A Bounded Path Propagator on Directed Graphs

Diego de Uña[1], Graeme Gange[1], Peter Schachte[1], and Peter J. Stuckey[1,2]
{d.deunagomez@student.,gkgange@,schachte@,pstuckey@}unimelb.edu.au

[1] Department of Computing and Information Systems, The University of Melbourne
[2] Data 61, CSIRO, Melbourne, Australia

Abstract. Path finding is an ubiquitous problem in optimization and graphs in general, for which fast algorithms exist. Yet, in many cases side constraints make these well known algorithms inapplicable. In this paper we study constraints to find shortest paths on a weighted directed graph with arbitrary side constraints. We use the conjunction of two directed tree constraints to model the path, and a bounded path propagator to take into account the weights of the arcs. We show how to implement these constraints with explanations so that we can make use of powerful constraint programming solving techniques using learning. We give experiments to show how the resulting propagators substantially accelerate the solving of complex path problems on directed graphs.

1 Introduction

Path-finding is an important task in (directed) networks. It arises in tasks such as graph layout [7], metabolic networks [25] or collaborative path-finding in video-games [22] among other examples. In many cases, though, side constraints make these problems highly combinatorial and no efficient algorithms exist.

In this paper, we focus on path-finding with distances. In order to do so, we go through preliminary steps to build two propagators from which we build a path propagator that works on the topology of the graph. Then, on top of that, we construct a propagator that takes into account the weights of the arcs to propagate distances.

Given a fixed directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we enforce properties of a graph variable $G = (V, E)$ subgraph of $\mathcal{G}$ using the following constraints:

- *dreachability*$(G, r, \mathcal{G})$ requires all nodes in G are reachable from root r;
- *dtree*$(G, r, \mathcal{G})$ requires that G forms a tree rooted at r;
- *path*$(G, s, d, \mathcal{G})$ ensures that G is a simple path from s to d;
- *bounded_path*$(G, s, d, \mathcal{G}, w, K)$ ensures that G is a simple path from s to d of length no more than K given the weight function w over the arcs of $\mathcal{G}$;

The focus of the present paper is the *bounded_path* propagator. We present two novel explanations for already existing propagation rules. Furthermore, we introduce a new stronger propagation technique with explanations as well. The explanations for this propagator and its new version are the main contributions of this paper.

Section 2 describes previous work as well as use cases for these propagators. Section 3 gives the necessary technical background to the reader as well as all the graph propagators for unweighted graphs. Section 4 describes *bounded_path*. Section 5 shows an extensive series of experiments justifying the use of these propagators and their explanations.

2 Related work

The three first constraints announced in the introduction were first introduced as part of CP(Graph) [6] in 2005, using a decomposition approach.

Later, Quesada *et al* [17] implemented the first reachability propagator and used it as a path constraint in their paper. They make use of simple propagation rules based on depth-first traversals of the graph and on the use of dominator nodes (i.e. nodes that appear in all paths). Nonetheless the asymptotic complexity of their algorithms is substantially greater than ours or those of Fages *et al.* [9] since they use a brute-force algorithm to compute dominator nodes.

Constraints for trees and forests were introduced in [1, 2, 9]. Although focused on forests, their work used better algorithms improving the work of Quesada *et al.* in [17] to make each individual tree connected. For explanations on the *dtree* constraint, we use the same algorithm as we previously introduced in [5] for undirected graphs.

In order to be self-contained, we describe *dreachability* and *dtree* in the preliminaries section. The propagations are based on previous work presented in [1, 9, 17]. The explanations are novel although the algorithms are similar to the undirected version which we already introduced [5].

Finding a simple path (no node repetitions) is a classic graph problem with wide applicability. The usefulness of the constraint arises when there are interesting side constraints. Our *path* propagator is based on the Ph.D. thesis by J.-G. Fages [8], which showed how to model the path constraint as a conjunction of *dtree* constraints:

$$path(G, s, d, \mathcal{G}) \Leftrightarrow dtree(G, s, \mathcal{G}) \wedge dtree(G, d, \mathcal{G}^{-1}) \tag{1}$$

This states that a path from s to d is the intersection of a subtree of $\mathcal{G}$ rooted at s and a tree in $\mathcal{G}^{-1}$ (the graph $\mathcal{G}$ with arcs reversed) rooted at d.

There exist other approaches to finding paths by using *circuit* style propagators [10]. We compare for the first time the tree-based and the circuit-based approaches where both use explanations.

Path finding with distances is one of the most well-studied graph problems, for which very well known fast algorithms exist. Many specific algorithms that handle some form of side constraint are also known. For instance, paths with resource constraints have been very well studied for electrical cars [23] and for bike routes [24]. Another application is the Generalized Shortest Path queries [18, 19] where a person needs to do a series of tasks during their journey and choose among different places to do them. The *bounded_path* constraint allows us to specify shorter path problems with arbitrary side constraints. It was introduced by Sellman [20, 21] with some propagation rules. Our work improves on this.

3 Preliminaries

3.1 Directed Graphs

A *directed graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a set of nodes $\mathcal{V}$ and arcs $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, where $e = (u, v)$ is an arc from $u = \mathit{tail}(e)$ to $v = \mathit{head}(e)$ (drawn '$u \longrightarrow v$', from tail to head). Given arc $e = (u, v)$ its *reverse* arc is $e^{-1} = (v, u)$. The *inverse* $\mathcal{G}^{-1}$ of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is $(\mathcal{V}, \{e^{-1} \mid e \in \mathcal{E}\})$. A *weighted directed graph* is a graph $\mathcal{G}$ with a weight function $w : \mathcal{E} \rightarrow \mathbb{N}^0$ mapping arcs to non-negative *weights*.

3.2 Lazy Clause Generation

Briefly, Lazy Clause Generation (LCG, [16]) is a technique by which CP solvers can *learn* from their mistakes. Propagators are extended to explain their propagations, and the failures they detect. These *explanations* are captured in clauses. When failure is detected, explanations are used to generate concise no-goods that explain why the failure occurred, and these are stored in the solver, preventing the same failure from occurring again. Using SAT technology to access and process explanations and no-goods allows very efficient handling of no-goods, and the reduction in search space for using explanation is usually substantial.

A critical consideration when constructing propagators for an LCG solver are the algorithms to generate concise and precise explanations of the propagation. Naive explanations may end up creating no-goods that are not reusable, while highly complex minimal explanations may require much more computation effort than propagation, and end up slowing down the solver.

3.3 Graph propagators with explanations

In order to model a graph variable $G = (V, E)$ subgraph of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in an LCG solver, we use Boolean variables c_n representing whether node $n \in \mathcal{V}$ is chosen to be in V and similarly Boolean variables c_e for each $e \in \mathcal{E}$ representing whether $e \in E$. Eventually the solution is the subgraph $G = (\{n | n \in \mathcal{V} \wedge c_n\}, \{e | e \in \mathcal{E} \wedge c_e\})$.

As we are searching for G, the variables c_n and c_e will become fixed by search or propagation. The propagators we describe here must infer new information as a consequence of the constraint they implement, hence reducing the future search. We say that an arc e is *mandatory* if at the current stage of the search c_e is *true* (and we draw it as '—' in the following figures), *forbidden* ('.....') if c_e is *false* and *unknown* ('- - -') for an unassigned arc. Similarly, we use the same terms for nodes: *mandatory* ('●'), *forbidden* ('◌') and *unknown* ('○'). Nodes or arcs that are mandatory or unknown are called *available*.

Basic graph propagation We assume that the graph variable G propagates basic graph properties: the endnodes of an arc in the graph are also in the graph. Explanations for this are given in previous work by the authors [5].

Reachability propagation The *dreachability* constraint guarantees that all nodes in the subgraph G are reachable from a given node r. Quesada *et al.* [17] first proposed this propagator, although our algorithm is substantially improved by making use of the Lengauer-Tarjan algorithm to find dominators in a digraph [13]. Fages *et al.* [9] already used this algorithm.

Detecting and explaining failure: In order to detect that the current assignment of arcs and nodes in G is invalid, we need to check if all the nodes in G (i.e. mandatory) are reachable from the given node r. We first perform a depth-first search (DFS) in $(\{n \mid c_n \neq \mathit{false}\}, \{e \mid c_e \neq \mathit{false}\})$ starting at r, saving all the nodes visited in a set R. If some mandatory node f is not in R, we need to fail.

To explain why the mandatory node f is not reachable, we need to find forbidden arcs that would have let it be in R if they were not forbidden, similarly to the work in [5] for undirected graphs.

To find these arcs, we perform a DFS in $\mathcal{G}^{-1}$ starting at f, this time following all (reversed) arcs, regardless of their current state. Whenever the head of a forbidden reversed arc $e^{-1} = (t, h)$ is in R, e could have been used to extend the reachable area further and eventually reach f. Therefore, e must be in the explanation (we do not cross e^{-1} in this DFS). We add such arcs to a set F_f. Then an explanation for failure is: $c_r \wedge c_f \wedge \bigwedge_{e \in F_f} \neg c_e \Rightarrow \mathit{false}$. This exact same rule can be applied for propagation to eliminate unreachable nodes.

Finding dominators: During the search, we also make inferences that will accelerate the search. We say that a node t is *dominated* by a node d from r if all paths from r to t go through d. The *immediate dominator* of a node is the dominator that is its closest ancestor. For reachability, immediate dominators of mandatory nodes must be mandatory, otherwise some node (namely t) would not be reachable from r.

Finding immediate dominators in a graph can be done using the Lengauer-Tarjan algorithm [13] in $\mathcal{O}(|\mathcal{E}|\alpha(|\mathcal{E}|, |\mathcal{N}|))$ where α is the inverse Ackerman function. Their algorithm builds an array representation of a so-called *dominator tree* where the parent of a node is its immediate dominator. For our purposes, we apply the algorithm to $(\{n \mid c_n \neq \mathit{false}\}, \{e \mid c_e \neq \mathit{false}\})$.

We assume that the reachability has been ensured and thus all mandatory nodes are reachable from r. To enforce dominators to be in G, we build a queue containing all the mandatory nodes and iterate through the queue until it is empty while making their immediate dominators mandatory (if they are not already) and enqueueing them. This way, all the nodes in the path between r and some mandatory node t in the dominator tree become mandatory.

Now, we need to explain why each dominator that we fix is mandatory. Explaining this inference comes down to explaining the failure that would happen if the dominator d of t was forbidden. We compute a partition of nodes P from which t can be visited without going through the dominator by performing a DFS starting at t in $(\{\mathcal{V} \backslash \{d\}, \{e^{-1} \mid c_e \neq \mathit{false}\})$. Now we find alternative ways to get to any mandatory node beyond the dominator (that is, not in P) without using d. For that, we perform another DFS in $(\{\mathcal{V} \backslash \{d\}, \{e^{-1} \mid e \in \mathcal{E}\})$ starting in t, this time allowing the use of forbidden arcs. Whenever a forbidden reversed arc $e^{-1} = (t, h)$ has its tail on the set P but its head is a node outside of P we know that e would have allowed us to short-circuit d if it was not forbidden. Let F_t be the set of such arcs. The explanation for making a dominator mandatory is: $c_r \wedge c_t \wedge \bigwedge_{e \in F_t} \neg c_e \Rightarrow c_d$.

Finding bridges: Additionally, if any mandatory node n (other than r) has only one incoming arc that is not forbidden, that arc can be set as mandatory (if it is not already). This is because lacking that arc would make that node unreachable. That arc is called a bridge. The explanation for including this arc e in G is trivial: $c_n \wedge \bigwedge_{e_i \in to(n) \setminus \{e\}} \neg c_{e_i} \Rightarrow c_e$, where $to(n)$ is the set of arcs incident to n. Similarly, if n no longer has any incident arcs available, we have to set it to $false$, or fail if it is mandatory, explained by $\bigwedge_{e_i \in to(n)} \neg c_{e_i} \Rightarrow \neg c_n$.

Directed Tree propagator Trees are connected graphs, therefore *dtree* inherits from *dreachability*. Additionally, trees cannot contain any cycle (whether it is directed or not). Maintaining this condition is the task of *dtree*.

For this propagator, the algorithm is trivial. The use of the adequate data structure to detect cycles is what makes the whole propagator. We use the Rerooted Union-Find (RUF) data structure [5] to detect cycles and retrieve explanations. This yields a propagator identical to the one for undirected graphs [5] since cycle detection in undirected and directed graphs is equivalent as far as trees are concerned.

4 Bounded Path propagator

As we will see in the experiments, the decomposition of the path constraint as two trees (eq. 1) is not competitive for solving shortest path problems when compared to the alternative circuit formulation by Francis *et al.* [10]. For this reason, we needed a bounded path propagator to enhance optimality proving.

In this section we present a $bounded\_path(P, s, d, \mathcal{G}, w, K)$ propagator that ensures that the weight of the simple path P from s to d in $\mathcal{G}$ is no more than K. The weights of the arcs are given by the function $w : \mathcal{E} \mapsto \mathbb{N}^0$. The propagations in section 4.1 and 4.2 were already introduced by Sellman [20, 21], without explanation. As we will see in the experimental section, the explanations greatly improve these propagations.

4.1 Propagating simple distances

This constraint fails when there is no path from s to d of cost no more than K. This property naturally extends to all nodes in the path: the distance from s to any node n in P must be no more than K. The best correct lower bound for this is obviously the shortest path from s to $n \in P$: if the shortest path is longer than K, then no solution of cost less than or equal to K exists.

We compute the shortest path from s to every node in $(\mathcal{V}, \{e \mid c_e \neq \mathit{false}\})$ (i.e. avoiding forbidden arcs) using Dijsktra's algorithm. This yields the shortest *available* path from s to all nodes. If the cost of the path to a node n is greater than K, we can forbid n. This reasoning can be applied in both directions: d cannot be further than K from any node n in the path. For this reason, we also apply this rule starting Dijkstra's algorithm at d on the reversed graph.

To explain this inference we need to find (at least a superset of) the arcs that made the path to n too expensive. Let F_n be that set of arcs (initially empty). Let δ_x be the shortest available path from s to some node x, and δ_x^{-1} the shortest available path from x to d. Any arc $e = (u, v)$ such that $\delta_u + w[e] + \delta_v^{-1} \leq K$ can be used to connect s to d in no more than K (we say e is in a *short-enough* path). We can easily keep track of those arcs since both runs of Dijkstra's algorithm yield δ_u and δ_v^{-1}. When such an arc is forbidden, a feasible path is removed from the graph. We then add e to F_n. Eventually, F_n contains *all* the arcs causing n to be too far from either s or d. We have the following Theorem:

Theorem 1. *Let $[\![\delta_d \leq K]\!]$ be the literal stating that δ_d (i.e. the length of P) should be less than or equal to K (K is typically a variable). Then, $[\![\delta_d \leq K]\!] \wedge \bigwedge_{e_f \in F_n} \neg c_{e_f} \Rightarrow \neg c_n$ is a valid explanation for why n cannot be in G.*

Note the explanation set F_n is the same for any node n further than K from the source, its not specific to a particular n. We will address this flaw and give an example in Fig. 2 later on. The explanations can be used to explain failure too.

These explanations can be computed very efficiently by storing a function giving constant time access to whether an arc has been in a short-enough path. Upon removal of an arc e, we add it to F_n if e has been in a short-enough path.

4.2 Propagating combined distances

The previous rule removes any node that is too far from the source or too far from the destination to be in the path P, or detects failure. In addition, we can consider nodes through which a path from s to d would be longer than K and filter them. Similarly we can filter arcs.

Proposition 1. *Let δ_u be the cost of the shortest path from s to u, and let δ_u^{-1} be the cost of the shortest path from u to d. If $\delta_u + \delta_u^{-1} > K$, then u cannot be in the path from s to d of cost less than or equal to K.*

Proposition 2. *Let $e = (u, v)$ be an arc of cost $w[e]$. Let δ_u and δ_v^{-1} be the cost of the shortest paths from s to u and v to d respectively. If $\delta_u + w[e] + \delta_v^{-1} > K$, then e cannot be in a path from s to d of cost less than or equal to K.*

We use these observations to filter out nodes and arcs that cannot participate in the path from s to d.

To explain these propagations, we note that if the filtered element (either node or arc) was mandatory, we would have to fail. Thus the explanations are the same as given in Theorem 1 (applied to the node or the arc we are propagating here). These explanations can be used for failure if either u or e is mandatory.

Algorithm 1 Shortest path from s to d containing all mandatory nodes M.

```
1:  procedure DPBound($\mathcal{G}, s, d, ns = \{c_n | n \in \mathcal{V}\}, es = \{c_e | e \in \mathcal{E}\}, w, M$)
2:      $Q \leftarrow newPriorityQueue()$; $Q.push((s, \{s\}, 0))$;
3:      $tables[s][\{s\}] \leftarrow 0$                                   ▷ One table per node
4:      while $\neg Q.empty()$ do
5:          $(u, m_p, \gamma) \leftarrow Q.top()$; $Q.pop()$
6:          if $tables[u].contains(m_p) \wedge tables[u][m_p] < \gamma$ then continue;
7:          for all $e = (u, v) \in \{e | e \in \mathcal{E}\}$ do
8:              if $c_e = false$ then continue;
9:              if $\neg tables[v].contains(m_p) \vee (tables[v][m_p] > \gamma + w[e])$ then
10:                 $tables[v][m_p] \leftarrow (p, \gamma + w[e])$
11:                 $Q.push(v, m_p \oplus v, \gamma + w[e])$          ▷ $S \oplus v$ adds $v$ to set $S$ iff $v \in M$
12:     return $tables[d][M]$
```

4.3 Stronger bounding using Dynamic Programming

Although the implementation of *bounded_path* explained above proves to be useful, the bound is too weak if there are many intermediate nodes. For this reason, we developed a dynamic programming (DP) lower bound. If the previous one does not prune, we run a more expensive DP algorithm to find the shortest path from s to d containing *all* the mandatory nodes.

The algorithm is similar to Dijkstra's, but our priority queue stores more information. Each entry is a tuple (u, m_p, γ): a node u, the set of mandatory nodes m_p visited in some path p leading to u and the cost of p. As usual, the priority is on the cost.

We associate a hash-table to each node n that maps sets of mandatory nodes (encoded as bit-sets in our implementation) to the cost of visiting those nodes before reaching n. Formally the tables are functions $tables[n] : (M' \subseteq M) \mapsto \mathbb{N}$.

Then, when a tuple (u, m_p, γ) is retrieved from the queue, the algorithm considers each available arc (u, v). For each neighbor node v, we first check if there is a set m'_p in its table such that $m'_p = m_p$. If m'_p exists and its associated cost is greater than $\gamma + w[(u, v)]$, we update the entry on v's table, and enqueue $(v, m_p \oplus v, \gamma + w[(u, v)])$ (where $\oplus$ adds v to m_p iff v is mandatory, and returns m_p otherwise). If such m'_p does not exist, we add that same entry to v's table and enqueue it. We do not need to enqueue or update any table if m'_p exists and its associated cost is less than $\gamma + w[(u, v)]$. The cost of the shortest path to d containing all the nodes will be found in d's table. If such path does not exist, we simply return an error code and fail with the naive explanation (all the fixed arcs and nodes). In practice, this rarely happens.

Notice that this algorithm does not give simple paths, and therefore it does not give an exact lower bound. Indeed, if we did, we would need to keep track of all the states in the path, making the state space grow too quickly. Instead we only keep track of the mandatory nodes visited.

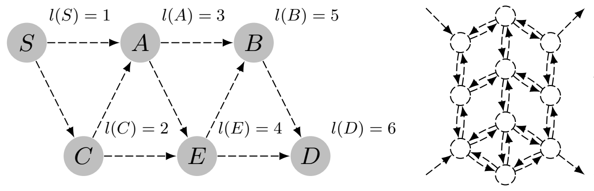
The explanation for pruning is the same as in Theorem 1, but we need to add the conjunction of c_n for all the mandatory nodes $n \in M$. Note that the asymptotic complexity of this algorithm is $\mathcal{O}(n2^{|M|}|M|log(n))$, hence the state may grow prohibitively. We will study solutions to this issue in the next subsection. Nonetheless, as we will see in the results, this explosion rarely happens since the higher the number of mandatory nodes, the smaller the choice in arcs.

Limiting state explosion in the DP propagation

Strongly Connected Components: Some basic inference we can take into account to reduce the state explosion is based on strongly connected components (SCC) of the current graph. There is no point for the DP algorithm to take an arc leaving SCC A if it has not yet visited all the mandatory nodes in A.

We use Kosaraju's algorithm [12] to compute SCCs. We then label the SCCs as follows. Let m be the number of SCCs containing at least one mandatory node (we call them mandatory SCCs). The SCC D containing the destination node d is labeled m. All other mandatory SCCs are numbered with the number of the lowest numbered SCC they can reach minus one. All non-mandatory SCCs are numbered with the lowest numbered SCC they can reach. It is easy to do this in linear time using a topological sort on the graph of SCCs. We call this *levels* and we denote the level of an SCC A by $l(A)$. Then, if an arc e goes from A to B such that $l(B) > l(A) + 1$, by crossing it we would skip some mandatory SCC to which we can never go back. Similarly, if A is mandatory and $l(B) = l(A) + 1$ we only cross e if we have visited all mandatory nodes of A, otherwise we would not be able to get back to A to finish visiting the mandatory nodes in A.

This process can greatly accelerate the DP algorithm without losing pruning power. Nonetheless, because during the search the partially assigned graphs tend to have a succession of SCCs of only one node followed by a big SCC containing all the unassigned nodes, we often did not see a benefit from this. It is, however, very worthwhile running at the root level. As a simple example, consider the graph in Figure 1: it takes 0.03 seconds to solve the problem using the SCC labeling, but 22.72s without it (same number of nodes and conflicts).



Each node on the left graph is an SCC of the form of the right graph (from [17]). Each SCC contains 3 random mandatory nodes.
All edges have weight 1.

Fig. 1. Example of use of SCCs to accelerate Algorithm 1.

Clustering mandatory nodes: Further acceleration can be achieved by reducing the number of mandatory nodes to visit. To decide which ones to ignore, we use the k-means clustering algorithm [11] on the set M of mandatory nodes. We use the centroids of the clusters only as mandatory nodes (i.e., we have as many mandatory nodes as clusters, treating the non-centroid nodes as unknown). Because the centroids tend to be equidistant to the other nodes in the cluster, the DP tends to also use some of the other mandatory nodes, thus visiting more than just the centroids. Also, since k-means has some inherent randomness, we have different clusters every time, which is also beneficial for the lower bound.

This has huge performance effects, but is a double-edged sword: the DP gets faster but we prune much less often as the bound is not as high. In order to regulate this, we use a simple heuristic based on the time spent by the DP. If the DP algorithm with C clusters takes less than x seconds, we increase C by 1, if it takes more than y seconds, we lower it. For the experiments where we used clustering, we chose $x = 0.5s$, $y = 8.0s$ and started with $C = 5$.

4.4 Improving the explanations

So far, the explanations for *bounded_path* have been the set of forbidden arcs that were in a short-enough path at some point (see Section 4.1). One problem with these explanations is that they are not targeted. It is easy to see that some of the arcs in the explanations may have nothing to do with the fact that some specific node n is too far from the source. We now provide better explanations.

Simple and combined distances propagation First, during the propagation we use Dijkstra's algorithm on the available graph. This leaves a label on each node indicating how far it is from the source. These labels are noted $\delta_n, \forall n \in V$. Nodes not visited have label $\delta_n = \infty$.

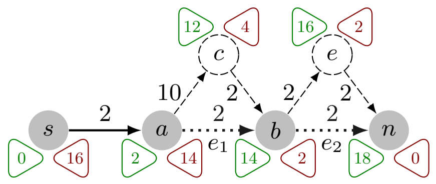
Let n be a node that is at distance δ_n more than the limit K from the source. Alg. 2 returns a set of forbidden arcs that explain why $\delta_n > K$.

Algorithm 2 Explaining why n is at distance more than K from the source.

```
1:  procedure ExplainDist($\mathcal{G}, s, n, \{\delta_u | u \in \mathcal{V}\}, K$)    ▷ We consider all arcs in $\mathcal{G}$
2:      $Q \leftarrow newPriorityQueue(); Q.push((n, 0));\ X = \emptyset;\ cost = [\infty | v \in \mathcal{V}]$
3:      while $\neg Q.empty()$ do
4:          $(u, \delta_u^{-1}) \leftarrow Q.top(); Q.pop()$    ▷ $\delta_u^{-1}$ : cost of the shortest path from $n$ to $u$
5:          if $u = s$ then break    ▷ Reached start $s$
6:          for all $e = (v, u) \in \{e | e \in \mathcal{E}\}$ do    ▷ Notice that we take arcs backwards
7:              if $e \in F \wedge \delta_v + w[e] + \delta_u^{-1} \leq K$ then $X = X \cup \{ e \}$
8:              else if $cost[v] > \delta_u^{-1} + w[e]$ then
9:                  $cost[v] = \delta_u^{-1} + w[e]$    ▷ Update cost
10:                 $Q.push(v, cost[v])$    ▷ Overwrites previous instances of $v$ in $Q$
11:     return $X$
```

Alg. 2 mimics Dijkstra's starting at n in $\mathcal{G}^{-1}$. For any arc $e = (v, u)$ of weight $w[e]$ we know δ_v (obtained during propagation). We also have the distance from u to n (the cost of the last current node in the loop, line 4). Let X be the initially empty set of arcs explaining why n is at distance more than K from s. When considering a forbidden arc, if $\delta_v + w[e] + \delta_u^{-1} \leq K$, e participates in a path from s to n no longer than K. Therefore we add it to the explanation **and we do not cross it**. Otherwise, we can cross it. Once we dequeue s we finish since all other paths are no shorter than $\delta_s^{-1} > K$. See Figure 2 for an example of explanation.

Theorem 2. *The clause $[\![\delta_d \leq K]\!] \wedge \bigwedge_{e_f \in X} \neg c_{e_f} \Rightarrow \neg c_n$ computed by Algorithm 2 is a correct and* ***minimal*** *explanation for why n is too far from s to be in G.*



K = 10, $\delta_n = 18 > K$, thus we fail.
Algorithm 2, starting at n (line 7):
e_2: $14 + 2 + 0 = 16 > K \;\Rightarrow$ cross it.
e_1: $2 + 2 + 2 = 6 \leq K \;\Rightarrow$ **explanation**.
Only e_1 is needed in the explanation. The basic explanation would have added both.

Fig. 2. Example of improved explanations. The labels for propagation ('▷', from Dijkstra's algorithm) and explanation ('◁', from Alg. 2) are given next to each node.

Proof. Let F be the set of forbidden arcs at the time of explanation. At any stage of Alg. 2, let F_p be the set of forbidden arcs not yet considered (initially F), X the arcs in the explanation, $d_{G'}(u, v)$ the shortest distance from u to v for any $G' \subseteq \mathcal{G}$, and u the top of Q. Let $G_R = \mathcal{G} \setminus (F_p \cup X)$. We ensure correctness and minimality by preserving the following invariants: (1) $d_{G_R}(s, n) > K$, (2) for all $(v', u') \in F_p$, $d_{G_R}(u, n) \leq d_{G_R}(u', n)$, and (3) for all $e \in X$, $d_{G_R \cup \{e\}}(s, n) \leq K$.

The three invariants hold initially: $G_R = \mathcal{G} \setminus F = G$, so (1) is the bound to be explained, (2) holds because n is initially the head of Q and all weights are non-negative, and (3) holds because X is initially empty.

At each iteration, we remove u from Q and process each arc $e = (v, u) \in \mathcal{E}$ (removing all forbidden arcs (v, u) from F_p, preserving (2) as nodes are processed in order of distance from n). We add arcs such that $\delta_v + w[e] + \delta_u^{-1} \leq K$ to X (preserving property (3)). Other forbidden arcs are now made available in G_R.

We show how adding these arcs to G_R maintains the invariants. Note $d_{G_R}(x, n)$ values for previously processed nodes x remain unchanged as any newly introduced path from n must be at least as long as δ_u^{-1}. Newly available arcs may, however, decrease $d_{G_R}(x, n)$ for some x which has not yet been processed. However, if $d_{G_R}(x, n)$ decreased as a result of (v, u) becoming available, then the shortest path from x to n must pass through u. But, x is not yet processed, so still $d_{G_R}(u, n) \leq d_{G_R}(x, n)$, preserving property (2). If the shortest path from s to n were to be reduced because now $\delta_x + d_{G_R}(x, n) < \delta_n$, there is a contradiction since this path goes through (v, u) meaning the arc should have been added to X and be unavailable. Hence property (1) is preserved. Adding arcs to G_R can only make paths shorter, hence property (3) is preserved.

Once s is popped from Q, our explanation is X. By (1), $F_p \cup X$ is a valid explanation; but by (2), no arc e remaining in F_p may be on a shorter path from s to n (as either n is unreachable via e, or the head of the arc is distance no less than δ_s^{-1} from n), so e may be omitted from the explanation. Thus X is also a valid explanation. Removing arcs from F_p, thus adding them to G_R preserves property (3). By (3), omitting any element of X introduces a path from s to n of length no greater than K, so X is also minimal. □

Clearly, Alg. 2 runs in $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}| log(|\mathcal{V}|))$, like Dijkstra's algorithm. We can use it to explain Prop. 1 and 2 as follows. For Prop. 1, we first obtain $X_1 = \textsc{ExplainDist}(\mathcal{G}, s, u, \{\delta_v | v \in \mathcal{V}\}, K - \delta_u^{-1})$, the explanation for u being at distance $K - \delta_u^{-1}$ from s. The call to Alg. 2 also yields the distance δ_u^* from s

to u in $\mathcal{G}$ that is still greater than $K-\delta_u^{-1}$. Let X_2 be the explanation for d being at distance $K - \delta_u^*$ from u. The final explanation is $X = X_1 \cup X_2$. The same idea can be used for Prop. 2, using the head and tail of the arc to be removed.

DP-based propagation We can also improve the explanations for the DP-based propagation. Similarly to the simple propagation, Alg. 1 leaves a table on each node stating the cost of visiting some subsets of mandatory nodes before getting to that node. If d is not reachable in less than $K+1$ visiting all mandatory nodes, we fail and explain the failure. To do so, we run the same Alg. 1 starting at d on the revered graph allowing forbidden arcs (similarly to Alg. 2).

Let $e^{-1} = (v, u)$ be some reversed forbidden arc of cost $w[e]$. On node u (the tail of e in the original graph) lies the table left from the propagation pass of Alg. 1. Each row of the table is a pair (m_u, γ_u) as defined in 4.3. Symmetrically, node v contains a table where each row (m_v, γ_v) indicates the mandatory nodes visited from d to v. If there exists an entry (m_u, γ_u) in u's table and an entry (m_v, γ_v) in v's table such that $m_v \cup m_u = M$, then e is an arc that could be used in a path from s to d containing all nodes in M. If additionally, $\gamma_v + w[e] + \gamma_u \leq K$, that path would be a valid path. Therefore, e being forbidden explains why we can't reach d visiting all mandatory nodes in no more than K. This corresponds to substituting the if-condition in line 8 of Alg. 1 with a call to Explain from Alg. 3 (where e is the reversed arc of whom we are visiting the tail, namely v).

State explosion for explanations: The explanation algorithm needs to use the same mandatory nodes as the propagation. Therefore, if we clustered, the same clustering is given to this algorithm. Also, we cannot use SCC levels here (other than the ones computed at the root) since we need to traverse forbidden arcs whether or not they skip entire mandatory SCCs as there may be other forbidden arcs leading to the skipped SCCs later.

A major problem with these explanations is that we need to traverse forbidden arcs. In dense graphs, this can be slow as there may be many possible paths to consider. For this reason, we use a simple stopping condition. Let t_p be the time it takes to run Algorithm 1 for propagation. If explaining is taking more than $x \times t_p$ (we choose x arbitrarily) we switch to the version of Explain in Algorithm 4 which corresponds to the basic explanations described in Theorem 1. We say that we *interrupt the explanation* when this change happens.

Algorithm 3 Better explanations

```
1: function EXPLAIN($e, \gamma_v, m_v$)
2:     for all $(m_h, c_h) \in table[head(e)]$ do
3:         if $m_h \cup m_v = M$ then
4:             if $\gamma_h + w[e] + \gamma_v \leq K$ then
5:                 $explanation.add(\neg c_e)$
6:                 return true
7:     return false
```

Algorithm 4 Avoiding state explosion

```
1: function EXPLAIN($e, \gamma, m$)
2:     ▷ $was\_short(e) = true \Leftrightarrow e$ was in a short-enough path at some point.
3:     if $\neg c_e \wedge was\_short(e)$ then
4:         $explanation.add(\neg c_e)$
5:         return true
6:     return false
```

5 Experimental results

In this section we test our *bounded_path* in different problems (all benchmarks available at [4]). We implemented all our work in the CHUFFED solver [3]. All tests are run on a Linux 3.16 Intel® Core™ i7-4770 CPU @ 3.40GHz machine.

We annotate the tests EXPL when learning is enabled, NOEXPL otherwise. We use EXPL* for the improved explanations. We name the tree decomposition for *path* PATH, BPATH the *bounded_path* propagator without the DP algorithm, and DPBPATH when using the DP algorithm. We compare failures (the number of times the solver has encountered a wrong valuation of the variables before proving optimality), the number of nodes (the size of the search space explored) and the time in seconds.

We found it beneficial to add an array of successors constrained as $c_e \Leftrightarrow succ[tail(e)] = head(e)$. Definitions of all search strategies are given in [15].

5.1 Node constrained shortest paths

Here we compare our path propagators with the results from [17] using their same benchmarks. The aim of these problems is to find the shortest path between two given nodes in a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ passing through a set of mandatory nodes M. We present the results in Table 1 using `first_fail` on the *succ* variables as the search strategy.

		[17]		[10]			PATH		PATH+BPATH		PATH+DPBPATH	
Benchmark	$\lvert\mathcal{N}\rvert$	Fails	Time(s)	Fails	Time(s)		Fails	Time(s)	Fails	Time(s)	Fails	Time(s)
Ham22	22	13	4.45	24	0.00	EXPL	139	0.03	19	0.01	16	0.01
Ham22full	22	0	1.22	2	0.00	EXPL	19	0.01	15	0.01	15	0.01
Ham52b	52	100	402	112	0.01	EXPL	1119	0.81	19	0.07	19	0.22
Ham52full	52	3	45.03	5	0.00	EXPL	90	0.13	72	0.11	72	0.58 (C)
Ham52order_a	52	16	57.07	97	0.02	EXPL	2203	2.54	189	0.45	76	3.80
Ham52order_b	52	41	117	1	0.00	EXPL	49	0.04	49	0.05	49	0.08
See [17] for details on						NOEXPL	202	0.02	34	0.01	22	0.01
on the benchmarks.						NOEXPL	35	0.01	27	0.01	13	0.74
"full" $\equiv M = \mathcal{N}$						NOEXPL	17579	6.04	1523	0.76	21	4.03
"order" $\equiv$ the nodes						NOEXPL	328	0.12	264	0.12	264	0.59(C)
in M must be visited						NOEXPL	17438	7.93	1409	0.83	407	0.38
in some given order.						NOEXPL	83	0.03	83	0.03	83	0.13

Table 1. Comparison between [17], [10], EXPL, NOEXPL, BPATH, DPBPATH and PATH. (C) indicates when clustering is used.

We clearly see that we solved the benchmarks faster than in [17]. We also see how BPATH and DPBPATH improve the results obtained by PATH, which is the point of having *bounded_path*. We can also see that the explanations reduced the number of failures greatly, specially for the two instances with biggest search space (52b and 52order_a). We do not show EXPL* as they don't improve on EXPL, because the search space is already very small, and EXPL* is more expensive than EXPL.

Although slow, the DPBPATH is still suitable for the Hamiltonian path of 22 nodes. For 52 nodes in such dense graph though, the state space explodes and we absolutely need to cluster.

We also compared against the circuit-based path propagator with explanations presented in [10]. Their propagator is surprisingly fast on these benchmarks and requires little search. This is because their propagator has much better reasoning over the topology of the graph. The topological reasoning of our case is done by the *path* propagator, which is a combination of directed trees (Eq. 1), whereas their propagator makes more inferences based on strongly connected components and starting the path at different nodes. This specific benchmarks are simple in terms of distance (all the arcs have the same weight), but hard in terms of topology, hence the advantage.

The take-away from this experiment is that for graphs that are topologically hard, using our propagator might be a burden whereas using other propagators with strong topological reasoning as [10] might be a better approach.

5.2 Metabolic networks

A metabolic network is a network of molecules and reactions. Biologists use this to understand how some molecules transform into others and cause some behavior in cells. For instance, this helps biologists understand how a protein behaves or how gene expression is regulated. This problem was modeled in [25] by creating a bipartite graph where molecules are in one partition of the nodes, and reactions in another partition. The arcs of the graph link the substrates and products participating in a reaction to the reaction itself.

Here, there is a set of mandatory nodes (because biologists are aware of their existence) and mutually exclusive nodes (corresponding to mutually exclusive reactions). Furthermore, each node is given a weight corresponding to its degree (this is to model highly connected molecules). The objective is to find a pathway from some given substrate to some given product minimizing the total weight of the path, where the weight is the sum of the degrees of the nodes.

Table 2 shows a comparison between our solver and the work in [25], which used the solvers GRASPER and CP(Graph) on an Intel Core 2 Duo 2.16GHz. Here BPATH stands for PATH+BPATH. There is one instance for each size.

	Glycosis				Lysine				Heme			
$\|\mathcal{N}\|$	GRASP.	CP(Graph)	PATH	BPATH	GRASP.	CP(Graph)	PATH	BPATH	GRASP.	CP(Graph)	PATH	BPATH
500	0.28	0.21	**0.05**	0.11	0.36	0.41	**0.06**	0.12	0.22	0.10	**0.05**	0.22
600	0.38	0.31	**0.07**	0.17	0.48	0.44	**0.06**	0.16	0.28	0.12	**0.06**	0.31
700	0.45	0.35	**0.19**	0.22	0.47	0.75	**0.08**	0.25	0.36	0.16	**0.08**	0.46
800	0.53	0.50	**0.24**	0.29	0.53	1.00	**0.12**	0.37	0.41	0.19	**0.11**	0.55
900	0.64	0.68	**0.15**	0.39	0.57	1.29	**0.16**	0.4	0.51	0.27	**0.15**	0.73
1000	0.77	0.84	**0.18**	0.51	0.60	1.37	**0.18**	0.46	0.62	0.32	**0.18**	0.95
1100	0.91	1.00	**0.17**	0.71	0.73	1.29	**0.19**	0.64	0.65	0.33	**0.32**	1.08
1200	0.96	1.08	**0.20**	0.75	0.86	2.23	**0.23**	0.79	0.80	0.41	**0.21**	3.62
1300	1.03	1.21	**0.81**	0.84	0.99	2.50	**0.28**	1.02	0.94	0.47	**0.4**	1.81
1400	1.23	1.56	**0.71**	1.05	1.12	2.84	**0.30**	1.17	1.11	0.51	**0.4**	2.1
1500	1.40	1.85	**1.25**	1.28	1.25	2.92	**0.39**	1.33	1.14	**0.52**	0.94	2.09
1600	1.67	2.14	**0.75**	1.49	1.30	2.97	**0.43**	1.36	1.35	**0.61**	0.74	2.55
1700	1.93	2.40	**0.82**	1.77	1.41	3.03	**0.67**	1.44	1.57	0.69	**0.4**	3.08
1800	2.11	2.77	**1.01**	2.01	1.53	3.69	**0.49**	1.69	1.72	0.77	**0.45**	3.69
1900	2.27	3.02	**1.19**	2.21	1.75	3.93	**0.60**	1.95	1.96	0.84	**0.48**	6.21
2000	2.40	3.14	**1.33**	2.3	1.96	2.18	**0.64**	2.39	2.18	0.91	**0.51**	4.86

Table 2. Solving metabolic pathways in real-world networks (same strategy as [25]).

The results show that BPath slows Path down. We interpret this as the effect of the overhead of *bounded_path*. Indeed, the instances are solved so quickly by Path that BPath has little to improve on. We also ran the same experiments with the VSIDS [14] search strategy. The times were very similar to those in Table 2 for Path, but 3 benchmarks (1200, 1300 and 1900 nodes for Heme) were much slower (around 30 seconds). We tested the BPath version on those three instances and noticed a big speedup (between 5 and 15 times faster). Nonetheless, note how BPath is still faster than GRASPER and CP(Graph) in two thirds of the tests. From this we conclude that bounding is only worthwhile if the instances are hard to solve (i.e. there is a big search space to explore).

5.3 Task constrained shortest path

In this problem, we are required to perform a set of tasks along a path. A task can be done at different nodes, and visiting a node where some task can be performed is enough, we do not need to visit more than one. As an example, consider on the drive home withdrawing money from an ATM, going to a carwash and buying some groceries. Any ATM, supermarket or carwash on the path is sufficient. This problem was studied in [18, 19] using dynamic programming only.

In [10], the authors used a circuit-based path propagator to solve a similar problem (minimizing the longest arc). We compare our implementation against theirs using the same instances (500 graphs of 20 nodes each) with the objective of minimizing the total length of the path. The aim of this experiment is to see if BPath and DPBPath can also improve the circuit-based path propagator.

In this experiment we compare the best runtimes of both approaches, even if they use two different strategies. Our best search strategy is `smallest` (i.e. branching on the *succ* variable with smallest domain) and their best search strategy is `first_fail` on the *succ* variables. Additionally, we combine our bounding propagator with theirs to see the benefits.

	Expl* (all use `smallest`)				Expl				NoExpl			
Version	Conflicts	Nodes	Time(s)	Opt(s)	Conflicts	Nodes	Time(s)	Opt(s)	Conflicts	Nodes	Time(s)	Opt(s)
[10]	48790	54254	3.18	2.14	48790	54254	3.18	2.14	308888	619304	7.95	6.48
[10]+BPath	18303	19883	2.90	1.49	27050	29995	3.70	2.84	174329	350327	15.99	13.67
[10]+DPBPath	636	1133	2.09	1.86	4933	6228	1.68	1.36	31256	188278	4.47	3.75
Path	26488	28801	7.05	2.27	26488	28801	7.05	2.27	200773	402943	32.63	9.81
Path+BPath	13175	14787	3.63	1.30	15238	16868	4.07	1.37	76701	156208	16.20	5.51
Path+DPBPath	**54**	**456**	**0.53**	**0.36**	**221**	**648**	**1.31**	**0.44**	**381**	**1253**	**2.96**	**1.14**

Table 3. Benefit from BPath & DPBPath with both Path and [10]. Geometric average over 500 instances of 20 nodes.

Table 3 gives the results, also showing the time (Opt) to find (but not prove) the optimal solution. The Path constraint finds optimal solutions very fast, but takes time to prove optimality. On the other hand, the version from [10] is superior in both these aspects. Adding BPath and DPBPath improves both these versions. The circuit based propagator does 89% less search when combined with DPBPath (in its fastest version, using Expl), and Path does 98% less search when combined with DPBPath (using Expl*). This shows how *bounded_path* with explanations can be used in combination with both tree-based and circuit-based paths to enhance propagation.

5.4 Profitable Tourist path

We introduce here a new problem (as far as we are aware) similar to the prize collecting TSP. Imagine you need to do a long layover during a trip and change airports. You might be interested in visiting the city while waiting for your connection flight. In this problem, we model every point of interest (POI) of a city with a minimum visit time (i.e. the least amount of time that a visit to some POI is worthwhile) and a profit (i.e. how much a person enjoys visiting some POI). The path can contain a node without necessarily visiting the corresponding POI, but in order to visit a POI the path must contain the corresponding node and spend the minimum visit time. The objective of the problem is to find the path with most profit such that the total time is less than a certain bound (i.e. the time we have available between connections). The total time is the cost of the path plus the time spent at each POI (either 0 or the minimum visit time).

We created two benchmarks, based on New York City (14 nodes, from LGA Airport to JFK Airport) and London (12 nodes, from Heathrow Airport to Liverpool Street Station). We added two side constraints: for London, we require that the visit to the Tower Bridge (if it happens) takes place between two narrow time frames (which would correspond to times where the bridge opens to let ships go through); for NYC, the ferry to Liberty Island leaves every hour and so there might be a waiting time added to the total time (if the visit happens).

We used Expl on all the tests to study the benefits of bounding. The results are in Table 4. Clearly, DPBPath and BPath largely improve Path for this problem. Again, there was no need to cluster or interrupt explanations.

	New York City (14 POI)			London (12 POI)		
Version	Fails	Nodes	Time(s)	Fails	Nodes	Time(s)
Path	≥5030898	>5034046	>3600.00	236010	237263	60.14
Path+BPath (Expl)	390985	391746	379.19	24061	25190	13.86
Path+DPBPath (Expl)	44015	44606	48.82	10645	11866	2.89
Path+BPath (Expl*)	360945	361971	350.26	18546	19881	8.78
Path+DPBPath (Expl*)	**2062**	**2690**	**37.45**	**224**	**670**	**0.16**

Table 4. Profitable tourist path. Search: `smallest` on *succ* variables.

Without explanations, though, NYC takes 1598s using DPBPath and London takes 13s, making them substantially slower than with explanations.

6 Conclusion

In this paper we have improved the *bounded_path* propagator by adding a new propagation technique that is clearly superior. Both propagations are enhanced by our two new versions of explanations. First, a fast way of computing valid but not minimal explanations is given. We then provided another version that generates more reusable explanations.

We have shown how combining *bounded_path* with path propagators (composition of directed trees or circuit-based) improves their performance, reaching the state of the art in bounded path propagation.

References

1. Beldiceanu, N., Flener, P., Lorca, X.: The tree constraint. In: Integration of AI and OR Techniques in constraint programming for combinatorial optimization problems, pp. 64–78. Springer (2005)
2. Beldiceanu, N., Katriel, I., Lorca, X.: Undirected forest constraints. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 29–43. Springer (2006)
3. Chu, G.G.: Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne (2011)
4. De Uña, D.: Directed graph benchmarks (2015), http://people.eng.unimelb.edu.au/pstuckey/bounded_path/bounded_path.zip
5. De Uña, D., Gange, G., Schachte, P., Stuckey, P.J.: Steiner tree problems with side constraints using constraint programming. In: Proceedings of the Thertieth AAAI Conference on Artificial Intelligence. p. to appear. AAAI Press (2016)
6. Dooms, G., Deville, Y., Dupont, P.: CP(Graph): Introducing a graph computation domain in constraint programming. In: van Beek, P. (ed.) Principles and Practice of Constraint Programming - CP 2005, Lecture Notes in Computer Science, vol. 3709, pp. 211–225. Springer Berlin Heidelberg (2005)
7. Eades, P., Wormald, N.C.: Edge crossings in drawings of bipartite graphs. Algorithmica 11(4), 379–403 (1994)
8. Fages, J.G.: Exploitation de structures de graphe en programmation par contraintes. Ph.D. thesis, École de Mines de Nantes (2014)
9. Fages, J.G., Lorca, X.: Revisiting the tree constraint. In: Principles and Practice of Constraint Programming–CP 2011, pp. 271–285. Springer (2011)
10. Francis, K.G., Stuckey, P.J.: Explaining circuit propagation. Constraints 19(1), 1–29 (2014)
11. Hartigan, J.A., Wong, M.A.: Algorithm as 136: A k-means clustering algorithm. Journal of the Royal Statistical Society. Series C (Applied Statistics) 28(1), 100–108 (1979)
12. Hopcroft, J.E., Ullman, J.D., Aho, A.V.: Data structures and algorithms, vol. 175. Addison-Wesley Boston, MA, USA: (1983)
13. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems (TOPLAS) 1(1), 121–141 (1979)
14. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th annual Design Automation Conference. pp. 530–535. ACM (2001)
15. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: Procs. of CP2007. LNCS, vol. 4741, pp. 529–543. Springer-Verlag (2007)
16. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009), http://dx.doi.org/10.1007/s10601-008-9064-x
17. Quesada, L., Van Roy, P., Deville, Y., Collet, R.: Using dominators for solving constrained path problems. In: Practical Aspects of Declarative Languages, pp. 73–87. Springer (2006)
18. Rice, M.N., Tsotras, V.J.: Engineering generalized shortest path queries. In: Data Engineering (ICDE), 2013 IEEE 29th International Conference on. pp. 949–960. IEEE (2013)

19. Rice, M.N., Tsotras, V.J.: Parameterized algorithms for generalized traveling salesman problems in road networks. In: Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. pp. 114–123. ACM (2013)
20. Sellmann, M.: Cost-based filtering for shorter path constraints. In: Principles and Practice of Constraint Programming–CP 2003. pp. 694–708. Springer (2003)
21. Sellmann, M., Gellermann, T., Wright, R.: Cost-based filtering for shorter path constraints. Constraints 12(2), 207–238 (2006), http://dx.doi.org/10.1007/s10601-006-9006-4
22. Silver, D.: Cooperative pathfinding. In: AIIDE. pp. 117–122 (2005)
23. Storandt, S.: Quick and energy-efficient routes: computing constrained shortest paths for electric vehicles. In: Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science. pp. 20–25. ACM (2012)
24. Storandt, S.: Route planning for bicycles-exact constrained shortest paths made practical via contraction hierarchy. In: ICAPS. vol. 4, p. 46 (2012)
25. Viegas, R.D., Azevedo, F.: Lazy constraint imposing for improving the path constraint. Electronic Notes in Theoretical Computer Science 253(4), 113–128 (2009)