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 $V = (v_1, \ldots, v_n)$ is the set of its vertices and $A = (a_1, \ldots, a_m)$ is the set of its arcs. n and m 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).
F 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 $F(a) = F(v_i, v_j)$ if $a = (v_i, v_j)$.
$R : A \to \mathbb{N}$ is the affectation function, and $R(e) = c$ if $c \in F(e)$. For simplicity, if $P = (a_1, \ldots, a_k)$ is a list of consecutive arcs whose length is k, then $F(P) = (F(a_1), \ldots, F(a_k)) = (\mathbb{C}_1, \ldots, \mathbb{C}_k)$ and $R(P) = (R(a_1), \ldots, R(a_k)) = (c_1, \ldots, c_k)$.
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, R_{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 R_{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 F, we delay a color switch as much as possible. Tha algorithm is decomposed in two main parts where the first one affect each arc a_i to a subset of colors of $F(a_i)$ and a second one which make a unique affectation for each arc.

For the first part of the procedure, we start from the colors associated to the first arc and we take the colors of the second arc in order to make the intersection of them. We repeat this operation for all the following arcs until the intersection returns a non-empty set or we do not reach the end of the path. At the i^{th} arc, if the intersection set is empty, we make a color switch. Then we repeat the same procedure from this arc starting with a set of colors equal to $F(a_i)$. This procedure returns for each arc a subset of its associated colors.

For the second part of this procedure, we have to make a unique affectation, this is possible by reading the list of set returned by the previous part of the algorithm. For the last arc we take one among all the possible colors. For the next subset of colors related to the before-last arc, we take the same color of the last arc if present, otherwise a random color from its set. We repeat this operation until reaching the first arc of the path.

An implementation of this procedure can be found in Algorithm 2.

Proof (First part of the procedure). Let $H_{\mathbb{C}} = (\mathbb{C}_1, \ldots, \mathbb{C}_k)$ 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 $k = 1$ we have $w(H_{\mathbb{C}}) = 0$ which is the optimal cost.
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 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:

- if $F(a_k) \cap F(a_{k+1}) = \varnothing$ then, for any color c_i chosen from $F(a_{k+1})$ $w(H'_{\mathbb{C}}) = w(H_{\mathbb{C}}) + 1$, *i.e.* a color switch is forced.

- otherwise, if $F(a_k) \cap F(a_{k+1}) \neq \varnothing$ we have two sub-cases to treat:
  - if it exist a subset of colors $\mathbb{C}_{k+1} \subseteq F(a_{k+1})$ which is included in $\mathbb{C}_k$, *i.e.* there exists at least a color in $F(a_{k+1})$, allowing us to have $w(H'_{\mathbb{C}}) = w(H_{\mathbb{C}})$.
  - this final case is the most interesting to treat because we have $\mathbb{C}_k \cap F(a_{k+1}) = \varnothing$ but of the other hand we have $F(a_k) \cap F(a_{k+1}) \neq \varnothing$. It means that we have done a particular choice of colors associated to the arc a_k causing a color switch ($w(H'_{\mathbb{C}}) = w(H_{\mathbb{C}}) + 1$) even if it would have been possible to affect a_k to the same color of its following arc.
    Let's suppose, by means of contradiction, that it exists a better affectation of subsets for the subpath $(a_1, \ldots, a_{k+1})$. Without loss of generality, let's suppose that the intersection of the first $F(a_i), 0 \leqslant i \leqslant k$ is not empty, *i.e.* there exists a color which is shared by all the a_i first arcs. The cost of the this subpath is 0 since all edges can have the same color. If we want to add the new edge $ak + 1$ to the path without increasing the number of color switch we need that there exists at least a colors in $F(ak + 1)$ which is common to all the $F(a_i), 0 \leqslant i \leqslant k$ first edges. However, this is not possible, otherwise the algorithm would have kept this color as a valid option for all the path. A contradiction.

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

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 return exactly the an affectation with the sae number of colors switches to conclude that our algorithm is optimal.
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 a_k. Then for the edge a_{k-1}, we choose the same color of a_len if possible and repeat the same procedure until reaching the first edge of the path.
By definition, the number of color switches of the generated path will be qual to the number of two by two different $i, j \in [0..k]$ such that $\mathbb{C}_i \cap \mathbb{C}_{i+1} = \varnothing$. □

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 s colors, then the intersection between two sets of that size will take $O(s)$. Finally, the global time complexity will be $O(2 * k * s) = O(k * s)$.

3.3 An example run

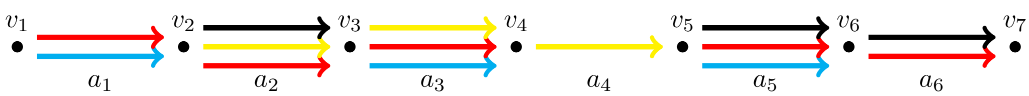

Figure 1: A path example

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

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

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

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

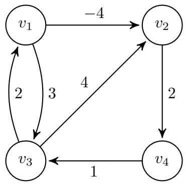

Figure 2: A directed weighted graph example

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. We see that the previous algorithm is no more effective, since we have to keep into account the potential color switch between its first and the last arcs. Despite this, 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. We now see that the affectation H of Section 3.3 is no more optimal: $w(H) = 3$, while the affectation $H' = (red, red, red, yellow, red, red)$ as a cost of 2.
In order to take into account this situation, we keep attention to 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 first and the last edges, will be given by the intersection of their corresponding sets if non-empty. Otherwise, nothing is change on the given affectation.
Concretely, take the example in Figure 1, we intersect $R'(a_1)$ with $R'(a_7)$. Since this intersection is non-empty, then $R'(a_1) = R'(a_7) = red$ $P_1 = (a_1, a_2, a_3)$. The final affectation will be return by applying the second part of the procedure.

4 Minimize color switches with matrices

The previous section provides a strategy to compute the smallest cost of a given path. The key idea is to delay color switches and in this section we try to rework this algorithm in order to apply it on general directed graph. The goal is to find paths made of a *fixed* number of edges between two vertices in order to minimize the number of color switches.

4.1 Floyd-Warshall algorithm

Floyd [1] and Warshall [2], in respectively 1959 and 1962, gave an implementation [5] of an algorithm able to compute the shortest path of a directed weighted graph.

Let G be a directed graph and c, such that for all couple of vertices i, j of V, if there exists no arc going from i to j in A then $w(i, j) = \infty$. Let M be the $n \times n$ adjacency matrix of G such that each cell M_{ij} equals $w(i, j)$. Note that for each $v \in V$, $w(v, v) = 0$.

The goal of the algorithm is to update the weight of each cell for every iteration. In particular at time 0, we have $M^1 = M$ representing all the shortest path of length *at most* 1 between two vertex. To improve the information about the global shortest path, we look for path from every couple $i, j \in V^2$ passing through a third vertex k and take the minimum distance. Therefore M^2 will indicate all shortest path for every pair of vertices of length *at most* 2.

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

Globally, the matrix should be updated n times. In fact, except for negative cycles, every shortest path between two vertices will pass through every vertex *at most* one time.

Let's take the directed graph represented in Figure 2. The corresponding adjacency matrix is indicated in Table 1a. At the iteration 2, the distance from vertex v_3 to vertex v_2 is updated to -2 since, while looking for all possible paths passing through an intermediate vertex, there is the path P' going from n_3 to n_1 and then from n_1 to n_2. The overall cost P' is made of $c_{3,1} + c_{1,2} = 2 + (-4) = -2$ which is less than the direct path v_3 to v_2 depicted in the adjacency matrix.

	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

(a) Iteration 1

	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

(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	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

4.2 Paths of fixed length with minimum cost

As explained in [6] and [7], 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 semi-rings [3].

Let M be the adjacency matrix of a graph whose cells on the diagonal have an infinity weights if there is no self-loop on the considered vertex. We say that M^k is the k-matrix where each cell M_{ij} contains the cost of the shortest path from i to j with *exactly*[1] k edges.

The update function of this generalized approach is slightly different from Equation (2) since the cost of the cell c_{ij} will depend of its cost at the previous iteration and the original adjacency matrix.

$$M^k_{ij} = \min_{v \in V}(M^{k-1}_{in} + M_{nj}) \tag{3}$$

An important remark of this approach is that while finding path with fixed length k, we consent to pass more than one time through each vertices (*i.e.*we can have non-simple path).

4.3 Minimize color switches with matrices

In this section we propose an adaptation of the generalized Floyd-Warshall algorithm in order to solve the shortest path problem minimizing the number of color switches in oriented graphs.

The main strategy as proved in Section 3.1 will be to delay color switches as further as possible by keeping trace of the set of colors for every edge.

What will change is the construction of the adjacency matrix M since we do not have exact costs associated to edges, but instead a set of colors. The weight of this edge will depend by the chosen affectation compared to the one of its neighbors. M is a $n \times n$-matrix where

$$M_{ij} = \begin{cases} F(ij), \text{ if } ij \in A \\ \varnothing, \text{ otherwise} \end{cases} \tag{4}$$

The cells of the M^k matrix is a pair *(w, cols)* where: w is the cost of the path and *cols* is the set of colors to choose in cell ij minimizing the number of color switches for the current path.

Similarly to the matrix computation illustrated in the previous section, M^k is computed from the matrix at time $k-1$ and the adjacency matrix M. Particularly, for all $i, j \in V$, $M^1_{ij} = \{w \leftarrow 0 \textit{ if } M_{ij} \neq \varnothing;\ \textit{cols} \leftarrow M_{ij}\}$ and the other matrices are computed thanks Algorithm 1

Firstly, the matrix M^{k+1} initialized and each cell as an empty set of colors and an infinity cost. In a second moment for each cell ij we try to find a minimal path passing through a vertex $p \in M$. If such path exists (*i.e.* the distance is not ∞), we have to check the intersection I of the color set of the cell ip at time k and the set of colors returned by the coloring function for the edge pj. The cost of the edge is either 0 if I is non-empty, 1 otherwise. The subset of colors associated to the edge pj at position k is I if it is non-empty, otherwise $F(pj)$.

The shortest path is obtained by finding the path ipk with minimum associated cost. If there are multiple minima then the colors we can use at vertex j is made by the union of all the colors associated to the vertices with the minimum cost.

Why take the union, and why it is the good strategy to find the minimum paths ?

Add binary exponentiation

5 Minimize colors switches with MDD

5.1 Multi-Valued Decision Diagram

A *Multi-Valued Decision Diagram* is

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

Algorithm 1: Compute M^{k+1}

Input: M^k, M respectively the matrix at time k and the adjacency matrix
Output: M^{k+1} the matrix at time $k+1$

$n = len(M)$;
`// Matrix initialization`
$M^{k+1} \leftarrow$ new $n \times n$ matrix ;
$\forall i, j \in [0..n]^2 : \; M_{ij}^{k+1} \leftarrow \{w \leftarrow \infty;\ cols \leftarrow \varnothing\}$;
`// Procedure start`
for $i = 1$ **to** n **do**
  for $j = 1$ **to** n **do**
    for $p = 1$ **to** n **do**
      $inter\_prov \leftarrow M_{ip}^{k}.cols \cap M_{pj}$;
      $cost \leftarrow M_{ip}^{k}.w +$ (if $inter\_prov = \varnothing$ then 1 else 0);
      $inter \leftarrow$ (if $inter\_prov = \varnothing$ then M_{pj} else $inter\_prov$);
      if $cost < M_{ij}^{k+1}.w$ **then**
        $M_{ij}^{k+1} \leftarrow \{w \leftarrow cost;\ cols \leftarrow inter\}$;
      else if $cost = M_{ij}^{k+1}.w$ **then**
        $M_{ij}^{k+1}.cols \leftarrow \; inter \cup M_{ij}^{k+1}.cols$;
      end
    end
  end
end
return M^{k+1};

5.2 MDD strategy

5.3 The all different constraint

5.4 Find simple paths

5.5 The NValue constraint

6 Conclusion

7 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] Mostafa Haghir Chehreghani. *Effectively Counting s-t Simple Paths in Directed Graphs*. Report. Teheran Polytechnic, 2022.

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

[6] *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.

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

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

A Algorithms

A.1 Minimize color switches in a path

Algorithm 2: Shortest path of a path

Input: $P = (a_1, \ldots, a_k), F :=$ 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;
```