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

# 2 Definitions and notations

In this section we fix some notations that will be reused next.
$G = (V, A)$ is a directed graph where $V = (v_1, \dots, v_n)$ is the set of its vertices and $A = (a_1, \dots, a_m)$ is the set of its arcs. $n$ and $m$ represents the cardinality of respectively $V$ and $A$. An arc $a_i \in A$ is a pair $(v_i, v_j) \in V^2$ saying that $a_i$ goes from $v_i$ to $v_j$. The arc $(v_i, v_j)$ is different from $(v_j, v_i)$.
colF is the coloring function and takes an arc $a$ as parameters. It returns the set of colors $\mathbb{C}$ associated to $a$. By abuse of notation we say that $\mathrm{colF}(a) = \mathrm{colF}(v_i, v_j)$ if $a = (v_i, v_j)$.
$R : A \to \mathbb{N}$ is the affectation function, and $R(e) = c$ if $c \in \mathrm{colF}(e)$. For simplicity, if $P = (a_1, \dots, a_k)$ is a list of consecutive arcs whose length is $k$, then $\mathrm{colF}(P) = (\mathrm{colF}(a_1), \dots, \mathrm{colF}(a_k)) = (\mathbb{C}_1, \dots, \mathbb{C}_k)$ and $R(P) = (R(a_1), \dots, R(a_k)) = (c_1, \dots, c_k)$.
Given a path $P$ of length $k$ and its corresponding affectations $H = R(P)$, the weight of $H$ is the number of its color switches. This cost is computed by $w(R(P))$.

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

$w_{OPT}(H)$ is the minimal weight of a path among all the possible affectation $H$ of $P$, $H_{OPT}$ is an optimal affectation.
Finally, we say that a shortest path from $v_i$ to $v_j$ in a graph $G$ is a path $P$ starting in $v_i$ and ending in $v_j$ whose optimal affectation $H_{OPT}$ is the minimal among all the other possible paths in $G$.

# 3 Minimize Switches in Paths

The goal of this section is to provide a greedy algorithm able to compute an optimal affectation $H$ of a given path $P$. The obtained result, will then be extended to general graphs using the matrix technique proposed in Section 4.3 or the MDD strategy of Section 5.2.

## 3.1 Procedure

This problem can be solved through a greedy strategy: taking a path $P$ and a coloring function colF, we delay a color switch as much as possible. The algorithm is decomposed in two main parts where the first affects each arc $a_i$ to a subset of colors of $\mathrm{colF}(a_i)$ and where the second makes a unique affectation for each arc.

**Procedure part 1.** In this part of the procedure, we affect each arc of a path $(a_1, \dots, a_k)$ to a subset of colors $(\mathbb{C}_1, \dots, \mathbb{C}_k)$. The colors associated to $a_1$, noted $\mathbb{C}_1$, are exactly $\mathrm{colF}(a_1)$. Next, the set of colors $\mathbb{C}_i$ of the arc $a_i$ $(i > 1)$ will be iteratively given by the intersection of $\mathbb{C}_{i-1}$ and $\mathrm{colF}(a_i)$ if the intersection is non-empty, otherwise, a color switch is imminent and, therefore, $\mathbb{C}_i$ will be $\mathrm{colF}(a_i)$.

**Procedure part 2.** In this second part of the procedure, we make a unique color affectation from the list $\mathcal{L} = (\mathbb{C}_1, \dots, \mathbb{C}_k)$ returned by the previous procedure. We read the $\mathcal{L}$ from right to left. The color $c_k$ affected to the last arc of $P$ is a color randomly chosen from $\mathbb{C}_k$. The color of $i^{th}$ arc $a_i$ $(i < k)$ is the color $c_{i+1}$ affected to the arc $a_{i+1}$, if $c_{i+1}$ is in $\mathbb{C}_i$, otherwise, we are facing a color switch, and, therefore, any color of $C_i$ can be indifferently affected to $a_i$.

An implementation of this procedure can be found in Algorithm 1.

*Proof (First part of the procedure).* Let $H_{\mathbb{C}} = (\mathbb{C}_1, \dots, \mathbb{C}_k)$ be a solution returned by the first part of our algorithm, we prove, by induction on the length of the path, that $H_{\mathbb{C}}$ minimizes the number of color switches. After this first proof, we will show that the number of color switches returned by the second part of the algorithm is the same as the one returned by the first part.
By definition of the weight function, if the length $k$ of the path is 1 we have $w(H_{\mathbb{C}}) = 0$ which is the optimal cost. And therefore, any color chosen from $\mathbb{C}_1$ will not cause any color switch.
Let's suppose that $H_{\mathbb{C}}$ is an optimal solution for every path of length at least $k$. We want to prove that the new affectation $H'_{\mathbb{C}}$ returned by the algorithm for a path of length $k + 1$ is still optimal.
We have to analyze two main situations:

**Algorithm 1:** Shortest path of a path

**Input:** $P = (a_1, \ldots, a_k)$, colF := a path and the color function
**Output:** $H$ := a path affectation minimizing the color switches

```
colSet ← [colF(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;
```

- if $\mathrm{colF}(a_k) \cap \mathrm{colF}(a_{k+1}) = \varnothing$ then, for any color $c$ chosen from $\mathrm{colF}(a_{k+1})$, $w(H'_{\mathbb{C}}) = w(H_{\mathbb{C}}) + 1$, *i.e.* a color switch is forced independently on the affectation we have chosen for the first part of the list. Since, by hypothesis, $H_{\mathbb{C}}$ is optimal, $w(H')$ remains optimal.

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

  - if it exists a subset of colors $\mathbb{C}_{k+1} \subseteq \mathrm{colF}(a_{k+1})$ which is included in $\mathbb{C}_k$, *i.e.* there exists at least a color in $\mathrm{colF}(a_{k+1})$, we are able to avoid a color switch. therefore, the cost of the affectation $H'_{\mathbb{C}}$ of the new path of length $k+1$ equals $w(H_{\mathbb{C}})$. Again, since the affectation $H_{\mathbb{C}}$ is optimal, and we do not increase the number of color switches then the new affectation $H'_{\mathbb{C}}$ is still optimal.

  - this final case is the most interesting to treat because the intersection between $\mathbb{C}_k$ and $\mathrm{colF}(a_{k+1})$ is empty, but, on the other hand, $\mathrm{colF}(a_k) \cap \mathrm{colF}(a_{k+1}) \neq \varnothing$. 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 $P$. The cost of the affectation $H'_{\mathbb{C}}$ is therefore, $w(H_{\mathbb{C}}) + 1$.
    Let's suppose, by means of contradiction, that it exists a better affectation $H_{\mathbb{C}opt}$ of subsets for $(a_1, \ldots, a_{k+1})$. 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, it must exist at least one color belonging to $\bigcap_{i=1}^{k} \mathrm{colF}(i)$. 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. A contradiction.

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

*Proof (Second part of the procedure).* In the previous proof, we have shown that the number of color switches return by the first part of the algorithm is minimal. We only have to prove that its second part returns an affectation with the same number of color switches.
Let $(\mathbb{C}_1, \ldots, \mathbb{C}_k)$ be the subset affectation returned by the previous part of the algorithm. Note that, by construction of the first algorithm, for each set $\mathbb{C}_i$, its successor $\mathbb{C}_{i+1}$ is either a subset of $\mathbb{C}_i$ or $\mathbb{C}_i \cap \mathbb{C}_{i+1} = \varnothing$. Starting from the last arc of the path, we can choose an arbitrary color $c_k \in \mathbb{C}_k$ 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 $\mathbb{C}_i$ and $\mathbb{C}_{i+1}$ is empty. □

## 3.2 Time Complexity

We can analyze the time complexity of this procedure from the implementation proposed in Algorithm 1. We have two loops of size $k$ (the length of the path). Inside them we make intersection between sets of at most $s$ colors, then the intersection between two sets of that size will take $\mathcal{O}(s)$. Finally, the global time complexity will be $\mathcal{O}(2 * k * s) = \mathcal{O}(k * s)$.

## 3.3 An example run

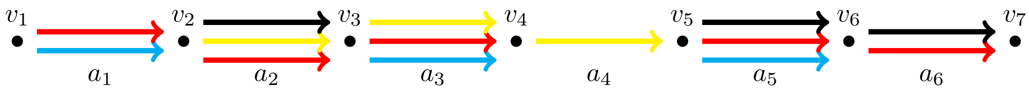



Figure 1: A path example

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

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

Here we give a solution of how the procedure proposed in Section 3.1 would solve it. The first procedure will return a list of subsets equal to

$$\begin{aligned}H_{\mathbb{C}} = (&\{cyan, red\}, \{red\},\\ &\{red\}, \{yellow\},\\ &\{cyan, red, black\}, \{red, black\})\end{aligned}$$

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 affectations, from Figure 1 we can choose $H_2 = (cyan, yellow, yellow, yellow, red, red)$, but in any case, any other affectations will not be less than $w(H)$.

## 3.4 Extension on cycles

A cycle in a path whose starting node coincide with its last one. 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. However, the procedure proposed in Section 3.1, can be easily modified to provide an optimal affectation 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 consider this situation, it is important to look at the intersection between the first and the last set of colors returned by the first part of the procedure. In particular, the subset associated to the extreme arcs of the path, will be modified into the intersection of their corresponding sets if non-empty. Finally, we can apply the second part of the algorithm.

Concretely, take the example in Figure 1, we intersect $H_{\mathbb{C}1}$ with $H_{\mathbb{C}2}$. Since this intersection $\mathcal{I}$ is non-empty, then $H_{\mathbb{C}1} \leftarrow \mathcal{I}$ and $H_{\mathbb{C}7} \leftarrow \mathcal{I}$. 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.

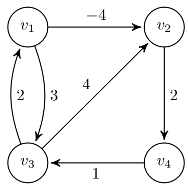



Figure 2: A directed weighted graph example

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| $v_1$ | 0 | $-4$ | 3 | $\infty$ |
| $v_2$ | $\infty$ | 0 | $\infty$ | 2 |
| $v_3$ | 2 | 4 | 0 | $\infty$ |
| $v_4$ | $\infty$ | $\infty$ | 1 | 0 |

(a) Iteration 1

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| $v_1$ | 0 | $-4$ | 3 | $-2$ |
| $v_2$ | $\infty$ | 0 | $\infty$ | 2 |
| $v_3$ | 2 | $-2$ | 0 | 0 |
| $v_4$ | $\infty$ | $\infty$ | 1 | 0 |

(b) Iteration 2

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| $v_1$ | 0 | $-4$ | 3 | $-2$ |
| $v_2$ | $\infty$ | 0 | $\infty$ | 2 |
| $v_3$ | 2 | $-2$ | 0 | 0 |
| $v_4$ | 3 | $-1$ | 1 | 0 |

(c) Iteration 3

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| $v_1$ | 0 | $-4$ | $-1$ | $-2$ |
| $v_2$ | 5 | 0 | 3 | 2 |
| $v_3$ | 2 | $-2$ | 0 | 0 |
| $v_4$ | 3 | $-1$ | 1 | 0 |

(d) Iteration 4

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 $G$ be a directed graph and a cost function $w$, 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 $v \in V$, $w(v, v) = 0$. Let $M$ be the $n \times n$ adjacency matrix of $G$ such that each cell $M_{ij}$ 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 0, we have $N^0 = M$ representing all the shortest path of length *at most* 1 between two vertices. This information should however be improved to find out paths of smaller length made of more than one arc. Therefore, for every pair $i, j \in V^2$, we seek if the shortest path from $i$ to $j$ passing through a third vertex $k$ exists.

$$N_{ij} = \min_{k \in V}(N_{ik} + N_{kj}) \tag{2}$$

At the second iteration, we obtain $N^2$ which contain 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 loop $n$ times through a $n \times n$ matrix.

**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 based on the theory of semirings [3].

**Semiring** A *semiring*[9] is a algebraic structure composed by a set of elements $R$ 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 $e$, it is right distributive over $\oplus$ and $z$ is absorbing over $\otimes$. A semiring differs from a ring because it is not required to have an inverse for the $\oplus$ operation.

**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 $N_{ij}$ contains the cost of the shortest path from $i$ to $j$ with *exactly* $k$ edges[1].

[1]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 $c_{ij}$ at time $k$ will depend of the cost at the previous iteration and the adjacency matrix.

$$N_{ij}^{k} = \min_{v \in V}(N_{iv}^{k-1} + M_{vj}) \tag{3}$$

The time complexity of this computation is $\mathcal{O}(n^3 k)$, 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 and say that $N^k = N^{k-1} \odot M = M^{\odot k}$. Finally, 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 of two adjacent color affectations. In our implementation, $M$ is represented by the coloring function where $\mathrm{colF}(ij) = \varnothing$ if there is no edge between the two vertices $i$ and $j$.

The cells of the $N^0$ 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, $M^k$ depends on the matrix at time $k-1$ and the coloring function (*i.e.* the modified adjacency matrix). For all $i, j \in V$, $N_{ij}^0 = \{w \leftarrow 0 \textit{ if } \mathrm{colF}(ij) \neq \varnothing;\ \textit{cols} \leftarrow \mathrm{colF}(ij)\}$. The $N^{k+1}$ is computed by Algorithm 2.

**Algorithm 2:** Compute $N^{k+1}$

**Input:** $N^k$, colF, 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 ;
$\forall i, j \in [0..n]^2 :\ N_{ij}^{k+1} \leftarrow \{w \leftarrow \infty;\ \textit{cols} \leftarrow \varnothing\}$;
`// Procedure start`
**for** $i = 1$ **to** $n$ **do**
  **for** $j = 1$ **to** $n$ **do**
    **for** $p = 1$ **to** $n$ **do**
      $\mathcal{I} \leftarrow N_{ip}^{k}.\textit{cols} \cap \mathrm{colF}(pj)$;
      $\textit{cost} \leftarrow N_{ip}^{k}.w + (\text{if } \mathcal{I} = \varnothing \text{ then } 1 \text{ else } 0)$;
      $\mathcal{S} \leftarrow (\text{if } \mathcal{I} = \varnothing \text{ then } \mathrm{colF}(pj) \text{ else } \mathcal{I})$;
      **if** $\textit{cost} < N_{ij}^{k+1}.w$ **then**
        $N_{ij}^{k+1} \leftarrow \{w \leftarrow \textit{cost};\ \textit{cols} \leftarrow \mathcal{S}\}$;
      **else if** $\textit{cost} = N_{ij}^{k+1}.w$ **then**
        $N_{ij}^{k+1}.\textit{cols} \leftarrow \mathcal{S} \cup N_{ij}^{k+1}.\textit{cols}$;
      **end**
    **end**
  **end**
**end**
**return** $N^{k+1}$;

**Analyze of Algorithm 2** The first step of the algorithm is to initiate the resulting matrix $N^{k+1}$ such that each cell as an empty set of colors and an infinity cost. After this initialization, we loop over each pair of vertices $ij$ and, as for the generalized version of the Floyd-Warshall algorithm, we look for minimal paths passing through each vertex $p \in V$. This distance is obtained following the $w$ function: if the intersection $\mathcal{I}$ of colF$(pj)$ and the color set of cell $ip$ of $N^k$ is not empty, we are able to avoid a color switch and therefore, the cost of the path $ip$, $pj$ is the same as the cost of the path $ip$. On the other hand, if the intersection is empty, the cost of the path will be 1 more than the cost of the path $ip$. $\mathcal{S}$ is the set of colors that can be associated to the arc $pj$. It is equal to $\mathcal{I}$ if $\mathcal{I}$ is non-empty (*i.e.* we are delaying the color switch), otherwise, it will be affected to colF$(pj)$, since any color in colF$(pj)$ will not avoid a color switch.

Let $0 \leq p' \leq p \leq n$. While looping over all the intermediate vertex $v_p$, we can have three possible scenarios:

- there exists a path passing through $v_{p'}$ which is less than the path passing through $v_p$, the path through $n_p$ can be ignored;
- the computed cost is strictly less than all the the previous path passing through $v_{p'}$, in this case the shortest path between $ij$ will take this new cost and its set of colors will $\mathcal{S}$;
- finally, the cost passing through $v_i$ equals a previous minimal one. In this case, the cost is not updated, but the colors we can give to the edge $ij$, to reduce the possibility of a switch, will be the union of the colors of the previous best affectations and $\mathcal{S}$.

Add binary exponentiation

# 5 Minimize colors switches with MDDs

## 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 by the 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 coded 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 A).

## 5.2 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 find shortest paths starting indistinctly from any node 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.

$$\text{state} = \{\text{name: String; cost: Int; colors: Set of Colors}\} \tag{4}$$

The states of the *MDD* will be represented by the record depicted in Equation (4) and the root will be {colors: Set.Full, cost:0}, where *name* is the name given to the current node in the graph and *Set.Full* is the set containing all the colors returned by colF.

Let $r$ the node chosen for the root of the *MDD*, at the $i^{th}$ iteration a new layer is added to the *MDD*, in order to represent the set of shortest paths of length $i$ rooted in $s$.

The algorithm aiming to build the *MDD* works as follow: for every state $s$ of the current state and for every successor $n$ of $s$ in $G$, let $\mathcal{S}$ = colF$(s, n) \cap$ s.colors. Let $\mathcal{L}$ be the new layer to build, if $\mathcal{S}$ is non-empty we add to $\mathcal{L}$ the record

$$\{\text{name: n; cost: s.cost; colors: } \mathcal{S}\}$$

otherwise the record

$$\{\text{name : n; cost: s.cost} + 1\text{; colors: colF(s.name, n)}\}$$

is added.

To avoid the exponential growth of the search tree, an *ad-hoc* strategy is applied in order to either ignore all dominated states or to merge two compatible states. A state $s_1$ is dominated by $s_2$ if they have same *name* and $s_1$.cost $< s_2$.cost, every dominated state is removed from $\mathcal{L}$. Two states $s_1$ and $s_2$

are compatible (and removed from $\mathcal{L}$) if they share the same *name* and the same *cost*. In this case a third state $s_3 = \{\text{name: } s_1.\text{name; cost: } s_1.\text{cost; colors: } s_1.\text{colors} \cup s_2.\text{colors}\}$ is built and added in $\mathcal{L}$.

The application of this reduction the *MDD* ensures us to only have can grow of at most $|V|$ at each level.

Let $n$ be the cardinal of $V$ and $l$ be the length of the path to build, This last remark can be exploited to give an information about the complexity of the algorithm which is bounded by $\mathcal{O}(l \cdot n^2)$ since at each layer we have at most $n$ states and for each state we should visit at most $n$ successor states. Moreover, given that $l$ is a fixed parameter, the complexity can be simplified to $\mathcal{O}(n^2)$.

Add proof ?

## 5.3 The all different constraint

The all different constraint (*allDiff*) is a very used constraint in CP whose goal is to affect each variable to a value of its domain such that there does not exist to variables with same affectation. This constraint is very simple to implement, but sometimes, it can complexify the problem we are dealing with.

Let's take the alphabet $\mathcal{A} = \{a \dots z\}$ an let $\mathcal{L}$ be the set of words of length 3. The *MDD* satisfying this problem will have on 4 states (including the root), whereas the *MDD* for the same problem with the *allDiff* constraint on the letter of the words will have $\mid \mathcal{A} \mid \times (\mid \mathcal{A} \mid -1) \times (\mid \mathcal{A} \mid -2) + 1$ states. This exponential growth is due to the inability of reducing the with of the layers since each state of the *MDD* has the particular role to "memorize" the letters stored previously in order to avoid any possible repetition.

## 5.4 Find simple paths

In this section we are going to adapt the *MDD* algorithm provided in Section 5.2 in order to apply the *allDiff* constraint on the nodes of the paths. 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 of the graph.

In order to solve this newly added constraint in the problem, we have to slightly modify the information stored in the states of the *MDD* in order to remember from which nodes we are coming from. Therefore, a state will now be represented by the record in Equation (5)

$$\text{state} = \{\text{name: String; cost: Int; colors: Set of Colors; parents: Set of Nodes}\} \tag{5}$$

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 $s$ of the previous layer and for every successor $n$ of $s$ we build the new state. The only new operation to do in this situation, is to update the *parent* field of $n$ which will be $\{\text{s.parent} \cup \text{s.name}\}$[2].

The important modification to focus on, is the *MDD* reduction. Currently, a state $s_1$ is dominated by a state $s_2$ if they have same *name*, same *parents* (*i.e.* $s_1.\text{parents}\Delta s_2.\text{parents} = \varnothing$) and the *cost* of $s_2$ is less than the *cost* of $s_1$, and two states are compatible if the have same *name, parents* and *cost*. This two conditions are useful respectively to remove dominated states and merge compatible states.

We can finally see that the introduction of the *allDiff* constraint causes a complexity blow up from a polynomial to an exponential one. In fact, the size of the layers in the *MDD* can grow of at most $|V|$ at each level.

# 6 An implementation of the stated procedures

The previous algorithms have been implemented in *OCaml*, following the procedures provided in the previous sections, 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.

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

**The color_function type.** 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 color_function = {
  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.

```
type action = MERGE | REPLACE | IGNORE

module type OrderedType = sig
  type t

  val merge : t -> t -> action
  val compare : t -> t -> int
end

module Make (Ord : OrderedType) = struct
  module MySet = MySet.Make (Ord)
  include MySet
end
```

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.

**The mdd_tree record.** The last but not the least, important data structure is the *mdd_tree*, the heart of the *MDD* procedure implementation.

```
type 'a mdd_tree = {
  node : int list;
  state : 'a;
  mutable children : (int, 'a mdd_tree) Hashtbl.t;
  mutable father : 'a mdd_tree list;
}
```

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

Ciao

# 8 Conclusion

## 8.1 Go further

The *NValue* constraint...

# 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.

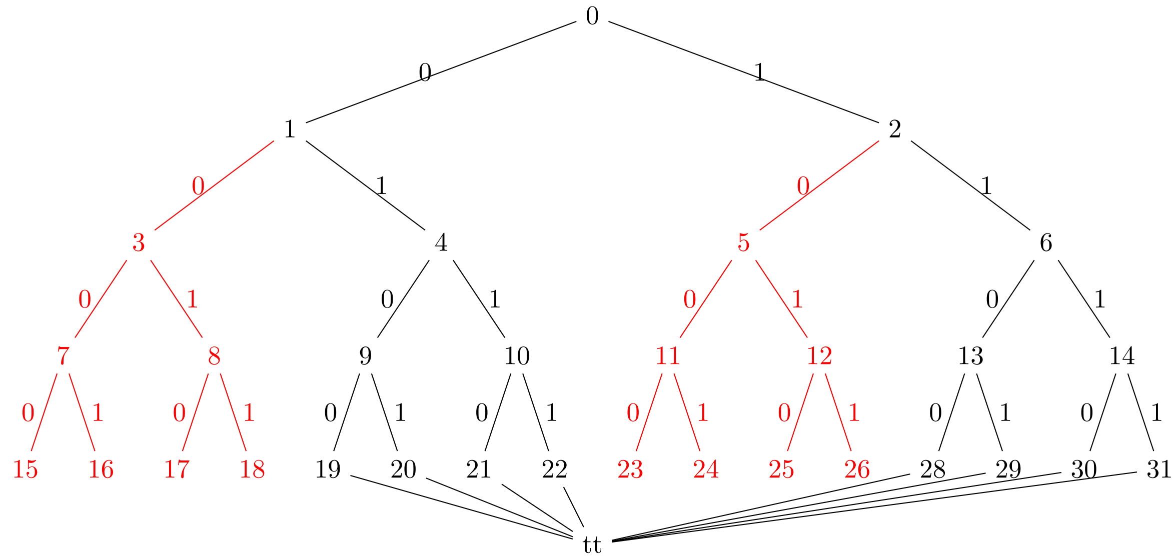
[8] *Number of paths of fixed length / Shortest paths of fixed length*. June 8, 2022. URL: https://cp-algorithms.com/graph/fixed_length_paths.html.

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

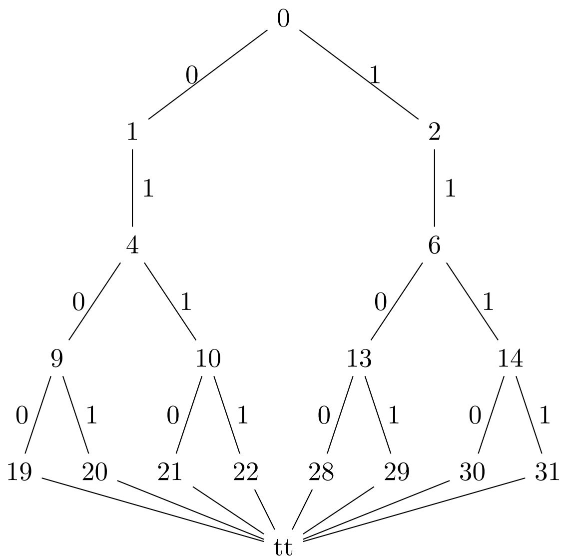
[10] *Functors in OCaml*. URL: https://ocaml.org/docs/functors.

# A MDD example

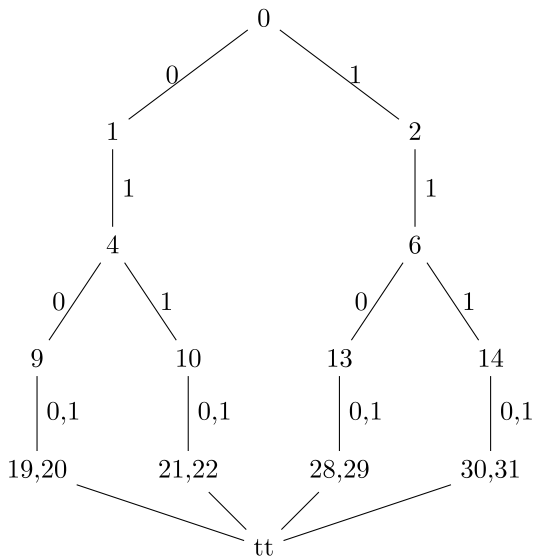
In this section we provide an example



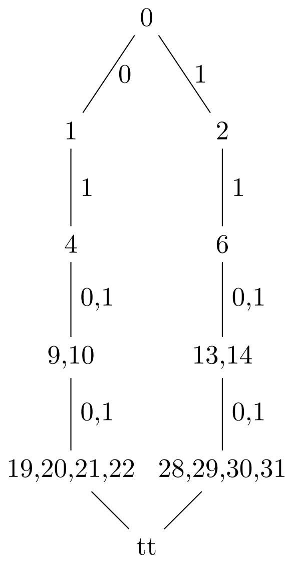

(a) Complete *MDD*



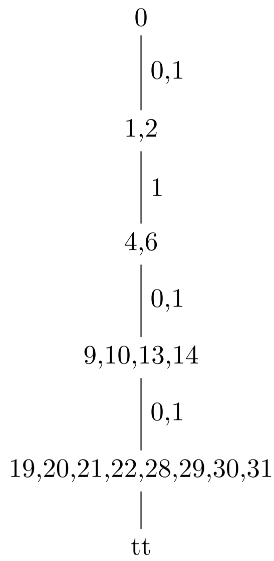

(b) Reduction 1




(c) Reduction 2




(d) Reduction 3




(e) Final reduction

Figure 3: *MDD* for $\mathcal{L} = \{\omega \in \{0,1\}^4 \mid \omega[2] = 1\}$