An Algorithm for Single-Source Shortest Paths Enumeration in Parameterized Weighted Graphs

Bastien Sérée, Loïg Jezequel, Didier Lime

▶ **To cite this version:**

Bastien Sérée, Loïg Jezequel, Didier Lime. An Algorithm for Single-Source Shortest Paths Enumeration in Parameterized Weighted Graphs. Language and Automata Theory and Applications, Sep 2021, Milan, Italy. pp.279-290, ⟨10.1007/978-3-030-68195-1_22⟩. ⟨hal-03561972⟩

HAL Id: hal-03561972

https://hal.science/hal-03561972v1

Submitted on 8 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

An algorithm for single-source shortest paths enumeration in parameterized weighted graphs

Bastien Sérée[1,3], Loïg Jezequel[2,3], and Didier Lime[1,3]

[1] École Centrale de Nantes
[2] Université de Nantes
[3] LS2N, UMR CNRS 6004, Nantes, France
firstname.lastname@ls2n.fr

Abstract. We consider weighted graphs with parameterized weights and we propose an algorithm that, given such a graph and a source node, builds a collection of trees, each one describing the shortest paths from the source to all the other nodes of the graph for a particular zone of the parameter space. Moreover, the union of these zones covers the full parameter space: given any valuation of the parameters, one of the trees gives the shortest paths from the source to all the other nodes of the graph when the weights are computed using this valuation.

Keywords: Shortest paths, weighted graphs, parameterized graphs

1 Introduction

For many real-world systems and problems there are natural discrete-event abstractions, which can be modelled by a graph or a derived formalism. In many cases, we can also identify resources that need to be optimized (distance, memory, energy, time, etc.) and then (extensions of) weighted graphs are a formalism of choice.

One of the most basic problems is to find optimal paths, for which the accumulated weight (also often called cost) is minimal.

When addressing systems that are not well-known, maybe because we are in the early phases of a design process, one way to cope with this uncertainty is to use parameters for the weights. The interesting problems are parameter synthesis problems, in which one tries to find the values of parameters such that some path is optimal, or such that a target vertex can be reached within a given bound on the accumulated weight, etc.

Surprisingly, those problems have not been studied in detail for the setting for which cost themselves are parameters. Parametric timed automata (PTA) [1] allow clocks to be tested against parameters, which may allow simulation of parametric weights to some extent. The optimal time reachability problem has recently been studied for PTA [2]. Similarly, the bounded-cost reachability problem has been addressed for a formalism related to PTA called parametric time Petri nets in [7]. In [3], the authors do extend PTA with parametric weights on edges, but the topology of the systems considered is always that of a tree.

Much differently, the parametric one-counter machines of [4] should allow one to model a parametric cost, however, the parameter synthesis problems are not addressed in that article.

Finally, in [6] the authors consider graphs in which weights are expressed as a function of a single parameter. They partition the real numbers in a finite way, such that for two parameter values in a given partition, the optimal paths from a given source vertex to all other vertices are the same, and exhibit the corresponding trees those paths form. Such parameteric graphs, with a single parameter, were later studied in [8] (directly improving [6]), and in [5] for example. Extending the results of [6] to parameterized graphs with multiple parameters makes the problem more complex since one needs to partition the n-dimensional real space (where n is the number of parameters). This is the subject of our work. To the best of our knowledge, this has not been done prior to this paper. The algorithm that we propose is exponential in the number of vertices and is polynomial in the number of edges, and in the number of parameters of the considered graph.

This article is organised as follows: in Section 2 we introduce the basic notations and definitions; in Section 3 we informally present our algorithm on a comprehensive example; in Section 4, we give the algorithm together with the associated proofs of correctness, completeness, and termination, as well as the complexity; in Section 5 we conclude.

2 Definitions

For all $(a, b, i) \in \mathbb{N}$, we denote by $i \in [\![a, b]\!]$ any integer i such that $a \leqslant i \leqslant b$.

2.1 Parametric graphs

Definition 1 (Parametric graph). *A* parametric graph *with n parameters is a tuple $G = (V, E, f, (\lambda_i)_1^n, (\Lambda_i)_1^n)$ where V is a set of* vertices*, $E \subseteq V \times V$ is a set of* edges*, $f : E \to \mathbb{R}$ is a non-parametric cost function, and for every $i \in [\![1, n]\!]$, λ_i is a parameter, and $\Lambda_i : E \to \{0, 1\}$ is a parametric cost function.*

The parameters can take any value in $\mathbb{R}$. With a slight abuse of notations, we denote by λ_i not only the parameters but also their values. We also write $\overrightarrow{\lambda} = (\lambda_1 \dots \lambda_n)$. All the following definitions are written for a parametric graph $G = (V, E, f, (\lambda_i)_1^n, (\Lambda_i)_1^n)$.

Definition 2 (Edge cost). *The* cost *of an edge $e \in E$ is:*

$$c(e) = f(e) - \sum_{i=1}^{n} \lambda_i \Lambda_i(e).$$

A *Path* p in G is a sequence of edges $p = e_0 e_1 e_2 \dots e_k$ such that $\forall i \in [\![0, k]\!], e_i = (v_i, v_{i+1})$. In such a path v_0 is the *initial vertex* and v_{k+1} is the *terminal vertex*. The integer k is called the *length* of p.

Definition 3 (Path cost). *Let $p = e_0 e_1 \dots e_k$ be a path of G, the* cost *of p is:* $c(p, \overrightarrow{\lambda}) = \sum_{j=0}^{k} c(e_j) = \sum_{j=0}^{k} \left(f(e_j) - \sum_{i=1}^{n} \lambda_i \Lambda_i(e_j) \right) = \sum_{j=0}^{k} f(e_j) - \sum_{i=1}^{n} \lambda_i \sum_{j=0}^{k} \Lambda_i(e_j).$

We call *path of minimal cost* for $\overrightarrow{\lambda} \in \mathbb{R}^n$ a path of G such that there is no other path of G with the same initial and terminal vertices, and with a strictly lower path cost for $\overrightarrow{\lambda}$.

Finally, we call a *cycle* a path where the initial vertex and the terminal vertex are the same. And we call a *negative cycle* for $\overrightarrow{\lambda} \in \mathbb{R}^n$ a cycle with a negative cost for $\overrightarrow{\lambda}$.

2.2 Trees over parametric graphs

In the following we suppose that there is a distinguished vertex s in the graph G, from which we will search for paths of minimal cost toward all other vertices. As we are interested in paths from s to each vertex, in the following we assume that the graphs we consider are such that these paths exist.

Definition 4 (Tree). *We define a* tree *of G rooted at s (or with source s) as any tuple $T = (V_T, E_T, f_T, (\lambda_{T,i})_1^n, (\Lambda_{T,i})_1^n)$ such that $V_T = V$, $E_T \subseteq E$ is such that for all $v \in V \setminus \{s\}$, $|\{(u,v) : (u,v) \in E_T\}| = 1$, $\{(u,s) : (u,s) \in E_T\} = \emptyset$ and there is no cycle in T, $f_T = f_{|E_T}$, $\forall i, \lambda_{T,i} = \lambda_i$, and $\forall i, \Lambda_{T,i} = \Lambda_{T,i|E_T}$.*

Notice that, in such a tree, there are always $|V| - 1$ edges. Moreover, there is exactly one path from s to each vertex.

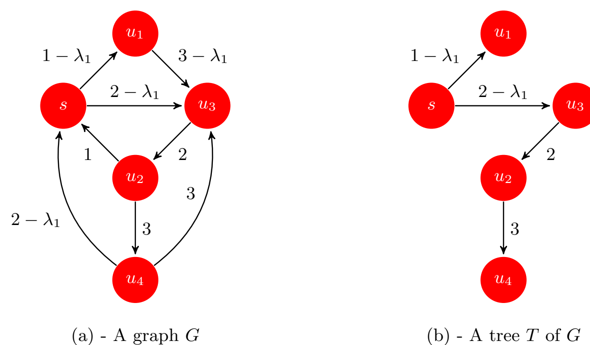

Fig. 1. A graph G and an example of a tree T of G.

In Figure 1, we have represented an example of a graph with one parameter, λ_1 (a) and a tree T of G rooted at s (b).

In all the following definitions T is a tree rooted at s.

Definition 5 (Distance). *Let v be a vertex of G, the* distance $d(T, v, \overrightarrow{\lambda})$ *is the cost of the unique path from s to v in T.*

Definition 6 (Partial distances). *Let v be a vertex of G and let $e_0 e_1 \cdots e_k$ be the unique path from s to v in T. The* partial non-parametric distance *between s and v in T is*

$$d_f(T, v) = \sum_{j=0}^{k} f(e_j).$$

The partial parametric distances *between s and v are the*

$$d_{\Lambda_i}(T, v) = \sum_{j=0}^{k} \Lambda_i(e_j),$$

for all $i \in [\![1, n]\!]$.

Notice that $d(T, v, \overrightarrow{\lambda}) = d_f(T, v) - \sum_{i=1}^{n} \lambda_i d_{\Lambda_i}(T, v)$.

Definition 7 (Tree of minimal distances). *We say that T is a* tree of minimal distances *for $\overrightarrow{\lambda} \in \mathbb{R}^n$ if for all $v \in V$, the unique path from s to v in T is a path of minimal cost $\overrightarrow{\lambda}$ from s to v in G.*

Moreover, if $S \subseteq \mathbb{R}^n$, T is a tree of minimal distances for all $\overrightarrow{\lambda} \in S$, we say that T is a tree of minimal distances on S.

Definition 8 (Neighbour). *Let $e = (u, v)$ be an edge in E, the* neighbour *of T generated by e is the tuple $N(T, e) = (V_N, E_N, f_N, (\lambda_{N,i})_1^n, (\Lambda_{N,i})_1^n)$ where:*

- $V_N = V$
- $E_N = (E_T \setminus \{(u', v) : (u', v) \in E_T\}) \cup \{e\}$, $f_N = f_{|E_N}$
- $\forall i, \lambda_{N,i} = \lambda_i$
- $\forall i, \Lambda_{N,i} = \Lambda_{i|E_N}$

In other words, $N(T, e)$ is obtained from T by deleting the only edge $e' = (u', v')$ such that $v' = v$ and adding e.

Notice that for all $e \in E_T$, $N(T, e) = T$ and that, for $(u, v) \in E \setminus E_T$, an edge (u', v) does not necessarily exist in T, since T is rooted at s. Indeed an edge e such that $e = (u, s)$ will not be in T (as T is a tree rooted at s) and can generate neighbours as any edge. In particular, this means that the neighbour of a tree is not necessarily a tree.

As an example, consider G from Figure 1 and let $e = (s, u_3)$, $e_1 = (u_1, u_3)$ and $e_2 = (u_4, u_3)$. Figure 1 (b) represents a tree T rooted at s, Figure 2 (a) represents $N(T, e_1)$, which is also a tree, and Figure 2 (b) represents $N(T, e_2)$, illustrating the fact that a neighbour of a tree is not necessarily a tree itself.

The following proposition specifies in which case a neighbour of a tree is actually a tree. Its proof is omitted due to space constraints

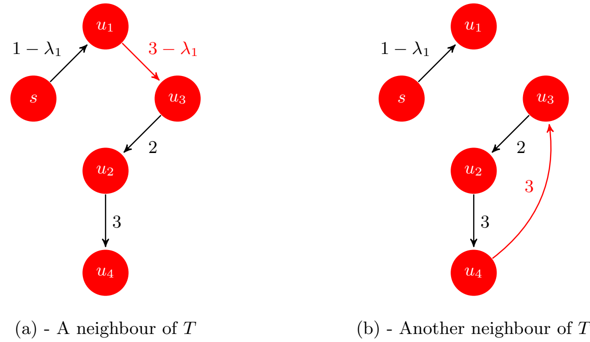

Fig. 2. Some neighbours of the tree T of Figure 1.

Proposition 1. *Let $e = (u, v) \in E$ be an edge, $N(T, e)$ is a tree if and only if v is not on the unique path from s to u in T.*

Before giving other properties of trees and their neighbours, we have to take a few notations. For an edge $e = (u, v) \in E$ we note:
$\Delta_f(T, e) = d_f(T, u) + f(e) - d_f(T, v)$,
$\forall i \in [\![1, n]\!], \Delta_{\Lambda_i}(T, e) = d_{\Lambda_i}(T, u) + \Lambda_i(e) - d_{\Lambda_i}(T, v)$.

These deltas represent the differences in distance from s to v between T and its neighbour generated by e. Δ_f is the difference in the non-parametric part of the distance. Each Δ_{Λ_i} represents the difference of the number of occurrences of the corresponding parameter λ_i. Notice that for any $e \in E_T$, one always has $\Delta_{\Lambda_i}(T, e) = 0$.

For example in Figure 1, with the same notations as before, $\Delta_f(T, e_1) = d_f(T, u_1) + f(e_1) - d_f(T, u_3) = 1 + 3 - 2 = 2$, and $\Delta_{\Lambda_1}(T, e_1) = d_{\Lambda_1}(T, u_1) + \Lambda_1(e_1) - d_{\Lambda_1}(T, u_3) = 1 + 1 - 1$.

Proposition 2. *Let $e = (u, v)$ be an edge not in T. If $N(T, e)$ is a tree then $\forall w \in V, d(T, w, \overrightarrow{\lambda}) = d(N(T, e), w, \overrightarrow{\lambda})$ if and only if $\Delta_f(T, e) - \sum_{i=1}^{n} \lambda_i \Delta_{\Lambda_i}(T, e) = 0$.*

Proposition 3. *Let $e = (u, v)$ be an edge. If $N(T, e)$ is not a tree then $N(T, e)$ has a cycle of cost 0 if and only if $\overrightarrow{\lambda}$ is such that $\Delta_f(T, e) - \sum_{i=1}^{n} \lambda_i \Delta_{\Lambda_i}(T, e) = 0$.*

The proofs of these propositions are omitted due to space constraints.

2.3 Constraints and zones associated to trees

Definition 9 (Constraint). *Let $e \in E \setminus E_T$ be an edge, the* constraint *associated with e is*

$$C_{T,e} = \Delta_f(T, e) - \sum_{i=1}^{n} \lambda_i \Delta_{\Lambda_i}(T, e).$$

Definition 10 (Zone). *Let $E_c \subseteq E \setminus E_T$ be a set of edges such that $\forall e_c \in E_c, \exists i \in [\![1, n]\!], \Delta_{\Lambda_i}(T, e_c) > 0$. The* zone defined by the constraints *associated to the edges in E_c is the set S such that: $\overrightarrow{\lambda} \in S$ if and only if $\forall e_c \in E_c, \Delta_f(T, e_c) - \sum_{i=1}^{n} \lambda_i \Delta_{\Lambda_i}(T, e_c) \geq 0$.*

We can notice that zones are convex by construction.

Definition 11 (Active constraint). *Let $E_c \subseteq E \setminus E_T$ be a set of edges such that $\forall e_c \in E_c, \exists i \in [\![1, n]\!], \Delta_{\Lambda_i}(T, e_c) > 0$. Let $e_c \in E_c$ be an edge. Let S be the zone defined by the constraints associated to the edges in E_c. Let $S_{/e_c}$ be the zone defined by the constraints associated to the edges in $E_c \setminus \{e_c\}$. The constraint C_{T,e_c} is said to be active if and only if $S_{/e_c} \neq S$.*

3 Presentation of our algorithm for minimal distances

In section 3, we propose an algorithm which, given a graph G returns a list of trees and a list of disjoint zones. Each tree T is associated with one zone S, such that T is a tree of minimal distances on S. Moreover, the union of the returned zones is the zone of $\mathbb{R}^n$ for which there is no negative cycle in G. In this section, we demonstrate how this algorithm works on the example of G_{ex}, a parametric graph with two parameters presented in Figure 3.

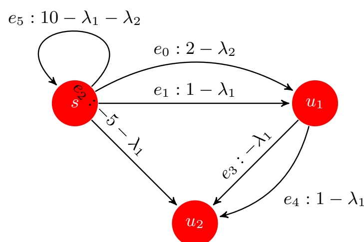

Fig. 3. Graph G_{ex}

The first step of the algorithm consists in finding the zone $Z_{noCycle}$ for which the concept of minimal distances makes sense, that is the values of λ_1 and λ_2 for which there is no negative cycle. Here, the only cycle is $e_5 e_5 e_5 \ldots$. So, the only possible negative cycle is when the cost of e_5 is negative, therefore, when $10 - \lambda_1 - \lambda_2 < 0$. Hence, we have $Z_{noCycle} = \{(\lambda_1, \lambda_2) \in \mathbb{R}^2 \ : \ \lambda_1 + \lambda_2 \leqslant 10\}$.

The goal will be to cover $Z_{noCycle}$ with zones associated to trees of minimal distances. For that the algorithm enumerates zones, associated with trees, going from one zone to another by considering the neighbours of the associated tree. The algorithm begins by computing a first tree T_0. This tree is a tree of minimal distances for a particular pair $(\lambda_{1,init}, \lambda_{2,init})$, where $-\lambda_{1,init} = -\lambda_{2,init} = 1 +$

$\sum_{e \in E} |f(e)|$. Here we have $(\lambda_{1,init}, \lambda_{2,init}) = (-20, -20)$ and T_0 is represented in Figure 4.

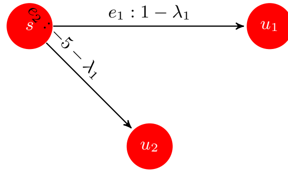

Fig. 4. T_0

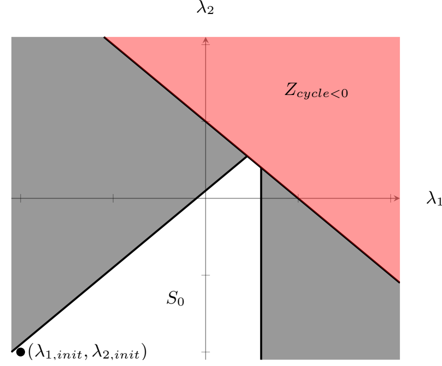

Fig. 5. $(\lambda_{1,init}, \lambda_{2,init})$ and S_0

Now look at the constraints associated with the neighbours of T_0 to characterize the associated zone S_0. The active constraints also tell which trees will be considered next. Here, the edges that generate neighbours are e_0, e_3, e_4 and e_5. So, one has to look at the constraints C_0, C_3, C_4 and C_5 associated respectively with $N(T_0, e_0)$, $N(T_0, e_3)$, $N(T_0, e_4)$ and $N(T_0, e_5)$.

We have $C_0 : 1 + \lambda_1 - \lambda_2 = 0$, $C_3 : 6 - \lambda_1 = 0$, $C_4 : 7 - \lambda_1 = 0$ and $C_5 : 10 - \lambda_1 - \lambda_2 = 0$. Among these constraints, C_0, C_3 and C_5 are active. Thus, we take $S_0 = \{(\lambda_1, \lambda_2) \in \mathbb{R}^2 \ : \ 1 + \lambda_1 - \lambda_2 \geqslant 0, 6 - \lambda_1 \geqslant 0, 10 - \lambda_1 - \lambda_2 \geqslant 0\}$, as represented in Figure 5.

As $N(T_0, e_0)$ and $N(T_0, e_3)$ are associated with active constraints and have not been considered yet, they are added in a list of trees to be considered later, called $listToDo$. $N(T_0, e_5)$ is not added to $listToDo$ because it is not a tree. The pair (T_0, S_0) is added to a list called $listExplored$. This list will be returned by the algorithm at the end of its computation.

From now on, the algorithm iteratively considers the trees of $listToDo$. Assume, for example, that it begins with $N(T_0, e_0) = T_1$, represented in Figure 6. The edges that generate neighbours are e_1, e_3, e_4 and e_5. The associated constraints are $C_1 : -1 - \lambda_1 + \lambda_2 = 0$, $C_3 : 7 - \lambda_2 = 0$, $C_4 : 8 - \lambda_2 = 0$ and $C_5 : 10 - \lambda_1 - \lambda_2 = 0$. Among them, C_1, C_3, and C_5 are active. From that we can define S_1, represented in Figure 7.. As $N(T_1, e_1) = T_0$ has already been considered, only $N(T_1, e_3)$ is added to $listToDo$ (recall that $N(T_1, e_5)$ is not a tree). (T_1, S_1) is added to $listExplored$.

At the moment, $listToDo = \{N(T_0, e_3), N(T_1, e_3)\}$ and $listExplored = \{(T_0, S_0), (T_1, S_1)\}$. Assume that the algorithm considers $N(T_0, e_3) = T_2$ next. This tree is represented in Figure 8. The constraints used to define S_2 are the ones associated with e_0, e_2 and e_5. e_4 is not considered because $\Delta_{\Lambda_1} T_2, e_4 =$

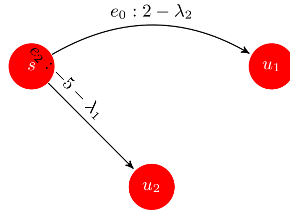

Fig. 6. T_1

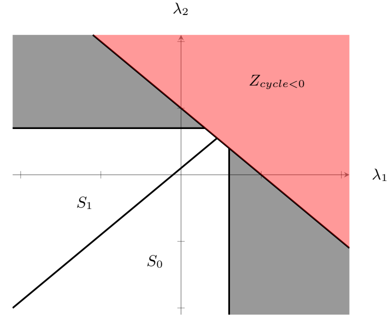

Fig. 7. S_0 and S_1

$\Delta_{\Lambda_2} T_2, e_4 = 0$. All constraints are active. The zone S_2 is represented in Figure 9. No tree is added to *listToDo* as $N(T_2, e_0) = N(T_1, e_3)$, which is already in *listToDo* and $N(T_2, e_2) = T_0$, which has already been considered. (T_2, S_2) is added to *listExplored*.

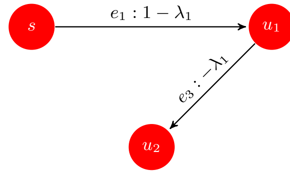

Fig. 8. T_2

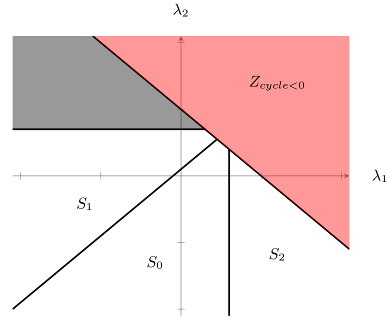

Fig. 9. S_0, S_1 and S_2

Then, it remains to consider $N(T_1, e_3) = T_3$, represented in Figure 10. Three constraints are considered, this time associated with e_1, e_2 and e_5. All these constraints are active. The obtained S_3 is represented in Figure 11. No tree is added to *listToDo* because all neighbours either have already been considered or are not trees. (T_3, S_3) is added to *listExplored*.

At that point, the algorithm terminates because $S_0 \cup S_1 \cup S_2 \cup S_3 = Z_{noCycle}$. It returns *listExplored*. Note that *listToDo* is empty.

4 Formal presentation of our algorithm

In this section, we formalize the algorithm that has been presented on the example in the previous section. We then prove that this algorithm is correct.

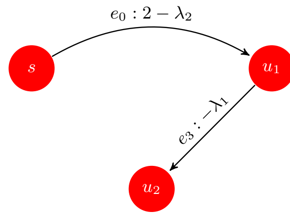

Fig. 10. T_3

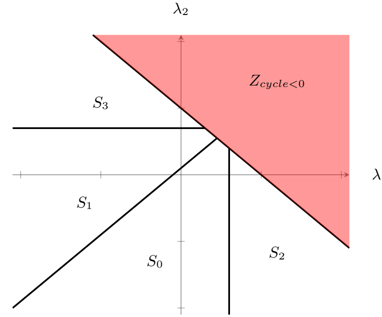

Fig. 11. S_0, S_1, S_2, and S_3

4.1 The algorithm

Algorithm 1 is an algorithm that, given a parametric graph G and a vertex s, returns a list of pairs (T, S) such that every T is a tree of minimal distances on S and that the union of all S in the list is equal to the zone such that there is no negative cycle in G. In this algorithm, $\overrightarrow{\lambda}_{init}$ is the n-components vector so that each component is equal to $-\left(1 + \sum_{e \in E} |f(e)|\right)$.

Algorithm 1 trees of minimal distances of $G = (V, E, f, (\lambda_i)_1^n, (\Lambda_i)_1^n)$

1: **let** $listExplored = \emptyset$.
2: **let** T_0 be a tree of minimal distances for $\overrightarrow{\lambda}_{init}$
3: **let** $listToDo = \{T_0\}$
4: **while** $listToDo \neq \emptyset$ **do**
5: **choose** a tree T in $listToDo$ (and delete it from $listToDo$)
6: **let** *Neighbours* be the set of all possible edges e such that $\exists i \in [\![1, n]\!]$, $\Delta_{\Lambda_i}(T, e) > 0$.
7: **let** S be the zone defined by the constraints associated to the edges of *Neighbours*
8: **let** *Active* be the subset of *Neighbours* containing the edges giving active constraints
9: **for** each edge e in *Active* such that $N(T, e)$ is a tree **do**
10: **let** $T_N = N(T, e)$
11: **if** $T_N \notin listToDo$ **and** $T_N \notin listExplored$ **then**
12: **add** T_N to $listToDo$
13: **end if**
14: **end for**
15: **add** (T, S) to $listExplored$
16: **end while**
17: **return** $listExplored$

In the following, we will refer to the zone where there are no negative cycles as $Z_{noCycle}$ and the union of the zones in *listExplored* as $Z_{explored}$.

Algorithm 1 starts by computing T_0 (the tree from which the space will be explored) as a tree of minimal distances for $\overrightarrow{\lambda}_{init}$. This initial tree is chosen to ensure that the algorithm will cover $Z_{noCycle}$ by only exploring toward greater λ_i, $i \in [\![1, n]\!]$ as proven in the next section. T_0 is added to *listToDo* which is the list of the trees the algorithm needs to consider.

For each tree T, the algorithm begins by enumerating all the edges e that can generate a neighbour. From the associated constraints it characterizes S, a zone where T is a tree of minimal distances. Then the algorithm does two things: (1) it adds to *listToDo* all the neighbour trees that have not been considered yet (those that are not already in *listToDo* or in *listExplored*) and (2) it adds the new result to the returned list by adding T and S to *listExplored*. We can notice that the zones in *listExplored* are disjoint by construction.

When *listToDo* is empty the algorithm returns *listExplored*.

Notice that when there is only one parameter this algorithm is equivalent to the algorithm presented in [6] as constraints have a better form that make computing active constraints equivalent to look for a maximum. So it is possible to do a more efficient exploration because of the fact that there is only one dimension.

4.2 Proof of the algorithm

A first Lemma expresses that – with the value that has been chosen for $\overrightarrow{\lambda_{init}}$ – it is not necessary to consider all the neighbours of each tree in the main loop of the algorithm. It is sufficient to consider neighbours with increasing number of occurrences of (at least) one parameter, as enforced by line 6 of Algorithm 1.

Lemma 1. *Let S_0 be the first zone computed by the algorithm. $\forall \overrightarrow{\lambda} \in Z_{noCycle}$, $\exists \overrightarrow{\lambda'} \in \mathbb{R}^n$ and $\overrightarrow{\lambda_{S_0}} \in S_0$ such that (1) $\exists i \in [\![1, n]\!], \lambda'_i \geqslant \lambda_{i,S_0}$ and (2) for all trees T of minimal distances for $\overrightarrow{\lambda'}$, T is also a tree of minimal distances for $\overrightarrow{\lambda}$.*

Proof is omitted due to space constraints

A second lemma exhibits a loop invariant that will be instrumental in showing the correctness of the results of the algorithm.

Lemma 2 (loop invariant). *Let $Z_{notExplored} = Z_{noCycle} \setminus Z_{explored}$, at each loop of the while loop we have:*

1: $Z_{explored} \cup Z_{notExplored} = Z_{noCycle}$ (in particular there is no $\overrightarrow{\lambda} \in Z_{explored}$ such that there is a negative cycle in G for $\overrightarrow{\lambda}$)

2: for all $(T, S) \in$ ListExplored, T is a tree of minimal distances for all $\overrightarrow{\lambda} \in S$.

Proof is omitted due to space constraints

Building on the two above lemmas, we give our main theorem, that states that the proposed algorithm terminates and returns correct results.

Theorem 1. *The algorithm terminates and returns ListReturned such that for all* $(T,S) \in ListReturned$, *T is a tree of minimal length for all* $\overrightarrow{\lambda} \in S$, $\bigcup_{(T,S)\in ListReturned} S = Z_{noCycle}$ *and for all* (T,S) *and* (T',S') *in ListReturned such that* $S \neq S'$, $S \cap S' = \emptyset$.

Proof. If the algorithm does not terminate it means that there is an infinite loop.

As we consider a new tree in each loop and there is a finite number of (possible) trees it is impossible to have an infinite loop so the algorithm does terminate. We also need that $Z_{explored} = Z_{noCycle}$ at the end of the algorithm, i.e. when *listToDo* is empty. If it is not the case it means that the algorithm has missed one or more zones and this is only possible if there is some zones such that $\overrightarrow{\lambda} \in \mathbb{R}^n$, $\nexists \overrightarrow{\lambda'} \in \mathbb{R}^n$ and $\overrightarrow{\lambda_{S_0}} \in S_0$ such that $\exists i \in [\![1,n]\!], \lambda'_i \geqslant \lambda_{i,S_0}$. Which is impossible by Lemma 1 so the algorithm terminates and we have $\bigcup_{(T,S)\in ListReturned} S = Z_{noCycle}$.

For each $(T,S) \in ListReturned$ we also have that T is a tree of minimal length for all $\overrightarrow{\lambda} \in S$ and $\bigcup_{(T,S)\in ListReturned} S = Z_{noCycle}$ by the loop invariant.

4.3 Complexity

We conclude the presentation of Algorithm 1 by giving its worst-case complexity.

Theorem 2. *The worst case complexity of Algorithm 1 is exponential in the number* $|V|$ *of vertices and is polynomial in the number* $|E|$ *of edges and in the number* n *of parameters. Moreover it is logarithmic in the largest constant value* M *appearing in the constraints.*

The proof of this theorem is omitted due to space constraints and shows that this complexity is $O(|V|^{|V|-1}(|E|-|V|)^3 n^3 \log(M))$.

5 Conclusion

We have proposed an algorithm to find the optimal paths from a single source to all other vertices in a weighted graph in which weights involve an arbitrary number of real-valued parameters. Since those paths change with the values of the parameters, the result of our algorithm is a finite set of trees, each with a zone of the parameter space on which it is optimal. Those zones cover the parameter space for which there are no negative cost cycles in the graph. This algorithm generalizes a previous work by Karp and Orlin in which only one parameter was considered [6].

Further work includes implementing the algorithm, and evaluating its efficiency on real-world case-studies.

References

1. Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *ACM Symposium on Theory of Computing*, pages 592–601, 1993.
2. Étienne André, Vincent Bloemen, Laure Petrucci, and Jaco van de Pol. Minimal-time synthesis for parametric timed automata. In Tomáš Vojnar and Lijun Zhang, editors, *TACAS, Part II*, volume 11428 of *Lecture Notes in Computer Science*, pages 211–228. Springer, 2019.
3. Étienne André, Didier Lime, Mathias Ramparison, and Marielle Stoelinga. Parametric analyses of attack-fault trees. In Jörg Keller and Wojciech Penczek, editors, *19th International Conference on Application of Concurrency to System Design (ACSD 2019)*, Aachen, Germany, June 2019. IEEE Computer Society.
4. Daniel Bundala and Joël Ouaknine. On parametric timed automata and one-counter machines. *Inf. Comput.*, 253:272–303, 2017.
5. Sourav Chakraborty, Eldar Fischer, Oded Lachish, and Raphael Yuster. Two-phase algorithms for the parametric shortest path problem. In Jean-Yves Marion and Thomas Schwentick, editors, *27th International Symposium on Theoretical Aspects of Computer Science, STACS 2010, March 4-6, 2010, Nancy, France*, volume 5 of *LIPIcs*, pages 167–178. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.
6. Richard M. Karp and James B. Orlin. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete Applied Mathematics*, 3(1):37–45, 1981.
7. Didier Lime, Olivier H. Roux, and Charlotte Seidner. Parameter synthesis for bounded cost reachability in time Petri nets. In Susanna Donatelli and Stefan Haar, editors, *40th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2019)*, volume 11522 of *Lecture Notes in Computer Science*, pages 406–425, Aachen, Germany, June 2019. Springer.
8. Neal Young, Robert Tarjan, and James Orlin. Faster parametric shortest path and minimum balance algorithms. *Networks*, 21(2):205–221, 1991.