



CONCOURS CENTRALE•SUPÉLEC

Sujet 02 — MPI

Jeu randomisé et route optimale




Durée : 3h

Centrale-Supélec

Préambule

Ce sujet d'oral d'informatique est à traiter, sauf mention contraire, en respectant l'ordre du document. Votre examinatrice ou votre examinateur peut vous proposer en cours d'épreuve de traiter une autre partie, afin d'évaluer au mieux vos compétences.

Le sujet comporte plusieurs types de questions. Les questions sont différenciées par une icône au début de leur intitulé :

- les questions marquées avec  nécessitent d'écrire un programme dans le langage demandé. Le jury sera attentif à la clarté du style de programmation, à la qualité du code produit et au fait qu'il compile et s'exécute correctement ;
- les questions marquées avec  sont des questions à préparer pour présenter la réponse à l'oral lors d'un passage de l'examinatrice ou l'examineur. Sauf indication contraire, elles ne nécessitent pas d'appeler immédiatement l'examinatrice ou l'examineur. Une fois la réponse préparée, vous pouvez aborder les questions suivantes ;
- les questions marquées avec  sont à rédiger sur une feuille, qui sera remise au jury en fin d'épreuve.

Votre examinatrice ou votre examinateur effectuera au cours de l'épreuve des passages fréquents pour suivre votre avancement. En cas de besoin, vous pouvez signaler que vous sollicitez explicitement son passage. Cette demande sera satisfaite en tenant également compte des contraintes d'évaluation des autres candidates et candidats.



En bleu sont indiquées des remarques et observations de la part du jury, ainsi que des précisions sur la réussite des questions et les interactions qui ont eu lieu avec les candidates et candidats.

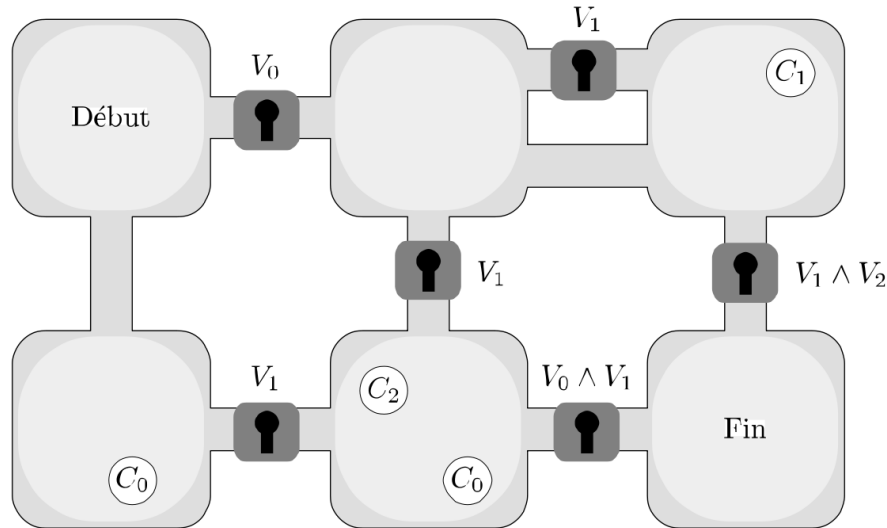


FIGURE 1 – Un exemple d’instance du jeu considéré dans ce sujet. Les salles sont modélisées par des nœuds et peuvent contenir des clefs C_i à ramasser, représentées ici par des ronds blancs. Les couloirs sont modélisés par des arêtes étiquetées par des éventuels verrous V_i qui indiquent les clefs nécessaires pour emprunter cette arête. L’objectif est d’atteindre le nœud *Fin* à partir du nœud *Début*.

1 Introduction

Dans ce sujet, on s’intéresse à la *randomisation de clefs* dans un jeu et à la recherche d’une *route optimale* permettant de gagner le plus rapidement possible.

1.1 Contexte

Dans l’optique d’offrir de la rejouabilité à un jeu-vidéo, certains programmes dits *randomizers* permutent les emplacements d’objets clefs au sein du jeu, afin de pousser le joueur à le compléter en suivant un chemin différent. Un autre défi intéressant consiste à compléter un jeu le plus rapidement possible, ce qui passe par la recherche d’une route optimale.

On s’intéresse dans ce sujet à un jeu simplifié, qui sera représenté par un multi-graphe (un graphe dans lequel plusieurs arêtes peuvent exister entre deux nœuds) non-orienté. Les nœuds représentent des lieux pouvant contenir des clefs C_i à récupérer. Pour prendre une arête vers un nœud voisin, il faut parfois déverrouiller un verrou V_i ou une conjonction de verrous $V_{i_1} \wedge \dots \wedge V_{i_n}$ qui étiquettent l’arête. Cela est uniquement possible si durant son parcours, le joueur a déjà récupéré toutes les clefs C_i nécessaires, sachant qu’une clef C_i permet de déverrouiller le verrou V_i , indexé par le même i . Lorsque le joueur déverrouille un verrou, il conserve toujours les clefs qui lui ont été nécessaires. De manière générale, une fois qu’une clef est ramassée, elle reste dans l’inventaire jusqu’à la fin du jeu. Le joueur commence initialement dans le nœud *Début* et son objectif est de se déplacer dans le multi-graphe en collectant des clefs pour atteindre le nœud *Fin*. Si cela est possible, on dit que le jeu est *satisfiable*.

1.2 Formalisation

On pose $k \in \mathbb{N}$ un nombre de clefs différentes, et l'on note les clefs par C_i pour $0 \leq i \leq k-1$. On note $K = \{C_0, \dots, C_{k-1}\}$ l'ensemble des clefs.

Une instance du jeu est définie par un multi-graphe $\mathcal{G} = (\mathcal{S}, \mathcal{C}, \mathcal{A})$ dans lequel \mathcal{S} est un ensemble fini de nœuds qui contient au moins deux nœuds particuliers notés D et F correspondant respectivement aux nœuds de début et de fin, $\mathcal{C} : \mathcal{S} \rightarrow \mathcal{P}(K)$ est une application associant à chaque nœud l'ensemble des clefs présentes dans ce nœud, et \mathcal{A} est un ensemble fini d'arêtes $a = (n, W, n') \in \mathcal{S} \times \mathcal{P}(K) \times \mathcal{S}$ où n est le nœud de départ, W l'ensemble de clefs nécessaires pour pouvoir prendre cette arête et n' le nœud d'arrivée.

On appelle *route* une succession finie d'arêtes

$$\mathcal{R} = a_0, a_1, \dots, a_r = (n_0, W_0, n'_0), (n_1, W_1, n'_1), \dots, (n_r, W_r, n'_r)$$

qui vérifie les conditions suivantes :

- $n_0 = D$

La route commence par le nœud Début.

- $\forall i < r, \quad n'_i = n_{i+1}$

Chaque arête part du nœud précédent.

- $\forall i \leq r, \quad W_i \subset \bigcup_{j=0}^i \mathcal{C}(n_j)$

Les clefs nécessaires à chaque verrou ont déjà été prises.

On dit qu'une route est *gagnante* si elle vérifie également la condition $n'_r = F$, c'est-à-dire si le dernier nœud visité est le nœud de fin de jeu. Un jeu est dit *satisfiable* s'il admet au moins une route gagnante. Dans le cas contraire, il est dit *insatisfiable*.

Pour une permutation $\sigma \in \mathfrak{S}_k$ (l'ensemble des permutations de $\llbracket 0, k-1 \rrbracket$) fixée, on appelle *randomisé* de $\mathcal{G} = (\mathcal{S}, \mathcal{C}, \mathcal{A})$ par σ le nouveau jeu défini par $\mathcal{G}_\sigma = (\mathcal{S}, \mathcal{C}_\sigma, \mathcal{A})$ où \mathcal{C}_σ est définie par :

$$\forall n \in \mathcal{S}, \quad \mathcal{C}_\sigma(n) = \{C_{\sigma(i)} \mid C_i \in \mathcal{C}(n)\}.$$

Intuitivement, il s'agit du même jeu mais dans lequel les indices des clefs présentes sont passés de i à $\sigma(i)$. Les verrous eux ne changent pas. Une *randomisation* est une opération qui prend un jeu et une permutation et renvoie le jeu randomisé associé.

La première partie porte sur la recherche de randomisations qui définissent un jeu randomisé satisfiable, en utilisant un système de règles de déduction. Une fois un jeu satisfiable généré, la seconde partie s'intéresse au temps minimal nécessaire pour le résoudre. Enfin, une troisième partie plus courte s'intéressera à une variante du jeu et à sa complexité. Les deux premières parties sont indépendantes mais il est conseillé de traiter le sujet dans l'ordre.

1.3 Description des fichiers fournis

Ce sujet est accompagné de deux squelettes de code :

- un fichier `randomizer.c` qui est à compléter pour la partie 2 ;
- un fichier `route_optimale.ml` qui est à compléter pour la partie 3 ;

2 Satisfiabilité d'un jeu randomisé (en C)

L'objectif de cette partie est de pouvoir générer des randomisations et tester la satisfiabilité du jeu randomisé obtenu, afin de proposer au joueur une version randomisée dans laquelle il reste possible de gagner.

▷ **Question 1.** 🚩 Proposer à l'oral une version randomisée du jeu donné en exemple qui est insatisfiable, et une autre qui est satisfiable mais différente du jeu initial.



Cette première question permettait de s'assurer que la formalisation du problème était bien comprise. Le jury a pu demander aux candidats d'écrire au brouillon les permutations proposées à l'oral et de montrer une route gagnante sur le schéma. Il a également été demandé de préciser quelles étaient les clefs présentes après la randomisation proposée, afin de s'assurer que les candidats aient bien compris que deux clefs de même indice restaient de même indice après la randomisation.

2.1 Génération d'une permutation

On s'intéresse ici à la génération d'une permutation $\sigma \in \mathfrak{S}_k$. Une telle permutation sera implémentée en C par un tableau T de type `int*` et de taille k , et tel que $\forall i \in \llbracket 0, k-1 \rrbracket$, $T[i] = \sigma(i)$.

▷ **Question 2.** 📖 Implémenter une fonction de prototype `void init_i(int* tableau, int k)` qui initialise un tableau d'entiers de taille k passé en argument, en plaçant l'entier i dans chaque case d'indice i .

▷ **Question 3.** 📖 Implémenter une fonction de prototype `void pp_permutation(int* permutation, int k)` qui permet d'afficher une permutation.



Pour ces deux questions, le jury a pu demander aux candidats de tester leurs fonctions si cela n'avait pas été fait automatiquement. Les candidats n'ayant pas réalisé de tests au passage suivant ont été pénalisés. En revanche, un test simple était satisfaisant.

On considère un algorithme qui parcourt les indices i du tableau par ordre croissant, et pour chaque indice échange les éléments $T[i]$ et $T[s]$ pour s un indice aléatoire tiré uniformément entre 0 et i inclus. Cet algorithme permet d'appliquer une permutation aléatoire σ tirée uniformément dans \mathfrak{S}_k aux éléments du tableau T .

▷ **Question 4.** 🚩 Donner un invariant de boucle permettant de prouver ce dernier point. L'algorithme reste-t-il correct en tirant s entre 0 et $i-1$ inclus à la place? Justifier.




Dans cette question il n'était pas attendu une preuve complète et rigoureuse, mais uniquement d'énoncer un invariant de boucle. Les candidats étaient accompagnés pour exhiber un invariant correct. Dans le cas des bons candidats qui avaient

immédiatement proposé un invariant correct, il a pu être demandé à l'oral comment ce dernier pourrait être prouvé, sans toutefois rentrer dans les détails.


Les rares candidats pensant que l'algorithme restait correct en tirant s entre 0 et $i - 1$ inclus ont été accompagnés grâce à des contre-exemples donnés par le jury. Dans le cas des bons candidats ayant immédiatement justifié ce point de manière correcte, le jury a pu demander un exemple concret de permutation qui ne pouvait pas être engendrée par cet algorithme modifié.

Il est possible de générer un entier (pseudo-)aléatoire entre 0 et k inclus en C en écrivant `rand() % (k+1)`.

▷ **Question 5.**  Implémenter une fonction de prototype `void melange(int* tableau, int k)` qui prend un tableau d'entiers et qui applique cet algorithme pour obtenir une permutation aléatoire.



Cette question a été très bien réussie dans l'ensemble. Certains candidats ont présenté des difficultés pour échanger deux éléments dans un tableau. Ces erreurs n'étaient en général pas détectées car elles provenaient de candidats ne testant pas leurs fonctions. Bien que cela ait été pénalisé, le jury a pris soin d'aiguiller les candidats dès que possible pour que cette erreur ne soit pas problématique pour la suite.

▷ **Question 6.**  Comment pourrait-on remettre les éléments d'un tableau quelconque dans l'ordre ? Citer deux algorithmes permettant de réaliser cette tâche, expliquer brièvement leurs fonctionnements et donner leurs complexités.



La discussion avec les candidats s'est ouverte à partir des diverses propositions faites. À partir du principe général de ces algorithmes de tri, il a été demandé de développer ou préciser certains aspects.

Par exemple : présenter l'opération de fusion du tri fusion, donner un exemple de pire cas pour le tri rapide, expliquer l'implémentation d'un tas et les complexités des opérations associées, justifier la complexité du tri à bulle ou du tri fusion, rappeler comment le tableau est coupé en deux dans le tri fusion, et éventuellement discuter de l'impact d'une coupe ailleurs qu'au milieu, donner les cas de bases des algorithmes de tri récursifs, etc.

Certains candidats ont confondu les noms de différents algorithmes de tri (notamment le tri insertion et le tri sélection). Toutefois les candidats qui ne se rappelaient pas du nom de l'algorithme proposé n'ont pas été pénalisés si par ailleurs ils se montraient capables de répondre clairement aux questions et de détailler avec précision l'algorithme en question.

Il n'est évidemment pas demandé de maîtriser l'ensemble de ces algorithmes, le tri par tas étant le seul au programme, mais le jury s'attend à ce que les candidats aient rencontré au moins un autre algorithme de tri et soient capables de l'expliquer.

Une corrélation importante a été observée entre la note finale du candidat et son recul lors de cette discussion.

2.2 Règles de déduction

On cherche désormais à savoir si un jeu est satisfiable ou non. Pour cela, on commence par numéroter les nœuds de \mathcal{S} avec un indice i . On représente le fait qu'il existe une route atteignant le nœud i à partir du nœud D par la proposition N_i , et le fait qu'il existe une route permettant de ramasser la clef C_i à partir du nœud D par la proposition O_i . En particulier on note N_D et N_F les propositions associées au nœud de départ et au nœud de fin.

Ces deux types de propositions N_i et O_i sont implémentées en C par la structure `prop`.

```
struct prop_ {
    bool N; // Booléen indiquant si c'est une proposition N_i ou O_i
    int i; // Indice i associé à la proposition
};
typedef struct prop_ prop;
```

Nous allons formaliser les dépendances du jeu comme un ensemble de règles d'inférences de la forme $\mathcal{P}_1 \wedge \dots \wedge \mathcal{P}_q \rightarrow \mathcal{Q}$, dans laquelle $\mathcal{P}_1, \dots, \mathcal{P}_q$ et \mathcal{Q} sont des propositions, $\mathcal{P}_1 \wedge \dots \wedge \mathcal{P}_q$ est appelée la *prémisse* et \mathcal{Q} est appelée la *conclusion*. Une telle règle d'inférence est représentée par :

$$\frac{\mathcal{P}_1 \quad \mathcal{P}_2 \quad \dots \quad \mathcal{P}_q}{\mathcal{Q}}$$

Intuitivement, si $\mathcal{P}_1, \dots, \mathcal{P}_q$ ont été déduites, alors \mathcal{Q} peut être déduite. Il est alors possible de modéliser une instance du jeu par un ensemble bien choisi de règles. En particulier nous nous intéressons ici à deux types de règles. Les premières modélisent le fait que si on se trouve dans un nœud donné, alors on peut récupérer une clef présente sur ce nœud. Les deuxièmes représentent le fait qu'une arête puisse être franchie. En tout il faudra définir une règle pour chaque clef dans un nœud, et deux règles pour chaque arête, car ces dernières peuvent s'emprunter dans les deux sens. La règle axiomatique sans prémisse et de conclusion N_D ne sera pas représentée explicitement dans l'ensemble des règles.

▷ **Question 7.** ⓘ Définir formellement ces deux types de règles, dans le cas d'une instance du jeu arbitraire. Donner le nom d'un autre système formel de règles connu.



Cette question a été problématique pour plusieurs candidats qui semblaient fragiles sur la notion de règles de déduction. L'objectif de cette question était avant tout d'éviter qu'un candidat soit bloqué à cause d'une mauvaise compréhension du sujet.

Bien que la déduction naturelle soit le système de règles au programme attendu ici, le jury a volontiers accepté tout système de règles proposé, tant que le candidat ait semblé à l'aise dans sa description. Il n'était pas attendu de réciter par cœur toutes les règles. Pour les candidats qui semblaient à l'aise, le jury a pu demander de justifier l'intérêt d'un tel système de règles, afin de tester leur recul scientifique. Une justification brève, comme « prouver la validité d'une formule sans vérifier exhaustivement une table de vérité » était complètement satisfaisante.

▷ **Question 8.** ✎ Écrire un arbre de preuve complet en utilisant vos règles définies pour le jeu en figure 1, dont la racine tout en bas qui représente la conclusion correspond à la proposition associée au nœud de fin. Le jeu est-il bien satisfiable ?



Cette question a été bien réussie dans l'ensemble, même si certains candidats ont eu besoin d'un petit rappel.

En C, ces règles sont implémentées par la structure suivante, formant une liste simplement chaînée :

```
struct regle_ {  
    struct regle_* suivant; // Pointeur vers la règle suivante  
    prop* premisses; // Tableau des propositions requises dans la prémisse  
    int taille_premisse; // Nombre de propositions dans la prémisse  
    prop conclusion; // Proposition déduite dans la conclusion  
};  
typedef struct regle_ regle;
```

Le fichier `randomizer.c` contient une fonction `libere_regles` qui permet de libérer cette structure de données, que vous serez libre d'utiliser.

▷ **Question 9.** 📖 Implémenter des fonctions de prototype `regle* ajoute_regles_clefs(regle* r, int n1, int* clefs, int nb_clefs)` et `regle* ajoute_regles_arete(regle* r, int n1, int n2, int* verr, int nb_verr)` qui permettent d'allouer et d'ajouter des règles basées sur l'instance du jeu considéré, en suivant le modèle proposé à la **Question 7**. La première fonction ajoute des règles du premier type qui expriment la présence de clefs d'indices contenues dans le tableau `clefs`, sur le nœud d'indice `n1`. La seconde ajoute deux règles du second type qui représentent une arête entre deux nœuds d'indices `n1` et `n2`, et dont les indices des verrous nécessaires sont fournis dans le tableau `verr`. Dans les deux cas, ces fonctions doivent renvoyer un pointeur vers la nouvelle liste de règles obtenue.



Cette question a été problématique pour de nombreux candidats et très discriminante sur le temps nécessaire pour la traiter. Malgré la formulation explicite de la question, certains candidats n'ont initialement pas utilisé `malloc`, et le jury a pu demander de détailler l'organisation mémoire d'un processus pour s'assurer que ces notions soient acquises. Le jury a également observé que la structure de liste chaînée en C et notamment l'ajout d'un élément posait souvent problème. Plusieurs candidats parcouraient toute la liste pour ajouter le nouvel élément à la fin, sans même faire attention au cas où la liste était vide ce qui provoquait pourtant une erreur qui aurait pu être détectée rapidement. De même, certains candidats se sont confrontés à des erreurs « double-free ». Le jury a pu alors discuter avec le candidat, d'abord en demandant ce que signifiait cette erreur, puis en demandant d'où celle-ci pouvait provenir. Les candidats qui étaient réceptifs aux pistes du jury ont pu rapidement réagir et corriger ces erreurs avant le passage suivant. Le jury rappelle que l'autonomie du candidat et notamment sa démarche de débogage sont

valorisées de manière importante dans cette épreuve. Il est préférable de voir un candidat activement faire face à ses erreurs qu'un candidat qui attend passivement le prochain passage du jury ou qui saute la question.

Le fichier `randomizer.c` contient une implémentation du jeu de la figure 1 ainsi que du jeu plus complexe figurant en annexe, à décommenter.

▷ **Question 10.** ☞ Implémenter une fonction qui permet d'afficher la liste des règles ainsi définies, et la tester sur les règles définissant le jeu de la figure 1 et le jeu en annexe.



Cette question permettait de s'assurer que les règles étaient correctement ajoutées. Le jury a pris soin de vérifier l'affichage obtenu. Certains rares candidats affichaient des règles erronées mais ne s'en étaient pas rendu compte avant le passage du jury. Pour une grande partie de candidats auxquels il restait moins de la moitié du temps après avoir fini cette question, le jury a demandé de passer à la partie OCAML, après avoir éventuellement traité rapidement les questions 11 et 12.

Nous cherchons désormais à utiliser ces règles pour savoir si le jeu est satisfiable ou non. Nous allons déterminer l'ensemble des propositions qui peuvent être déduites à partir de N_D en utilisant ces règles, en les stockant dans deux tableaux de booléens `noeuds` et `clefs` dont l'élément d'indice i indique si N_i (respectivement O_i) a pu être déduit avec ces règles. Au départ, aucune proposition n'est déduite, mis à part N_D qui est un axiome.

▷ **Question 11.** ☞ Implémenter une fonction de prototype `void init_faux(bool* tableau, int taille_tableau)` qui initialise toutes les cases d'un tableau de booléens à faux.

Nous allons itérativement déduire de nouvelles propositions en parcourant les règles à la recherche d'une prémisse satisfaite. Dès lors, on peut appliquer cette règle, c'est-à-dire marquer sa conclusion à vrai dans le bon tableau des propositions déduites. On peut alors supprimer cette règle. Ces nouvelles déductions pourront à leur tour permettre de vérifier d'autres prémisses et donc d'obtenir de nouvelles conclusions à partir de la simple hypothèse de départ que le nœud initial est atteint.

▷ **Question 12.** ☞ Implémenter une fonction de prototype `bool verifie_premisse(regle r, bool* noeuds, bool* clefs)` qui indique si la prémisse d'une règle est satisfaite, c'est-à-dire si toutes les propositions dans la prémisse ont pu être déduites.



Ces deux questions n'ont en général pas posé problème.

▷ **Question 13.** ☞ Implémenter une fonction récursive de prototype `regle* parcours_regles(regle* r, bool* noeuds, bool* clefs, bool* progres)` qui parcourt la liste des règles, et pour chaque règle vérifie si la prémisse est satisfaite. Si c'est le cas, elle met à jour les tableaux `noeuds` et `clefs` avec la conclusion, passe à vrai la variable booléenne pointée par `progres`, signifiant qu'au moins une règle a été appliquée, puis libère la règle. Cette fonction renvoie un pointeur vers la liste de règles ainsi modifiée. Il est possible de parcourir les règles dans n'importe quel ordre.



Les candidats qui ont correctement traité cette question et les questions suivantes étaient des candidats qui ont su être rapides et efficaces sur les questions précédentes. En particulier, presque tous les candidats ayant traité la question 14 ont réussi à terminer intégralement cette première partie du sujet dans un temps raisonnable. D'autres candidats ont passé un certain temps sur la question 13 avant que le jury ne conseille de passer à la partie OCAML. En revanche le jury a pris en compte les tentatives d'implémentation même si la fonction n'était pas complètement aboutie ou si elle était incorrecte.

Pour déterminer l'ensemble des variables qui peuvent être déduites, nous allons parcourir les règles tant que chaque parcours permet au moins de satisfaire une prémisse. On admettra que si la variable associée au nœud de fin de jeu n'a pas pu être déduite, alors le jeu est insatisfiable.

▷ **Question 14.** Implémenter une fonction de prototype

```
bool est_sat(regle* r, int nb_noeuds, int nb_clefs, int D, int F)
```

qui prend un jeu décrit par des règles `r`, qui possède `nb_noeuds` nœuds et `nb_clefs` clefs, et dont les nœuds de début et de fin sont respectivement d'indices `D` et `F`, et qui renvoie un booléen indiquant si ce jeu est satisfiable. C'est cette fonction qui sera désormais responsable de libérer la mémoire allouée par les règles restantes.

▷ **Question 15.** Quelle est la complexité de cette fonction ? Prouver sa terminaison.

▷ **Question 16.** Modifier vos fonctions pour qu'elles prennent en argument une permutation et que le tableau `clefs` soit mis à jour en utilisant l'indice $\sigma(i)$ au lieu de l'indice i . Cela revient à travailler sur la version randomisée du jeu. La liste des règles ne doit pas être modifiée.



Ces trois questions n'ont pas été problématiques pour les candidats qui y sont parvenus. Le jury a apprécié les candidats qui décrivaient leurs fonctions de manière claire et concise, sans rentrer dans des détails inutiles mais en faisant part des subtilités rencontrées. Le recul scientifique de ces candidats a permis des interactions riches et rigoureuses qui ont été valorisées par le jury.

▷ **Question 17.** Tester la fonction `est_sat` sur des randomisations du jeu de la figure 1. Les résultats obtenus sont-ils cohérents ?

▷ **Question 18.** Indiquer pour les cinq permutations définies dans `randomizer.c` si le randomisé du jeu en annexe par cette permutation est satisfiable ou non. Tester sur d'autres permutations tirées au hasard et noter vos résultats.



Ces deux dernières questions permettaient de tester les fonctions implémentées jusque-là et de récompenser les candidats efficaces ayant réussi à terminer intégralement cette partie. Le jury a apprécié le réflexe de reprendre les résultats de la question 1 pour implémenter des tests à la question 17, notamment de tester des randomisations satisfiables et insatisfiables. Les bonnes pratiques de code et notamment

les tests fréquents au cours du sujet (même lorsqu'ils n'étaient pas explicitement demandés) ont permis aux candidats ayant atteint cette question de passer ces derniers tests du premier coup et de rapidement enchaîner sur la suite du sujet.

3 Route optimale (en OCaml)

Dans cette partie on cherche à résoudre le plus rapidement possible un jeu supposé satisfiable. Pour cela, on introduit une notion de temporalité à notre problème. Désormais, \mathcal{C} est une application $\mathcal{C} : \mathcal{S} \rightarrow \mathcal{P}(K \times \mathbb{N})$ qui associe à un nœud un ensemble de couples (C_i, t_i) avec $t_i \in \mathbb{N}$ le temps nécessaire pour ramasser cette clef dans ce lieu. En ce qui concerne les arêtes, ce sont désormais des quadruplets $a = (n, W, t, n') \in \mathcal{S} \times \mathcal{P}(K) \times \mathbb{N}^* \times \mathcal{S}$ dans lequel on a rajouté un temps strictement positif t nécessaire pour franchir cette arête. Ainsi, en arrivant sur un nœud n , il est désormais possible de choisir quel sous-ensemble de clefs $K_n = \{C_{i_1}, \dots, C_{i_q}\}$ on ramasse, ce qui prend un temps $\sum_{C_i \in K_n} t_i$.

Le temps d'une route est la somme des temps des arêtes franchies et des temps mis à ramasser des clefs. On dit qu'une route est *optimale* si c'est une route gagnante de temps minimal. Une randomisation ne modifie pas le temps de la clef : la permutation ne s'applique qu'à la clef elle-même mais pas à son temps t_i associé qui lui reste identique. Ainsi, les clefs présentes dans un nœud auront toujours les même temps de ramassage, peu importe la randomisation.

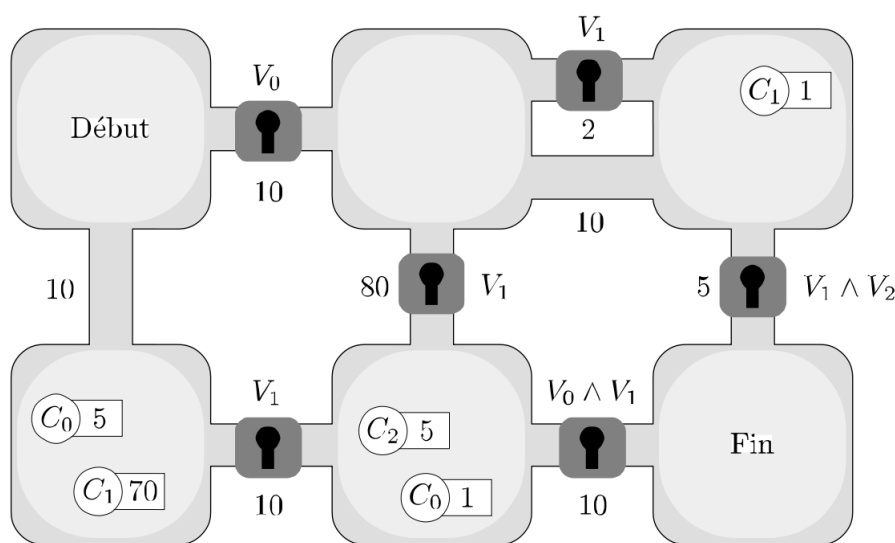


FIGURE 2 – Désormais, les clefs sont étiquetées par le temps nécessaire pour les récupérer, et les arêtes sont également étiquetées par le temps nécessaire pour les emprunter. Deux clefs de même indice peuvent avoir des temps de ramassage différents dans deux nœuds différents.

Commençons par définir la notion d'*état*, qui symbolise un joueur en pleine partie. Un état est constitué de trois éléments :

- Un nœud, correspondant à la position actuelle dans le jeu,

- Un temps, correspondant au temps écoulé depuis le début du jeu,
- Un *inventaire*, correspondant aux clefs rencontrés sur la route menant à cet état, et le temps minimal qu'il faut dépenser pour obtenir ces clefs. Si ce temps vaut **None**, c'est que cette clef n'a jamais été rencontrée. S'il vaut **Some** 0, c'est que la clef a déjà été ramassée, il n'y a pas besoin de dépenser du temps supplémentaire pour qu'elle soit disponible car elle l'est déjà. Si ce temps vaut une autre valeur **Some** t , cela correspond au temps t_i minimum pour la ramasser rencontré jusqu'à cet état. La clef C_i aurait donc pu être disponible en payant le temps t_i , et on peut l'obtenir en payant ce temps t_i , signifiant qu'elle avait en fait été ramassée auparavant, dans le nœud où cela coûtait t_i de la ramasser.

Afin de trouver une route optimale dans un jeu, nous allons parcourir les états accessibles depuis l'état initial à la recherche de celui qui se trouve dans le nœud final et qui possède le temps le plus petit. Pour cela nous allons récursivement explorer les états accessibles en visitant prioritairement ceux de temps minimal.

▷ **Question 19.** ✎ Écrire de manière informelle en quoi consiste l'état initial dans le jeu 1, et une suite d'états qui permet de franchir un verrou.



Cette première question permettait de s'assurer que la notion d'état était bien assimilée pour ne pas être bloqué par la suite. Le jury a été bienveillant et a pu poser quelques questions s'il a senti que les définitions n'étaient pas bien comprises, mais cela a été rarement le cas.

▷ **Question 20.** ⓘ Citer un algorithme qui utilise une priorité particulière pour explorer un graphe. Détailler brièvement son fonctionnement et donner sa complexité.



L'algorithme A* a aussi été accepté. Certains rares candidats n'ont pas pas su répondre à cette question du tout, malgré des indications du jury. Lorsqu'un candidat mentionnait un simple parcours en profondeur ou en largeur, le jury prenait soin de préciser la question pour guider vers l'algorithme de Dijkstra. La complexité de l'algorithme de Dijkstra a été presque tout le temps incorrecte, même pour les candidats qui semblaient très à l'aise jusque-là. Le jury a demandé de vérifier la complexité à l'oral en détaillant chaque étape de l'algorithme. Ceux qui ont su retrouver la bonne complexité rapidement n'ont été que peu pénalisés, mais ce n'était pas le cas de la majorité des candidats.

Le temps écoulé sera représenté par un entier, et un nœud sera représenté par un enregistrement dont les champs représentent l'identifiant du nœud, une liste d'indices de clefs et de temps de ramassage associé, et un booléen indiquant si ce nœud est le nœud de fin ou non. On rappelle qu'il est possible d'accéder aux champs de cet enregistrement avec `noeud.id`, `noeud.clefs` et `noeud.est_final`, et de créer un tel enregistrement avec la syntaxe `{id=...; clefs=...; est_final=...}`.

```
type temps = int
type noeud = {id : int ; clefs : ((int * temps) list) ; est_final : bool}
```

Un verrou sera simplement une liste des indices des clefs du verrou, et le multi-graphe correspondant à l'instance du jeu sera un tableau d'adjacence dans lequel la case i contient la liste des arêtes sortantes du nœud d'indice i . Une telle arête est de type `temps * verrou * noeud`, représentant respectivement le temps pris pour emprunter cette arête, son verrou et le nœud d'arrivée.

```
type verrou = int list
type mgraphe = (temps * verrou * noeud) list array
```

Un inventaire comme défini plus haut sera implémenté par un tableau dont les éléments sont de type `int option`. Un état sera un enregistrement dont les champs correspondent respectivement au nœud de l'état, au temps écoulé depuis le début, et à l'inventaire de l'état.

```
type inventaire = int option array
type etat = {noeud : noeud; temps : temps; inventaire : inventaire}
```

Le jeu de la figure 1 et le jeu figurant en annexe sont implémentés dans le fichier `route_optimale.ml`, et ils pourront être utilisés pour tester vos fonctions.

▷ **Question 21.** ☞ Implémenter une fonction

`mise_a_jour_inventaire : (int * temps) list -> inventaire -> unit` qui prend une liste d'indices de clefs ramassables avec leurs temps associés, et met à jour l'inventaire en gardant, pour chaque clef de la liste, le nouveau temps minimal associé à cette clef.

▷ **Question 22.** ☞ Implémenter une fonction

`verrou_franchissable : verrou -> inventaire -> bool` qui vérifie s'il est possible de franchir un verrou, c'est-à-dire si aucune clef nécessaire n'a pour valeur `None` dans l'inventaire.



Ces deux questions ont été bien réussies, en revanche elles ont été peu testées. Il y avait par ailleurs une erreur de type dans la signature de ces deux fonctions (les signatures données étaient `int list -> inventaire -> unit` pour l'une et `verrou -> inventaire -> temps` pour l'autre) qui ne semble pas avoir posé de problème pour l'équité de l'évaluation. La très grande majorité des candidats corrigeaient cette erreur et la signalait au jury lors de son passage ; les autres candidats interpellaient respectueusement le jury pour vérifier qu'ils avaient bien compris la question.

▷ **Question 23.** ☞ Implémenter une fonction

`prendre_arete : verrou -> inventaire -> temps` qui calcule le temps total minimal qu'il faut passer à ramasser des clefs pour déverrouiller le verrou d'après l'inventaire actuel, et passe à 0 les temps de ces clefs dans l'inventaire. Cette fonction suppose que le verrou considéré est bien franchissable.

▷ **Question 24.** ☞ Implémenter une fonction

`voisins : etat -> mgraphe -> etat list` qui à partir d'un état donné, renvoie la liste de tous les états voisins, en prenant bien soin de mettre à jour les temps et inventaires de chacun de ces états. Il sera possible d'utiliser la fonction

`Array.copy : 'a array -> 'a array` qui permet de réaliser la copie d'un tableau.



Cette dernière question possède une subtilité. Il faut obligatoirement mettre à jour le temps du nouvel état avant de mettre à jour son inventaire, sinon on peut ouvrir un verrou avec une clef apparaissant dans le nœud de destination.

3.1 File de priorité

▷ **Question 25.** ⓘ Présenter différentes implémentations d'une structure de file de priorité, ainsi que les complexités associées à leurs opérations élémentaires d'insertion d'un élément, de recherche d'un élément de priorité minimal et d'extraction d'un tel élément. Quelle implémentation vous semble pertinente ici et pourquoi ?

▷ **Question 26.** 📖 Implémenter une file de priorité de type `t` pour stocker les états en attente d'exploration, dont la priorité est donnée par le temps de l'état. Votre structure de données devra permettre d'insérer un état, d'accéder rapidement à l'état le plus prioritaire et de l'extraire. En fonction du temps qu'il vous reste, vous pouvez choisir d'implémenter une structure de données plus ou moins optimisée.

▷ **Question 27.** ⚡ Noter l'approche que vous avez retenue, ainsi que les complexités des fonctions que vous avez implémentées.



Peu de candidats ont pu correctement traiter cette partie, par manque de temps.

3.2 Recherche de route optimale



Aucun candidat n'a traité les questions suivantes. Pour cette première session, le jury a proposé par prudence des sujets qui pouvaient être parfois trop longs mais cela a été pris en compte dans le barème.

▷ **Question 28.** 📖 Implémenter une fonction qui permet d'explorer les états accessibles à partir d'un état initial, en priorisant les états de temps les plus faibles. Cette fonction doit renvoyer le premier état final rencontré en suivant cette approche.

▷ **Question 29.** 📖 Combien de temps faut-il pour résoudre le jeu de la figure 1 en suivant une route optimale ?

Sur des jeux de grandes tailles, il peut vite devenir problématique de remplir la file de priorité avec des états non intéressants (par exemple des allers-retours entre deux nœuds). En particulier, il est inutile d'explorer un état ayant le même nœud et inventaire qu'un état déjà visité.

▷ **Question 30.** 📖 Définir une nouvelle fonction d'exploration qui résout ce problème.

▷ **Question 31.** ⚡ Rédiger un résumé en quelques lignes de votre approche, ainsi que les fonctions que vous avez définies et les modifications apportées. Existe-t-il d'autres alternatives ?

▷ **Question 32.** ⓘ Vérifier que votre approche permet de trouver une route optimale en un temps raisonnable sur l'instance du jeu définie en annexe.

4 Complexité

Dans cette dernière partie, on s'intéresse à une variante du jeu dans laquelle certaines clefs sont consommées lors de leur utilisation, c'est-à-dire qu'elles disparaissent de l'inventaire une fois qu'elles ont été utilisées pour ouvrir un verrou. En revanche tout verrou ouvert le reste jusqu'à la fin du jeu.

▷ **Question 33.** ♣ Est-il possible d'adapter les algorithmes proposés au cours de ce sujet pour traiter cette variante ? Justifier.

Nous allons montrer dans cette partie que cette petite modification rend le problème de satisfiabilité d'un jeu NP-complet. Pour cela, nous supposons que le problème suivant nommé CNF-SAT est NP-complet :

ENTRÉE

Une formule de la logique propositionnelle, mise sous Forme Normale Conjonctive :

$$\varphi = \bigwedge_i \bigvee_j l_{i,j} \quad \text{avec } l_{i,j} \text{ un littéral qui représente une variable ou sa négation.}$$

SORTIE

OUI si φ est satisfiable, NON sinon.

▷ **Question 34.** ♣ Citer deux autres problèmes qui appartiennent à cette classe de complexité.

La taille d'une telle formule φ est donnée par son nombre de connecteurs logiques, et la taille d'un jeu est donnée par :

$$|\mathcal{S}| + |\mathcal{A}| + \sum_{s \in \mathcal{S}} |\mathcal{C}(s)| + \sum_{a \in \mathcal{A}} |\mathcal{V}(a)| \quad (1)$$

où $\mathcal{V}(a)$ désigne l'ensemble des verrous de a .

Dans un premier temps, nous allons montrer que ce problème est NP-difficile.

▷ **Question 35.** ♣ Proposer une transformation en temps polynomial en la taille de l'instance, qui à toute formule φ sous CNF associe une instance du jeu qui est satisfiable si et seulement si φ est satisfiable. Donner des exemples de jeux ainsi construits à partir de quelques formules simples.

▷ **Question 36.** ♣ Démontrer que pour tout jeu satisfiable, il existe une route gagnante de taille polynomiale en la taille du jeu. Conclure.

5 Annexes

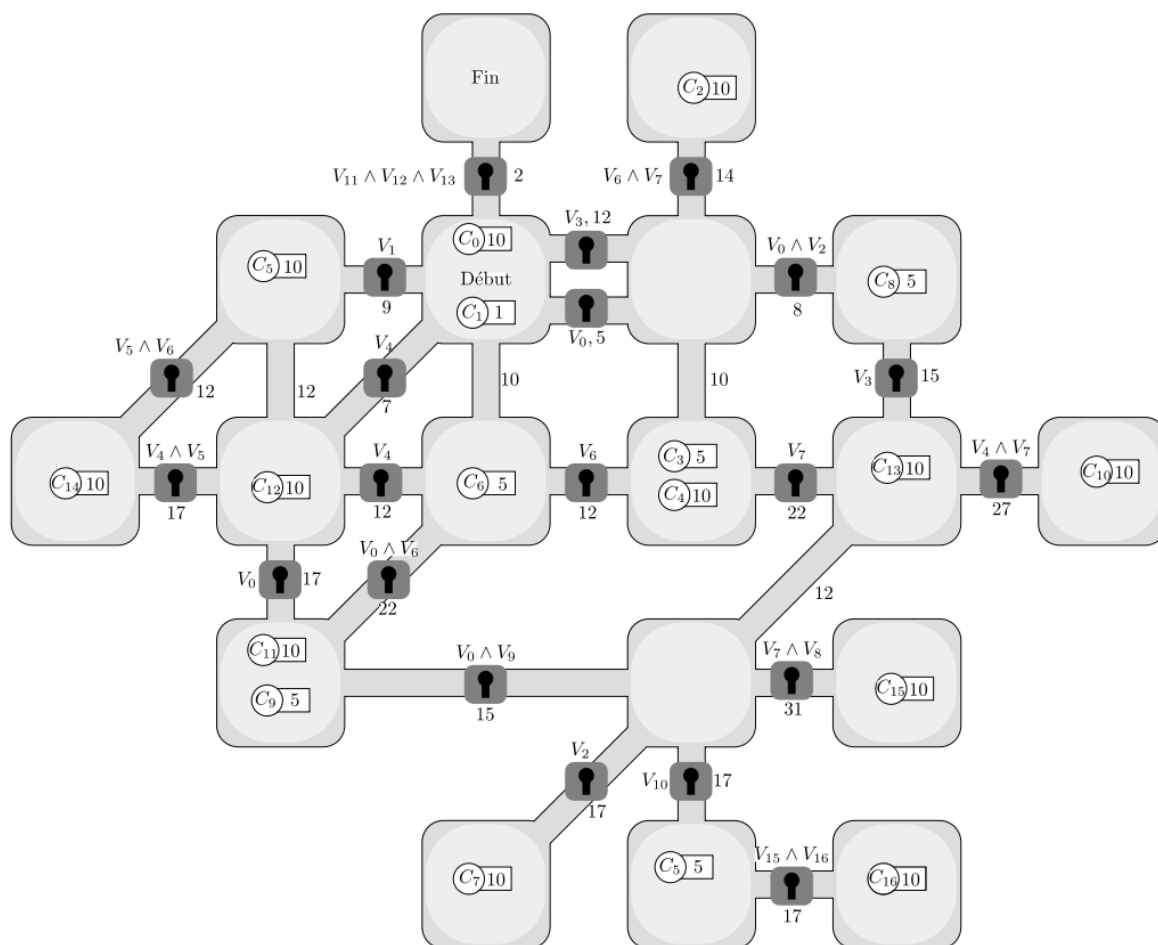


FIGURE 3 – Un deuxième exemple d'instance du jeu, inspirée d'un jeu réel. Cette instance est implémentée dans les fichiers `randomizer.c` et `route_optimale.ml`.