

Algorithmique appliquée

Projet UNO

Paul Dorbec, Cyril Gavaille

The aim of this project is to encode a program as efficient as possible to find the best sequence of cards that can be played by a single player in a hand of UNO. We first recall quickly the rules (simplified) of UNO. Cards bear a number and are of some color. A first card is chosen (here by the player), and then every card played must be of the same color or bear the same number as the previous card.

It is proven that this problem is NP-hard, with a reduction from the problem of finding a longest path in a cubic graph (see that paper). Remark that the reduction in the other direction is quite straightforward, so it seems natural to adapt algorithms for the longest path problem to our problem here.

In this project, we try to find an algorithm as efficient as possible that would solve the problem.

1 Definition of the file format for storing UNO games

We want to be able to deal with predefined instances of the problem. So we want to read a file wherer the instance is described. We propose to simply use the following format:

- a card is described with two numbers, one for its number, one for its color. E.g., “5 3” stand for the 5 of the third color. The cards will be described by that pair of values, separated by a blank.
- a deck (set of cards) is described as a file where each line contains one card. So the file contains as many line as there are cards, with two numbers on each line.

Here is an example of a game with 12 cards, with a longest sequence on 9 cards (written on 3 columns).

0 1	1 5	5 2
1 0	2 1	4 2
5 3	0 3	0 5
2 6	3 4	5 4

2 Brute force: backtracking

We first consider backtracking algorithms (see Wikipedia).

The idea is to propose a recursive algorithm. Suppose that you have already chosen a (possibly empty) beginning of a sequence. For each card that can be possibly chosen to cover the current last card, select it, look recursively for the best sequence using that card. Select the best sequence among all this sequences and return it.

This is a simple algorithm, that has a very high complexity in the worst case. Indeed, if all cards are of the same color, after picking each cards, you may select all the other cards. So possibly, you try all the permutations of the list of cards, that is $n! = \Theta\left(\left(\frac{n}{e}\right)^n \sqrt{n}\right)$.

3 First algorithm based on dynamic programming

The idea of dynamic programming is to compute some partial optimal solution and to extend it step by step to the solution of the whole problem.

3.1 DFS

Here, we first use a separation of the deck of cards into some maximal sequences obtained within a depth first search algorithm DFS. Explore the whole card set with a DFS. All the paths from the root to a leaf in are maximal paths. They form a sequence of subsets of cards S_1, S_2 , etc. They are represented with colors in Fig. 1 (S_1 , S_2 , S_3).

The key observation of this exploral is that a valid sequence of cards that uses two cards within different subsets S_i and S_{i+1} must pass by cards which are at the intersection $S_i \cap S_{i+1}$ of these subsets. For example, in Fig. 1, it is not possible to join the node with label 4 to the node with label 6 without

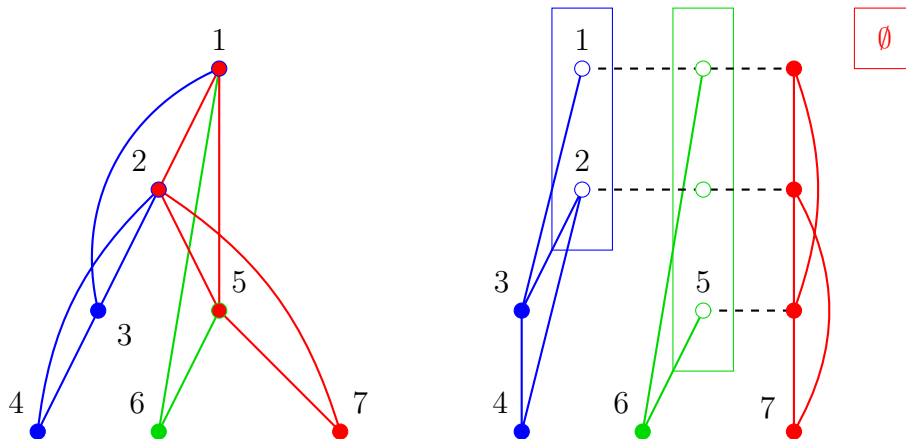


Figure 1: DFS on a graph and the sets obtained, together with their intersections.

going through 1 or 2. We will use this to make an efficient algorithm when the intersections of the subsets are small.

3.2 Principle of the algorithm

Suppose we have a sequence of subsets of cards S_1, S_2, \dots, S_p as computed by the DFS. Now we want to compute all the maximum sequences by considering each subset independently. Valid sequences of cards may remain in one subset, or may visit more than one. For example, the sequence $6 - 5 - 7 - 2 - 4 - 3 - 1$ starts in the green set, then goes through the red set before returning to the blue set (see Figure 2, left). This can be decompose into a sequence ending in 5 in the green set, then joining 5 to 2 in the red set, and finishing from 2 in the blue set. For the computation within one set, we don't need to know the details of what happens in the other set, but just how they interact with the current set, i.e. with the intersections.

Here is the process we follow. In a subset S_i , we compute a maximal sequence, taking into account possibilities of joining to cards by sequences in neighbouring sets. Precisely, we need to define the *signature* of a sequence, that is how the sequence behave on the intersection. Then, for each subset, we consider a maximal sequence of cards together with possible extra moves, and associate the length of the sequence to its signature for further computation. This can be done iteratively, using the value of the signatures of the set S_{i-1} to define the value of the signature of S_i that will be considered for further computation.

3.3 Signatures

We now discuss precisely how the signature should be defined. There is not a single way of defining the signatures, you can check Pr. Gavaille's webpage for another proposal (in French). We describe here one way of computing the signature.

We consider the set of cards in the intersection $S_i \cap S_{i+1}$ that are used by a sequence. A card may be used for jumping from the card of one subset to the other either from S_i to S_{i+1} (from left to right) or in the other direction. Note that if it jumped from left to right, the next jump must be from right to left, and vice versa. In the signature, we store all the cards that are used to jump from one set to the other in a sequence, and we remember if the first jump is from left to right or from right to left with a signature type. This implies in the signature:

- an ordered sequence of cards (e.g. (1, 2, 5)), that describes the order in which the cards are jumped through in the intersection.
- a type (A or B) specifying whether the first card in the order is visited from left to right (type A) or from right to left (type B).

Another case that must be considered is when some card of the intersection is used in the sequence but not as a jump from one subset to the other. In that case, the same card should not be used in a later part of the sequence. So in the signature, we need to store the set of cards that are forbidden for later use.

- a set of forbidden card, that have been used already

For each signature, we want to store the maximal length of a sequence with that signature. It is simpler to store the length of the signature as a number of jumps (edges of the corresponding graph) than as a number of cards, since the cards are in multiple sets. If two sequences have the same signature, only the longest sequence need to be stored to find a global sequence of maximal length. From the example of Fig. 2, the first set gives the

join cards in $S_{i-1} \cap S_i$, and that we may use subsequence in future subsets for joining cards in $S_i \cap S_{i+1}$. So we virtually add jumps between these cards in the sets, and assume they may be used in the backtrack. Then it should happen that the jump between two cards in $S_i \cap S_{i+1}$ can be done either by a direct jump or through a sequence in the future. In that case, it is better to always consider this should be a sequence in the future, since it can be decided then whether to finally take the direct jump or not. However, if two cards in $S_{i-1} \cap S_i$ can be jumped directly, we should consider both cases. Similarly, if two cards are in $S_{i-1} \cap S_i \cap S_{i+1}$, we need to consider both cases.

For each maximal sequence found by the backtrack in S_i , we compute the signature of this sequence in $S_{i-1} \cap S_i$ and get the value v computed earlier for this signature. Then we add to v the number of jumps made among cards from $S_i \setminus S_{i+1}$, and set to the signature of our sequence in $S_i \cap S_{i+1}$ the maximum of the newly computed value and the possible value it was already attributed. We also need to consider the empty sequence and report the signature for the empty intersection to the following.

Note that for the first and the last set, the same process can be used considering that the intersection with the set that does not actually exist is empty. Finally, the optimal length found should appear as the optimal length for the empty signature after the last set.

4 Probabilistic approach: dynamic programming using color coding.

The third algorithms rely on a simple observation. If you assign a random label from 1 to k to the UNO cards in your hand, there is a non zero probability that you find a sequence of cards of length k such that each card in the sequence is the only card with its label. (In terms of graphs, if you assign a random color to the vertices of the graph, there is a chance that a path of length k is rainbow, i.e. each pair of vertices in the path bear a different color.)

Then finding a path of length k with only different colors can be done with dynamic programming. The idea is to compute for each subset of labels $S \subseteq \{1, \dots, k\}$ and for each label $\ell \in S$ the set $C_{S,\ell}$ of cards bearing label ℓ of a sequence that uses all the labels from S . This can be done inductively on the order of S :

- if S is of order 1, i.e. $S = \{\ell\}$, then every card with label ℓ is the last card of a sequence of cards having all color in S . So $C_{\{\ell\},\ell}$ is all cards with label ℓ for each $\ell \in \{1, \dots, k\}$.

- otherwise, a card C with label ℓ is the end of such a sequence for $S \cup \{\ell\}$ if and only if there is a label $\ell' \in S$ such that C is reachable from a card with label ℓ' that is the last card in a sequence using all colors from S . Thus $C_{S \cup \{\ell\}, \ell}$ contains all cards that are of label ℓ and of same number or color than cards in $C_{S, \ell'}$ for some label $\ell' \in S$.

The algorithm works as follows:

- assign a random label to each card, with range 1 to k .
- for each label ℓ , define $C_{\{\ell\}, \ell}$ as the set of cards with label ℓ .
- for each size s from 2 to k , for each subset S of labels of size s in $\{1, \dots, k\}$, for each $\ell \in S$, compute the cards of $C_{S, \ell}$ following the rule above.
- if there exist a label ℓ such that $C_{\{1, \dots, k\}, \ell}$ is not empty, this is a success
- if it failed, restart up to e^k times.