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

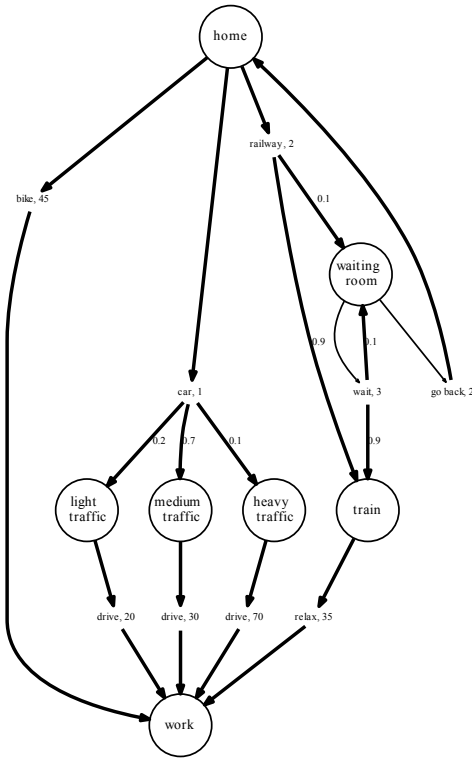
Janvier-Juin 2019

Sommaire

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.



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.

Modèle mathématique et algorithmes

Dans cette section, nous présentons le modèle 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 $\{S, A, T, C\}$ 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

Dans le cadre de ce projet, nous faisons le choix de reprendre les notations de MM. Stauffer et Guillot. 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 :

- $S = \{1, 2, \ldots, n\}$ est fini
- $A = \{1, 2, \ldots, m\}$ est fini
- $J \in \mathscr{M}_{m,n}(\{0,1\})$, avec $J(a,s) = 1$ ssi l'action a est disponible dans l'état s
- $P \in \mathscr{M}_{m,n}(\mathbb{R})$, 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 $\{1, \ldots, m\}$, 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

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

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, c'est-à-dire si la règle de décision associant une action à un état ne dépend que de l'état courant
- **propre** (au sens de MM Stauffer et Guillot) 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. 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.

2.2 Algorithmes de résolution

On va maintenant s'intéresser aux différents algorithmes pré-établis 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 :

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

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 on 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++.
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).:

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 "alglin.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
```

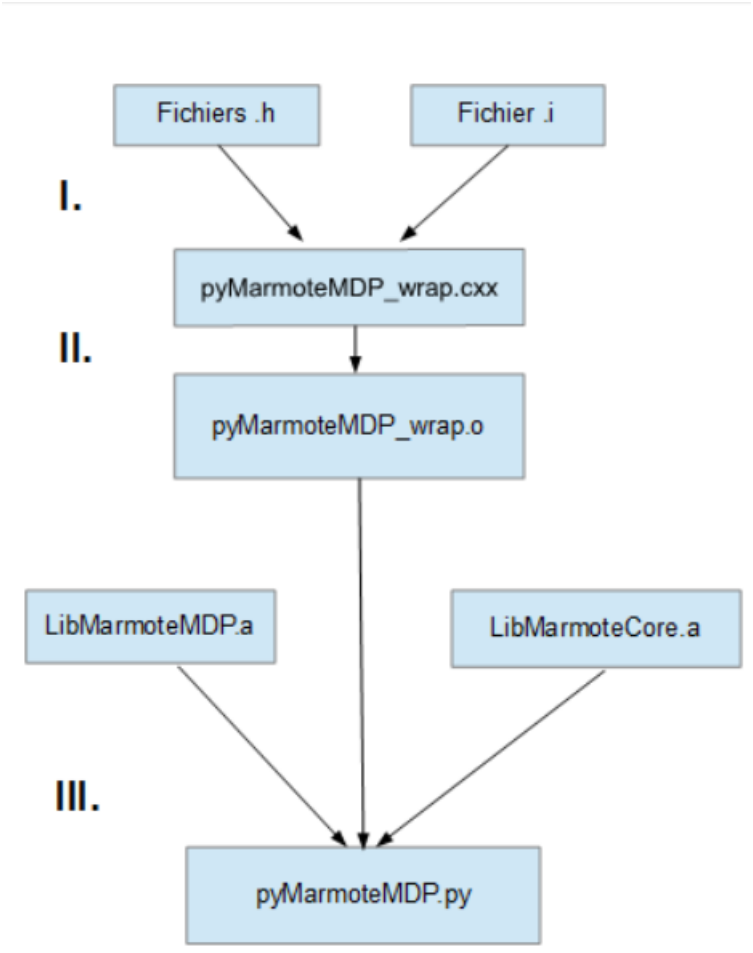
Puis lancer la commande :

```
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 interfacer.



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

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

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).

3.2 Programmation linéaire

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

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

Le dual de ce programme linéaire s'écrit ainsi :

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

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). A 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), soit 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 aient 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 préfacer chacune de ces lignes par c + un espace.

3.4.2 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.

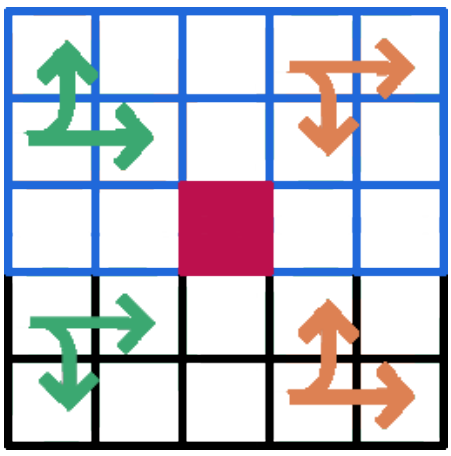


Figure 3.1: Instance type montagne

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 + ε
    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 rend l'instance moins intéressante).

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

- Si `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 $n <$ `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 $2n <$ `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 fichier instance (indicé 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).

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 language, dans la mesure du possible.

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 fonction 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 reprécisons donc pas 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 lite 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 instanciations 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 commence 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 instanciations de la politique de son choix (une même politique peut mener à différents instanciations du fait que les actions ont des résultats aléatoires).

Voici un exemple de visualisation pour `n = 15` avec `npuits = 16` et `relief = 30`.

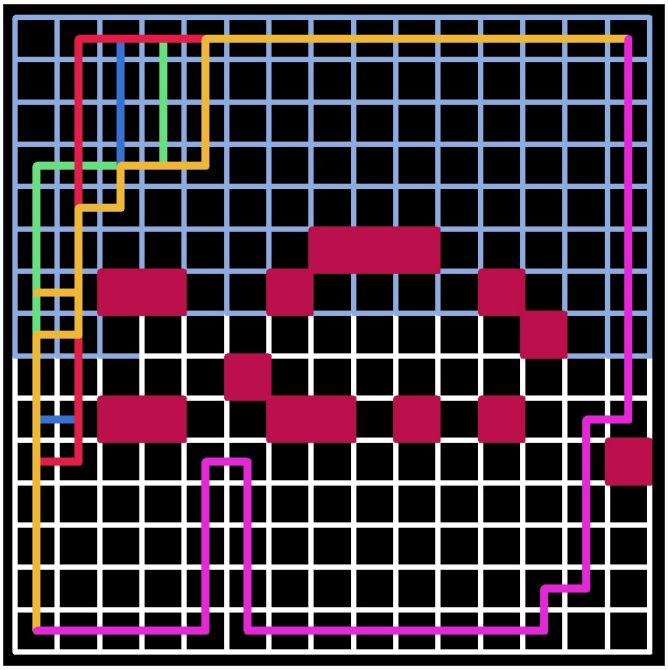


Figure 4.1: Chaque tracé correspond à une instanciation d'une politique optimale

4.3 Comparaison de vitesse sur les instances montagne

- **Pour n$\leq$10 :** Pour la PL ainsi que pour l'algorithme policy iteration, on obtient une politique optimale quasiment instantanément (2 centièmes de seconde pour policy iteration, 15 centièmes de seconde pour la PL pour n=10). Pour l'algorithme value iteration, il faut déjà compter 2 secondes avant d'obtenir un résultat (pour n=10).

- **Pour n=15 :** Pour la PL, on obtient une politique optimale en environ 0.8 seconde, contre 0.1 seconde pour policy iteration. Enfin, pour l'algorithme value iteration, on a un temps d'exécution de plus d'une minute en moyenne.
  On conjecture que le choix de la valeur limite causant l'arrêt de l'algorithme (i.e. le delta de valeur entre 2 itérations) est mauvais. Il semble difficile de le choisir de manière adéquate, particulièrement dans ce modèle où les puits ont tous des poids extrêmement importants (dans le code, on a pris un poids de n^{10}, donc quand n augmente, les poids des puits augmentent de façon exponentielle et donc la différence en norme des valeurs entre deux itérations successives augmente aussi exponentiellement avec n). **Pour la suite, on se limite aux tests PL et policy iteration.**

- **Pour n=20 :** L'algorithme policy iteration continue à prévaloir sur la PL (d'autant plus qu'il tourne en Python et est donc désavantagé par rapport à l'algorithme de PL) : entre 0.4 et 0.5 seconde contre entre 2.9 et 4 secondes pour la PL.

- **Pour n$\geq$25 :** Le rapport de rapidité d'exécution de 1/8 semble se maintenir entre les deux algorithmes : autour d'une seconde pour policy iteration contre autour de 8.5 secondes pour la PL pour n=25.

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.

Nous avons quelques réserves à exprimer au sujet des résultats obtenus : nous n'avons pas fait assez de tests pour pouvoir conclure avec précision quant à l'efficacité de l'algorithme value iteration. Le choix de la valeur d'arrêt semble centrale et demander de la réflexion (ce qui n'aurait peut être pas été le cas si nous avions réussi à faire tourner les algorithmes fournis par M. Hyon). En revanche, pour 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), l'algorithme policy iteration est de loin plus performant que l'algorithme de 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. En effet, les tests effectués semblent indiquer qu'il faudrait au moins que l'algorithme primal-dual soit 8 fois plus performant que l'algorithme de programmation linéaire "classique" pour rivaliser avec l'algorithme policy iteration.

Annexe

6.1 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
```

6.2 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();
```

```
  /**
  * @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 */
};
```

Bibliography

[1] Gautier Stauffer, Matthieu Guillot. *The Stochastic Shortest Path Problem: A polyhedral combinatorics perspective*. European Journal of Operational Research, 2018.

[2] Frédérick 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.