Master in Computer Science

Course : TER

Generation of sequences controlled by their "complexity"

Author:
Fissore Davide

Supervisor:
Jean-Charles Régin

Abstract

We want to generate sequences of musical "chords" (a chord is a set of notes basically) with some known constraints (allDiff, etc.) as well as control on the complexity of the sequence. This complexity in turn is defined by a dynamic programming algorithm working on the instantiated sequence, which makes the whole problem difficult.

Fall 2022

Contents

1 Problem description

The goal of this project is to analyze and find a solution to a problem of Spotify[1] in collaboration with Mr. Jean-Charles Régin[2]. The goal of this problem is to generate sequences of musical chords with some known constraints. Due to the company secret, we have not been communicated the application of this problem in the real world, but we can explain explain the subject of the problem in term of a graph problem.

We are given a directed graph $\mathcal{G} = (V, A)$ and a finite set of colors $\mathcal{C}$. Each arc of the graph is associated to a subset of $\mathcal{C}$. Let $\mathcal{P} = (v_1, \ldots, v_n)$ be a path from starting from v_1 and ending in v_n, an assignement of $\mathcal{P}$ is the selection of a unique color c for each couple of adjacent nodes v_i, v_{i+1} in $\mathcal{P}$ such that c belongs to the colors of the edge (v_i, v_{i+1}). The cost (or weight) of a path is the number of *color switches* (or color break) in $\mathcal{P}$, that is the number of times we find two adjacent edges with different colors assignements. A path with minimal cost is a path minimizing the number of colors switches.

The goal of the problem is to compute, for a given starting node $v_i \in V$ and a given ending node $v_j \in V$ the set of paths from v_i to v_j with minimal cost.

2 Definitions and notations

In this section we fix some notations that will be reused in the following sections.
As said in the previous section, $\mathcal{C}$ represents a finite set of colors and $\mathcal{G} = (V, A)$ is a directed graph where $V = (v_1, \ldots, v_n)$ is the set of its vertices and $A = (a_1, \ldots, a_m)$ is the set of its arcs; n and m represent the cardinality of respectively V and A. An arc $a_i \in A$ is made of an ordered pair $(v_i, v_j) \in V \times V$ of adjacent vertices, therefore, the arc (v_i, v_j) is different from the arc (v_j, v_i).
$\mathcal{F} : A \to 2^{\mathcal{C}}$ is the coloring function mapping each arc to its corresponding subset of colors. By abuse of notation we say that $\mathcal{F}(a) = \mathcal{F}(v_i, v_j)$ if $a = (v_i, v_j)$.
$\mathcal{G} : A \to \mathcal{C}$ is a function representing an assignement of a color $c \in \mathcal{F}(a)$ for the current arc a. For simplicity, if $\mathcal{P} = (a_1, \ldots, a_k)$ is a list of consecutive arcs whose length is k, then $\mathcal{F}(\mathcal{P}) = (\mathcal{F}(a_1), \ldots, \mathcal{F}(a_k)) = (\mathcal{C}_1, \ldots, \mathcal{C}_k)$ and $\mathcal{G}(\mathcal{P}) = (\mathcal{G}(a_1), \ldots, \mathcal{G}(a_k)) = (c_1, \ldots, c_k)$.
Given a path $\mathcal{P}$ of length k and its corresponding assignement $H = \mathcal{G}(\mathcal{P})$, the cost of H is given by $w(H)$. The definition of the cost function $w : \mathcal{C}[] \to \mathbb{N}$ is depicted in Equation (1).

$$w(H) = \sum_{i=1}^{k-1} (\text{if } c_i \neq c_{i+1} \text{ then } 1 \text{ else } 0) \tag{1}$$

An assignement H_{OPT} is the minimal, and therefore optimal, if there does not exists a second assignement H' such that $w(H') < w(H_{OPT})$.
Finally, we say that $\mathcal{P}_{OPT}$ with optimal assignement H_{OPT} is the minimal path in $\mathcal{G}$, if there does not exist a second path $\mathcal{P}'$ in $\mathcal{G}$, with same extremities as $\mathcal{P}_{OPT}$, having an optimal cost smaller than H_{OPT}.

3 Minimize Switches in Paths

The goal of this section is to provide a greedy algorithm able to compute an optimal assignement H_{OPT} of a given path $\mathcal{P}$. The obtained result, will then be extended to general graphs using a matrix technique proposed in Section 4.3 and the MDD strategy proposed in Section 5.2.

3.1 Procedure

Let $\mathcal{P} = (v_1, \ldots, v_k)$ be a path in $\mathcal{G}$, the greedy strategy to find an optimal assignement H_{OPT} is to *delay* a color switch as much as possible. The algorithm is decomposed in two main parts: the first (*part. A*) assigns each arc $a_i \in \mathcal{P}$ to a subset of colors chosen from $\mathcal{F}(a_i)$ and the second (*part B*) returns the optimal assignement H_{OPT}.

Part A. In this part of the procedure, we affect each arc of $\mathcal{P}$ to $\mathcal{L} = (\mathcal{C}_1, \ldots, \mathcal{C}_k)$, such that forall $1 \leq i \leq k$, $\mathcal{C}_i$ is a subset of $\mathcal{F}(a_i)$. Firstly, $\mathcal{C}_1$ is exactly $\mathcal{F}(a_1)$. Next, forall $1 < i \leq k$, the set of colors $\mathcal{C}_i$ attributed to the arc a_i will be iteratively set to the intersection between $\mathcal{C}_{i-1}$ and $\mathcal{F}(a_i)$ if the intersection is non-empty, otherwise $\mathcal{C}_i$ will be affected to $\mathcal{F}(a_i)$.

[1] https://open.spotify.com/
[2] http://www.constraint-programming.com/people/regin/

Part B. In this second part of the procedure, we make a unique color affectation from the list $\mathcal{L}$ returned by the *part A*. This time, we read $\mathcal{L}$ from right to left. The last arc is assigned to a random color c chosen from $\mathcal{C}_k$. Then forall $0 \leq i < k$, the color of then i^{th} arc is c_{i+1}, if c_{i+1} belongs to the set $\mathcal{C}_i$, otherwise, we are facing a color switch, and, the arc a_i can be assigned to an arbitrary color c chosen from $\mathcal{C}_i$.

An implementation of this procedure, containing both part of the algorithm, can be found in Algorithm 2.

Here, we want to give a formal proof to show that the stated procedure returns an optimal assignement for any given path. This proof is decomposed in two parts, one for each subpart of the global algorithm. In the first part we show that the list $\mathcal{L}$ minimizes the number of color switches and in second part we show that the cost of the assignement returned by *part B* is the same of the one returned by *part A*.

Proof of part A. Let $H = (\mathcal{C}_1, \ldots, \mathcal{C}_k)$ be the solution returned by *part A*, we prove, by induction on the length of the path, that H minimizes the number of color switches.
By definition of the weight function, if the length k of the path is 1, then $\mathcal{P} = (e_1)$ and we have $w(H) = 0$ which is the optimal cost: any color chosen from $\mathcal{F}(a_1)$ will cause no color switch.
In this inductive part of the proof, we suppose that H is an optimal solution for every path of length at least k. We want to prove that the new affectation H' returned by the algorithm for a path of length $k+1$ is still optimal.
This proof can be done by a case-by-case analyze:

- if $\mathcal{F}(a_k) \cap \mathcal{F}(a_{k+1}) = \varnothing$ then we are forced to make a color switch between a_k and a_{k+1}, since, the intersection of the colors of the two arcs is empty. In this particular scenario, the cost of the affectation returned for the path of length $k+1$ will be $w(H') = w(H)+1$. Since, by the induction hypothesis, H is optimal, $w(H')$ remains optimal.

- otherwise, if $\mathcal{F}(a_k) \cap \mathcal{F}(a_{k+1}) \neq \varnothing$ we have two sub-cases to treat:

  - if it exists a subset of colors $\mathcal{C}_{k+1} \subseteq \mathcal{F}(a_{k+1})$ which is included in $\mathcal{C}_k$, we are able to avoid a color switch since we are able to attribute the same color to a_k and a_{k+1}, therefore, the cost of the affectation H' of the new path of length $k+1$ equals $w(H)$. Again, since the affectation H is optimal, and we do not increase the number of color switches then the new affectation H' is still optimal.

  - this final case is the most interesting to treat because the intersection between $\mathcal{C}_k$ and $\mathcal{F}(a_{k+1})$ is empty, but, on the other hand, $\mathcal{F}(a_k) \cap \mathcal{F}(a_{k+1}) \neq \varnothing$. It means that the particular choice of colors associated to the arc a_k is causing a color switch, even if it had been possible to make no color break between the k^{th} arc and the $(k+1)^{th}$ arc of $\mathcal{P}$. The cost of the affectation H' is therefore, $w(H)+1$.
    Let's suppose, by means of contradiction, that there exists a better affectation H_{OPT}. Without loss of generality, let's suppose that the intersection of the colors of the first k arcs of the path is not empty, *i.e.* there exists at least one color shared by all the a_i $(0 \leq i \leq k)$ first arcs. The cost of this subpath is 0 since all of the arcs can have the same color. If we want to add the new arc a_{k+1} to the path, without increasing the number of color switches, then it must exist at least one color belonging to $\bigcap_{i=1}^{k+1} \mathcal{F}(i)$. However, this condition is not possible, otherwise the algorithm would have kept this subset of color as a valid option for every arc of the path, but, by hypothesis we have that $\mathcal{C}_k$ and $\mathcal{F}(a_{k+1})$ is empty. A contradiction.

We can conclude that the number of color switches returned by the first part of the procedure is minimal, therefore, optimal. □

Proof of part B. In the previous proof, we have shown that the number of color switches returned by *part A* is minimal. We only have to prove that the second part of the procedure returns an assignement with the same number of color switches.
Let $\mathcal{L} = (\mathcal{C}_1, \ldots, \mathcal{C}_k)$ be the subset affectation returned by *part A*. By construction of the *part A*, for each set $\mathcal{C}_i$ of $\mathcal{L}$, either $\mathcal{C}_{i+1}$ is a subset of $\mathcal{C}_i$ or $\mathcal{C}_i \cap \mathcal{C}_{i+1} = \varnothing$. Starting from the last arc of the path, we can choose an arbitrary color $c_k \in \mathcal{C}_k$ for the arc a_k. Then for the arc a_{k-1}, we choose the same color of a_k if possible and repeat the same procedure until reaching the first arc of the path.
We have, therefore, a color switch only when the intersection of $\mathcal{C}_i$ and $\mathcal{C}_{i-1}$ is empty. □

3.2 Time Complexity

We can analyze the time complexity of this procedure from the implementation proposed in Algorithm 2. We have two loops of size k (the length of the path). Inside them we make intersection between sets of at most $|\mathcal{C}|$, knowing that the intersection between two sets of size $|\mathcal{C}|$ is $\mathcal{O}(|\mathcal{C}|)$. The final time complexity is therefore $\mathcal{O}(2*k*|\mathcal{C}|) = \mathcal{O}(k*|\mathcal{C}|)$ which is an optimal time complexity wrt the input of the problem.

3.3 An example run

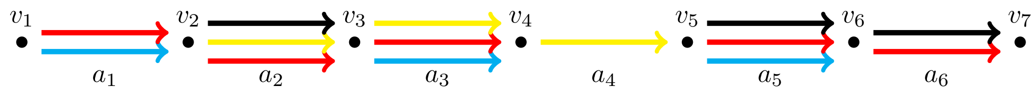


Figure 1: A path example

Let's take Figure 1, where $\mathcal{P} = (a_1, \dots, a_6)$ and $\mathcal{F}$ such that

$$\begin{aligned}\mathcal{F}(\mathcal{P}) = (&\{cyan, red\}, \{red, yellow, black\},\\ &\{cyan, red, yellow\}, \{yellow\},\\ &\{cyan, red, black\}, \{red, black\})\end{aligned}$$

Here we give a solution of how the procedure proposed in Section 3.1 would solve it. The list $\mathcal{L}$ returned by *Part A* will be

$$\begin{aligned}\mathcal{L} = (&\{cyan, red\}, \{red\},\\ &\{red\}, \{yellow\},\\ &\{cyan, red, black\}, \{red, black\})\end{aligned}$$

Then the second part of the algorithm would return an optimal solution which is, in this case, $H = (red, red, red, yellow, black, black)$, with $w(H) = 2$.

One can note that there can exist other optimal solutions, from Figure 1 we can choose $H_2 = (cyan, yellow, yellow, yellow, red, red)$, but none of them will have a cost smaller than $w(H)$.

3.4 Extension on cycles

A cycle is a path whose starting end ending nodes coincide. In this situation, the previous algorithm is no more effective, since we need to keep into account the potential color switch between the first and the last arcs. We can, however, easily modify the procedure proposed in Section 3.1, to compute optimal assignement on cycles.

Let's take the path of Figure 1 and imagine that nodes n_1 and n_7 coincide. The affectation H of Section 3.3 is no more optimal since $w(H) = 3$, whereas the cost of the affectation $H' = (red, red, red, yellow, red, red)$ is 2.

In order to take into account this situation, we have to look at the intersection between the first and the last set of colors returned by *part A*. If the intersection between the sets of $\mathcal{C}_1$ and $\mathcal{C}_k$ is not empty, we set them into their intersection.

Concretely, let's consider the example in Figure 1, we intersect $\mathcal{C}_1$ with $\mathcal{C}_6$. Since this intersection $\mathcal{I}$ is non-empty, then $\mathcal{C}_1 \leftarrow \mathcal{I}$ and $\mathcal{C}_6 \leftarrow \mathcal{I}$. The resulting affectation will be exactly H' which has the optimal cost.

4 Minimize color switches with matrices

The previous section provides a strategy to compute the smallest cost of a given path. It has been shown that an optimal strategy is to delay color switches as mush as possible. In this section we reuse this concept in order to find paths with a *fixed* number of edges between two vertices, minimizing the number of color switches.

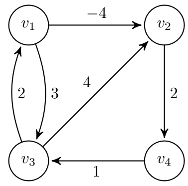


Figure 2: A directed weighted graph example

(a) Iteration 1

	v_1	v_2	v_3	v_4
v_1	0	−4	3	∞
v_2	∞	0	∞	2
v_3	2	4	0	∞
v_4	∞	∞	1	0

(b) Iteration 2

	v_1	v_2	v_3	v_4
v_1	0	−4	3	−2
v_2	∞	0	∞	2
v_3	2	−2	0	0
v_4	∞	∞	1	0

(c) Iteration 3

	v_1	v_2	v_3	v_4
v_1	0	−4	3	−2
v_2	∞	0	∞	2
v_3	2	−2	0	0
v_4	3	−1	1	0

(d) Iteration 4

	v_1	v_2	v_3	v_4
v_1	0	−4	−1	−2
v_2	5	0	3	2
v_3	2	−2	0	0
v_4	3	−1	1	0

Table 1: Floyd-Warshall algorithm execution of Figure 2

4.1 Floyd-Warshall algorithm

Floyd [1] and Warshall [2], in respectively 1959 and 1962, gave an implementation [6] of an algorithm able to compute the shortest path between any pair of vertices of a directed weighted graph. The solution is found in polynomial time over the number of vertices of the graph.

In particular, let $\mathcal{G} = (V, A)$ be a directed graph and a cost function $w : A \to \mathbb{N}$, such that for all pair of vertices i, j of V, if there exists no arc going from i to j in A then $w(i, j) = \infty$ and for each $v \in V$, $w(v, v) = 0$ (*i.e.* we are creating self loops on every vertex of the graph, that is we can stay in a vertex at no cost). Let M be the $n \times n$ adjacency matrix of $\mathcal{G}$ such that each cell $M[i][j]$ equals $w(i, j)$.

The goal of the algorithm is to build a new matrix N whose cells contain the weight of the shortest path for every pair of vertices. This matrix is updated iteratively: at time 1, we have $N^1 = M$: N^0 contains all the shortest path of length *at most* 1 between two vertices. This information should however be reworked because it can exist paths of smaller length made of more than one arc. Therefore, for every pair $i, j \in V \times V$, we seek if it exists a shortest path from i to j passing through a third vertex k exists.

$$N^k[i][j] = \min_{v \in V}(N^{k-1}[i][v] + N^{k-1}[v][j]) \tag{2}$$

At the second iteration, we obtain N^2 which contains all the shortest paths of length *at most* 2 for every pair of vertices. Globally, the matrix should be updated n times since, except for negative cycles, every shortest path between two vertices will pass through every vertex *at most* one time.

The overall time complexity of the Floyd-Warshall algorithm is $\mathcal{O}(n^3)$, we need to make at most n update of a $n \times n$ matrix.

Example 1 (Floyd-Warshall algorithm run)**.** Let's take the directed graph represented in Figure 2. The corresponding matrix M is indicated in Table 1a. At the iteration 4, the distance from the vertex v_1 to vertex v_3 is updated to -1 since there is a shorter path going from n_1 to v_4 and then from v_4 to v_3. Its overall cost is given by $c_{1,4} + c_{4,3} = -2 + 1 = -1$ which is less than the direct path v_1 to v_3.

4.2 Paths of fixed length with minimum cost

As explained in [7] and [8], the Floyd-Warshall algorithm can be generalized in order to compute shortest paths on directed weighted graphs having a *fixed* number of edges. This approach is called the Floyd-Warshall generalized algorithm and is based on the theory of semirings [3].

Semiring A *semiring*[9] is a algebraic structure composed by a set R and two binary operators $\oplus$ and $\otimes$. $(R, \oplus)$ forms a commutative monoid with an identity element z. $(R, \otimes)$ forms a monoid with an identity element called e. $\oplus$ is left and right distributive over $\oplus$ and z absorbs $\otimes$. A semiring differs from a ring because the $\oplus$ does not need to have an inverse element for $r \in R$.

Floyd-Warshall generalized algortithm Let M be the adjacency matrix of a graph whose cells on the diagonal have infinity weights if there is no self-loop on the considered vertex. We say that N^k is the matrix where each cell $N[i][j]$ contains the cost of the shortest path from i to j with *exactly* k edges[3].

[3]Note that in Section 4.1 we spoke about path of *at most* k edges.

The update function of this generalized approach differs from Equation (2) since the cost of the cell $N[i][j]$ at time k will depend of the cost of the iteration N^{k-1} and the adjacency matrix M.

$$N^k[i][j] = \min_{v \in V}(N^{k-1}[i][v] + M[v][j]) \tag{3}$$

The time complexity of this computation is $\mathcal{O}(n^3k)$, since to pass from N^i to N^{i+1} we must read n time the $n \times n$ matrix and globally the matrix is updated k times.

Link with semirings It is possible to rewrite this equation in a more concise way using the definition of semiring. In fact, if, from Equation (3), the min operator is the $\oplus$ and the $+$ operator is the $\otimes$. We have that $N^k = \oplus(N^{k-1} \otimes M)$. We can further simplify the notation knowing that $N^1 = M$ and $N^k = N^{k-1} \odot M$. In fact $N^k = M^{\odot k}$ and since min and + are associative, we can improve the previous complexity using the binary exponentiation [5] and get $\mathcal{O}(n^3 \log k)$.

4.3 Minimize color switches with matrices

In this section we propose an adaptation of the generalized Floyd-Warshall algorithm in order to compute shortest paths of fixed length minimizing the number of color switches in oriented graphs. This adaptation wants to merge this procedure with the idea of delaying color switches proposed in Section 3.1.

The adjacency matrix M is defined differently, since we do not have exact costs associated to arcs: the cost depends on the color assignation of two adjacent edges. In our implementation, $M[i][j]$ is replaced by the coloring function $\mathcal{F}(v_i, v_j)$ with the particularity that $\mathcal{F}(v_i, v_j) = \varnothing$ if there is no arc from i to j.

The cells of the N^1 matrix is a pair *(w, cols)* where: w is the cost of the path and *cols* is the set of colors minimizing the number of color switches for the path going for each vertex v_i to v_j.

Similarly to the matrix computation illustrated in the previous section, $M^k[i][j]$ depends on the matrix at time $k-1$ and $\mathcal{F}$. For all $v_i, v_j \in V \times V$, $N^1_{ij} = \{w \leftarrow 0 \ \textit{if}\ \mathcal{F}(v_i, v_j) \neq \varnothing$ otherwise ∞; $\textit{cols} \leftarrow \mathcal{F}(v_i, v_j)\}$. The N^{k+1} is computed by Algorithm 1.

Algorithm 1: Compute N^{k+1}

Input: N^k, $\mathcal{F}$, respectively, the matrix at time k and the coloring function
Output: N^{k+1} the matrix at time $k+1$

$n \leftarrow$ the number of vertices of the graph;
`// Matrix initialization`
$N^{k+1} \leftarrow$ new $n \times n$ matrix ;
$\forall i, j \in [0..n]^2 :\ N^{k+1}[i][j] \leftarrow \{w \leftarrow \infty;\ \textit{cols} \leftarrow \varnothing\}$;
`// Procedure start`
for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
        for $v = 1$ **to** n **do**
            $\mathcal{I} \leftarrow N^k[i][v].\textit{cols} \cap \mathcal{F}(v, j)$;
            $\textit{cost} \leftarrow N^k[i][v].w + (\text{if } \mathcal{I} = \varnothing \text{ then } 1 \text{ else } 0)$;
            $\mathcal{S} \leftarrow (\text{if } \mathcal{I} = \varnothing \text{ then } \mathcal{F}(v, j) \text{ else } \mathcal{I})$;
            if $\textit{cost} < N^{k+1}[i][j].w$ **then**
                $N^{k+1}[i][j] \leftarrow \{w \leftarrow \textit{cost};\ \textit{cols} \leftarrow \mathcal{S}\}$;
            else if $\textit{cost} = N^{k+1}[i][j].w$ **then**
                $N^{k+1}[i][j].\textit{cols} \leftarrow \mathcal{S} \cup N^{k+1}[i][j].\textit{cols}$;
            end
        end
    end
end
return N^{k+1};

Analyze of Algorithm 1 The first step of the algorithm initiates the matrix to return N^{k+1}. The cells of this matrix have an empty set of colors and an infinity cost. After this initialization, we loop over each pair of vertices (i, j) and, as for the generalized version of the Floyd-Warshall algorithm, we look for minimal paths passing through each vertex $v \in V$. This distance is obtained wrt the result of the intersection $\mathcal{I}$ between the color set of $N^k[i][v]$ and $\mathcal{F}(v,j)$. If $\mathcal{I}$ is not empty then we are able to avoid a color switch and, therefore, the cost of the path from i to j passing through v is the same as the cost of the path from i to v. On the other hand, if the intersection is empty, the cost of the path will be 1 more than the cost of the path from i to v. $\mathcal{S}$ is the set of colors that can be associated to the arc (v,j). It is equal to $\mathcal{I}$ if $\mathcal{I}$ is non-empty, otherwise, it will be affected to $\mathcal{F}(v,j)$, since any color in $\mathcal{F}(v,j)$ will force a color switch.

5 Minimize colors switches with *MDD*s

In this section we will provide a second approach to the problem using the *MDD* data structure in order to compute the shortest paths of *fixed* length for path starting from a given vertex of the graph.

5.1 Multi-Valued Decision Diagram

A *Multi-Valued Decision Diagram (MDD)*[4] is a generalization of a *Binary Decision Diagram*. It is represented as a directed acyclic graph whose nodes and arcs are called respectively states and transitions. *MDD*s are often used to solve constraint satisfaction problems where each layer of the *MDD* represents a variable of the problem and the number of transitions exiting from a state is upper bounded by the cardinal of the domain of the considered variable.

Even if the number of states may grow exponentially wrt the number of states, if well encoded the problem can be solved with an *MDD* whose size grows polynomially wrt its input. A well known example of this, is the representation of the language $\mathcal{L}$ accepting binary words with *fixed* length k having a 1 in the n-th last position (an example is provided at Appendix B).

5.2 The *MDD* strategy

The problem of minimizing the number of color switches in a colored graph can be solved with an *MDD*. This strategy is less generic then the matrix method: with the Floyd-Warshall matrix approach we compute the shortest paths from all the nodes of the graph, however the *MDD* should have a root and therefore this strategy will find all the shortest paths of *fixed* number of edges from a chosen node.

$$\text{type state} \triangleq \{\text{name: String},\ \text{cost: Int},\ \text{colors: Set of Colors}\} \tag{4}$$

The states of the *MDD* will be represented by the record depicted in Equation (4) and the root of the *MDD* will have $\{\text{name: } \mathcal{C},\ \text{cost: } 0,\ \text{colors: } n\}$ where n is the name of the starting node of the paths. The $\mathcal{C}$ is the set containing all the colors of the problem.

Let r be the node chosen for the root of the *MDD*, at each iteration i a new layer is added to the *MDD*. The i^{th} layer represents the set of shortest paths of length i rooted in r.

The algorithm which builds the *MDD* works as follow: for every state s_i with name n_i of the current layer and for every successor n_j of n_i in $\mathcal{G}$, let $\mathcal{S} = \mathcal{F}(n_i, n_j) \cap s_i.\text{colors}$. Let $\mathcal{L}$ be the new layer to build, if $\mathcal{S}$ is non-empty we add to $\mathcal{L}$ the state

$$s_j = \{\text{name: } n_j,\ \text{cost: } s_i.\text{cost},\ \text{colors: } \mathcal{S}\}$$

otherwise the new state

$$s_j = \{\text{name: } n_j,\ \text{cost: } s_i.\text{cost} + 1,\ \text{colors: } \mathcal{F}(n_i, n_j)\}$$

is added[4].

***MDD* reduction** Let $\mathcal{L}$ be the current layer of an *MDD*, to avoid the exponential growth of the search tree, an *ad-hoc* strategy is applied in order to either ignore dominated states or to merge two s-compatible states. A state s_1 is dominated by s_2 if they have same *name* and the cost of s_1 is smaller than the cost of s_2, the dominated states are removed from $\mathcal{L}$. Two states s_1 and s_2 are s-compatible if they share the same *name* and the same *cost*. In this case, s_1 and s_2 are removed from $\mathcal{L}$ and a third state $s_3 = \{\text{name: } s_1.\text{name},\ \text{cost: } s_1.\text{cost},\ \text{colors: } s_1.\text{colors} \cup s_2.\text{colors}\}$ is added to $\mathcal{L}$.
Finally, the application of the domination and the s-compatible laws ensures that the *MDD* to only have layers with a size upper-bounded by $|V|$.

[4]Note that n_i and n_j are two nodes belonging to the graph $\mathcal{G}$.

Proof sketch of the algorithm. For each new-created state, we set its color to the subset of the colors in common with its father. This is equivalent to the *Part A* of the procedure depicted in Section 3.1. We only have to show that the *MDD* reduction is valid for two states s_1 and s_2.

Case 1: s_1 dominates s_2. In this case, the algorithm is going to remove the state s_2, it means that, starting from the root r, we have found two different paths going to the node $s_1.name$ (that is the same of s_2.name) but the path going to s_2 is more expensive than the path s_1. We can therefore ignore s_2 from the current layer.

Case 2: s_1 and s_2 are s-compatible. In this situation we see that we build a new state s_3 having the same *name* and *cost* of s_1 (that are the same of s_2), but whose colors are the union of the colors of s_1 and s_2. This union is motivate by the fact that we are saying Continue this proof !

We can conclude that the reduction phase is valid. In the end we can all the minimal-cost paths from the root r of the *MDD* by applying the *part B* of the procedure of Section 3.1. □

Time complexity of this procedure Let n be the cardinal of V and k be the length of the path to build, thanks to the application of the *MDD* reduction, we can determined that the overall time complexity is $\mathcal{O}(k \cdot n^2 \cdot \mid \mathcal{C} \mid)$ since at each layer we have at most n states and for each state we should visit at most n successors and for each new-created state, we have to perform an intersection between the colors of each state. Given that k is a fixed parameter, the complexity can be simplified to $\mathcal{O}(n^2 \times \mid \mathcal{C} \mid)$.

5.3 An example run

5.4 The all different constraint

The all different constraint (*allDiff*) is a very used constraint in constraint programming (CP). The goal of this constraint is to assign each considered variable to a value of its domain such that there does not exist two variables with same assignment. Even if *allDiff* is simple to implement sometime it can considerably increase the time complexity of the problem we are dealing with.

Let's take the alphabet $\mathcal{A} = \{a \ldots z\}$ and let $\mathcal{L}$ be the set of words of length 3. The *MDD* representing all the words of the problem will have 4 states (the root plus one state per letter in the word). On the other hand, if we add the *allDiff* constraint on the letters of the words, the corresponding *MDD* will have $\mid \mathcal{A} \mid \times (\mid \mathcal{A} \mid -1) \times (\mid \mathcal{A} \mid -2) + 1$ states. This exponential growth is justified by the inability to efficiently apply the reduction operation on the layer of the *MDD* in every layer, each state of the *MDD* has the particular role to "memorize" the letters stored previously in order to avoid any possible repetition.

5.5 Find simple paths

A variation of the color-switching problem of the graph is the application of the *allDiff* constraint on the nodes of the graph. The goal of this section is to adapt the *MDD* algorithm provided in Section 5.2 in order to apply the *allDiff* constraint. A path now will be valid only if it is "simple" that is we can't pass two times or more on any already visited node.

The main modification we must apply to the previous algorithm is to slightly modify the information stored in the states of the *MDD*: a state must remember the sequence of nodes visited to join it from the root. The new state will be represented by the record in Equation (5)

$$\text{type state } \triangleq \{\text{name: String, cost: Int, colors: Set of Colors, parents: Set of Nodes}\} \tag{5}$$

The first part of the algorithm of Section 5.2 remains valid: when we add a new layer $\mathcal{L}$, we loop through every state s_i of the previous layer and, for every successor n_j of $n_i = s_i$.name, we build the new state s_j. The only new operation to do in this variation, is to update the *parents* field of s_j which will be set to s_i.parent $\cup \{n_i\}$[5].

Furthermore, the *MDD* reduction should be modified to keep into account the *fathers* of each state. Let s_1, s_2 be two states belonging to the same layer of the *MDD* s_1 dominates s_2 if they have same *name*, same *parents*[6] and the *cost* of s_1 is smaller than the *cost* of s_2, moreover s_1 and s_2 are s-compatible if the have same *name, parents* and *cost.*

[5] Note that the *parents* field of the root is the empty set.

[6] The symmetrical difference between the set of parents of s_1 and the set of parents of s_2 is empty

Time complexity The *allDiff* constraint forces to compute paths of length at most $| V |$, since a longer path should contain repetitions of nodes. A path of length $| V |$ passing exactly one time per node is called an *Hamiltonian Path* and compute such a path is a *NP-Complete* problem.

6 An implementation of the stated procedures

The previous algorithms have been implemented in *OCaml* and here we want to provide a sketch of the main data structures used to fulfill the requirements.

MySet. A useful data structure extending the classical *Set* module of *OCaml*. In particular, when we start to build a path, in order to maintain the *standard* update function over colors for each couple on adjacent nodes of a path, we need to represent the "*Full*" set (*i.e.* the complement of $\varnothing$). *MySet* adds this specifications. The classical operation over sets have been overrode if needed, so that, for example, the intersection of a set $\mathcal{S}$ and the *Full* set gives $\mathcal{S}$ and their union gives *Full*. The main advantage of sets in *OCaml* is that they are an immutable data structure. In fact, every binary operation over a set does not modify the current set, but it builds rapidly a fresh copy with the wanted content. This is very useful in our *MDD* implementation, for example, since the *parents* filed of each child of a state should contain the intersection of the colors of the father and the colors of the current arc.

TODO: say that Full is the full set of colors

The coloring function $\mathcal{F}$. Working with the implementation of the proposed algorithms, we remarked that a clear type for the color function would have improved a lot the clarity of the code.

```
type colorFunction = {
  is_sym : bool;
  tbl : (int * int, ColorSet.t) Hashtbl.t;
  get_col : int * int -> ColorSet.t;
}

let get tbl (v1, v2) =
  Option.value ~default:ColorSet.empty (Hashtbl.find_opt tbl (v1, v2))

let init ?(is_sym = false) () =
  let tbl = Hashtbl.create 2048 in
  { is_sym; tbl; get_col = get tbl }

let add { tbl; is_sym; _ } v1 v2 col =
  Hashtbl.replace tbl (v1, v2) col;
  if is_sym then Hashtbl.replace tbl (v2, v1) col
```

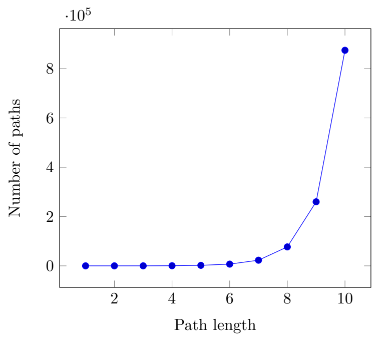
The *is_sym* field is used internally to know if the graph is not directed: in this case every edge $\langle a, b\rangle$ of the undirected graph is transformed to a couple of directed arcs (a, b) and (b, a). The *tbl* field is an hash-table where to a couple of nodes we associate the set of colors of the arc they represent. The *get_col* which is a function taking a couple of node which gives back either the set of colors of the arc if the arc between the two nodes exists otherwise the empty set. Finally, the *color_function* can be instantiated through the *init* function and the *add* function allows to add new arcs of the graph to it.

In fact, the *color_function* record can be seen as an classical *Java* object in the light *OCaml* functional style.

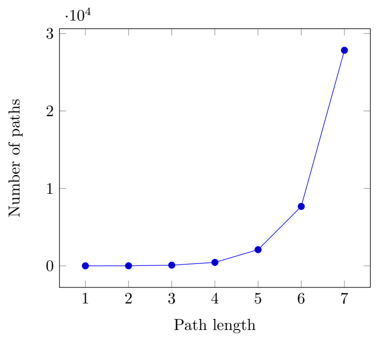
A state of a *MDD*. The states of a MDD are implemented with a functor[10] parametrized by a module implementing the *merge* operation, since we have to know if, having two states, we have to merge them, to keep the first or the second state.

This functor is particularly useful if we want to implement the classic algorithm, or the *allDiff* variant, in both case, in every case, we will only need to respectively implement the *merge* function.

Moreover, we can verify if two states are compatible, *i.e.* ready to be merged, through the *compare* function. In fact, a layer of a *MDD* is a set of states. We know if two state are to be merged if the *compare* method returns 0.



(a) Classic algorithm



(b) *allDiff* constraint

Figure 3: Number of paths of a given length from the node 1

The *MDD* functor. The last but not the least, important data structure is the *MDD* functor, the heart of the *MDD* procedure implementation. This functor is parametrized by a *State* of a *MDD*.

```
module Make : functor (T : State) ->
  sig
    ...
    type mdd_layers = MySet.Make(T).t ref
    val add : S.t -> S.elt -> S.t
    val add_succ : int list -> (int -> T.t2) -> T.t -> S.t
    val update_layers :
      (int -> int list) ->
      ColorFunction.colorFunction -> mdd_layers -> unit
  end
```

This functor signature need an implementation of some useful function, such as the *update_function*, the *initiate*, *make_iteration*, the *count_paths* and the *run* functions.

A *mdd_tree* is a recursive data structure made of a state, the root of the current sub *MDD*, a hash-table of children a list of fathers of the current state. Thanks to this ...

Rename mdd_tree to decorated state and add mdd_layer

Others

Matrix mathod, adjacency matrix

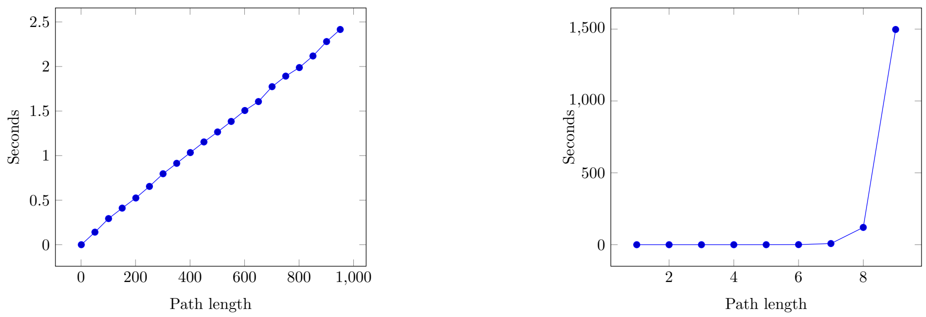
7 A benchmark of the *MDD* implementation

We have been provided, by Spotify, a graph representing a sample of the problem this company is dealing with. This graph is on the form of a *json* where a list of nodes are associated to colors and a list of pairs (a_1, a_2) representing the arcs of the graph.

The *MDD* version of the algorithm has been tested on this graph in order to obtain some of statistics, in particular we have tested the number of solutions and the time taken to get them for a given path length. The source of the path taken into account, in order to build the results, is the node 1. The are no big differences of performance for other sources.

Time and number of solutions In Figure 3a and Figure 3b, we can see that the number of solutions computed by respectively the classic algorithm or the algorithm with the *allDiff* constraint plots a curve with an exponential growth. However, we can also remark that the introduction of the *allDiff* constraint reduce drastically the number of solutions, in particular, for a path with 10 edges, there are about 9×10^5 solutions in the classic version against the about 3×10^4 of the *allDiff* version. It is quite normal to obtain this difference, since, as said in the previous sections, the *allDiff* constraints more the domain of the variables.

Another interesting phenomenon, which confirms the complexities of the algorithms, is that computing the solutions for the classic version takes a number of time which grows linearly with the length of the path. On the other hand, the *allDiff* version goes enough fast to compute paths of length up to 6, but, we need about 8 seconds for path of length 7, 2 minutes for paths of length 8, 25 minutes for paths of length 9, therefore with an exponential trend.



(a) Classic algorithm (b) *allDiff* constraint

Figure 4: Time taken to compute paths of a given length from the node 1

length	min_cost	*max_cost*
1	0	0
2	0	0
3	0	0
4	0	2
5	0	3
50	0	1
1000	0	1

(a) Classic algorithm

length	min_cost	*max_cost*
1	0	0
2	0	0
3	0	1
4	0	2
5	0	3
6	0	4
7	0	5

(b) *allDiff* version

Table 2: Number of solution stats

Cost of the paths for a given length In this paragraph we want to give some results about the cost of the paths, computed through the Equation (1). It is interesting to see that thanks to this property, we can deduce some properties of the graph and the behavior of the of the algorithm in these situations.

Lets look at the results proposed in Table 2.

8 Conclusion

8.1 Go further

The *NValue* constraint...

Coq implementation of fst algo

Make example of mdd algorithm run

9 References

[1] Robert W. Floyd. *Algorithm 97*. 1959.

[2] Stephen Warshall. *A Theorem on Boolean Matrices*. 1962.

[3] John E. Hopcroft et Jeffrey D. Ullman Alfred V. Aho. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Jan. 11, 1974, pp. 195–206. ISBN: 978-0-201-00029-0.

[4] Jean-Charles Régin. *ERC Advanced Grant 2019*. Research proposal [Part B2]. Université Côte d'Azur, 2019.

[5] *Binary Exponantiation*. June 8, 2022. URL: https://cp-algorithms.com/algebra/binary-exp.html.

[6] *Floyd-Warshall algorithm*. 2022. URL: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm.

[7] *Generalized Floyd-Warshall algorithm*. 2022. URL: https://fr.wikipedia.org/wiki/Probl%C3%A8me_de_plus_court_chemin#Algorithme_de_Floyd-Warshall_g%C3%A9n%C3%A9ralis%C3%A9.

[8] *Number of paths of fixed length / Shortest paths of fixed length*. June 8, 2022. URL: https://cp-algorithms.com/graph/fixed_length_paths.html.

[9] *Semiring*. Oct. 6, 2022. URL: https://en.wikipedia.org/wiki/Semiring.

[10] *Functors in OCaml*. URL: https://ocaml.org/docs/functors.

A Minimize color switches in a path

Algorithm 2: Shortest path of a path

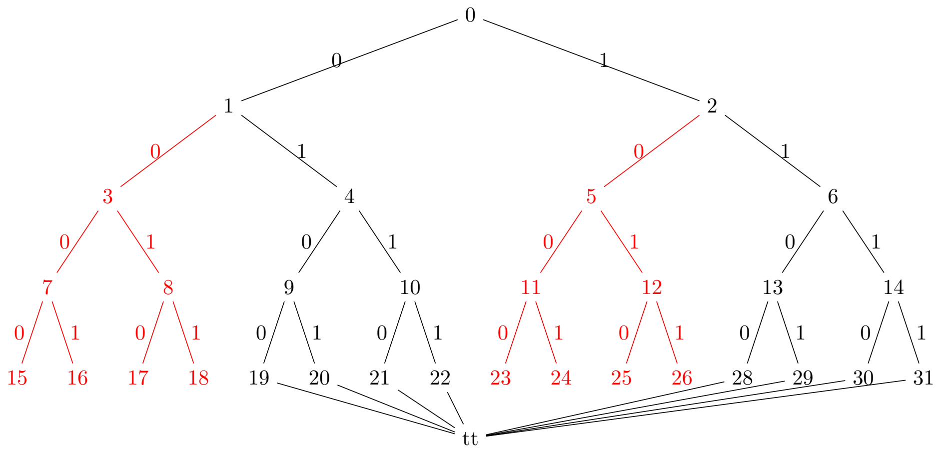
Input: $\mathcal{P} = (a_1, \ldots, a_k), \mathcal{F}$:= a path and the color function

Output: H := a path affectation minimizing the color switches

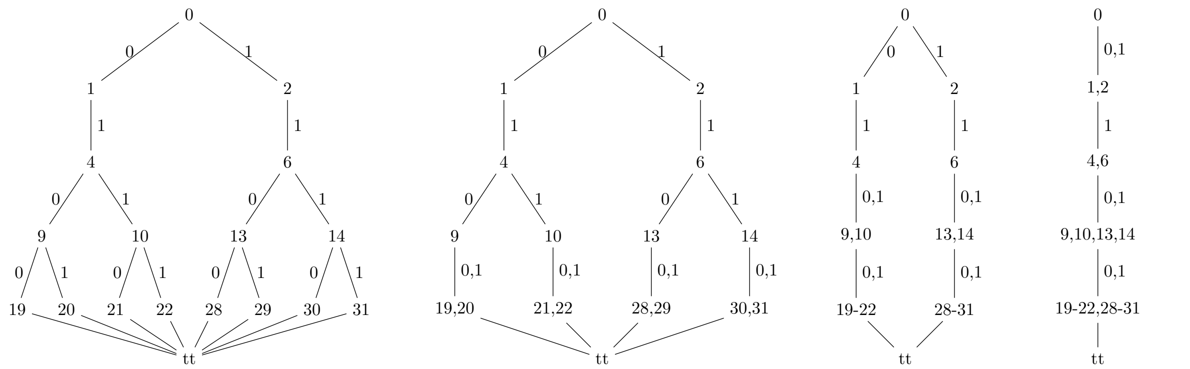
```
colSet ← [F(a_i) for i ∈ [1..k]];
for i ← 2 to k do
    inter ← colSet[i − 1] ∩ colSet[i] ;          // Delay a color switch
    if inter ≠ ∅ then
        colSet[i] ← inter;
    end
end
H ← [colSet[i].choose() for i ∈ [1..k]];
for i = k − 2 downto 1 do
    if H[i + 1] ∈ colSet[i] ∧ H[i] ≠ H[i + 1] then
        H[i] ← H[i + 1] ;          // If possible the R(e_i) equals R(e_{i+1})
    end
end
return H;
```

B MDD example

In this section we provide an example of a *MDD* reduction for the language $\mathcal{L} = \{\omega \in \{0,1\}^4 \mid \omega[2] = 1\}$, that are all the binary of length 4 with a 1 in the second position as a constraint. In Figure 5a we have the full *MDD* representation, and we can immediately see that the red branches can be ignored since they do not respect the constraint. After their elimination (Figure 5b), we see that the nodes 19 and 20 have same father and are both accepting states. Therefore, they are compatible and apt to be merged (a same reasoning case be made on the pairs of nodes 21 and 22, 28 and 29, 30 and 31 with their respectively fathers). In *Figure* 5c, we see that the two sub-paths going from state 4 to *tt* can be merged: they both lead to the accepting state and their suffix is identical w.r.t the labels on their edges. We continue this way until the reduction operation in no more feasible in order to obtain the final *MDD* depicted in Figure 5e.



(a) Complete *MDD*



(b) Deletion of dead branches (c) Reduction 1 (d) Reduction 2 (e) Reduction 3

Figure 5: *MDD* for $\mathcal{L}$