Université Libre de Bruxelles

INFO-F311
Intelligence Artificielle

Synthèse IA

Étudiants :
Rayan Contuliano Bravo

Enseignants :
T. Lenaerts

11 octobre 2023

Contents

1 Introduction

Définition 1.1. Qu'est-ce que l'ia

L'intelligence artificielle est une branche de l'informatique qui crée des systèmes qui pensent de manière **rationnelle**

Définition 1.2. Décisions rationnelles

Penser de manière rationnelle signifie qu'on va se concentrer sur le **choix de décisions** qui *maximisent la probabilité* d'atteindre un objectif donné. On va faire agir les systèmes de manière **optimale**

Remarques:

1. Être rationel signifie donc **maximiser** l'utilité attendue.
2. On définis un objectif par son **utilité**.

Définition 1.3. Agent

Un agent est un système qui perçoit son environnement par des **capteurs** et agit sur celui-ci par des **effecteurs**.

Définition 1.4. Agent rationnel

Un agent rationnel est un agent qui agit de manière à maximiser son utilité attendue.

Remarque: Les **capteurs**, **effecteurs** et l'**environnement** permettent à l'agent de percevoir et d'agir sur le monde de manière **rationnelle**. L'**agent** est le système qui prend les décisions.

Définition 1.5. Fonction agent

La fonction agent est une fonction qui prend en entrée une séquence de perceptions et retourne une action. $f : \mathcal{P}^* \to \mathcal{A}$

Exemple:

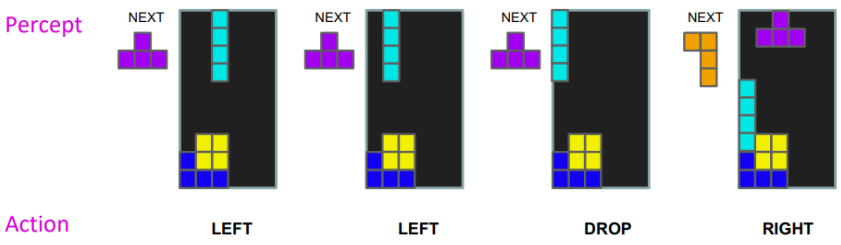

FIGURE 1.1 – Représentation fonction agent dans le jeu Tetris

Définition 1.6. Programe Agent

Un **programe agent** l est exécuté sur une **machine** M afin d'implémenter la fonction agent f.

Remarque: Les machines dans le monde réel sont **imparfaites** et **limitées** en temps et en mémoire.

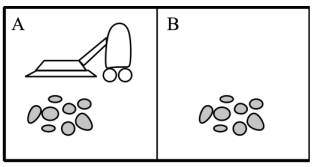

FIGURE 1.2 – Etat de l'environnement de l'aspirateur

Exemple: Nous pouvons représenter un aspirateur comme un agent qui perçoit son environnement par des capteurs et agit sur celui-ci par des effecteurs.

- **Perception** : capteurs qui détectent la saleté et sa localisation dans l'espace
- **Action** : effecteurs qui déplacent l'aspirateur dans l'espace et aspire ou non

En imaginant la situation en figure 1.2, nous pouvons définir la fonction agent de l'aspirateur comme suit :

TABLE 1 – Fonction agent de l'aspirateur

Sequence de perception	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [B, Clean]	Left
[A, Clean], [B, Dirty]	Suck
etc...	etc...

Pour que notre agent soit bien rationnel, il nous faut une manière de **mesurer** la **performance** de celui-ci. Pour cela, nous allons définir une **fonction de performance** qui va mesurer la qualité des actions de l'agent.

Exemple: On peut lui faire gagner des points ou bien lui en retirer en fonction d'une action

De cette manière, l'agent va savoir quelles actions lui permettent de **maximiser** son utilité attendue.

Afin de bien déterminer un environnement, les particularité de notre agents, il nous faut **avant toute chose** définir **PEAS**

Définition 1.7. PEAS

- **Performance** : mesure de la qualité des actions de l'agent
- **Environnement** : type d'environnement dans lequel l'agent va évoluer
- **Actuateurs** : les effecteurs de l'agent
- **Sensors** : les capteurs de l'agent

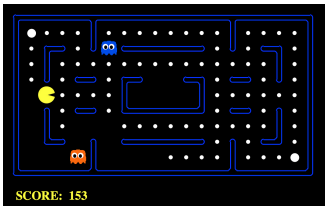

FIGURE 1.3 – Environnement Pacman

Exemple: Pour l'environnement Pacman de la figure 1.3, nous pouvons définir PEAS comme suit :

- **Performance** : -1/pas, +10/nourriture, +500/partie gagnées, -500/mort, +200/tuer un fantôme effrayé
- **Environnement** : labyrinthe **dynamique** de pacman
- **Actuateurs** : Haut, Bas, Gauche, Droite
- **Capteurs** : L'état entier visible

Définition 1.8. Types d'environnement

Il y a plusieur type d'environnement :

- **Mono-agent** : un seul agent
- **Multi-agent** : plusieurs agents qui maximisent leur **propre** tâche (coop ou concurentiel)
- **Déterministe** : l'état de l'env est déterminé **seulement** par les actions de l'agent
- **Stochastique** : l'environnement est non déterministe
- **Épisodique** : les actions de l'agent n'affectent pas les actions futures
- **Séquentiel** : les actions de l'agent affectent les actions futures
- **Dynamique** : l'environnement peut changer pendant que l'agent réfléchit
- **Statique** : l'environnement ne change pas pendant que l'agent réfléchit
- **Complètement observable** : les capteurs de l'agent perçoivent l'état complet de l'environnement
- **Partiellement observable** : les capteurs de l'agent perçoivent une partie de l'état de l'environnement
- **Discret** : un nombre fini d'états
- **Continu** : un nombre infini d'états
- **Connu** : l'agent connait les lois de l'environnement

Il existe plusieurs types d'agents qui répondent à des environnements plus complexes :

- **Agent réflexe simple** : l'agent choisit son action en fonction de la **dernière** perception
- **Agent réflexe basé sur un modèle** : l'agent choisit son action en fonction de la **dernière** perception et d'un **état interne**(dépend de l'**historique** des perceptions)
- **Agent fondés sur des buts** : l'agent choisit son action en fonction de la dernière perception ainsi que des infos relatives à l'objectif
- **Agent fondés sur l'utilité** : l'agent choisit son action en fonction de sa satisfaction par rapport à l'état résultant

2 Recherche Non-Informée

Définition 2.1. Recherche non-informée

La recherche **non-informée** est une stratégie de recherche qui n'utilise **pas** d'information sur l'état de l'environnement afin de le **guider** pour trouver une solution. Elle explore simplement l'espace de recherche de manière systèmatique. En utilisant souvent des *algorithmes* comme **DFS**, **BFS**.

Remarque: La recherche non-informée est utilisée quand on ne connait pas l'état de l'environnement. Lorsqu'on ne peut quantifier la qualité d'un état en utilisant des **information heuristiques**

Définition 2.2. Agent de Plannification

Les agents de plannification font des **hypothèses** sur les conséquences des actions entreprises et utilisent un **modèle** de l'environnement pour trouver un plan qui atteint son objectif.

Résolution de problèmes par la recherches :

1. **Formulation de l'objectif** : L'agent doit avoir un objectif afin de pouvoir organiser son comportement. Ca permet de limiter l'espace de recherche (*actions entreprises*)
2. **Formulation du problème** : L'agent doit avoir un moyen de représenter les actions et les états afin de pouvoir les manipuler
3. **Recherche de la solution** : Avant d'agir dans le monde réel, l'agent fait une simulation de séquences d'actions dans son modèle de l'environnement jusqu'à trouver un séquence qui mène à l'objectif. C'est la *solution*
4. **Exécution de la solution** : L'agent exécute la séquence d'actions dans le monde réel

Note:-

Un plan est une séquence d'actions qui mène à l'objectif.

2.1 Problème de recherches

Définition 2.3. Problème de Recherche

Un problème de recherche est défini par :

- **Ensemble d'État** S : Une situation dans lequel l'environnement peut être agencé
- **État initial** s_o : l'état dans lequel le problème commence
- **Actions** $A(s)$: les actions possibles dans l'état s
- **Modèle de Transition** $Result(s, a)$: la fonction qui définit les conséquences des actions
- **Solution** : Une séquence d'actions qui mène de l'état initial à l'état final
- **État final** : l'état que l'on veut atteindre

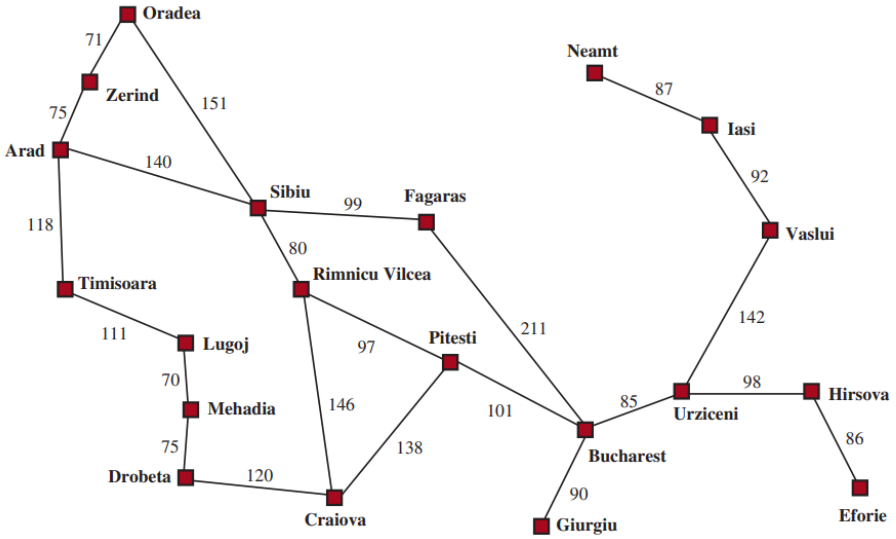

FIGURE 2.1 – Représentation simple de la Roumanie en graphe

Exemple: Voyage en Roumanie :
- **États** : les villes de Roumanie
- **État initial** : Arad
- **Actions** : les routes entre les villes adjacentes
- **Modèle de transition** : Atteindre une ville adjacente
- **Cout de l'action** : distance entre les villes
- **État final** : Bucharest

2.2 Graphe d'espace d'état

Définition 2.4. Graphe d'espace d'état

Un graphe d'espace d'état est un graphe qui représente les états et les actions possibles.
- **Noeuds** : les états
- **Arêtes** : les actions

L'état initial est le noeud racine et l'état final est un noeud (ou plusieurs ?).

⚠ Dans ce genre de graphe, chaque état n'est représenté qu'**une seule fois**.

Remarque: Il est fortement possible de ne pas pouvoir représenter un problème de recherche par un graphe d'espace d'état car il y a **trop d'états** ou que les états sont **continus**.

2.3 Arbres de Recherches

Définition 2.5. Arbre de Recherche

Un arbre de recherche est un arbre qui représente les états et les actions possibles.
- **Noeuds** : les états, plans pour arriver à ces états
- **Arêtes** : les actions
- **Enfants** : les états suivants (*succeseurs*)

L'état initial est le noeud racine et l'état final est un noeud (ou plusieurs ?). Les noeuds peuvent être représentés plusieurs fois, (**il est donc plus grand qu'un graphe d'espace**

d'état.)

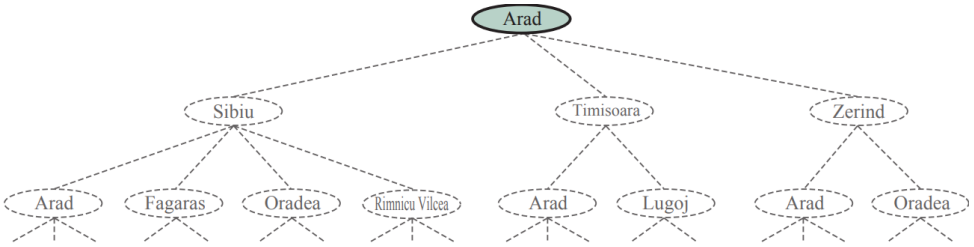

FIGURE 2.2 – Arbre de recherche de la figure 2.1

2.3.1 Recherche dans un arbre de recherche

Recherche dans un Arbre 1 Algorithme de recherche

```
function TREE-SEARCH(problème, stratégie)
    initialise un noeud avec l'état initial du problème
    loop
        if il ne peut plus y avoir d'état à explorer then
            return Erreur
        end if
        choisis un noeud non exploré selon la stratégie
        if le noeud est l'état final then
            return le plan qui mène à l'état final
        else
            Développe le noeud et ajoute ses enfants à l'arbre
        end if
    end loop
end function
```

Remarques:

1. La frontière est l'ensemble des noeuds construits non explorés de l'arbre
2. Pour développer un noeud de la frontiere, on le retire de la frontière et on l'ajoute à l'ensemble des noeuds explorés ses enfants sont ajoutés à la frontière
3. La recherche dans un graphes est similaire à la recherche dans un arbre sauf qu'on doit vérifier si un noeud a déjà été visité avant de l'ajouter à la frontière

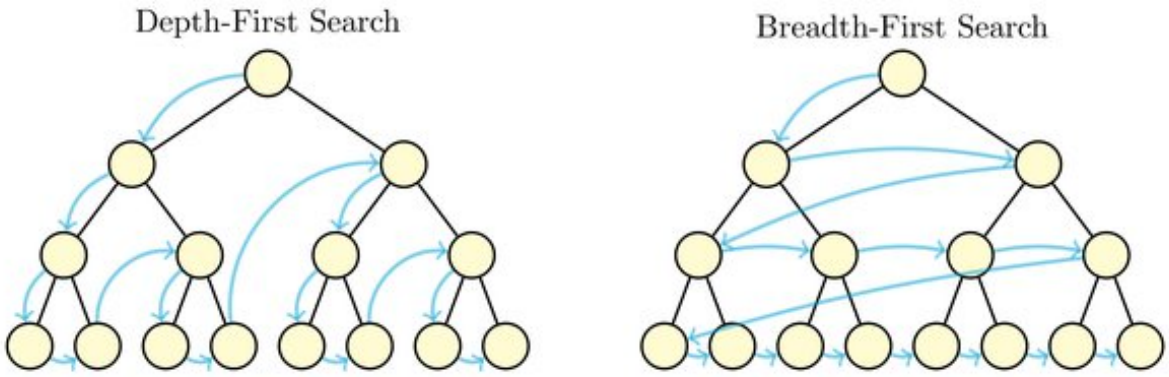

FIGURE 2.3 – Sens d'éxécution BFS et DFS

2.4 DFS

Définition 2.6. DFS

La recherche en profondeur (**DFS**) est une stratégie de recherche qui explore l'arbre en allant le plus loin possible dans une branche avant de revenir en arrière.

- **Frontière** : une pile (LIFO)
- **Stratégie** : on choisit le noeud le plus profond de la frontière

2.5 BFS

Définition 2.7. BFS

La recherche en largeur (**BFS**) est une stratégie de recherche qui explore l'arbre en allant le plus large possible dans une branche avant de revenir en arrière.

- **Frontière** : une file (FIFO)
- **Stratégie** : on choisit le noeud le moins profond de la frontière

DFS est meilleur que **BFS** dans les cas suivant :

- Si il y a des limitations de mémoire

BFS est meilleur que **DFS** dans les cas suivant :

- Si on veut trouver la solution la plus courte

2.6 Iterative deepening

Note:-

L'idée est d'avoir les avantages mémoire de **DFS** et la solution optimale de **BFS**

Définition 2.8. Iterative deepening

L'exploration itérative en profondeur (**Iterative deepening**) est une stratégie de recherche qui explore l'arbre en faisant une recherche en profondeur avec une limite de profondeur de 1, puis 2, puis 3, etc. jusqu'à ce que la solution soit trouvée.

Remarque: Même si cette algorithme visite plusieurs fois les mêmes noeuds, ça n'a pas vraiment d'impact car le nombre de noeuds est réduit (car on mise de le trouver avant d'atteindre la limite de profondeur)

2.7 UCS

Définition 2.9. UCS

La recherche par coût uniforme (**UCS**) est une stratégie de recherche qui explore l'arbre en allant le plus loin possible dans une branche avant de revenir en arrière.

— **Frontière** : une file de priorité
— **Stratégie** : on choisit le noeud ayant le plus petit coût de la frontière

Pour ananlyser un algorithme, on va utilser ces différentes propriétés :

— **Complet** : l'algorithme trouve toujours une solution si elle existe
— **Optimal** : l'algorithme trouve toujours la solution optimale (avec le plus petit coût)
— **Complexité en temps** : Combien de temps l'algorithme prend pour trouver une solution
— **Complexité en espace** : Combien de mémoire l'algorithme prend pour trouver une solution

TABLE 2 – Comparaison stratégie d'exploration

Critère	Largeur	Cout uniforme	Profondeur	Profondeur itérative
Complet	Oui	Oui	Non	Oui
Optimal	Oui	Oui	Non	Oui
Temps	$O(b^d)$	$O(b^{\frac{C^*}{\epsilon}})$	$O(b^m)$	$O(b^d)$
Espace	$O(b^d)$	$O(b^{\frac{C^*}{\epsilon}})$	$O(bm)$	$O(bd)$

3 Recherche Informée

Définition 3.1. Recherche Informée

La recherche **informée** est une stratégie de recherche qui utilise **des informations** sur l'état de l'environnement afin de le **guider** pour trouver une solution. Elle explore l'espace de recherche de manière **intelligente** grâce à des **heuristiques**. Ces heuristiques permettent de **quantifier** la **qualité** d'un état et donc de guider la recherche vers les états les plus prometteurs.

Remarque: La performance de ces algorithmes dépendent de la qualité de l'heuristique utilisée.

Définition 3.2. Heuristique

Une heuristique est une fonction qui permet d'estimer à quel point un état est **proche** de l'état final. Elles sont designées pour des problèmes spécifiques.

Exemples:

1. **Distance Euclidéenne** : la distance entre deux points en ligne droite

2. **Distance de Manhattan** : la distance entre deux points en ligne droite mais en ne pouvant se déplacer que sur les axes (pas en diagonale)

3.1 Greedy Best-First Search

Définition 3.3. Greedy Best-First Search

La recherche gloutonne (**Greedy Best-First Search**) est une stratégie de recherche qui explore l'arbre en choisissant le noeud qui semble le plus prometteur.

— **Frontière** : une file de priorité
— **Stratégie** : on choisit le noeud ayant la plus petite heuristique de la frontière

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

FIGURE 3.1 – Tableau de valeure d'heuristique

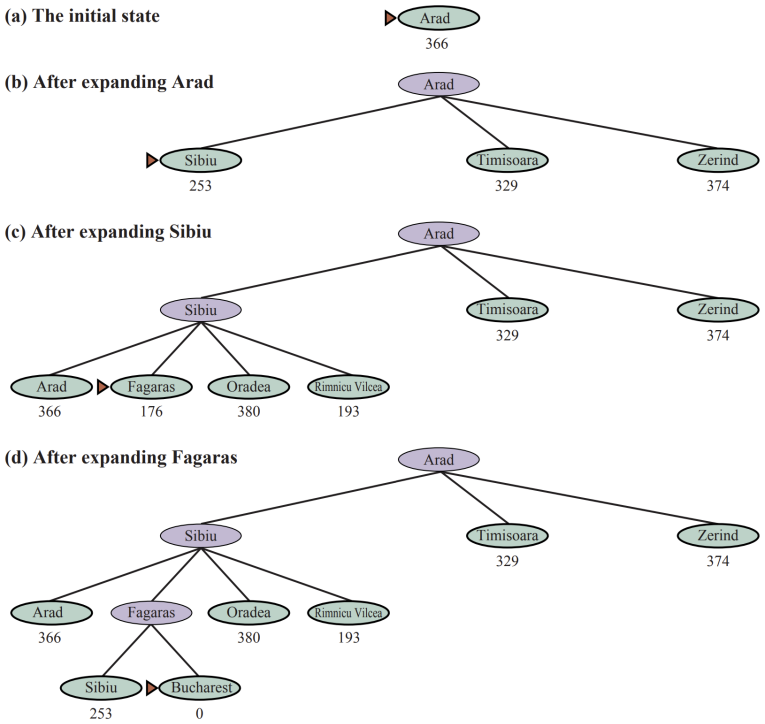

FIGURE 3.2 – Exécution de l'algo greedy

Remarques:

1. **Complet** : Non, peut aller dans des profondeurs infinies
2. **Optimal** : Non
3. **Complexité en temps** : $O(b^m)$
4. **Complexité en espace** : $O(b^m)$
5. Il est en général meilleur que **DFS**

3.2 A*

Définition 3.4. A*

L'algorithme **A*** est une stratégie de recherche qui explore l'arbre en choisissant le noeud qui semble le plus prometteur.

— **Frontière** : une file de priorité
— **Stratégie** : on choisit le noeud ayant le plus petit coût de la frontière

La fonction d'évaluation est la suivante : $f(n) = g(n)+h(n)$ où $g(n)$ est le coût pour atteindre le noeud n depuis la racine et $h(n)$ est le cout du noeud n pour aller jusqu'au but.

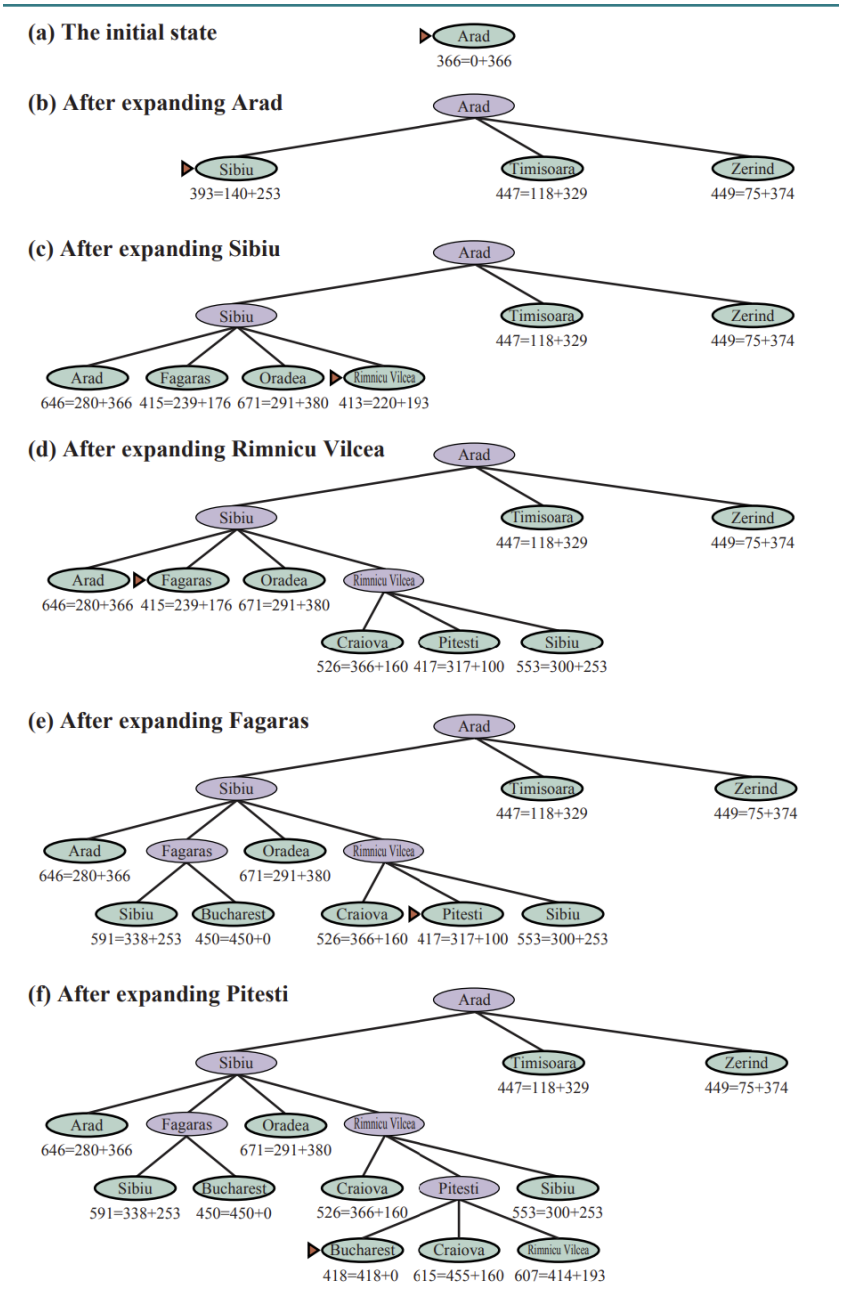

FIGURE 3.3 – Exécution de l'algo A*

Remarques:

1. **Complet** : Oui
2. **Optimal** : Oui si $h(n)$ est admissible
3. **Complexité en temps** : $O(b^d)$
4. **Complexité en espace** : $O(b^d)$

3.3 Creer des Heuristiques admissibles

Définition 3.5. Admissible

Une heuristique est admissible si elle ne surestime jamais le coût pour atteindre l'état final. $h(n) \leq h^*(n)$ où $h^*(n)$ est le coût réel pour atteindre l'état final. C'est une fonction **optimiste**

Définition 3.6. Cohérent

Une **heuristique** est **cohérente** si, pour chaque noeud n et un successeur n' de n généré par une action a, le coût estimé pour atteindre l'état final depuis n est inférieur ou égal au coût de l'action a plus le coût estimé pour atteindre l'état final depuis n'. $h(n) \leq c(n, a, n') + h(n')$ où $c(n, a, n')$ est le coût de l'action a pour aller de n à n'.

Remarque: Une heuristique cohérente est toujours admissible.

Note:-

Lorsqu'on compare 2 heuristiques, on peut dire que l'heuristique h_1 est meilleure que h_2 si $h_1(n) \leq h_2(n) \forall n$

Afin de créer une heuristique admissible, on peut utiliser les méthodes suivantes :

- **Relaxation** : on simplifie le problème en enlevant des contraintes
- **Décomposition** : on décompose le problème en sous-problèmes qui sont plus facile à résoudre
- **Combinaison** : on combine plusieurs heuristiques

Il est aussi possible de stocker les valeures réelles dans une **base de donnée** et de les utiliser pour calculer l'heuristique (qui est dcp dans la bdd).

4 Recherche Locale

à faire

5 Recherche Adversarial

Dans les chapitres précédents, nous avons vu comment résoudre des problèmes de recherche. Dans ce chapitre, nous allons nous concentrer sur les problèmes où nous devons **battre nos adversaire** dans des jeux à **deux ou plusieurs joueurs**.

5.1 Définition d'un jeu

Définition 5.1. Jeu

Un jeu est un problème de recherche avec les caractéristiques suivantes :

- **État initial,** s_0 : la position initiale du jeu.
- **Joueur,** $Player(s)$: le joueur qui doit jouer.
- **Actions,** $Action(s)$: les coups possibles pour un joueur.
- **Modèle de transition,** $Result(a, s)$: L'état s' qui résulte de l'action a dans l'état s
- **Fonction de terminal,** $isTerminal(s)$: la fonction qui détermine si le jeu est terminé
- **Utilité,** $Utility(s, p)$: la fonction qui détermine le score du jeu sur les états terminaux. Qui gagne,et combien. p est le joueur.

Note:-

Il y a plusieurs types de jeux :

- **Jeux à somme nulle** : la somme des utilités des joueurs est toujours égale à 0. Si l'un **gagne**, l'autre **perd**.
- **Jeux à somme général** : Les agents peuvent avoir des utilités différentes. Si l'un **gagne**, l'autre **ne perd pas forcément**. (*Coop*, ...)
- **Jeux d'équipes** : Les agents jouent en équipe contre d'autres agents.

5.2 Minimax

Définition 5.2. Minimax

L'algorithme **Minimax** est un algorithme qui permet de trouver le meilleur coup à jouer dans un jeu **déterministe** à **somme nulle**. Le principe est simple, nous imaginons 2 joueurs qui jouent l'un contre l'autre, **Min** et **Max**. Le but de **Max** est de maximiser l'utilité et le but de **Min** est de minimiser l'utilité sachant que l'autre joueur va jouer de manière optimale.

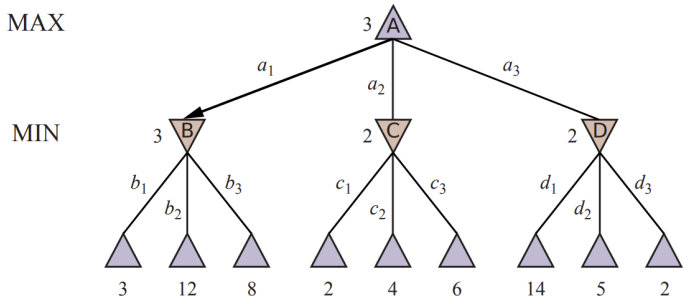

FIGURE 5.1 – Minimax dans un arbre

Minimax 2 Algorithme Minimax

```
function MINIMAX-SEARCH(game, state)
    player ← game.TO-MOVE(state)
    if player = max then
        value, action ← MAX-VALUE(game, state)
    else
        value, action ← MIN-VALUE(game, state)
    end if
    return action
end function

function MAX-VALUE(game, state)
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ← −∞, null
    for action in game.ACTIONS(state) do
        value2, action2 ← MIN-VALUE(game, game.RESULT(state, action))
        if value2 > value then
            value, action ← value2, action2
        end if
    end for
     return value, action
end function

function MIN-VALUE(game, state)
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ← ∞, null
    for action in game.ACTIONS(state) do
        value2, action2 ← MAX-VALUE(game, game.RESULT(state, action))
        if value2 < value then
            value, action ← value2, action2
        end if
    end for
    return value, action
end function
```

L'algorithme va généré l'arbre de recherche jusqu'aux étas **terminaux** pour utiliser la fonction d'utilité. Il traite ensuite ces valeurs en remontant l'arbre et les choisit en fonction du joueur qui doit jouer (**Min** ou **Max**)

Note:-

Pour ne pas généré tous l'arbre, on peut utiliser une **profondeur limitée**.

Si il y a plus de 2 joueurs, nous pouvons assigner à chaque noeud un tuple de valeurs qui représente l'utilité pour chaque joueur. Chaque joueur va alors choisir le coup qui maximise son utilité.

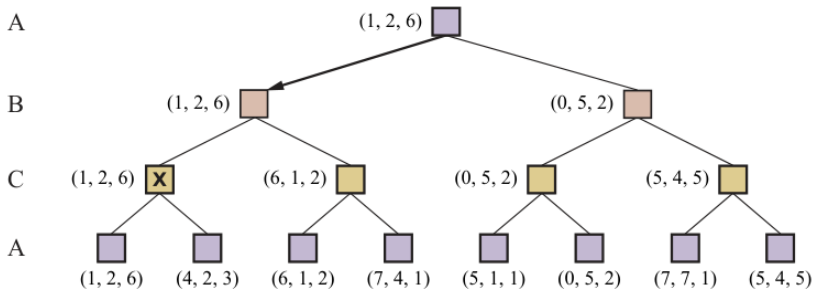

FIGURE 5.2 – Représentation en Multijoueur

Remarque: Cet algortihme est basé sur **DFS**.
- **Complexité en temps** : $O(b^m)$
- **Complexité en espace** : $O(bm)$

5.3 Alpha-Beta pruning

Définition 5.3. Alpha-Beta pruning

L'algorithme **Alpha-Beta pruning** est un algorithme qui permet de trouver le meilleur coup à jouer dans un jeu **déterministe** à **somme nulle**. Il est basé sur l'algorithme **Minimax** mais il va **couper** les branches qui ne sont pas intéressantes. Il utilise 2 paramètres, α et β qui représentent les valeurs minimales et maximales que l'on peut atteindre.
- α est la meilleure option (valeure la plus haute) que le joueur **Max** est assuré d'obtenir. **Au MOINS**.
- β est la meilleure option (valeure la plus basse) que le joueur **Min** est assuré d'obtenir. **Au PLUS**.

Remarque: Le meilleur situation pour cette algorithme est quand les meilleurs coups sont toujours le premier coup à être évalué (*plus à gauche*)

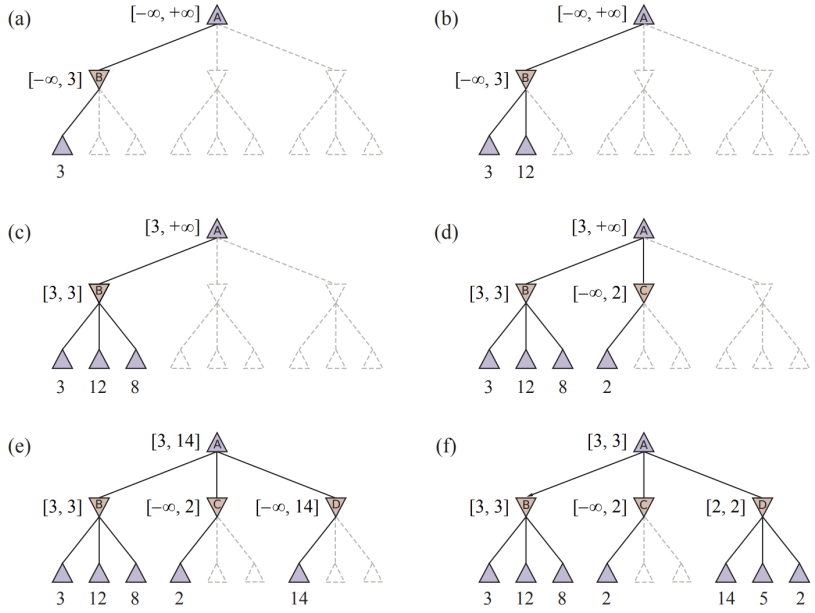

FIGURE 5.3 – Alpha-Beta pruning dans un arbre

Remarque: La complexité en temps est généralement plus petite que celle de minimax

- **Complexité en temps** : $O(b^{m/2})$ avec les noeuds dans le **bon** ordre et $O(b^{3m/4})$ si les noeuds sont visités aléatoirement.
- **Complexité en espace** : $O(bm)$ et $O(\sqrt{b})$ si les noeuds sont dans le bon ordre.

Si cet algorithme n'est pas suffisant, on peut utilisé une fonction d'évaluation qui permet d'évaluer les noeuds qui ne sont pas des états terminaux pour estimer leur utilité.

$\alpha - \beta$ pruning 3 Algorithme $\alpha - \beta$ pruning

```
function MINIMAX-SEARCH(game, state)
    player ← game.TO-MOVE(state)
    value, action ← MAX-VALUE(game, state, −∞, +∞)
    return action
end function

function MAX-VALUE(game, state, α, β)
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ← −∞, null
    for action in game.ACTIONS(state) do
        value2, action2 ← MIN-VALUE(game, game.RESULT(state, action), α, β)
        if value2 > value then
            value, action ← value2, action2
            α ← MAX(α, value)
        end if
        if v ≥ β then
            return value, action
        end if
    end forreturn value, action
end function

function MIN-VALUE(game, state, α, β)
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ← ∞, null
    for action in game.ACTIONS(state) do
        value2, action2 ← MAX-VALUE(game, game.RESULT(state, action), α, β)
        if value2 < value then
            value, action ← value2, action2
            β ← MIN(β, value)
        end if
        if v ≤ α then
            return value, action
        end if
    end for
    return value, action
end function
```