École Nationale Supérieure
d'Électrotechnique, d'Électronique, d'Informatique,
d'Hydraulique et des Télécommunications

Institut National Polytechnique

Rapport Projet Théorie des Graphes : Graphes et Donjons

Aymen Chla
Mohamed Maghfour

SOMMAIRE

Introduction **1**

1 Partie 1 : Les donjons à clés **2**
1.1 L'algorithme utilisé . . . 2
1.2 Les fonctions principales . . . 3
1.2.1 La fonction *v_reach* . . . 3
1.2.2 La fonction *cles_accessibles* . . . 3
1.2.3 La fonction *ouvre_porte* . . . 4
1.2.4 La fonction *faisable* . . . 5
1.2.5 La fonction *parcours_salles_principales* . . . 5
1.2.6 La fonction *parcours* . . . 6
1.3 Résultats . . . 7
1.3.1 Test sur un donjon à clés faisable : . . . 7
1.3.2 Test sur un donjon à clés non-faisable : . . . 8

2 Partie 2 : Les donjons à interrupteurs **9**
2.1 L'algorithme utilisé . . . 9
2.2 Mise en place de la solution . . . 10
2.2.1 Les combinaisons . . . 10
2.2.2 Génération des sous graphes . . . 10
2.2.3 Dessiner le graphe global . . . 10
2.2.4 Vérification de le faisabilité du donjon . . . 11
2.3 Résultats . . . 12
2.3.1 Test sur un donjon à interrupteurs faisable : . . . 12
2.3.2 Test sur un donjon à clés non-faisable : . . . 13
2.3.3 Test sur un donjon à interrupteurs non-faisable : . . . 13

Conclusion **14**

Table des figures

1 le donjon faisable 7
2 test partie 1 sur le donjon faisable 7
3 le donjon non-faisable 8
4 test partie 1 sur le donjon non-faisable 8
5 le donjon faisable 12
6 test partie 2 sur le donjon faisable 12
7 le donjon non-faisable 13
8 test partie 2 sur le donjon non-faisable 13

Introduction

Dans ce projet, on était demandé de contribuer dans la création d'un jeu vidéo. Le concept du jeu en général c'est que le joueur commence dans une place (salle initiale), et il doit parcourir le donjon pour trouver le trésor caché. les passages se font d'une salle à une autre. les salles se sont connectés par des portes de différents types (pratiquable ou non). Des salles peuvent contenir des clés (avec une certaine couleur) pour déverrouiller les couloirs fermés(de la même couleur). Notre tâche est donc de réaliser un programme qui vérifie que les donjons sont faisables pour éviter le blocage du jeu.

1 Partie 1 : Les donjons à clés

La première partie consiste à vérifier la faisabilité des donjons à clés. en prenant comme entrée notre donjon, la salle initiale, la salle finale et les clés, notre programme doit être capable décider s'il existe une solution pour aller de la salle initiale vers la salle finale en consommant les clés.

1.1 L'algorithme utilisé

On va modéliser le problème de donjon sous forme d'un graphe. les salles du donjon sont les sommets et les couloirs sont des arêtes menant d'une salle à une autre. Chaque élément de la liste des clés est de la forme d'un couple de la forme (sommet où se trouve la clé , Liste des arcs à ajouter dans le graphe).
Grâce à cette modélisation, le problème des donjon sera plus simple à résoudre.
Pour plus de clarté, on présentera les algorithmes adopté dans notre implantation de la solution.

L'algorithme suivi pour résoudre le problème de donjons à clés est le suivant :

Algorithme 1 Donjon faisable ou non

Entrée: $graphe, sommetInitial, sommetFinal, cls$
Sortie: $faisable\ ou\ non - faisable$

```
tant que true faire
  si ExisteChemin de sommetInitial → sommetFinal alors
    faisable
  sinon
    si ExisteCls atteignables alors
      mettreAjourgraphe
    sinon
      non − faisable
    fin si
  fin si
fin tant que
```

1.2 Les fonctions principales

Dans cette partie, on s'est servi du module Pack.Graph (ocamlgraph) pour réaliser les fonction décrites ci-dessous. Ce module contient plusieurs fonctions, des itérateurs prédéfinis qui vont faciliter la manipulation des graphes. On va décomposer notre problème de départ en plusieurs sous problèmes pour faciliter le traitement.

1.2.1 La fonction *v_reach*

Cette fonction est la base de notre programme. En effet, la fonction *v_reach* prend en paramètre un graphe g et un sommet s, et elle renvoie la liste des sommets atteignables à partir de s dans g.

1.2.2 La fonction *cles_accessibles*

Cette fonction prend en paramètre le graphe g, le sommet vi et une liste des clés (liste de couples), et elle renvoie la liste des clés accessibles à partir de notre sommet vi.
le principe est le suivant :

Algorithme 2 cles_accessibles

Entrée: *graphe* g, *sommet* vi, liste des clés
Sortie: Liste des clés accessible *lc*
Début :
sommetsAccessibles ← *v_reach g vi*
tant que *la liste des* clés *n'est par termine* **faire**
prendre clé
si SommetClé existe dans *sommetsAccessibles* **alors**
Ajouter la clé dans lc
sinon
passer à la clé suivante
fin si
fin tant que
Retourne lc
FIN.

1.2.3 La fonction *ouvre_porte*

Cette fonction prend en paramètre un graphe g et une clé et elle retourne une copie de g avec ajout des arêtes contenues dans la clé.

Algorithme 3 *ouvre_porte*

Entrée: graphe *gi*, clé
Sortie: nouveau graphe g
Début :
g ← copy_graph *gi*
tant que la liste des arêtes de la clé n'est par terminée **faire**
Prendre arête
ajoute arête dans g
passer au prochaine arête
fin tant que
Retourne g
FIN.

1.2.4 La fonction *faisable*

La fonction faisable est elle qui va décider si notre graphe est faisable ou non. cette fonction prend en paramètre g vi et vf et renvoie un booléen true si le graphe est faisable, false sinon. Cette fonction récapitule toutes les fonctions définies précédemment.

Algorithme 4 *ouvre_porte*

Entrée: graphe gi, sommet Initial vi, sommet final vf, cles
Sortie: Boolean : $faisable$

```
Début :
g ← copy_graph gi
cle_utilise1 ← [ ]
cle_utilise2 ← [ ]
tant que true faire
  si vf ∈ v_reach g vi alors
    retourne True
  sinon
    cle_utilise1 ← cle_utilise2
    cle_utilise2 ← cles_accessibles g vi cles
  fin si
  si cle_utilise1 == cle_utilise2 alors
    retourne false
  fin si
fin tant que
FIN.
```

1.2.5 La fonction *parcours_salles_principales*

Cette fonction nous donne les points de contrôle principaux pour arriver à notre destination. Les salles principales sont l'entrée, la liste des salles successives où prendre les clés ,et la sortie. *parcours_salles_principales* prend en paramètre le graphe, sommet initial, sommet final et les clés. Cette fonction renvoie donc les salles principales par lesquelles passer dans l'ordre pour parvenir au trésor si le donjon et renvoie la liste vide [] sinon.

1.2.6 La fonction *parcours*

La fonction *parcours* renvoie la liste des salles à parcourir dans l'ordre pour atteindre le trésor. La fonction *parcours* sur la fait que les salles principales sont connexes, et pour chaque paire de salles principales successives, on y appliquera l'algorithme de Dijkstra (shortest_path) pour déterminer le chemin global.

Algorithme 5 *parcours*

Entrée: graphe g, sommet initia vi,sommet final vf, liste des clés
Sortie: l : Liste des sommets pour atteindre vf

```
Début :
l ← [ ]
listeSalles ← parcours_salles_principales g vi vf
tant que listeSalles n'est pas terminée faire
    prendre vp et vp+1 ∈ listeSalles successives
    l ← append(l,fst (shortest_path g vp vp+1))
fin tant que
Retourne l
FIN.
```

1.3 Résultats

1.3.1 Test sur un donjon à clés faisable :

La figure du donjon faisable sur lequel on a appliqué le test :

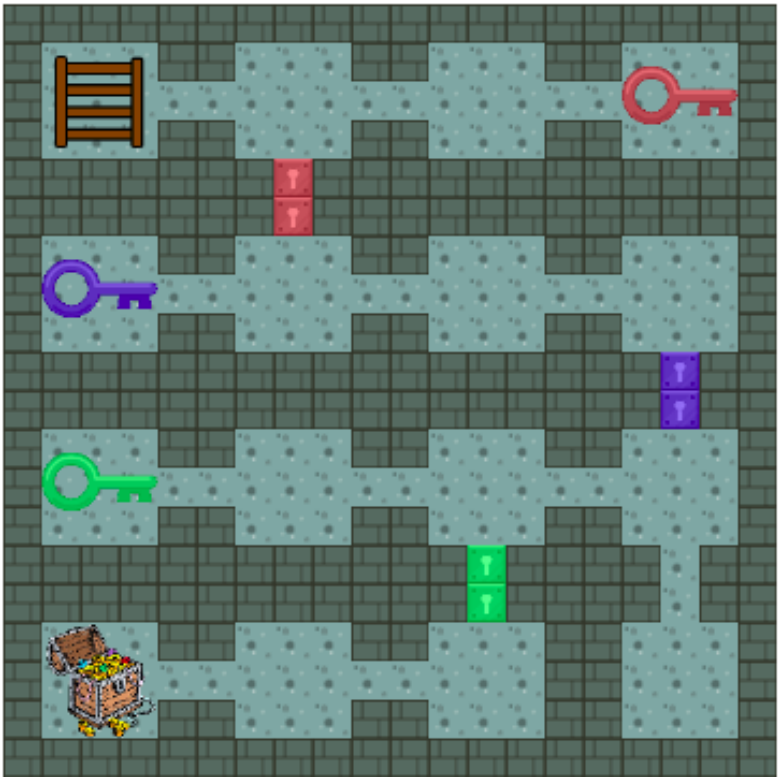

Figure 1 – le donjon faisable

Le résultat obtenu :

```
-( 22:28:50 )-< command 2 >-------------------------------------------------------------------{ counter: 0 }-
utop # Projet_donjon.faisable Projet_donjon__Test_c.dj Projet_donjon__Test_c.vi Projet_donjon__Test_c.vf Projet_donjon__
Test_c.doors;;
- : bool = true
-( 22:29:16 )-< command 3 >-------------------------------------------------------------------{ counter: 0 }-
utop # Projet_donjon.parcours_salles_principales Projet_donjon__Test_c.dj Projet_donjon__Test_c.vi Projet_donjon__Test_c
.vf Projet_donjon__Test_c.doors;;
- : Graph.vertex list = [v_11; v_14; v_21; v_31; v_41]
-( 22:29:18 )-< command 4 >-------------------------------------------------------------------{ counter: 0 }-
utop # Projet_donjon.parcours Projet_donjon__Test_c.dj Projet_donjon__Test_c.vi Projet_donjon__Test_c.vf Projet_donjon__
Test_c.doors;;
- : Graph.vertex list =
[v_11; v_12; v_13; v_14; v_13; v_12; v_22; v_21; v_22; v_23; v_24; v_34; v_33;
 v_32; v_31; v_32; v_33; v_43; v_42; v_41]
-( 22:29:31 )-< command 5 >-------------------------------------------------------------------{ counter: 0 }-
utop #
```

Figure 2 – test partie 1 sur le donjon faisable

1.3.2 Test sur un donjon à clés non-faisable :

La figure du donjon faisable sur lequel on a appliqué le test :

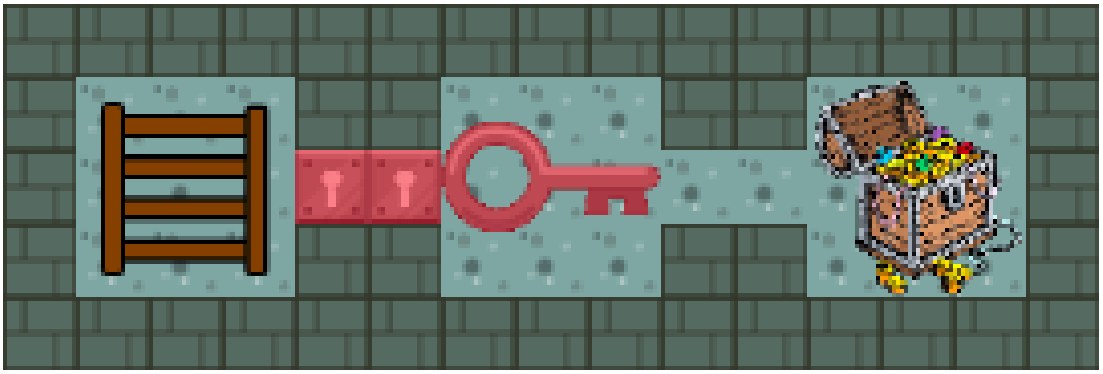

Figure 3 – le donjon non-faisable

Le résultat obtenu :

```
-( 22:32:39 )-< command 9 >-------------------------------------------------------------------{ counter: 0 }-
utop # Projet_donjon.faisable Projet_donjon__Test_ci.dj Projet_donjon__Test_ci.vi Projet_donjon__Test_ci.vf Projet_donjo
n__Test_ci.doors;;
- : bool = false
-( 22:32:50 )-< command 10 >------------------------------------------------------------------{ counter: 0 }-
utop # Projet_donjon.parcours_salles_principales Projet_donjon__Test_ci.dj Projet_donjon__Test_ci.vi Projet_donjon__Test
_ci.vf Projet_donjon__Test_ci.doors;;
- : Graph.vertex list = []
```

Figure 4 – test partie 1 sur le donjon non-faisable

2 Partie 2 : Les donjons à interrupteurs

La Deuxième partie consiste à vérifier le faisabilité des donjons à interrupteurs. en prenant comme entrée notre donjon, la salle initiale, la salle finale et les interrupteurs.Cette fois-ci, le donjon peut contenir des couloirs bloqués par des pointes au sol qui peuvent être levées ou baissées. Lorsque les pointes sont baissées, il est possible de passer. En revanche, lorsqu'elles sont levées, le couloir est impratiquable. Le rôle de l'interrupteur(d'une couleur donnée) est de faire alterner les pointes baissées et les pointes levées (de la même couleur). En effet, notre programme doit être capable de décider s'il existe une solution pour aller de la salle initiale vers la salle finale en passant par les interrupteurs.

2.1 L'algorithme utilisé

Dans la résolution de ce problème, on s'est basé sur l'algorithme proposé dans le sujet de projet.on synthétise l'algorithme dans les 3 étapes suivantes :

Étape 1 : Pour chacune des 2^n combinaisons d'états des n différentes couleurs d'interrupteurs disposés dans le donjon(cette liste de combinaison sera déduite à partir de la liste des interrupteurs), créer une copie du graphe en lui ajoutant/retirant les arêtes correspondant aux interrupteurs activés ou non.

Étape 2 : Par la suite, on lie entre le même interrupteur en ajoutant des arêtes de transition entre graphes correspondant à son activation.

Étape 3 : Appliquer l'algorithme de Djikstra sur le graphe globale pour déterminer l'existence d'un chemin entre l'état initial et les états finaux (chaque état final d'un sous graphe sera un état final du graphe global).

2.2 Mise en place de la solution

Dans cette partie, on présentera les solutions techniques adoptées pour l'implantation de l'algorithme.

2.2.1 Les combinaisons

Pour renvoyer n combinaisons états de n couleurs d'interrupteurs différentes, on s'est basé sur la structure de données des interrupteurs :

Les interrupteurs sont considérer comme une liste de couple de la forme (sommet où se trouve l'interrupteur, liste des arêtes contrôlées par cet interrupteur).

on a remarque que les interrupteurs de la même couleur ont la même partie seconde (car ils contrôlent la même liste d'arêtes). Pour faciliter la génération des combinaisons d'états, on a créé une nouvelle liste de couples, chaque couple est de la forme (Liste de sommets de même couleur , liste des arêtes contrôlées par ces interrupteurs).

Par la suite on génère les 2^n combinaisons possibles avec n désigne le nombre de couleurs existantes.

2.2.2 Génération des sous graphes

Chaque combinaison d'états a son propre graphe. Dans cette étape, on prend une combinaison (qui est de la forme d'une liste de couple), pour chaque couple, on prend sa deuxième partie qui est une liste des arêtes, si cette arête existe dans le graphe de départ on la supprime sinon on la rajoute dans le graphe.

Dans cette partie on rajoute aussi une nouvelle liste qui va nous aider à dessiner le graphe global qui représente l'état global de la combinaison. chaque élément de la liste est de la forme :
([(liste des sommets des interrupteurs d'une couleur C , 1 si C existe dans la combinaison 0 sinon), ...], (graphe de cette état, indice de cet état)).

2.2.3 Dessiner le graphe global

A partir de cette structure précédemment décrite, on va créer notre nouveau graphe global.
on a décomposé cette étape en deux :

Étape 1 : Graphe sans transition entre interrupteurs Dans cette partie on parcours la structure précédente. Pour chaque élément, on prend son graphe et on rajoute ses sommets et ses arêtes dans le nouveau graphe global. On crée la différence entre les sommets et les arêtes grâce à l'indice du graphe.

par exemple le sommet de label = k dans le sous graphe d'indice i correspond au sommet de label égal à (nombre de sommets) * i + k

Étape 2 : Ajout des transitions dans le graphe global C'est la partie la plus difficile du problème. En effet, pour mettre une transition entre des interrupteurs d'une couleur dans un sous graphe g1 et ses équivalents dans un autre sous graphe g2, on doit s'assurer que :

- 1 - on a fait un changement d'état, c'est-à-dire que dans la structure globale précédente, on doit vérifier que dans le premier sous graphe g1 et de dans le deuxième g2 on a un changement d'état pour cette couleur.
- 2 - le nombre de changement global des couleurs entre les deux sous graphe ne dois pas dépasser un, en d'autres termes une seule couleur qui doit changer son état dans les deux sous graphes.

Alors pour chaque couleur on vérifie si elle répond aux conditions précédentes pour chaque 2 sous graphes pour créer la transition.

2.2.4 Vérification de le faisabilité du donjon

Pour déterminer la faisabilité du donjon, on doit tout d'abord rassembler les états finaux du graphe global. Puisqu'on a sait la manière avec laquelle on a renommé les sommets, et on connaît le sommet final du graphe de départ, on peut donc les récupérer via la formule de renommage des sommets.

Pour le sommet de départ du graphe global, il a le même label que le sommet de départ , puisque l'indice sous graphe de départ (État où tous les couleur ont un indice 0) est égal à 0.

Pour chaque sommet final V_{f_i} du graphe global, on vérifie s'il y a un chemin menant du sommet initial V_i vers V_{f_i} en utilisant l'algorithme de Djikstra ($shortest_path$).
S'il existe, donc le donjon est faisable
sinon le donjon est non-faisable.

La fonction $shortest\_path$ est exploitée aussi pour expliciter le plus court chemin à parcourir.

2.3 Résultats

2.3.1 Test sur un donjon à interrupteurs faisable :

La figure du donjon faisable sur lequel on a appliqué le test :

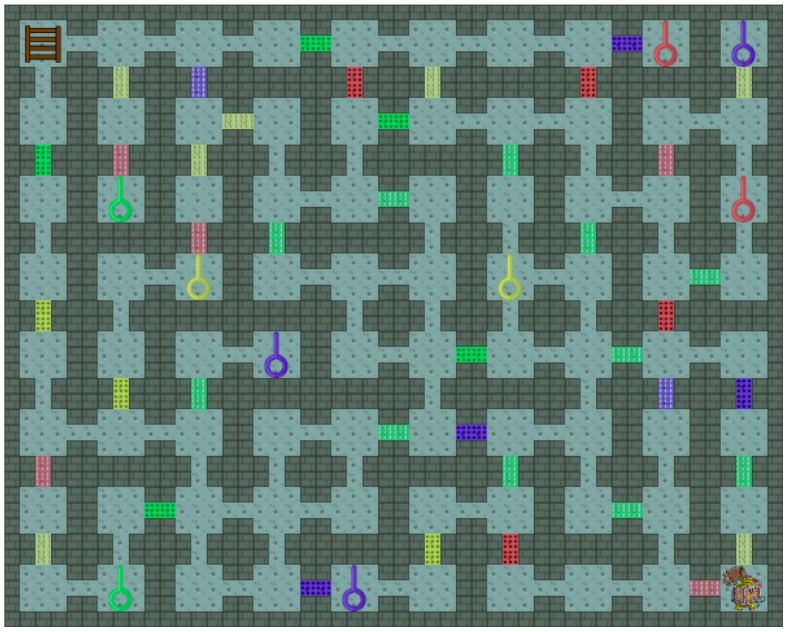

Figure 5 – le donjon faisable

Le résultat obtenu :

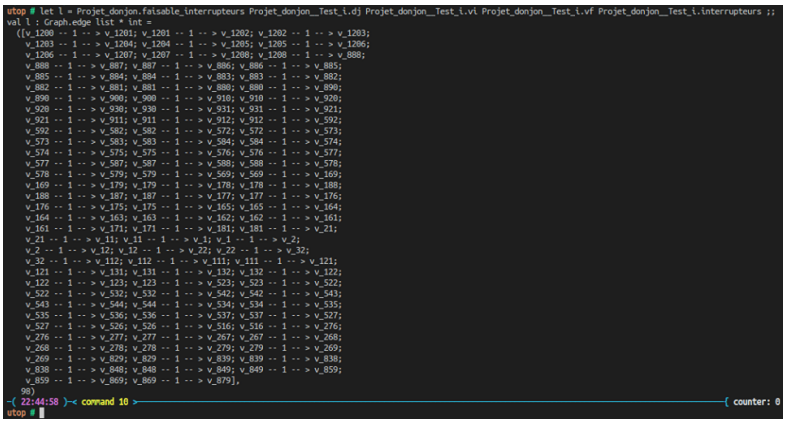

Figure 6 – test partie 2 sur le donjon faisable

2.3.2 Test sur un donjon à clés non-faisable :

2.3.3 Test sur un donjon à interrupteurs non-faisable :

La figure du donjon faisable sur lequel on a appliqué le test :

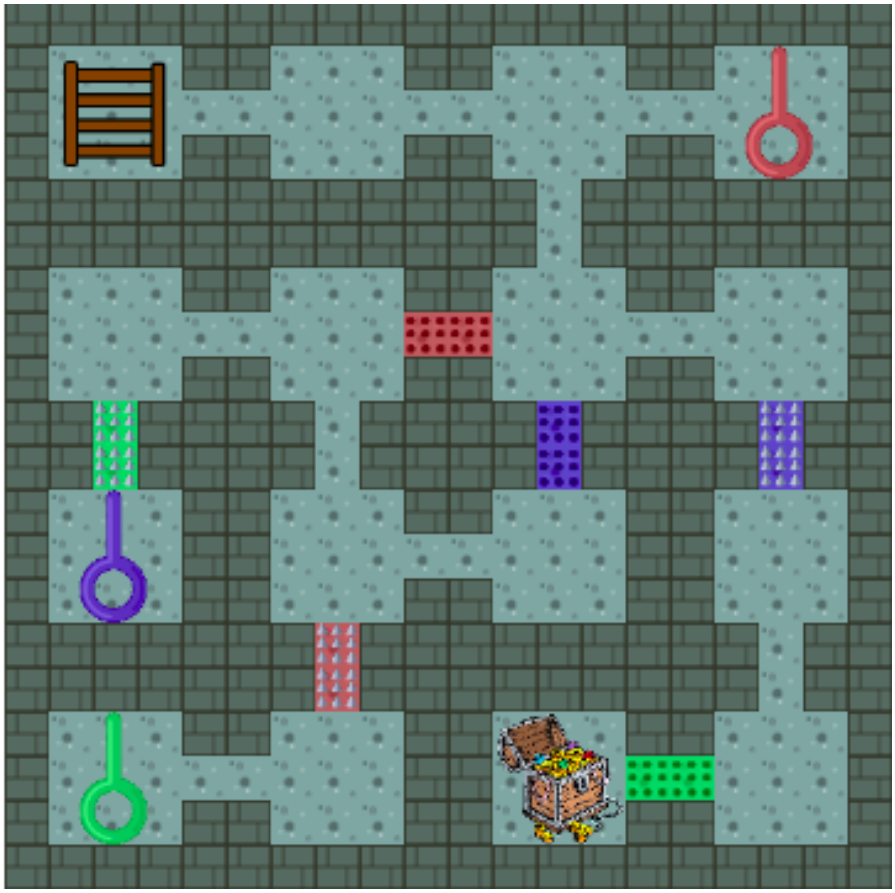

Figure 7 – le donjon non-faisable

Le résultat obtenu :

```
-( 22:41:16 )-< command 8 >-------------------------------------------------------------------{ counter: 0 }-
utop # let l = Projet_donjon.faisable_interrupteurs Projet_donjon__Test_ii.dj Projet_donjon__Test_ii.vi Projet_donjon__T
est_ii.vf Projet_donjon__Test_ii.interrupteurs ;;
val l : Graph.edge list * int = ([], 0)
-( 22:43:20 )-< command 9 >-------------------------------------------------------------------{ counter: 0 }-
utop #
```

Figure 8 – test partie 2 sur le donjon non-faisable

Conclusion

Dans ce rapport, on a présenté les différentes solutions implantées pour assurer la faisabilité du jeu. Ce projet nous a donné l'opportunité de bien manipuler les modules des graphes existantes dans le langage OCAML à savoir les itérateurs et les fonctions d'édition des graphes. De plus, ce projet nous a permis de mettre en pratique les connaissances acquises pendant les séances de cours et de Tps de l'élément de module Théorie des Graphes pour une meilleure compréhension des approches étudiées.