



UE PROJET

M1 ANDROIDE

Processus décisionnels de Markov et plus court chemin stochastique

Réalisé par:

Alexandre DUPONT-BOUILLARD

Adrien BROUCHET

Supervisé par:

Emmanuel HYON

Pierre FOUILHOUX

Janvier-Juin 2019

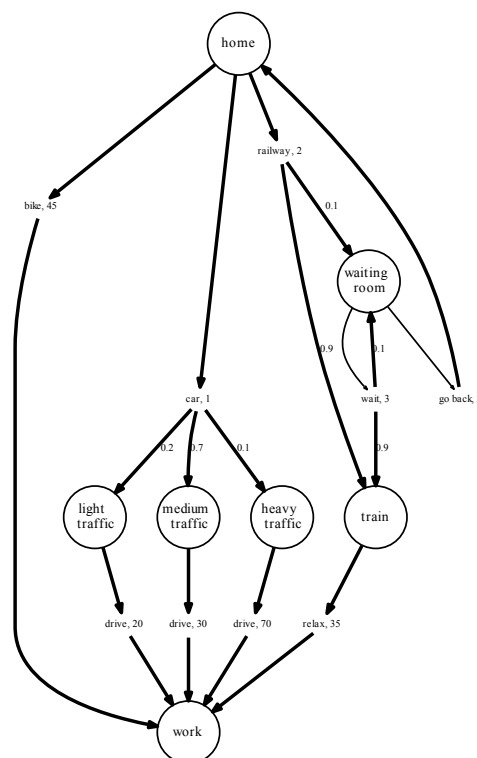
Sommaire

1	Présentation générale	2
1.1	Contextualisation du problème SSP-E	2
1.2	Présentation du rapport	3
1.3	Cadre du projet	3
2	Modèle mathématique et algorithmes	4
2.1	Processus décisionnels markoviens	4
2.1.1	Introduction	4
2.1.2	Formalisation	4
2.2	Algorithmes de résolution	7
2.2.1	Programmation linéaire	7
2.2.2	Programmation dynamique	7
3	Implémentations et instances	8
3.1	Value iteration et Policy iteration	8
3.1.1	Présentation de SWIG	8
3.1.2	pyMarmoteMDP	11
3.2	Programmation linéaire	12
3.3	Algorithme primal dual	12
3.4	Génération d'instances	13
3.4.1	Format d'instance	13
3.4.2	Description de l'instance montagne type	13
3.4.3	Génération d'instance montagne	14
4	Tests et comparaisons	16
4.1	Remarques générales	16
4.2	Manuel utilisateur	17
4.3	Comparaison sur une instance de petite taille (sous linux)	20
4.4	Comparaison sur les instances montagne (sous Windows)	20
5	Conclusion	23
6	Annexe	24
6.1	Le fichier mrDupont.txt :	24
6.2	Le fichier solutionMDP.h :	24
6.3	Le fichier solutionMDP_SWIG.h :	27

Présentation générale

1.1 Contextualisation du problème SSP-E

Imaginons la situation suivante: M. Dupont se rend au travail tous les jours, et a le choix entre prendre le vélo, la voiture, ou le train. Le temps de trajet varie d'un moyen de transport à l'autre, mais peut également varier pour le même moyen de transport : en voiture, par exemple, suivant le trafic, M. Dupont mettra plus ou moins longtemps pour arriver à son travail [4].



Dans le cadre du projet, on s'intéresse à l'étude de la question du plus court chemin stochastique en moyenne (en anglais : *Shortest Stochastic Path - Expectancy*, d'où le nom du problème SSP-E). Pour l'exemple de M. Dupont, cela revient à trouver les instructions à donner pour que M. Dupont mette un temps minimal en moyenne pour se rendre sur son lieu de travail, et ce quel que soit l'endroit d'où il part.

1.2 Présentation du rapport

Dans ce rapport, nous allons dans un premier temps formaliser le cadre dans lequel s'inscrit le problème SSP-E, puis nous présenterons les algorithmes utilisés pour répondre à ce problème : programmation linéaire, algorithme primal-dual, algorithme policy iteration, et algorithme value iteration. Enfin, dans un dernier temps, nous définirons des instances de test afin de pouvoir comparer l'efficacité de ces différents algorithmes.

1.3 Cadre du projet

Premièrement un travail de bibliographie sera effectué afin de comprendre précisément le problème de plus court chemin stochastique, suivi d'un approfondissement des articles afin de comprendre les différents modèles formels proposés dans la littérature : celui classique des processus de décision markovien [2] et celui introduit par Gautier Stauffer [1] puis faire un choix entre les deux pour la suite du rapport.

Parallèlement deux tâches seront ensuite effectuées : la première consistera à implémenter le programme linéaire [1], la seconde consistera à interfacer le code fournit écrit en c++, (marmoteMDP et marmoteCore) en python à l'aide de SWIG (un travail d'approfondissement de SWIG aura donc été nécessaire [6]).

Ensuite, pour effectuer des tests il sera nécessaire de mettre en place un ensemble d'instances jouets pertinentes et de tailles différentes, pour cela Mr Hyon nous a décrit un type d'instances "montagnes" que nous décrirons par la suite.

Et pour finir nous implémenterons quelques tests d'efficacité, sur windows et linux, des différents algorithmes.

Modèle mathématique et algorithmes

Dans cette section, nous présentons le modèle formel utilisé permettant de formaliser le problème SSP-E : les processus décisionnels markoviens. Dans un deuxième temps, nous utilisons le formalisme ainsi introduit pour décrire les algorithmes mis en place pour résoudre le problème.

2.1 Processus décisionnels markoviens

2.1.1 Introduction

Le problème SSP-E est un problème de décision séquentielle dans l'incertain, formalisé mathématiquement par les processus décisionnels markoviens (en anglais: Markov Decision Process, ou MDP - on utilisera cette abbréviation par la suite). Un MDP est un processus de contrôle stochastique discret dans lequel évolue un agent : à chaque instant, cet agent est dans un état, et choisit d'effectuer une certaine action, associée à un coût, dont le résultat est incertain.

2.1.2 Formalisation

Un MDP est généralement défini par un quadruplet $\{S, A, T, C\}$ [2] où :

- S est un ensemble d'états
- A est un ensemble d'actions
- T est une fonction dite de transition de $S \times A \times S$ dans $[0, 1]$, pour laquelle $T(s, a, s')$ est la probabilité d'arriver en s' en partant de l'état s en faisant l'action a
- C est une fonction de coût associée à chaque action, notée $c(s, a)$, coût de l'action a dans l'état s

Dans le cadre de ce projet, nous faisons le choix de reprendre les notations de MM. Stauffer et Guilloit [1]. Nous utiliserons ces notations pour la suite du projet. Une instance de plus court chemin stochastique est définie par un quintuplet $\{S, A, J, P, c\}$, où :

- $S = \{1, 2, \dots, n\}$ est fini
- $A = \{1, 2, \dots, m\}$ est fini
- $J \in \mathcal{M}_{m,n}(\{0, 1\})$, avec $J(a, s) = 1$ si et seulement si l'action a est disponible dans l'état s
- $P \in \mathcal{M}_{m,n}(\mathbb{R})$, avec $P(a, s) = p(s|a)$ probabilité d'arriver en s en faisant l'action a

- $c \in \mathbb{R}^m$ où pour tout i dans $\{1, \dots, m\}$, c_i est le coût de l'action i , supposé positif ou nul

Il s'agit en fait d'une définition équivalente, la différence étant que la "fonction" T définie précédemment a été "coupée" en deux matrices P et J : nous invitons le lecteur à vérifier qu'on a bien, dans le cas où $P(a, s)$ est la même pour tous les sommets :

$$\forall (s, a, s') \in S \times A \times S, \quad T(s, a, s') = J(a, s) P(a, s')$$

Définissons maintenant la notion centrale de politique : il s'agit d'une fonction, que l'on notera π par la suite, qui permet d'indiquer les choix d'actions de l'agent dans chaque état. Une politique est dite :

- déterministe si elle associe à tout état une unique action
- stochastique si elle associe à tout couple (état, action) une certaine probabilité
- stationnaire si elle est constante au cours du temps, c'est-à-dire si la règle de décision associant une action à un état ne dépend pas du temps
- **propre** [5] si pour tout état initial, en suivant une telle politique on a une probabilité 1 d'arriver à l'état objectif (et donc l'espérance du nombre de visites d'un état donné est fini).

Enfin, il convient de définir la notion de critère de performance. Pour une politique donnée, on peut en effet s'intéresser à différents critères d'optimisation suivant le problème.

Prenons l'exemple d'un robot devant se rendre à un point de charge. Sa batterie ne lui permettant d'effectuer qu'un nombre limité d'actions, on va dans ce cas s'intéresser au problème de trouver une politique optimale selon un **critère fini** : on cherche à minimiser le coût en un nombre fini d'étapes.

Imaginons maintenant que ce robot fonctionne à l'énergie solaire, et qu'il se déplace sur une comète s'éloignant du soleil ; l'objectif étant d'explorer un maximum de la surface de la comète. On donne donc plus de valeur aux premiers déplacements qu'aux suivants, puisqu'en s'éloignant du soleil le robot aura de moins en moins d'énergie ; on peut aussi supposer, par exemple, que l'éclairage se détériorant, ce que le robot observe devient de moins en moins exploitable.

On cherche alors une politique optimale selon le **critère gamma-pondéré** : on pondère les poids des actions effectuées par un facteur $\gamma < 1$ élevé à la puissance du nombre d'étapes séparant cette action de l'état initial.

Dans le cadre de ce projet, on s'intéresse au cas limite du critère précédent, c'est-à-dire au cas $\gamma = 1$. On appelle ce critère le **critère total** : on ne limite pas le nombre maximal d'actions, et l'importance de toute action effectuée ne dépend pas du moment où elle a été effectuée.

Il existe un dernier critère usuel, appelé **critère moyen**, pour lequel on regarde la moyenne des coûts le long d'un chemin : on associe alors à une politique le coût moyen par étape. On utilise typiquement ce critère dans des applications de gestion de file d'attente, de réseau de communication, de stock de marchandise, etc... ([2])

Résoudre un problème SSP-E revient à trouver une politique propre optimale, au sens du critère

total, qui soit déterministe et stationnaire (Fonction de valeur d'une politique pour le critère total : $V^\pi(s) = E^\pi[\sum_{t=0}^{\infty} r_t | s_0 = s]$ [2]). En d'autres termes, on cherche une solution qui associe à tout état une unique action, de telle sorte que le coût moyen pour se rendre de n'importe quel état à l'état objectif soit minimal. La présence d'un état s_∞ absorbant, à récompense nulle ($\exists a \in A$ tel que $c(s_\infty, a) = 0$ et $p(s_\infty | s_\infty, a) = 1$), et accessible depuis chacun des états assure l'existence d'une politique propre.

2.2 Algorithmes de résolution

On va maintenant s'intéresser aux différents algorithmes classiques pour la résolution de MDP que nous avons utilisés pour résoudre le problème SSP-E.

2.2.1 Programmation linéaire

On peut montrer ([1]) que résoudre un problème SSP-E revient à résoudre le programme linéaire suivant :

$$\begin{aligned} \min \quad & c^T x \\ (J - P)^T x \quad &= \frac{1}{n} \mathbf{1} \\ x \quad &\geq 0 \end{aligned} \tag{P}$$

Il est donc naturel d'utiliser une méthode classique de résolution de programme linéaire, qui sera présentée dans la partie suivante du rapport.

2.2.2 Programmation dynamique

Présentons maintenant les algorithmes de programmation dynamique, ils sont au nombre de deux : l'algorithme *value iteration* et l'algorithme *policy iteration*.

Value iteration

Définissons tout d'abord ce qu'on appelle valeur d'un état. Il s'agit, pour une politique propre donnée, de la somme des coûts des actions effectuées en partant de cet état et en suivant cette politique. On note pour tout état s sa valeur $V(s)$.

Alors, ainsi que son nom l'indique, l'algorithme *value iteration* consiste à itérer sur V de manière à tendre vers V^* la valeur optimale. On peut montrer qu'à chaque itération k de cet algorithme, il est possible d'obtenir une politique propre, déterministe et stationnaire Π_k en un temps fortement polynomial telle que V^{Π_k} tende vers V^* ([1]). En pratique on donne en entrée de l'algorithme un double ε et on aura convergence si $|V_i - V_{i+1}| < \varepsilon$ et un entier maxIter qui sera le nombre d'itérations maximum de l'algorithme

Policy iteration modified

On vient de voir qu'il était possible d'obtenir une politique optimale en itérant sur la valeur ; ne pourrait-on pas itérer sur la politique elle-même ? C'est ce qui est proposé par l'algorithme *policy iteration*.

On peut montrer qu'il est possible d'obtenir une politique propre, stationnaire, et déterministe en un temps de l'ordre de $\mathcal{O}((n+1)(m+1))$ ([1]). Partant d'une telle politique, on utilise un algorithme du simplexe en itérant sur la politique : on a convergence en un nombre fini d'étapes à condition que le problème considéré ne possède pas de cycle de coût négatif, et qu'il existe un chemin entre n'importe quel état et l'état objectif.

Ici en pratique on donne en entrée de l'algorithme un double ε , un double δ , un entier maxIter et un entier maxInIter . ε et maxIter ont le même rôle que précédemment, δ et maxInIter ont aussi pour rôle un critère de convergence pour une autre boucle de l'algorithme ([2]).

Implémentations et instances

3.1 Value iteration et Policy iteration

3.1.1 Présentation de SWIG

SWIG est un outil permettant d'interfacer du code c ou c++ en python, ou d'autres langages. Il permet de manipuler des classes écrites en c++ à partir de python, de cette manière on conserve la syntaxe simple de python mais on garde la rapidité du c++.

Tout d'abord un wrapper devra être créé afin d'encapsuler le code c++ , à partir duquel il sera possible d'obtenir un wrapper.so et enfin un fichier .py qui permettra l'import de tous les objets encapsulés.

Pour cela on doit fournir à SWIG un fichier de configuration dans lequel toutes les classes à interfacer sont indiquées en donnant le fichier .h puis une copie de ces .h sans les #include, #ifndef, #define (un exemple est donné en annexe avec les fichiers solutionMDP.h et solutionMDP_SWIG.h). On peut voir ci dessous qu'il y a deux parties dans le fichier de configuration. Dans la seconde on retrouve tous les objets à encapsuler, ceux que nous allons vouloir instancier et manipuler en python. Ainsi qu'un ensemble de modules .i qui permettent de la gestion de classes simples comme les suites ou les chaînes de caractères, les types primitifs semblent être gérés automatiquement. Dans la première partie, quant à elle nous devons indiquer tous les objets nécessaires au bon déroulement des objets à encapsuler. C'est à dire tous les objets que l'on pourrait retrouver dans la partie #include en tête des fichiers .h correspondant aux objets à encapsuler.

fichier de configuration pymarmoteMDP.i pour créer le module python, pymarmoteMDP :

```
%module pyMarmoteMDP
%{
#include "header/marmoteSet.h"
#include "header/marmoteInterval.h"
#include "header/sparseMatrix.h"
#include "header/solutionMDP.h"
#include "header/feedbackSolutionMDP.h"
#include "header/genericMDP.h"
#include "header/totalRewardMDP.h"
#include "algin.h"
%}

#include "cpointer.i"
```

```

#include "std_string.i"
#include "std_vector.i"
#include "header/solutionMDP_SWIG.h"
#include "header/feedbackSolutionMDP_SWIG.h"
#include "header/marmoteSet_SWIG.h"
#include "header/marmoteInterval_SWIG.h"
#include "header/sparseMatrix_SWIG.h"
#include "header/totalRewardMDP_SWIG.h"
namespace std {
    %template(sparseMatrixVector) vector<sparseMatrix*>;
}

```

Ensuite il faut lancer la ligne de commande suivante afin de générer le fichier "pyMarmoteMDP_wrap.cxx":

```
swig -Wall -python -c++ pyMarmoteMDP.i
```

Puis lancer la commande :

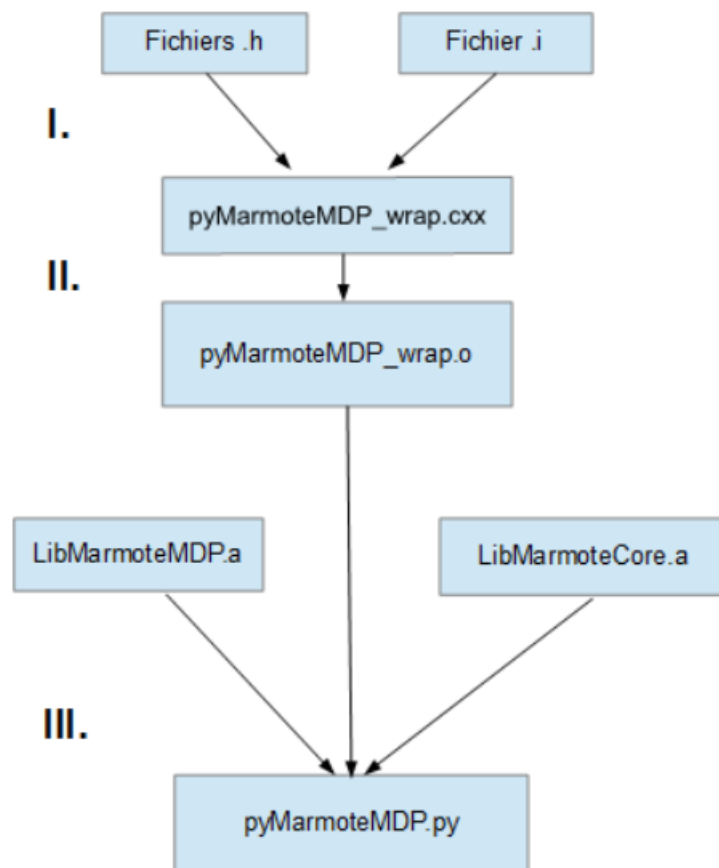
```
g++ -ansi -Wall -fPIC -O3 -c pyMarmoteMDP_wrap.cxx
    -I/usr/include/python3.6 /header
```

A condition que tous les .h nécessaires soient dans le dossier header. ATTENTION : python dev doit absolument être installé.

Et enfin :

```
g++ -shared pyMarmoteMDP_wrap.o -o _pyMarmoteMDP.so -L$(LIBRARIESDIR)
-l$(LIBRARIES)
```

Avec la variable LIBRARIESDIR contenant le répertoire dans lequel se trouvent les librairies présentes dans la variable LIBRARIES, nécessaires au fonctionnement des classes à interfacier.



Une partie du fichier n'a pas encore été présentée :

```
namespace std {
    %template ( sparseMatrixVector ) vector <sparseMatrix*>;
}
```

Cette ligne permet en python de pouvoir manipuler des objets qui seront interprétés par SWIG comme des `vector c++`. La conversion de listes python vers `vector c++` n'est pas automatique, SWIG fournira donc à l'aide de ce code un constructeur `sparseMatrixVector` qui prendra en entrée la taille souhaitée du `vector` et dans lequel on pourra ajouter des objets interfacés, `sparseMatrix`. Si l'on veut aussi pouvoir créer des `vector` de types différents il faudra ajouter au fichier de configuration autant de lignes que de types différents à ajouter.

L'objet `totalRewardMDP` n'a pas de constructeur par défaut et nécessite, entre autres, un `vector` d'objets `sparseMatrixMDP` qui représente les actions disponibles. Cette partie permet, en python, de créer un `vector` afin de le donner en paramètre du constructeur de `totalReward` de la manière suivante.

```
vectorLength = 4
sparseMatrixDimension = 5 #ces deux valeurs sont juste un exemple
x = sparseMatrixVector(vectorLength)
for i in range (vectorLength):
    x[i] = sparseMatrix(sparseMatrixDimension)
```

3.1.2 pyMarmoteMDP

Pour obtenir la librairie `pyMarmoteMDP` il suffit de lancer la commande `make` dans le dossier contenant le `makefile`, un fichier `pyMarmoteMDP.py` sera généré, on pourra alors l'importer et se servir des constructeurs suivants :

-`marmoteInterval(int borneMin, int borneMax)`

-`sparseMatrix(int dimension)` pour une matrice carrée

-`sparseMatrix(int dimension1, int dimension2)` pour une matrice non carrée

-`totalRewardMDP(String critere,marmoteInterval stateSpace, marmoteInterval actionSpace,vector<sparseMatrix> actions,sparseMatrix reward)` critere peut prendre les valeurs suivantes : "min" ou "max".

Pour ajouter des coefficients dans les matrices on utilisera la méthode : `addToEntry(int i, int j,double valeur)`

Pour exécuter l'un des algorithmes de programmation dynamique on utilisera les méthodes :

-`policyIterationModified(double epsilon,int maxIter, double delta, int maxInIter)` ces paramètres sont les conditions d'arrêt de l'algorithme.

-`valueIteration(double epsilon,int maxIter)` ces paramètres sont aussi des conditions d'arrêt.

Notre fichier `totalReward.py` fournit deux fonctions :

-`valueIteration(listAction,listCout,maxIter = 500,epsilon = 0.00001)`

-`policyIterationModified(listAction,listCout,maxIter = 500,epsilon = 0.00001,maxInIter = 1000,delta = 0.001)`

Ces deux fonctions exécutent l'algorithme correspondant à partir d'une liste de matrices d'actions

(l'action i sur le sommet x a une probabilité $m[x][i][j]$ de renvoyer sur le sommet j), et d'un vecteur de coût ($v[i]$ est le coût de l'action i).

Ces objets sont ceux nécessaires à la définition totale d'un problème de plus court chemin stochastique, une instance jouet fournie par Mr Hyon nous a permis de les identifier parmi tous les objets de `marmoteCore` et `marmoteMDP`.

3.2 Programmation linéaire

On rappelle le programme linéaire (P) défini au chapitre précédent :

$$\begin{aligned} \min \quad & c^T x \\ (J - P)^T x \quad &= \frac{1}{n} \mathbf{1} \\ x \quad &\geq 0 \end{aligned} \tag{P}$$

Le dual de ce programme linéaire s'écrit ainsi :

$$\begin{aligned} \max \quad & \frac{1}{n} \mathbf{1}^T y \\ (J - P)y \quad &\leq c \end{aligned} \tag{D}$$

Il est facile de coder le primal (P) ou le dual (D) sous Python puis d'utiliser le solveur Gurobi pour le résoudre. On obtient alors une politique optimale.

3.3 Algorithme primal dual

Il est également possible de résoudre le problème SSP-E en utilisant une autre méthode de programmation linéaire : l'algorithme du primal-dual.

Concrètement, cela consiste à partir d'une solution réalisable du dual (D) ci-dessus (on peut par exemple partir de $y = 0$ puisque les coûts sont positifs ou nuls par hypothèse). À partir de cette solution, on génère ce qu'on appelle un primal restreint, plus simple à résoudre, ainsi que le dual associé. Alors, à chaque étape, on regarde si le primal restreint a une solution nulle (auquel cas on est arrivé à convergence), sinon on résout le dual restreint associé et on réitère.

On est assuré d'avoir convergence en un nombre fini d'étapes dans le cadre du problème SSP-E ([1], [3]), et on converge vers une solution optimale. Cette méthode est plus efficace que la méthode classique du simplexe si on parvient à trouver une méthode astucieuse permettant de résoudre les primaux ou duaux restreints à chaque étape de manière rapide (autrement, si on doit appliquer une méthode du simplexe à chaque itération, le temps total d'exécution sera beaucoup plus long).

Nous avons essayé d'appliquer une méthode fonctionnant pour le problème du plus court chemin déterministe ([3]), mais du fait que les actions ont un résultat aléatoire, cette méthode n'a pas fonctionné. En l'état actuel des choses, nous n'avons pas encore réussi à mettre en place un algorithme primal-dual qui fonctionne pour résoudre le problème SSP-E.

3.4 Génération d'instances

3.4.1 Format d'instance

Tout d'abord, il est nécessaire de choisir un format d'instance, sur lequel on puisse baser les algorithmes. Nous n'avons pas trouvé de format de référence dans la littérature et nous avons donc fait le choix de prendre un format de données proche du format DIMACS.

Pour une instance donnée, la première ligne (commençant par p pour pouvoir être identifiée lors de la lecture) indique le nombre de noeuds, d'actions, et de redirections (c'est-à-dire la somme sur toutes les actions du nombre de noeuds accessibles depuis une action). Le groupe suivant de lignes indique toutes les actions disponibles et est sous la forme

```
a [numero noeud] [numero action] [cout de l'action]
```

Le dernier groupe de lignes indique tous les noeuds accessibles par les actions effectuées et est sous la forme

```
e [numero action] [numero noeud] [proba d'arriver sur ce noeud]
```

Il est possible de rajouter autant de lignes commentaire en début d'instance qu'on le souhaite, à condition de préfacier chacune de ces lignes par c + un espace.

Un exemple est fourni en annexe : `mrDupont.txt`, c'est la représentation sous notre format de l'exemple 1.1 du rapport.

3.4.2 Description de l'instance montagne type

Nous avons fait le choix de générer des instances de type montagne, assez adaptées aux problèmes de plus court chemin (stochastique).

Commençons par décrire l'exemple type de ce genre d'instance: on scinde une grille $n \times n$ horizontalement en deux parties égales, chaque partie correspondant à un versant de la montagne. La case centrale de la grille est à éviter absolument (coût infini), tandis que toutes les autres cases ont un coût unitaire.

On part de la case en bas à gauche, l'objectif étant de rejoindre la case en haut à droite avec un coût minimal. Pour ce faire, on a le choix entre 2 actions:

- Action A : On a une probabilité p_1 d'aller à droite, et une probabilité $1 - p_1$ d'aller en bas si on est sur le versant sud, et en haut sinon, avec p_1 faible
- Action B : On a une probabilité p_2 d'aller à droite, et une probabilité $1 - p_2$ d'aller en haut si on est sur le versant sud, et en bas sinon, avec p_2 proche de 0.5

Qualitativement, cela signifie que lorsqu'on essaye de "grimper" vers le sommet de la montagne (action B), on a plus de chance d'aller à droite, tandis que si on essaye d'aller à droite (action A), on a plus de chance de redescendre. Un dessin valant mieux qu'un long discours, voici une image représentant l'instance décrite ci-dessus.

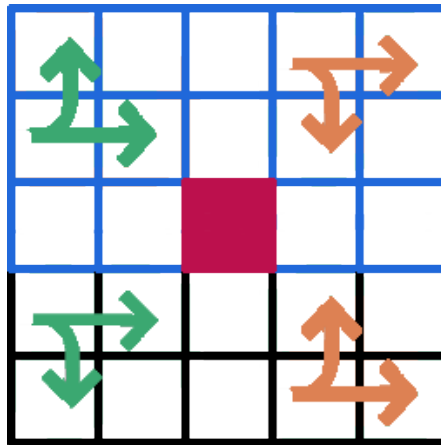


Figure 3.1: Instance type montagne

En vert, l'action A, et en orange, l'action B, avec leurs résultats respectifs. Les actions sont inversées suivant sur quel versant on se trouve (le "sommet" se trouvant en haut dans un cas, et en bas dans l'autre). Les actions A et B ont toujours un coût unitaire, sauf si on est sur la case rouge, auquel cas toute action sortant de cette case a un coût infini.

3.4.3 Génération d'instance montagne

Modifications apportées sur les instances

Pour les besoins du projet, on a généralisé le cas de l'instance montagne "type" de manière à avoir des instances plus intéressantes à étudier. Tout d'abord, on a rajouté 2 actions A^- et B^- , symétriques par rapport aux actions A et B : il s'agit des mêmes actions avec les mêmes probabilités, à la différence près qu'on a remplacé droite par gauche. Ceci est utile si jamais l'instance considérée admet une solution optimale pour laquelle il faut faire demi-tour.

Ensuite, on a rajouté la possibilité de modifier le versant de la montagne (à la place d'une ligne droite, on peut avoir un versant accidenté).

Enfin, on a également rajouté l'option d'avoir plus d'un puits (afin de rendre la politique optimale plus complexe que haut haut haut haut droite droite droite droite).

Nous avons donc codé une fonction (en Python) générant des instances de type montagne avec en paramètres :

- `n` la taille de la grille (longueur du côté)
- `npuits` le nombre de puits, ou cases à éviter (en rose sur le dessin)
- `p1` et `p2` ainsi que définis précédemment
- `relief` un nombre positif représentant la linéarité du versant (0 = linéaire, 100 = terrain très accidenté)
- `i` un indice servant lors du stockage en mémoire de l'instance

Description de la génératrice d'instance

La fonction commence par générer le versant de manière aléatoire :

Algorithm 1 Versant

```

initialiser  $L :=$ 
pour  $i$  allant de 2 à  $n$  faire
    courant  $\leftarrow L[-1]$ 
    courant  $\leftarrow$  courant +  $\varepsilon$ 
    rajouter courant à la fin de  $L$ 
fin pour
renvoyer  $L$ 

```

Le ε de l'algorithme dépend du paramètre *relief* de la fonction, et est généré aléatoirement de telle sorte que le versant peut monter ou descendre à chaque étape, en ayant tendance à revenir vers le centre (pour éviter que le versant ne fasse que monter et disparaisse, ce qui rendrait l'instance moins intéressante).

La fonction génère ensuite une liste de positions pour les puits suivant la valeur du paramètre *npuits*:

- Si $\text{npuits} \leq n$, alors les puits sont placés au niveau du versant, à raison de 1 puits par colonne, le choix de la colonne se faisant de manière aléatoire pour chaque puits.
- Si $n < \text{npuits} \leq 2n$, les puits sont placés sur 2 lignes situées de part et d'autre du versant à distance égale, le choix de la colonne se faisant toujours de manière aléatoire pour chacune des 2 lignes
- Si $2n < \text{npuits} \leq 3n$, les puits sont placés sur 3 lignes, la ligne du centre étant de nouveau le versant
- ... et ainsi de suite.

La fonction crée alors deux listes *listeA* et *listeE*, avec

- *listeA* une liste pour laquelle chaque élément a 3 éléments (numéro de noeud, numéro d'une action (disponible sur ce noeud), coût associé à l'action)
- *listeE* une liste pour laquelle chaque élément a 3 éléments (numéro d'action, numéro d'un noeud accessible depuis cette action, probabilité d'arriver sur ce noeud)

Intéressons-nous à la longueur des listes. On a 4 actions différentes disponibles sur chaque noeud (A, B, A⁻ et B⁻), et chacune de ces actions a deux noeuds de sortie possibles. Donc, *listeA* est de longueur $4n^2$ et *listeE* est de longueur $8n^2$.

Précisons que ces listes sont générées en tenant compte des conditions de bord : par exemple, si on essaye d'aller en bas alors qu'on est déjà sur la ligne du bas, alors on ne bouge pas. De même avec le bord droit, gauche, et haut.

Une fois ces listes *listeA* et *listeE* générées, la fonction crée le fichier instance (indiqué par le paramètre *i* de la fonction) et écrit ces listes l'une après l'autre : on obtient un fichier *.txt* respectant le format de données présenté en début de section.

Tests et comparaisons

4.1 Remarques générales

En raison du choix du format de données, et de la manière de les traiter par la suite avec Python, nous avons observé que la taille maximale d'instance sur laquelle on puisse faire tourner l'algorithme de programmation linéaire est de 32 par 32 (pour une instance montagne).

Cette limite est contraignante pour pouvoir comparer l'efficacité des différents algorithmes avec une précision suffisante, en effet dans le cadre de la programmation linéaire Gurobi résout le PL en un dixième de seconde environ.

Nous nous sommes rendus compte de ce problème assez tardivement et nous avons choisi de poursuivre avec le format choisi, tout changer à cette étape nous aurait pris beaucoup de temps. Il s'agit donc d'un point qui pourrait être repris et amélioré dans le cadre de la poursuite de ce projet (optimisation du format de données / choix d'une représentation différente dans Python pour prendre moins de place en mémoire).

Le choix de format de données aurait pu être amélioré en compactant les données, en effet il aurait été possible de définir seulement 4 actions pour les instances de montagnes, alors que nous en avons ici 4 par état (deux pour les états du bord de la grille). Ce modèle plus compact aurait probablement permis aux algorithmes policy iteration et value iteration d'être plus performants sur ces instances puisqu'ils exploitent plutôt ce genre de modèles compacts.

Un autre problème que nous avons rencontré tout au long du projet a été de réussir à créer les objets nécessaires avec un makefile, puis les wrapper via SWIG pour faire tourner Marmote, fourni par M. Hyon, sous Python (l'idée étant d'avoir un fichier unique pour lequel on puisse faire tourner tous les algorithmes afin de les comparer). Cela nous aurait en effet permis de tester les algorithmes de programmation dynamique, utilisés dans Marmote.

Nous n'avons cependant pas réussi à aboutir, et à défaut, nous avons codé les algorithmes ainsi que décrits dans [2] sous Python. Le défaut majeur de cette solution est que ces algorithmes tournent du coup en Python, alors que l'algorithme de programmation linéaire utilise Gurobi qui tourne en C. Lors de la comparaison des algorithmes, il faudra donc que nous prenions en compte la différence de langage, dans la mesure du possible.

Précisons toutefois que nous avons réussi à instancier l'exemple jouet présenté en introduction du rapport (le modèle représentant les chemins possibles pour que M. Dupont se rende sur son lieu de travail). Le problème que nous avons rencontré a été d'automatiser le processus de création d'objets. Le modèle jouet est de taille très raisonnable, et il est donc possible de créer les objets "à la main", ligne par ligne. En revanche, pour les instances montagne, il devient très rapidement impossible de les traiter manuellement, et lorsqu'on passe à des boucles

implémentant automatiquement les objets, on obtient une erreur de type `segfault`.

4.2 Manuel utilisateur

Le code que nous avons produit se décompose ainsi : un fichier `main.py`, servant à effectuer les tests. Ce fichier fait lui-même appel à trois fichiers :

- `prog_lin.py` et `prog_dyn.py` dans lesquels se trouvent les fonctions trouvant la politique optimale respectivement par méthode de programmation linéaire et dynamique (on détaille les fonctions ci-dessous)
- `instance_montagne.py` dans lequel se trouvent deux fonctions, une générant une instance de type montagne et une autre permettant de visualiser les résultats obtenus

Les fichiers `prog_lin.py` et `prog_dyn.py` font appel à un fichier `lecture_fichier.py`. Ce fichier permet de traiter un fichier `.txt` respectant le format de données détaillé plus haut, afin de générer les matrices et listes dont on se sert ensuite pour les différents algorithmes de programmation linéaire et dynamique - plus précisément, les matrices J et P ainsi que définies précédemment, et le vecteur de coûts c .

Détaillons maintenant les différentes fonctions présentes.

- Fichier `prog_lin.py` : une fonction `sSSPdual` prenant en paramètres les matrices J et P , et le vecteur c . Cette fonction applique alors une méthode classique de programmation linéaire (sur le dual), et renvoie une politique optimale solution sous forme de liste : cette liste est de longueur le nombre de noeuds moins 1 (pas besoin de stratégie sur le noeud but), et à chaque élément de cette liste correspond l'action optimale à choisir. Ce format de politique optimale reste le même pour les autres algorithmes présentés ci-dessous, nous ne le préciserons donc plus par la suite.
- Fichier `prog_dyn.py` : deux fonctions, `valueIteration` et `policyIteration`.
`valueIteration` prend en paramètres J , P , c , ainsi qu'un paramètre supplémentaire `epsilon` correspondant à la valeur d'arrêt de l'algorithme (si d'une itération à l'autre, la différence de valeur est inférieure en norme à cet `epsilon`, alors on s'arrête). Voir la suite pour le choix de la valeur de ce paramètre. Cette fonction renvoie une politique optimale obtenue par la méthode `value iteration`.
`policyIteration` prend en paramètres J , P , et c , et renvoie une politique optimale obtenue par la méthode `policy iteration`.
- Fichier `instance_montagne.py` : deux fonctions, `creeInstance` et `dessinInstanceM`.
La fonction `creeInstance` a été détaillée plus haut, elle prend en paramètres la taille de la grille n (maximum $n = 32$, erreur au-delà), le nombre de puits présents sur cette grille (maximum $n/4$, erreur au-delà), le numéro de l'instance (afin de pouvoir en générer plusieurs dans une boucle), les deux probabilités p_1 et p_2 ainsi que définies précédemment, et le niveau de relief (0 si le versant est rectiligne, 100 s'il est très accidenté). Cette fonction renvoie les coordonnées du versant ainsi que celles des puits (utiles pour la fonction `dessinInstanceM`).
La fonction `dessinInstanceM` prend en paramètres le nombre d'états, la liste des positions des puits, la liste des coordonnées du versant, l'état de départ, la politique considérée, et une liste intermédiaire fournie par `lecture_fichier.py` donnant, pour toute

action, l'ensemble des noeuds accessibles depuis cette action avec la probabilité associée. La fonction `dessinInstanceM` trace alors la grille considérée ainsi que plusieurs instanci-ations de la politique fournie à l'aide du module Python `turtle`.

Dans le fichier `main.py`, on a laissé pour l'utilisateur un exemple d'utilisation. On com-mence par générer des instances de type montagne, tout en stockant les listes des positions des puits et des versants (si jamais on souhaite les visualiser par la suite).

Ensuite, on lit et on transforme chacune des instances ainsi générées, afin de pouvoir déterminer la politique optimale sur chacune d'entre elles. On mesure le temps d'exécution des différentes méthodes, et on les stocke sous forme de liste, qu'on affiche ensuite.

Enfin, si on le souhaite, on peut visualiser l'instance montagne de son choix, avec 5 instanci-ations de la politique de son choix (une même politique peut mener à différents instanci-ations du fait que les actions ont des résultats aléatoires).

Voici un exemple de visualisation pour $n = 15$ avec $n_{\text{puits}} = 16$ et $\text{relief} = 30$. En blanc, le versant sud, en bleu, le versant nord, en rose, les puits à éviter ; on a ici appliqué l'algorithme `policy iteration`.

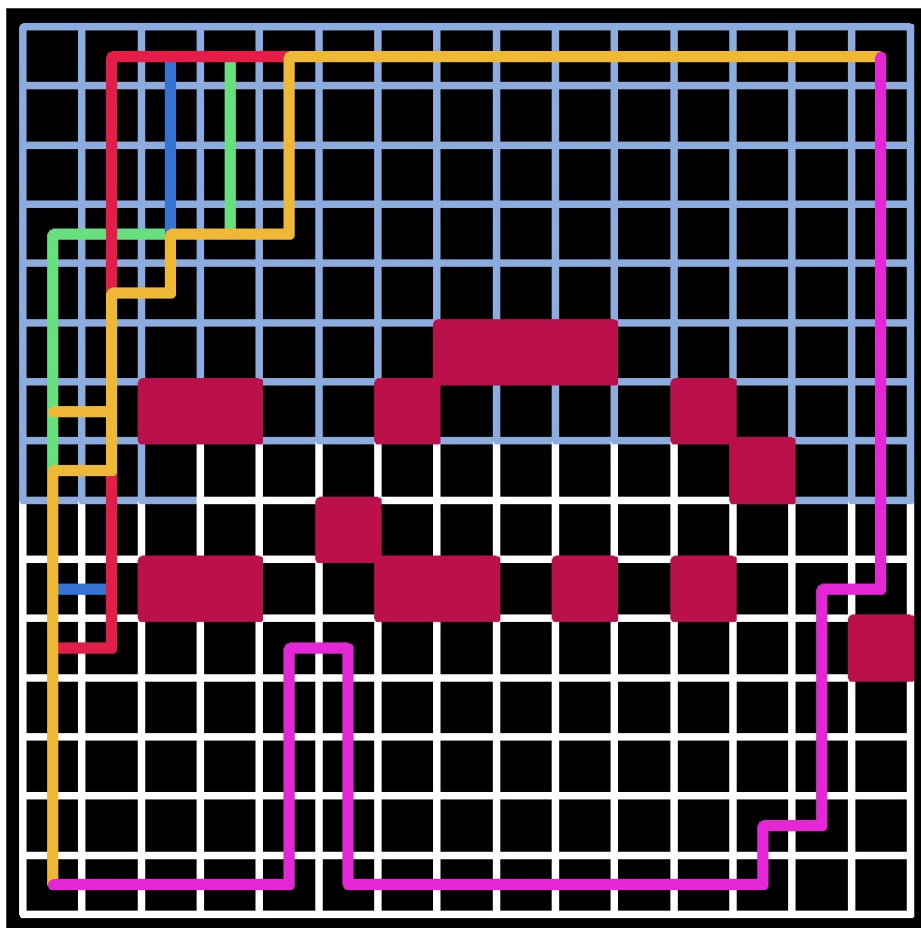


Figure 4.1: Chaque tracé correspond à une instanci-ation d'une politique optimale

Comme on peut le voir sur la figure, pour une politique donnée, on n'obtient pas toujours (et même en fait quasiment jamais) le même chemin (en effet, le cadre est stochastique et non déterministe). C'est ce qui conduit à la différence notable entre le chemin rose et les 4 autres chemins : sur le noeud initial, l'action indiquée par la politique optimale est l'action B (droite

ou haut), et suivant l'issue de cette action, des chemins très différents en découlent (passage à gauche des obstacles versus passage à droite).

4.3 Comparaison sur une instance de petite taille (sous linux)

Tout d'abord nous n'avons pas réussi à implémenter de fonction permettant de créer un objet `totalReward` seulement à partir d'un fichier `.txt`. Nous avons quand même effectué une comparaison sur une instance.

L'instance `mrDupont` est assez simple ce qui a permis d'écrire le code nécessaire à l'instanciation de l'objet `totalRewardMDP` lié à la main. Sur cette instance on obtient que :

- l'algorithme `value iteration` s'exécute en 0.00014328956604003906 sec
- l'algorithme `policyIterationModified` s'exécute en 0.00045490264892578125 sec
- le programme linéaire quant à lui s'exécute en 0.3608238697052002 sec.

`Value iteration` semble être plus rapide sur de petites instances.

4.4 Comparaison sur les instances montagne (sous Windows)

On observe que l'efficacité des algorithmes varie suivant la quantité de puits présents dans l'instance.

On commence par conjecturer l'efficacité des algorithmes en testant sur des instances de petite taille, disons $n \leq 10$. On observe les résultats suivants : quand on a peu de puits, l'algorithme `policy iteration` est franchement plus efficace que l'algo `PL` et l'algo `value iteration`. Pour $n=10$ par exemple, avec 1 puits, `policy iteration` fournit une politique optimale en environ 0.05 seconde, contre environ 0.15 seconde pour les deux autres algorithmes.

Quand on fait augmenter le nombre de puits, les algorithmes de programmation dynamique ont un temps d'exécution qui augmente, alors que l'algorithme de programmation linéaire fournit un résultat en un laps de temps identique. Pour $n=10$, on a toujours 0.15 seconde pour la `PL`, mais pour `policy iteration`, on passe de 0.05 à 0.15 seconde, et pour `value iteration`, on passe de 0.15 à entre 1 et 3 secondes.

Augmentons maintenant n pour voir si ces résultats se maintiennent.

- **Pour $n=15$:** avec un seul puits, l'algorithme `policy iteration` est le plus performant (autour de 0.65 seconde), suivit par la programmation linéaire (autour de 0.85 seconde), et en dernier, `value iteration` (1 seconde environ). Quand on augmente le nombre de puits, les temps d'exécution deviennent rapidement très grands pour `value iteration` (sans doute à cause des coûts très importants sur les puits, en théorie infinis, en pratique égaux à n^3). Ainsi pour $n=15$, pour 20 puits, il faut de 4 à 6 secondes pour obtenir un résultat ; pour $n=15$, pour 25 puits, il faut jusqu'à 39 secondes (pire cas observé). Par la suite, on va donc se limiter à comparer `policy iteration` et programmation linéaire.

On remarque que pour la programmation linéaire, le temps d'obtention moyen d'une politique optimale reste de 0.85 seconde, peu importe le nombre de puits. De plus la variance est très faible : dans le pire des cas, on avoisine les 0.90 seconde, dans le meilleur des cas, on se rapproche de 0.80 seconde. En revanche, pour l'algorithme `policy iteration`, le temps moyen augmente : ainsi, pour 20 noeuds, on a un temps moyen de 1.00 seconde, avec un minimum à 0.9 seconde et un maximum à 1.3 seconde. Pour 50 noeuds, le temps moyen est de 1.30 seconde, avec un minimum à 1.00 seconde et un maximum à

2.3 seconde. On remarque en fait qu'en moyenne, le temps augmente de manière limitée avec le nombre de puits, en revanche dans le pire des cas, on a une grosse augmentation du temps d'exécution.

- **Pour $n=20$:** avec 1 seul puits, le programme linéaire met 3 secondes en moyenne pour fournir une politique optimale. On vérifie par la suite que ce temps reste le même, quel que soit le nombre de puits. De son côté, l'algorithme policy iteration converge en 1.9 seconde en moyenne. Si maintenant on augmente le nombre de puits, (on en met 45), il faut environ 4.2 secondes pour que policy iteration converge, avec près de 6 secondes dans le pire des cas. Pour 90 puits, le temps de convergence de policy iteration reste le même, environ 4.2 secondes, en revanche le temps maximal "dans le pire des cas" semble augmenter puisqu'on dépasse les 7 secondes.

- **Pour $n=25$:** l'algorithme de programmation linéaire converge en 9.5 secondes en moyenne. Pour un seul puits, policy iteration est meilleur puisqu'il converge en moyenne en 6 secondes. Pour 40 puits, il faut 12 secondes en moyenne pour que policy iteration converge, 9 dans le meilleur des cas, 14 dans le pire des cas. Pour 80 puits, il faut 15 secondes en moyenne, avec un pic à 20 secondes. Dans le meilleur des cas, on arrive à 11 secondes, ce qui est moins bien que le pire des cas pour la programmation linéaire. Pour 120 puits, de manière intéressante, le temps d'exécution diminue pour l'algorithme policy iteration, la moyenne retombant à 11 secondes avec un minimum à 8 secondes et un maximum à 17 secondes.
- **Pour $n=30$:** l'algorithme de programmation linéaire converge en 21 secondes en moyenne. Pour un seul puits, l'algorithme policy iteration prend seulement 10 secondes en moyenne (minimum de 6 secondes, maximum de 13 secondes). Si on augmente le nombre de puits: pour $n_{\text{puits}}=50$, policy iteration met en moyenne 25 secondes, mais met jusqu'à 45 secondes dans le pire cas. Au-delà, si on continue à faire augmenter la valeur des puits, on arrive à des erreurs mémoire fréquemment ce qui rend les tests assez compliqués à effectuer.

Conclusion

Au travers de ce projet, nous avons implémenté des algorithmes de programmation linéaire et dynamique répondant au problème de plus court chemin stochastique, puis nous avons modélisé et généré des instances de processus décisionnels markoviens afin de pouvoir tester l'efficacité de ces différents algorithmes.

Il semble que le choix d'instance effectué rende inefficace l'algorithme value iteration, sans doute en raison des très forts coûts sur les puits (coût égal à n^3). Ces coûts font sans doute "exploser" la valeur de la différence en norme entre deux itérations, ce qui rallonge considérablement le nombre d'itérations à effectuer avant d'atteindre le seuil epsilon. En revanche, pour les deux autres algorithmes, on a pu observer des résultats intéressants sur les instances de type montagne : nous avons fait varier tous les paramètres à notre disposition, à savoir, la taille, le nombre de noeuds à éviter, les probabilités, la forme du relief, et les performances des algorithmes policy iteration et programmation linéaire semblent comparables (n'oublions pas que policy iteration est codé en Python alors que la programmation linéaire utilise Gurobi qui est codé en C). En particulier, pour des instances avec peu de puits, policy iteration est largement préférable à la programmation linéaire.

Il reste donc la question de savoir dans quelle mesure un algorithme primal-dual pourrait être compétitif avec des algorithmes de programmation dynamique.

Annexe

6.1 Le fichier mrDupont.txt :

```
c graphe tire de : Planning Journey in an Uncertain Environment The Stock  
p 7 9 13  
a 1 2 2  
a 1 3 1  
a 1 4 45  
a 2 1 2  
a 3 6 35  
a 4 7 20  
a 5 8 30  
a 6 9 70  
a 2 5 3  
e 1 1 1  
e 2 2 0.1  
e 2 3 0.9  
e 3 4 0.2  
e 3 5 0.7  
e 3 6 0.1  
e 4 7 1  
e 5 2 0.1  
e 5 3 0.9  
e 6 7 1  
e 7 7 1  
e 8 7 1  
e 9 7 1
```

6.2 Le fichier solutionMDP.h :

```
/* Marmote and MarmoteMDP are free softwares: you can redistribute  
it and/or modify it under the terms of the GNU General Public  
License as published by the Free Software Foundation, either  
version 3 of the License, or (at your option) any later version.
```

Marmote is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Marmote. If not, see <<http://www.gnu.org/licenses/>>.

Copyright 2018 EMmanuel Hyon, Alain Jean–Marie Abood Mourad*/

```
#ifndef SOLUTIONMDP_H
#define SOLUTIONMDP_H
```

```
/**
 * @brief Class solutionMDP: implementation of an abstract
 * solutionMDP class.
 *
 * This class is to be inherited by other classes.
 * @author Abood Mourad, and E. Hyon, lip6
 * @date 18 jan 2018
 * @version 2
 *
 */
class solutionMDP
{
public:

    /**
     * @brief Constructor to create solutionMDP object.
     * @author Hyon
     * @version 1
     * @date feb 2018
     */
    solutionMDP();

    /**
     * @brief the destructor of the object solutionMDP
     * @author EH
     * @version 1
     * @date feb 2018
     */
    virtual ~solutionMDP();

    /**
     * @brief A function to print the solution object.
     * @author Abood Mourad.
     * @date feb 2018
     * @return none.
     */
```

```
*/
    virtual void writeSolution();

/**
 * @brief setter of the size.
 * @author EH
 * @date mar 2018
 * @return none.
 */
void setSize(int s);

protected:
    int _size;                                /**< _size of the solution */
};

#endif // SOLUTIONMDP_H
```

6.3 Le fichier solutionMDP_SWIG.h :

`/* Marmote and MarmoteMDP are free softwares: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.`

`Marmote is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.`

`You should have received a copy of the GNU General Public License along with Marmote. If not, see <http://www.gnu.org/licenses/>.`

`Copyright 2018 EMmanuel Hyon, Alain Jean-Marie Abood Mourad*/`

```
/**
 * @brief Class solutionMDP: implementation of an abstract
 * solutionMDP class.
 *
 * This class is to be inherited by other classes.
 * @author Abood Mourad, and E. Hyon, lip6
 * @date 18 jan 2018
 * @version 2
 */
```

```
class solutionMDP
{
public:
```

```
    /**
     * @brief Constructor to create solutionMDP object.
     * @author Hyon
     * @version 1
     * @date feb 2018
     */
    solutionMDP();
```

```
    /**
     * @brief the destructor of the object solutionMDP
     * @author EH
     * @version 1
     * @date feb 2018
     */
    virtual ~solutionMDP();
```

[illegible]

Bibliography

- [1] Gautier Stauffer, Matthieu Guilloit. *The Stochastic Shortest Path Problem: A polyhedral combinatorics perspective*. European Journal of Operational Research, 2018.
- [2] Frédéric Garcia. *Processus Décisionnels de Markov en Intelligence Artificielle*. Groupe PDMIA, INRIA Lille, 2008.
- [3] Christos H. Papadimitriou, Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Inc, 1982.
- [4] Mickael Randour, Jean-François Raskin, Ocan Sankur. *Variations on the Stochastic Shortest Path Problem*. Invited paper for VMCAI 2015.
- [5] D. P. Bertsekas. *Dynamic programming and optimal control. Volume II*. Athena Scientific optimization and computation series. Belmont, Mass. Athena Scientific, 2012.
- [6] *Documentation SWIG* disponible sur : <http://www.swig.org/doc.html>, 2019