutset Propagator

We must be able to find the k-cutsets with $k = 1, 2, 3$ and two mandatory edges for $k = 3$. If we split the problem, finding the 1-cutset, which are actually bridges, can be done with the Tarjan algorithm [11] in $O(m+n)$; finding 2-cutset can be done with the Tsin algorithm [13] in $O(m+n)$. The strength of Tsin's algorithm is that it also allows us to find bridges, so we can manage $k = 1, 2$ at the same time. We now have to manage $k = 3$ with at least two mandatory edges.

By the cut definition, if you remove a k-cutset edge, then it becomes a $(k-1)$-cutset. We can then propose a first simple algorithm:

For each mandatory edge $a^* \in M$, we look for the 2-cutsets of $G - \{a^*\}$. In this way, each of the 2-cutset found forms a 3-cutset with at least one mandatory edge. It is then sufficient to keep only the 3-cutsets with 2 mandatory edges.

The number of considered mandatory edges can be reduced. To do this, we will build a special structure called CST. The CST is not a required structure for the proper functioning of the k-cutset propagator, just an improvement.

4.1 CST : Cycled Spanning Tree

A CST is a 2-edge-connected subgraph of G such that for each edge a of G there is a cycle in G formed only by edges of the CST and a.

One way to build a CST is to calculate T a spanning tree, then add some edges to T until all the edges, those of T and those outside of T, belong to a CST cycle. Any edge $a \notin T$ belongs to a cycle composed of a and only T edges.

For the edges of T, the CST is built by adding edges to the spanning tree such that each tree edge belongs to a cycle of CST. This can be done in linear time by marking the tree edges each time a cycle is found. More precisely, we consider three graphs at the same time: G the graph, T the spanning tree, and CST the CST, initially equal to T. All tree edges are unmarked. We traverse the non tree edges of G until we find $a_{\overline{T}} = (i, j) \notin T$ such that there is a cycle formed by at least an unmarked edge of T, some edges of T and $a_{\overline{T}}$. We add $a_{\overline{T}}$ to CST and we mark all the tree edges of C. We repeat this operation until there is no more unmarked edge in T. Clearly, at the end, each tree edge which has been marked belongs to a cycle. In addition, there is at least one tree edge in each cycle, so the number of added edges is bound by n. An example of a construction is shown in Fig. 5. This algorithm can be efficiently implemented, similarly to Kruskal's algorithm, by using a union-find data structure to avoid traversing each edge of each cycle. If we consider first the non mandatory edges for the construction of the spanning tree and for the construction of the CST, then we can expect to reduce the number of mandatory edges in the CST.

W.l.o.g. we assume that G is a connected bridgeless graph. Thus, there exist a CST in G.

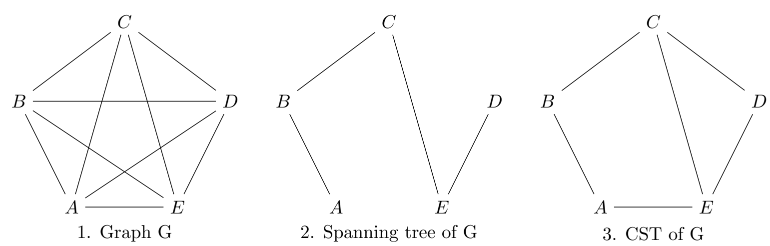

Fig. 5. Example of building a CST.

Corollary 2. *Given $k > 1$. If there is a k-cutset in G, then at least two edges of the cutset are in the CST.*

Proof. By construction, the CST is connected and covers the graph with cycles. So each cut has a cardinality greater than or equal to two. □

Corollary 3. *If there is a 3-cutset containing at least two mandatory edges, then at least one mandatory edge belongs to the CST.*

Proof. Immediate from Corollary 2. □

Definition 2. *The* **identification edges** *are the mandatory edges for which a 2-cutset algorithm is run.*

From Corollary 3, the simple algorithm can be improved by reducing the number of mandatory edges that are considered. Considering the identification edges as each mandatory edge a^* of CST, the algorithm becomes: for each identification edges, search for the 2-cutsets of $G - \{a^*\}$. For each 3-cutset found, we obtain either a 3-cutset with three mandatory edges, or a 3-cutset with two mandatory edges or a 3-cutset with one mandatory edge. Then, we apply Properties 3, 4, 5, 6 and 7.

Since mandatory edges outside the CST are not considered as identification edges and the edges in the CST are chosen during construction, it is a good idea to minimize the number of mandatory edges in the CST.

4.2 Additional improvement

The proposed algorithm is highly dependent on the number of identification edges. From Corollary 2, if two edges belong to the same 2-cutset and are mandatory, then they are identification edges. However, when searching for the 3-cutsets with an identification edge, it is not necessary to repeat the search for all the edges forming a 2-cutset with it. More precisely, the problem of searching for 3-cutsets with a^* as an identification edge has the same set of solutions as the problem of searching for 3-cutsets with each of the edges forming a 2-cutset with a^*. Fig. 6 illustrates it well since the 2-cutset is a path.

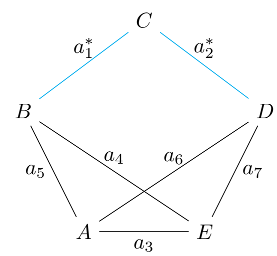

Fig. 6. $\{a_1^*,a_2^*\}$ is a 2-cutset. $\{a_1^*,a_4,a_5\}$ and $\{a_1^*,a_6,a_7\}$ are 3-cutsets including a_1^*. We can deduce that $\{a_2^*,a_4,a_5\}$ and $\{a_2^*,a_6,a_7\}$ are 3-cutsets including a_2^*.

Property 9. Let S_1 be a k-cutset and S_2 be a 2-cutset such that $k > 1$ and $S_2 \not\subseteq S_1$. If $\exists a \in S_1$ such that $a \in S_2$ then $(S_1 \cup S_2) - \{a\}$ forms a k-cutset.

Proof. Given $S_2 = \{a_1, a_2\}$ a 2-cutset and $a_1 \in S_1$. Removing S_1 from the graph disconnects it into two connected components. In the modified graph, $S_2 - \{a_1\} = \{a_2\}$ is a bridge. Removing a_2 further increases the number of connected components: there are now three. If we put back a_1, G is disconnected into two connected components, its cutset is $(S_1 - \{a_1\}) \cup \{a_2\} = (S_1 - \{a_1\}) \cup (S_2 - \{a_1\}) = (S_1 \cup S_2) - \{a_1\}$. Since S_1 is a k-cutset, there is no subset of it that disconnects the graph other than the k-cutset itself. If $(S_1 \cup S_2) - \{a_1\}$ disconnects the graph then it is a k-cutset because we delete and add an edge in a set of initial cardinality k. □

Consider S_1 a 3-cutset, $S_2 = \{a_1, a_2\}$ and S_3 two distinct 2-cutsets. From Property 9 the number of identification edges is reduced:

- If $a_1 \in S_1$, then $(S_1 - \{a_1\}) \cup \{a_2\}$ is a 3-cutset.
- If $a_1 \in S_3$, then $(S_3 - \{a_1\}) \cup \{a_2\}$ is a 2-cutset.

Thus, the set of identification edges is defined by the mandatory edges of the CST that do not belong to any 2-cutset and the subset of edges belonging to all 2-cutsets of G that maximizes its cardinality such that there is no combination of it forming a 2-cutset.

To avoid any inconsistency, all 2-cutsets must be searched before performing the 3-cutset search. Otherwise, there may be a 2-cutset containing at least one non-mandatory edge. This may result in a edge being marked as removable when searching for 3-cutsets while it is necessary for the existence of a Hamiltonian cycle. In addition, deleting an edge in a 3-cutset may create a 2-cutset and so either we perform a 2-cutset search immediately or we wait until the end of the search of all 3-cutsets to make the deletions effective. The first possibility is too time-consuming, a better solution is to postpone the deletions.

With this method we consider a subset of the identification edges. The higher the mandatory number of edges required, the more likely it is that the number of edges considered will be reduced.

Finally, CST has another advantage: it is incremental. Indeed, as long as no CST edges are removed, all edges outside the CST belong to a cycle composed of CST edges, so there is no need to rebuild it.

4.3 Implementation

Algorithm 1 is a possible implementation of the k-cutset filtering. The main function is PROPAGKCUTSET(G,M). Function PROPAG2CUTSET(G, M, set) defines a 2-cutset filtering. Function PROPAG3CUTSET(G, M, a^*) defines a 3-cutset filtering. Both filtering functions use FIND2CUTSET$(G, bridge, 2cutsandFound)$ which finds all 2-cutsets in G as proposed in [13] with a complexity in $O(n+m)$, this function is used as a black box. Filtering functions also have two subfunctions, bridge() and 2cutsetFound(M,a_1,a_2) describing the behavior that the FIND2CUTSET$(G, bridge, 2cutsetFound)$ algorithm must have when it finds a bridge or a 2-cutset in G. Function MERGECUTPAIRS(S, set, id) allows the use of the improvement proposed in section 4.2. We will now describe the overall behavior of the algorithm. In Function PROPAGKCUTSET(G, M), we define *set* as a set of pairs of edges forming 2-cutsets in G. Then, we use the filtering PROPAG2CUTSET(G, M, set) to find and make mandatory all the edges belonging to a 2-cutset in G, the 2-cutsets are stored in *set*. The *id* array represents for each edge its 2-cutset identifier. In order to create sets of edges forming 2-cutsets between them Function MERGECUTPAIRS(S, set, id) is called. Each disjoint set will finally have a different identifier and each edge belonging to the same set will have the same identifier. Then, we define an array *visited* to allow us to consider only one edge per set described above. The $identificationEdges$ set contains the mandatory edges which are in the CST. Then, we consider one edge per set calculated by MERGECUTPAIRS(S, set, id) and all the edges of the CST not being in any set. For each of its edges, the filtering PROPAG3CUTSET(G, M, a^*) is performed, i.e. the Properties 4, 5 and 7 are used. As recommended in section 4.2, deletions are postponed. The final complexity of the Algorithm 1 is $O(k*(n+m))$ where $k <= |M| <= n$. Tsin's algorithm $(O(n + m))$ is called k times.

Algorithm 1: k-CUTSET(G,M)

```
propagKCutset(G, M) :
    set ← ∅    /* set of pairs of edges representing the 2-cutset */
    if not propag2Cutset(G,M,set) then return False;
    ∀e ∈ U(G) : id[e] ← nil    /* contains the 2cutset identifier of each edge */
    mergeCutpairs(identificationEdges,set,id)
    U' ← U ∀e ∈ U(G) : visited[e] ← False
    identificationEdges ← CST(G).getMandatoryEdges()
    for each a* ∈ identificationEdges do
        if id[a*] = nil or ¬visited[id[a*]] then
            if not propag3Cutset(G, M, U', a*) then  return False;
            if id[a*] ≠ nil then visited[id[a*]] ← True;
    G ← (X,U')    /* As deletion are postponed, update G */
    return True

propag2Cutset(G,M,set):
    /* Return False if the graph isn't 3-edge-connected */
    define bridge(){Exit propagation}
    define 2cutsetFound(M, a1, a2){
        if a1 ∉ M then M ← M ∪ {a1};
        if a2 ∉ M then M ← M ∪ {a2};
        set ← set ∪ (a1, a2);
    }
    return find2Cutset(G,bridge,2cutsetFound)

propag3Cutset(G,M,U',a*):
    /* Return False if the graph contains a 3-cutset with 3 mandatory edges */
    define bridge(){ continue; }
    define 2cutsetFound(M, a1, a2){
        if a1 ∈ M  and a2 ∈ M then Exit propagation;
        else if a1 ∈ M then U' ← U' − {a2};
        else if a2 ∈ M then U' ← U' − {a1};
    }
    G' ← (X(G), U(G) − {a*})
    return find2Cutset(G',bridge,2cutsetFound)

mergeCutpairs(S,set,id):
    cpt ← 0
    for each (a1, a2) ∈ set do
        /* if both a1 and a2 do have an identifier */
        if id[a1] ≠ nil and id[a2] ≠ nil then
            /*(a1, a2) is a 2-cutset: id[a1] must be equals to id[a2] : id are merges*/
            for each s' ∈ S do
                if id[s'] = id[a1] then
                    id[s'] ← id[a2]
        /* if both a1 and a2 do not have an identifier */
        if id[a1] = nil and id[a2] = nil then
            id[a1] ← id[a2] ← cpt
            cpt ← cpt + 1
        /* if a2 does not have an identifier and a1 have one */
        if id[a1] ≠ nil and id[a2] = nil then
            id[a2] ← id[a1]
        /* if a1 does not have an identifier and a2 have one */
        if id[a1] = nil and id[a2] ≠ nil then
            id[a1] ← id[a2]
```

5 Experiments

The algorithms are implemented in Java 11 in a locally developed constraint programming solver. The experiments were performed on a Windows 10 machine using an Intel Core i7-3930K CPU @ 3.20 GHz and 64 GB of RAM. The reference instances are from the TSPLib [9], a library of reference graphs for the TSP. For fairness, we naturally took up the instances given by the state of the art [5]. All instances considered are symmetrical graphs.

We present the results in tables. Each of them reports the solving time in milliseconds. Timeout ($t.o$) is set at 30 minutes. The number of backtracks is denoted by #bk. Tables include a column expressing the ratio of solving time and number of backtracks.

	maxCost (1)		maxCost k-cutsetNotImproved (2)		Ratios (1)/(2)		maxCost k-cutset (3)		Ratios (1)/(3)	
Instance	time	#bk	time	#bk	time	#bk	time	#bk	time	#bk
gr96	13456	14970	3308	1492	4.1	10.0	3064	1492	4.4	10.0
rat99	132	40	321	40	0.4	1.0	196	40	0.7	1.0
kroA100	82296	96252	18594	9442	4.4	10.2	17632	9442	4.7	10.2
kroB100	243514	294148	15736	7286	15.5	40.4	15382	7286	15.8	40.4
kroC100	5937	4238	3677	1540	1.6	2.8	3646	1540	1.6	2.8
kroD100	806	480	944	286	0.9	1.7	819	286	1.0	1.7
kroE100	1213859	1628090	24986	9352	48.6	174.1	22968	9352	52.9	174.1
eil101	309	116	489	112	0.6	1.0	326	112	0.9	1.0
gr120	6610	3872	3089	980	2.1	4.0	2730	980	2.4	4.0
pr124	1876	566	1611	310	1.2	1.8	1530	310	1.2	1.8
bier127	822	402	770	146	1.1	2.8	641	146	1.3	2.8
ch130	18520	11810	7466	2250	2.5	5.2	6465	2250	2.9	5.2
pr136	t.o.	1733604	155283	35150	$\geq$ 11.6	$\geq$ 49.3	137675	35150	$\geq$ 13.1	$\geq$ 49.3
gr137	27828	11968	24223	6788	1.1	1.8	22579	6788	1.2	1.8
pr144	1622	466	1999	434	0.8	1.1	1603	434	1.0	1.1
ch150	11983	5684	6190	1424	1.9	4.0	5314	1424	2.3	4.0
kroA150	1290205	620080	174954	48892	7.4	12.7	171972	48892	7.5	12.7
kroB150	t.o.	791880	1222756	304630	$\geq$ 1.5	$\geq$ 2.6	1124443	304630	$\geq$ 1.6	$\geq$ 2.6
brg180	250527	2957988	t.o.	1000666	$\leq$ 0.1	$\leq$ 3.0	492962	2741812	0.5	1.1
rat195	t.o.	638322	1166767	271352	$\geq$ 1.5	$\geq$ 2.4	980190	271352	$\geq$ 1.8	$\geq$ 2.4
d198	440621	171294	273510	47838	1.6	3.6	179474	47838	2.5	3.6
kroB200	t.o.	647992	1586292	303282	$\geq$ 1.1	$\geq$ 2.1	1432978	303282	$\geq$ 1.3	$\geq$ 2.1
gr202	19681	9812	13385	2282	1.5	4.3	8261	2282	2.4	4.3
pr264	9520	1502	7817	256	1.2	5.9	6852	256	1.4	5.9

Table 1. Improvement of k-cutset filtering.

Table 1 shows the performance of adding k-cutset filtering to the WCC. This table is composed of three main columns (1, 2 and 3) showing the following results respectively: WCC without k-cutset filtering, WCC with k-cutset filtering without the improvement proposed in 4.2 and k-cutset improved filtering. A static strategy such as maxCost, selecting arcs by decreasing costs allows us to compare the performance of the filtering without any disruption due to the strategy. These results show that using structural filtering is very interesting. For example, the search space of pr136 has been reduced by a factor of 49.3 and its solving time by a factor of 11.6 if the improvement proposed in 4.2 is not considered, by a factor of 13.1 otherwise. Indeed, the number of backtracks is generally reduced by a large factor (mean equal to 14.4, geometric mean equal to 4.3), which allows a good reduction in solving time (mean equal to 5.3, geometric mean equal to 2.4). The improvement allows the results to be refined by further improving the solving times.

We are now considering different strategies, including LCFirst maxCost introduced by Fages et al. [5]. It keeps one of its two extremities for the last branching edge and selects the edges from the neighborhood of the kept node by decreasing costs. It is currently considered the best current strategy for resolving the TSP in CP.

	(1) LCFirst minDeltaDeg		(2) LCFirst minDeltaDeg k-cutset		Ratios (1) / (2)		(3) LCFirst maxCost		(4) LCFirst maxCost k-cutset		Ratios (3) / (4)		(5) LCFirst minRepCost		(6) LCFirst minRepCost k-cutset		Ratios (5) / (6)	
Instance	time	#bk	time	#bk	time	#bk	time	#bk	time	#bk	time	#bk	time	#bk	time	#bk	time	#bk
gr96	2327	1376	744	212	3.1	6.5	1951	1272	3113	1372	0.6	0.9	1534	746	1818	610	0.8	1.2
rat99	291	88	323	80	0.9	1.1	271	56	278	46	1.0	1.2	278	50	256	28	1.1	1.8
kroA100	9092	6278	4315	1846	2.1	3.4	5643	4048	7305	3726	0.8	1.1	3602	1884	3559	1288	1.0	1.5
kroB100	5321	3392	8380	3764	0.6	0.9	6359	4868	23181	10812	0.3	0.5	8232	4022	4419	1514	1.9	2.7
kroC100	2025	1126	2601	1076	0.8	1.0	1434	902	4451	2070	0.3	0.4	693	202	721	160	1.0	1.3
kroD100	868	410	917	290	0.9	1.4	705	286	778	240	0.9	1.2	410	76	453	80	0.9	1.0
kroE100	30414	26932	4304	1776	7.1	15.2	5488	4218	5604	2316	1.0	1.8	7650	3790	3479	1152	2.2	3.3
eil101	302	104	343	86	0.9	1.2	319	74	337	74	0.9	1.0	294	52	279	40	1.1	1.3
gr120	1311	468	685	112	1.9	4.2	1200	548	1791	578	0.7	0.9	1014	312	1062	214	1.0	1.5
pr124	6358	2336	7898	2462	0.8	0.9	1611	448	2387	582	0.7	0.8	1851	424	1415	208	1.3	2.0
bier127	520	128	466	56	1.1	2.3	609	216	728	180	0.8	1.2	533	84	1203	194	0.4	0.4
ch130	6953	3902	5301	1804	1.3	2.2	5287	2726	10243	3682	0.5	0.7	5028	1852	2826	750	1.8	2.5
pr136	19710	9822	28683	7448	0.7	1.3	262470	144980	160126	48370	1.6	3.0	181842	65974	55240	9926	3.3	6.6
gr137	8130	3640	6418	2092	1.3	1.7	5580	2158	13664	4208	0.4	0.5	4953	1548	3053	602	1.6	2.6
pr144	2742	648	3060	668	0.9	1.0	1463	256	1892	316	0.8	0.8	782	88	972	92	0.8	1.0
ch150	7189	2954	4824	1310	1.5	2.3	5100	1988	12350	3514	0.4	0.6	5034	1422	5348	1042	0.9	1.4
kroA150	34168	14996	14197	3874	2.4	3.9	21362	9510	63307	17526	0.3	0.5	14018	3724	8747	1702	1.6	2.2
kroB150	730330	320634	726592	207550	1.0	1.5	799195	373076	1194191	319360	0.7	1.2	1096412	331548	563570	114116	1.9	2.9
brg180	706	86	760	86	0.9	1.0	13423	125018	56323	267004	0.2	0.5	535	62	574	62	0.9	1.0
rat195	60531	17460	110822	25566	0.5	0.7	132012	41758	732018	178312	0.2	0.2	189821	40362	240102	32958	0.8	1.2
d198	26347	7062	27677	5686	1.0	1.2	71567	23740	93713	24048	0.8	1.0	119257	31262	51608	8044	2.3	3.9
kroB200	614139	191058	315601	67666	1.9	2.8	346683	114372	1393679	288336	0.2	0.4	360004	66452	149824	21622	2.4	3.1
gr202	4949	1582	7268	2004	0.7	0.8	8043	3248	7073	1906	1.1	1.7	5285	1066	6007	876	0.9	1.2
pr264	5816	190	6682	290	0.9	0.7	6631	322	7194	278	0.9	1.2	6663	206	6119	122	1.1	1.7
geo mean	6431	2274	5418	1324	1.2	1.7	6788	2911	11559	3490	0.6	0.8	5376	1271	4341	731	1.2	1.7
mean	65856	25695	53703	14075	1.2	1.8	71017	35837	158155	49119	0.4	0.7	83989	23217	46361	8225	1.8	2.8

Table 2. Dynamic strategies.

Surprisingly, Table 2 shows that the k-cutset filtering is not interesting for the LCFirst maxCost strategy. The fact is that for the selected instances, the geometric mean of the solving times increases from 6788ms to 11559ms when k-cutset filtering is used. From our experiments, the strategy seems very ad hoc in regards to the propagator of the WCC constraint and in particular to the Lagrangian relaxation. It seems to partially correct the lack of structural constraints of the WCC. However, Fages et al. [5] have proposed other strategies: LCFirst minDeltaDeg and LCFirst minRepCost, with performances comparable to LCFirst maxCost. The strategy LCFirst minRepCost is more suited to our model. It consists in selecting the edges by increasing replacement costs [2] with the LCFirst policy. This strategy has a slightly better sensitivity to the addition of k-cutset filtering and has the advantage of being generally more efficient. Indeed, between LCFirst minDeltaDeg and LCFirst minRepCost, we notice that when the k-cutset filtering is present, the geometric mean of the solving time of LCFirst minDeltaDeg is 5418 while that of LCFirst minRepCost is 4341, which is approximately 25% better. LCFirst minRepCost shows a significant reduction of the search space and a smaller reduction of the reduction time. For example, kroB200 gains a factor of 2.4 on solving time and a factor of 3.1 on the size of the search space.

	(1) LCFirst maxCost		(2) LCFirst minRepCost k-cutset		Ratios (1) / (2)	
Instance	time	#bk	time	#bk	time	#bk
gr96	1951	1272	1818	610	1.1	2.1
rat99	271	56	256	28	1.1	2.0
kroA100	5643	4048	3559	1288	1.6	3.1
kroB100	6359	4868	4419	1514	1.4	3.2
kroC100	1434	902	721	160	2.0	5.6
kroD100	705	286	453	80	1.6	3.6
kroE100	5488	4218	3479	1152	1.6	3.7
eil101	319	74	279	40	1.1	1.9
gr120	1200	548	1062	214	1.1	2.6
pr124	1611	448	1415	208	1.1	2.2
bier127	609	216	1203	194	0.5	1.1
ch130	5287	2726	2826	750	1.9	3.6
pr136	262470	144980	55240	9926	4.8	14.6
gr137	5580	2158	3053	602	1.8	3.6
pr144	1463	256	972	92	1.5	2.8
ch150	5100	1988	5348	1042	1.0	1.9
kroA150	21362	9510	8747	1702	2.4	5.6
kroB150	799195	373076	563570	114116	1.4	3.3
brg180	13423	125018	574	62	23.4	2016.4
rat195	132012	41758	240102	32958	0.5	1.3
d198	71567	23740	51608	8044	1.4	3.0
kroB200	346683	114372	149824	21622	2.3	5.3
gr202	8043	3248	6007	876	1.3	3.7
pr264	6631	322	6119	122	1.1	2.6
geo mean	6788	2911	4341	731	1.6	4.0
mean	71017	35837	46361	8225	2.5	87.4

Table 3. General results

Table 3 underlines the interest of using a structural filtering such as the k-cutset filtering. In comparison to the state of the art, we reduced the size of the search space for most instances by a very significant factor in order to obtain an improvement in solving time. There is a huge gain (solving time improved by 23.4) for the problem brg180. If we exclude this problem we improve the mean of the solving times by a factor of 1.5 and the mean of the number of backtracks by a factor of 3.6. The number of backtracks is reduced for each instance. The solving time is improved for 92% of the instances.

Note that the interaction of the k-cutset filtering with Lagrangian relaxation is not clear (the WCC is built around Lagrangian relaxation), a more in-depth study will have to be conducted to better understand it. Adding filtering can then disrupt the convergence of the latter and sometimes slow it down [10]. This explains why the gain factor of the number of backtracks is always much higher than that of the solving time.

6 Conclusion

We introduced a new structural constraint in the WCC based on the search for k-cutsets in the graph. The experimental results show the interest of our approach in practice. We observed that the number of backtracks is reduced by an order of magnitude depending on the chosen strategy and resolution times are significantly improved. The interactions between this constraint and the research strategy, as well as between this constraint and the Lagrangian model of the WCC, deserve further study.

7 Acknowledgements

We would like to thank Pr. Tsin for sending us his 2-cutset search algorithm implementation.

References

1. Applegate, D.L., Bixby, R.E., Chvatal, V., Cook, W.J.: The traveling salesman problem: a computational study. Princeton university press (2006)
2. Benchimol, P., Régin, J.C., Rousseau, L.M., Rueher, M., van Hoeve, W.J.: Improving the held and karp approach with constraint programming. In: Lodi, A., Milano, M., Toth, P. (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. pp. 40–44. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
3. Cohen, N., Coudert, D.: Le défi des 1001 graphes. Interstices (Dec 2017), https://hal.inria.fr/hal-01662565
4. Ducomman, S., Cambazard, H., Penz, B.: Alternative filtering for the weighted circuit constraint: Comparing lower bounds for the tsp and solving tsptw. In: AAAI (2016)

5. Fages, J.G., Lorca, X., Rousseau, L.M.: The salesman and the tree: the importance of search in cp. Constraints **21**(2), 145–162 (2016)
6. Haythorpe, M.: Fhcp challenge set: The first set of structurally difficult instances of the hamiltonian cycle problem (2019)
7. Held, M., Karp, R.M.: The traveling-salesman problem and minimum spanning trees. Operations Research **18**(6), 1138–1162 (1970)
8. Held, M., Karp, R.M.: The traveling-salesman problem and minimum spanning trees: Part ii. Mathematical Programming **1**(1), 6–25 (1971)
9. Reinelt, G.: Tsplib—a traveling salesman problem library. ORSA Journal on Computing **3**(4), 376–384 (1991)
10. Sellmann, M.: Theoretical foundations of cp-based lagrangian relaxation. In: Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings. pp. 634–647 (2004)
11. Tarjan, R.E.: A note on finding the bridges of a graph. Inf. Process. Lett. **2**, 160–161 (1974)
12. Tarjan, R.E.: Data Structures and Network Algorithms. CBMS-NSF Regional Conference Series in Applied Mathematics (1983)
13. Tsin, Y.H.: Yet another optimal algorithm for 3-edge-connectivity. Journal of Discrete Algorithms **7**(1), 130 – 146 (2009), selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP)
14. Yeh, L.P., Wang, B.F., Su, H.H.: Efficient algorithms for the problems of enumerating cuts by non-decreasing weights. Algorithmica **56**(3), 297–312 (2010)