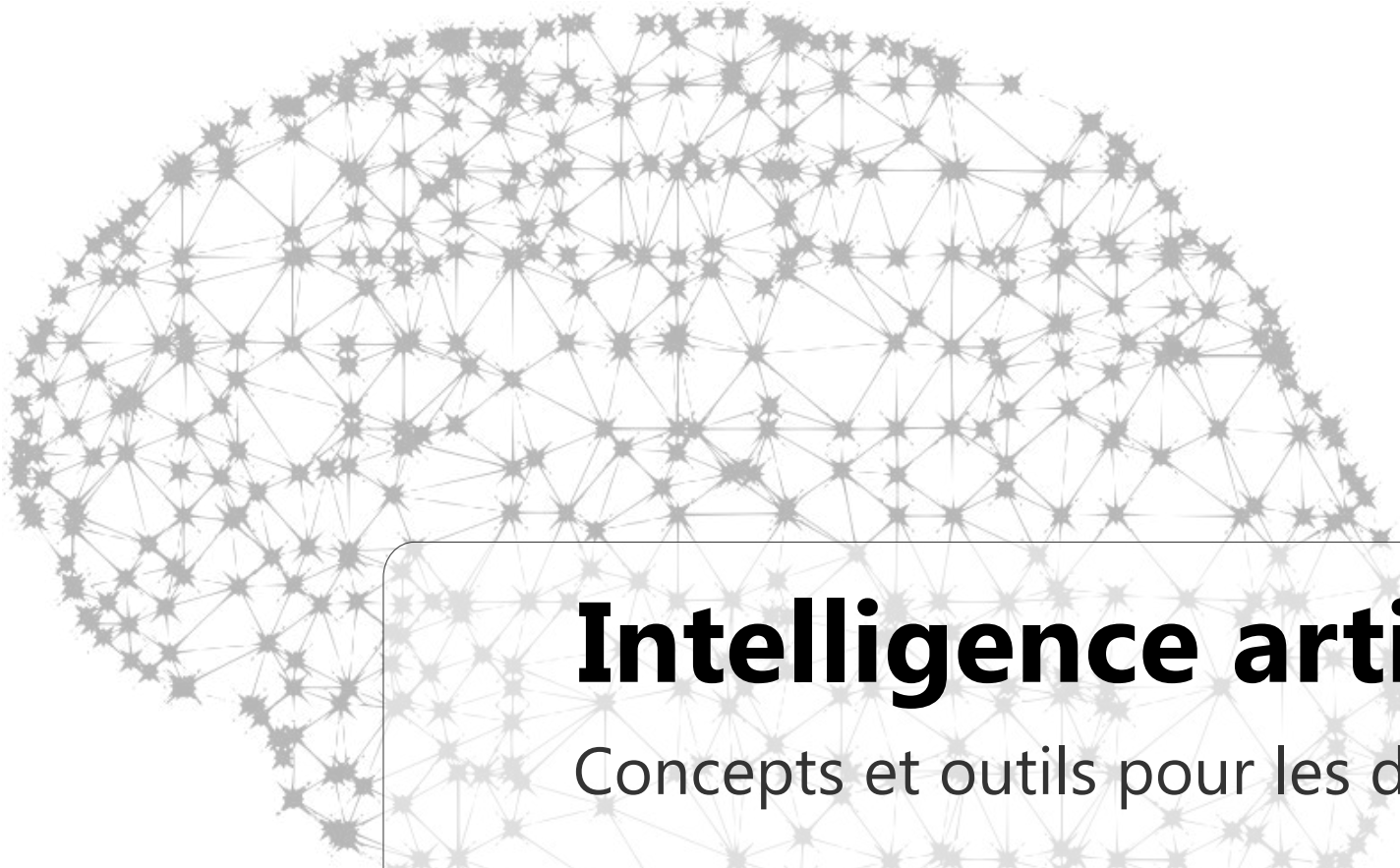


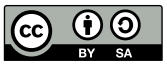
Intelligence artificielle

Concepts et outils pour les développeurs

Alain Lebret

Spécialité Informatique par apprentissage

L'École des INGÉNIEURS Scientifiques



Intelligence artificielle

Spécialité Informatique par apprentissage 2e année

Alain Lebret, 2024

Présentation de l'enseignement

Objectifs

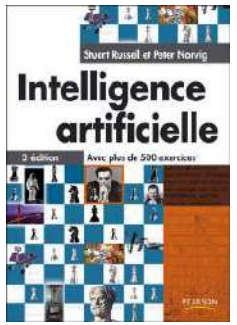
- Connaître différentes méthodes de l'intelligence artificielle
- Être capable de choisir une méthode afin de résoudre un problème
- En pratique : 11 h de cours, 8 h de TD et 14 h de TP
- Note finale : $\frac{1}{2} \times N_{\mathrm{E}} + \frac{1}{2} \times N_{\mathrm{TP}}$

Prérequis

- Algorithmique avancée
- Un langage objet (C++ ou Java)

Références

- Stuart Russel et Peter Norvig. *Intelligence artificielle* 3e édition. Pearson, 2010



Codes sources du livre : http://aima.cs.berkeley.edu

Introduction

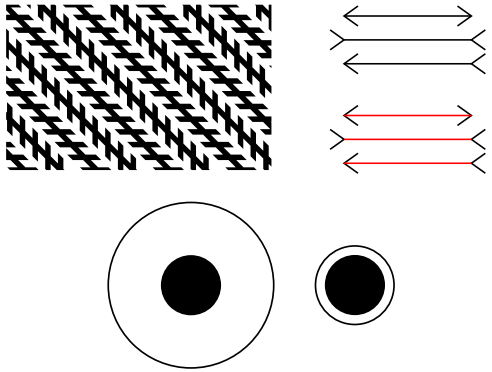
Présentation

L'intelligence artificielle est la "recherche de moyens susceptibles de doter les systèmes informatiques de capacités intellectuelles comparables à celles des êtres humains." La Recherche, janv. 1979, no 96, vol. 10, p. 61

Son objectif :

- Résoudre des problèmes difficiles
- Reproduire des raisonnements intelligents
- Simuler des phénomènes complexes
- S'adapter
- Optimiser une solution à un problème
- Apprendre
- etc.

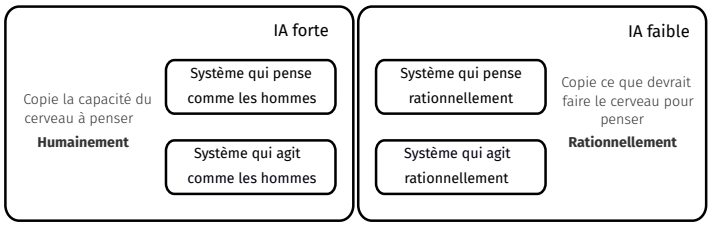
Certains problèmes sont triviaux ... pour l'ordinateur



D'autres problèmes sont triviaux ... pour l'humain

IA forte et IA faible

- L'IA considère que le mécanisme de la pensée peut être "copié"
- 4 façons d'aborder l'IA



IA forte et IA faible : exemple de l'IA d'un jeu d'échecs

IA Forte

- Compréhension de la façon de jouer des maîtres d'échecs
- Simulation de leurs comportements avec des règles
  - Occuper le centre
  - Occuper des cases de mêmes couleurs avec les pions
  - etc.

IA Faible

- Évaluer l'ensemble des coups possibles pour ce tour de jeu
- Évaluer l'ensemble des coups possibles de l'adversaire au tour d'après
- …
- Jouer le meilleur coup

Informatique classique et IA

Informatique classique	Vision IA
Exécute des instructions	Raisonne sur les connaissances
Adaptée au traitement numérique	Adaptée au traitement symbolique
Déterministe	Non déterministe, comprend choix et imprécision
N'est généralisable qu'à une classe de problèmes semblables	Généralisable à des domaines différents

"L'IA commence là où l'informatique classique s'arrête. Tout problème pour lequel il n'existe pas d'algorithme connu ou raisonnable permettant de le résoudre relève a priori de l'IA." Jean-Louis Laurière

Test de Turing

Comment savoir si un programme est intelligent ?

- Est-ce qu'il agit comme un humain ?

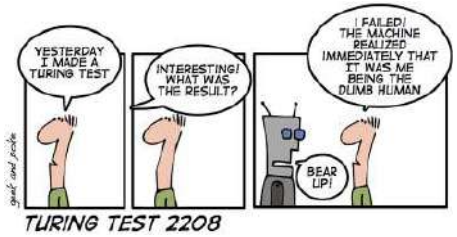
En 1950, Alan Turing décrit une méthode pour tester un système d'intelligence artificielle

Source : Wikimedia, domaine public

Test de Turing et Chatterbot

Test de Turing

- Un individu communique à l'aide d'un ordinateur avec un interlocuteur et doit décider si celui-ci est un humain ou un agent conversationnel (*chatterbot*)
- Turing avait prédit qu'avant l'an 2000, une machine aurait 30 % de chances de tromper une personne non avertie pendant 5 minutes



Source : Flickr (Geek And Poke – Turing Test), licence CC-BY-SA 2.0

Test de Turing et Chatterbot

Chatterbot

- Un agent conversationnel agit humainement et est plutôt une IA forte

The Loebner Prize: "The First Turing Test"

- Compétition du meilleur "système intelligent" / *chatterbot*
- https://aisb.org.uk

Chatterbots connus

- ELIZA : psychothérapeute virtuel, premier prix Loebner en 1966, inclus dans l'éditeur Emacs
- Rosette, CleverBot, **Chat-GPT**, etc.

Historique : la préhistoire

Les premières machines "intelligentes" sont des machines de calcul

- La Pascaline de Blaise Pascal (1645) permet de faire des additions et soustractions

Source : Wikimedia (David Monniaux), licence CC-BY-SA 2.0

- L'arithmomètre (1820) qui permet les 4 opérations tout le 19e siècle, etc.

Les premiers modèles de raisonnement

- La logique propositionnelle (1854), Georges Boole

Historique : années 45-55 – naissance de l'IA "moderne"

- Fort intérêt pour les mathématiques et les jeux
- **1943** McCulloch et Pitts : modèle de cerveau à partir de circuits booléens $\rightarrow$ réseaux de neurones
- **1947** Bardeen, Brattain et Schockley : transistor bipolaire
- **1948** C. Shannon : théorie de l'information
- **1950** A. Turing : théorie de la calculabilité
- **1954** A. Newel et H. Simon : IPL un langage de traitement symbolique

1955 Réunion de Dartmouth

- Invention du terme "intelligence artificielle" par John McCarty
  - "Every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it".

Historique : années 55-70 – l'âge d'or de l'IA

***Logic theorist* (55-56) A. Newel et H. Simon**

- 1er programme de démonstration de théorèmes utilisant IPL
- Démonstration de 38 des 52 premiers théorèmes du 2e chapitre de *Principia Mathematica*
- Recherche dans un arbre en utilisant des heuristiques

Deux voies principales

- Méthodes à bases de raisonnements logiques
  - *General Problem Solver* (1956), A. Newel et H. Simon, programme qui reformule un problème pour le résoudre → pense humainement
  - Checkers (1959), programme apprenant à devenir meilleur au jeu
  - LISP (1959), McCarthy, langage dédié à l'IA (éditeur Emacs !!!)
- Méthodes à base d'apprentissage : réseaux de neurones
  - Perceptron (1961), Rosenblatt

Historique : années 70 et 80

Années 70 : utilisation de connaissances

- Modélisation et utilisation de connaissances
- Développement de systèmes experts
  - MYCIN (1974) : diagnostic d'une infection bactérienne
  - PROSPECTOR (1978) : aide aux géologues pour l'évaluation du potentiel minéral d'un terrain
- Prolog/PROgrammation en LOGique (1972) : programmation de faits et règles, exécution en posant une question

Années 80 : industrialisation de l'IA

- Utilisation de systèmes experts pour faire des économies dans les grandes entreprises

Historique : années 90 – réalisme et modestie

Les systèmes experts déçoivent

- Résultats pas à la hauteur
- Certains problèmes sont intrinsèquement compliqués
- Manque de modélisation de la connaissance
- Manque de modélisation de l'incertain

De nouvelles pistes de recherche s'ouvrent

- Modélisation de l'incertain
- Agents autonomes
- Apprentissage
- Fouille de données
- etc.

IA forte ou IA faible ?

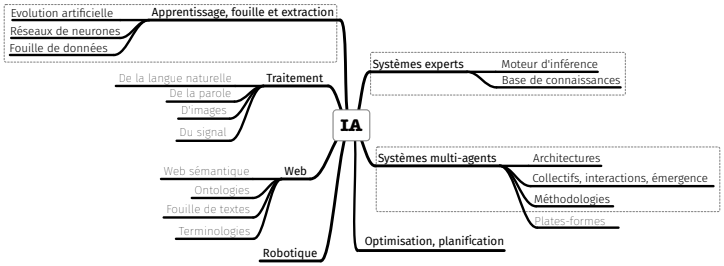
Aujourd'hui, la distinction IA faible/forte n'existe plus

- L'IA est un mélange des deux courants :
  - L'IA forte est complexe à mettre en oeuvre et ne donne pas les résultats attendus (quoi que Chat-GPT...)
  - L'IA faible est trop limitée par les machines (mémoire, temps)

L'IA est un mélange d'algorithmes génériques et spécialisés

- Générique :
  - Apprentissage
  - Fouille de données
  - Optimisation
  - Preuve de théorèmes
- Spécialisé
  - Reconnaissance de formes
  - Modélisation du langage

À quoi sert l'IA aujourd'hui ?



Systèmes multi-agents

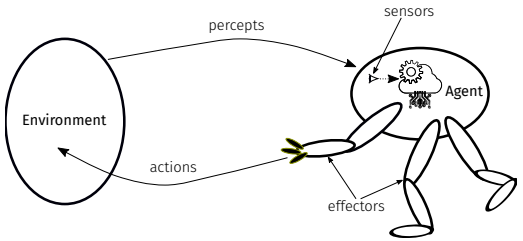
Systèmes multi-agents

Agents

Agent : définition

Agent

- Entité "*intelligente*" douée d'**autonomie**
- **Perçoit** et **agit** avec son environnement
- Poursuit un **objectif**



Agent

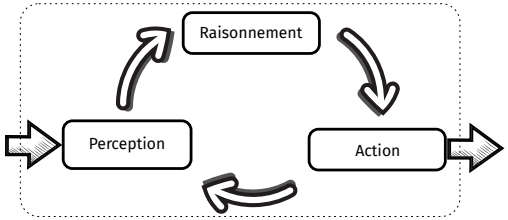
Un agent est-il un objet au sens informatique ?

- Points communs :
  - les agents ont un **état** / des **attributs**
  - les agents ont un **comportement** / **des méthodes**
- Différences :
  - L'objet répond lorsqu'on fait appel à une de ses méthodes
  - L'agent effectue des actions de lui-même
- On parle de POA : "Programmation Orientée Agents"

Agent

Un agent est une entité qui

- Est capable de **percevoir** (au moins partiellement) son environnement à l'aide de **capteurs** (*sensors*) et d'**agir** sur celui-ci à l'aide d'**effecteurs** (*effectors*)
- Se comporte de manière "intelligente" : ses actions sont le produit d'un **raisonnement** plus ou moins compliqué



Agent : exemples

Agent de diagnostic médical

- **Environnement** : patient, hôpital
- **Objectif** : proposer un traitement, minimiser le coût
- **Percepts** : recherche de symptômes, réponses des patients / **Capteurs** : entrée standard, micro, capteurs
- **Actions** : poser des questions, mesurer des paramètres / **Effecteurs** : sortie standard, haut-parleurs

Chat-GPT

- **Environnement** : ensemble des utilisateurs
- **Objectif** : apporter des réponses aux questions
- **Percepts** : suite de mots entrés / **Capteurs** : entrée standard
- **Actions** : afficher les réponses ou suggestions / **Effecteurs** : sortie standard

Agent : modèle

Choisir le **modèle** d'un agent consiste à définir :

- Son/ses **objectif(s)** (ce que l'agent cherche à faire)
- Sa/ses **perception(s)** (ce qu'il perçoit de l'environnement)
- Son/ses **action(s)** (capacité à interagir avec son environnement et/ou d'autres agents)
- Son intelligence
  - ce qui motive le déclenchement d'actions (**règles**, **état**)
  - sa façon de faire des choix
  - etc.

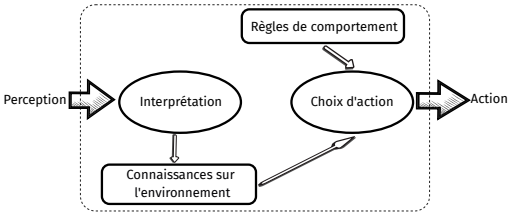
Agent : exemple du taxi autonome

- Ses **objectifs**
  - Destination, confort du passager, sécurité, etc.
- Ses **percepts**
  - Vision de la route, données GPS, compteur de vitesse, bruit du moteur, etc.
- Ses **actions**
  - Accélérer, freiner, tourner, klaxonner, changer de vitesse, etc.
- Son **intelligence**
  - Déterminer le plus court chemin
  - Respecter les limitations de vitesse
  - Respecter les distances de sécurité
  - Maximiser le bénéfice
  - etc.

Agent : types

Agent réactif (ou à "raisonnement réflexe")

- Capacités décrites par des paires de conditions-actions
- Raisonnement simple : **si condition alors action**
- S'inspire du monde animal : fourmis, abeilles, poissons, etc.
- L'intelligence provient du grand nombre d'agents mis en oeuvre



Agent : types

Agent réactif (ou à "raisonnement réflexe")

```java
public abstract class Agent<P, A> {
  private Set<Rule<A>> rules;

  public Agent(Set<Rule<A>> rules) {
    this.rules = rules;
  }

  public abstract Optional<A> act(P percept);

  // ...
}
```


Agent : types

Agent réactif (ou à "raisonnement réflexe")

```java
public Optional<A> act(P percept) {
  // Analyse du percept en entrée pour mettre à jour l'état
  State state = interpretInput(percept);

  // Recherche la règle correspondant à la situation
  Rule<A> rule = ruleMatch(state, rules);

  // Choisit l'action à réaliser en fonction de la règle
  action = (rule != null) ? rule.getAction() : null;

  // Retourne l'action à faire
  return Optional.ofNullable(action);
}
```

Agent : types

Agent réactif avec un but et un état interne

```
public abstract class GoalBasedAgent<P, A> {
    private State state;
    private Set<Rule<A>> rules;

  public GoalBasedAgent(Set<Rule<A>> rules) {
    this.rules = rules;
  }

  public abstract Optional<A> act(P percept);

  // ...
}
```


Agent : types

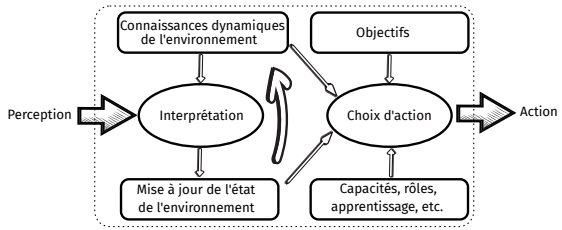
Agent réactif avec un but et un état interne

```
public Optional<A> act(P percept) {
  // Analyse du percept en entrée pour mettre à jour l'état
  state = updateState(state, percept);

  // Recherche la règle correspondant à la situation
  Rule<A> rule = ruleMatch(state, rules);

  // Choisit l'action à réaliser en fonction de la règle
  action = (rule != null) ? rule.getAction() : null;

  // Retourne l'action à faire
  return Optional.ofNullable(action);
}
```


Agent : types

Agent cognitif (ou "rationnel")

- Plus "intelligent" qu'un agent réactif, mais plus lent
- Modélisation des informations possibles selon l'approche *BDI* (*Belief*, *Desire*, *Intention*)



Cf. aima/core/logic/propositional/agent/KBAgent.java (agent basé connaissances dans Russel et Norwig, 2010)

Agent : approche BDI

- ***Belief* / croyances** : informations provenant de l'environnement
  - Peuvent s'avérer fausses
  - Si nécessairement vraies, on parle alors de **connaissances**
- ***Desire* / désirs** : objectifs à atteindre par l'agent
- ***Intentions*** : ensemble d'actions qui vont permettre de mener à bien les désirs

Exemple

Un agent désire aller boire un café. Il croit que le distributeur à café fonctionne, et il croit qu'il a suffisamment de monnaie. Il a alors l'intention de sortir de son bureau, l'intention d'aller au distributeur et l'intention de se servir un café.

Agent : différences entre agents réactif et cognitif

Propriété	Cognitif	Réactif
Type	Complexe / intelligent	Action / réaction
Représentation de l'environnement	Explicite	Aucune
Mémorisation	Oui (apprend)	Non (n'apprend pas)
Nombre d'agents en interaction	Petit	Grand

De nombreux cas intermédiaires sont possibles

- Par exemple, agent réactif à mémoire → conserve un historique de quelques états passés

Systèmes multi-agents

Systèmes multi-agents

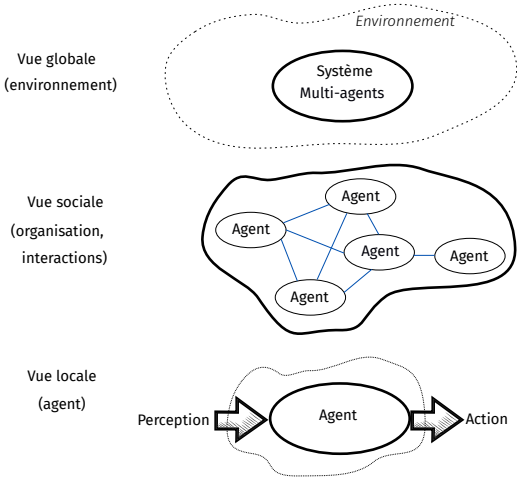
Systèmes multi-agents

Les systèmes multi-agents (SMA) peuvent être utilisés pour

- Simuler des phénomènes complexes (incendie, mouvements de foules, simulation d'un pont, etc.)
- Décomposer un problème complexe (recherche sur Internet, etc.)

Source : Wikimedia (Rose Thumboor), licence CC-BY-SA 4.0

Systèmes multi-agents : niveaux de détails



Systèmes multi-agents : organisation

- **Organisation** : ensemble des relations entre agents (hiérarchie, groupe, etc.)
- L'organisation dépend de nombreux facteurs à déterminer
  - **Homogène** ou **hétérogène**
  - **But unique** ou **buts multiples**
  - **Implicite** ou **explicite**
  - **Statique** ou **dynamique**
  - **Prédéfinie** ou **émergent**
  - **Centré agent** ou **organisation**
  - **Coopérativité** entre agents
  - etc.

Systèmes multi-agents : coopérativité

La **coopérativité** des agents peut prendre plusieurs formes :

- **Indépendance** : les agents ont chacun leurs buts/désirs et peuvent les atteindre individuellement
- **Encombrement** : les agents sont indépendants, mais les ressources sont limitées et empêchent une partie des agents d'atteindre leur but
- **Collaboration** : les agents ont des compétences insuffisantes et coopérent pour atteindre le but
- **Compétition** (collectif ou individuel) : des agents ont des objectifs incompatibles et se battent pour atteindre le(s) but(s)
- **Conflit pour des ressources** (collectif ou individuel) : des agents en compétition avec des ressources limitées se battent pour l'accès à ces ressources

Systèmes multi-agents : coopérativité

- Trois catégories : **indépendance**, **coopération** et **antagonisme**

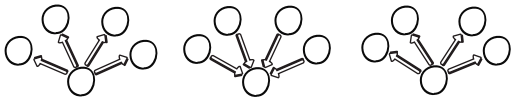
Buts	Ressources	Compétences	Situation
Compatibles	Suffisantes	Suffisantes	Indépendance
Compatibles	Insuffisantes	Suffisantes	Encombrement
Compatibles	Suffisantes	Insuffisantes	Collaboration simple
Compatibles	Insuffisantes	Insuffisantes	Collaboration coordonnée
Incompatibles	Suffisantes	Suffisantes	Compétition individuelle pure
Incompatibles	Suffisantes	Insuffisantes	Compétition collective pure
Incompatibles	Insuffisantes	Suffisantes	Conflits individuels pour des ressources
Incompatibles	Insuffisantes	Insuffisantes	Conflits collectifs pour des ressources

Systèmes multi-agents : interactions

- Les **interactions** concernent toute relation dynamique des agents entre eux
  - **infrastructure(s)**
  - **langages** et **protocoles** d'interactions entre agents
- Il y a interaction lorsque la dynamique d'un agent est modifiée par l'influence d'autres agents
- Plusieurs modèles d'interactions possibles liés à l'organisation choisie

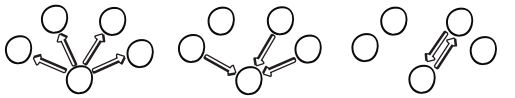
Systèmes multi-agents : exemples d'interactions

Vote / sondage



Le client diffuse le vote | Le coordinateur récupère les votes | Le coordinateur retourne les résultats

Appel d'offres



Le client diffuse l'appel d'offre | Le client reçoit les offres | Le client passe un contrat avec le plus offrant

Systèmes multi-agents : environnement

Environnement : "espace" commun aux agents du système

Propriétés de l'environnement

- **Accessible** ou **inaccessible** : si accessible, l'agent a accès à tout instant à l'environnement complet (pas de trace à conserver)
- **Déterministe** ou **non-déterministe** : un environnement est déterministe si son prochain état ne dépend que de son état courant et de l'action de l'agent
- **Statique** ou **dynamique** : un environnement qui ne change pas lorsque l'agent "réfléchit" est dit *statique*
- **Discret** ou **continu** : si le nombre de séquences perceptives et d'actions est limité alors l'environnement est *discret*

Systèmes multi-agents : environnement

Exemples

Système	Accessible	Déterministe	Statique	Discret
Diagnostic médical	Non	Non	Non	Non
Chat-GPT	Non	Non	Semi	Non
Taxi automatique	Non	Non	Non	Non
Analyseur d'images sat.	Oui	Oui	Semi	Non
Projet LV-223	Non	Non	Semi	Oui

Analyseur d'images satellite :

- Environnement : images satellitaires
- Objectif : classification des images
- Percepts : pixels d'intensité variable, couleurs / fichier
- Actions : afficher une classification de l'image / fichier

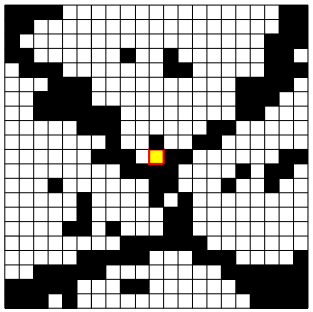
Systèmes multi-agents

À propos du projet

À propos du projet

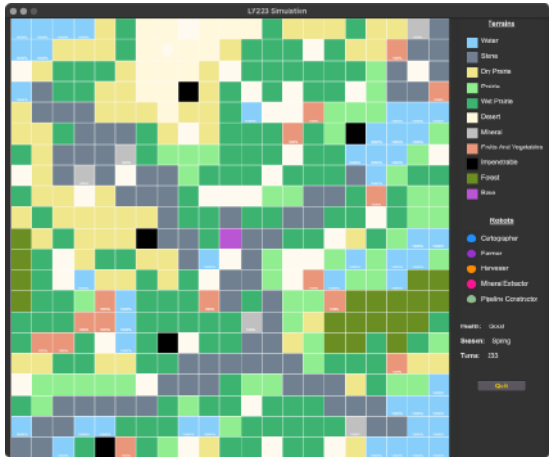
- LV-223 est un agent réactif à mémoire temporaire
- LV-223 correspond à l'environnement pour la colonie de robots
- La colonie de robots est un système multi-agents
- Le système multi-agents correspond à l'environnement de LV-223

Exosquelette de LV-223



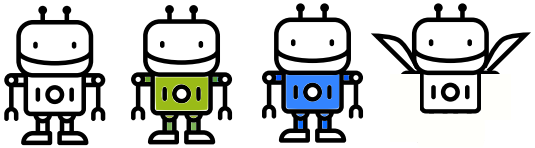
À propos du projet

Cartographie de LV-223



À propos du projet

Robots de la colonie

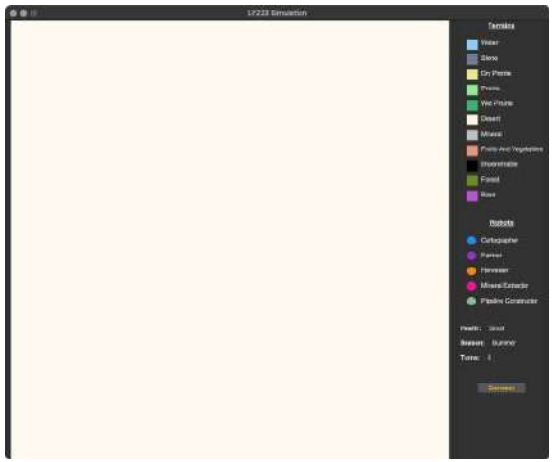


Centralisateur — 2 agriculteurs — 9 ouvriers — 2 cartographes

3 récolteurs nourriture
3 extracteurs minerai
3 constructeurs pipelines

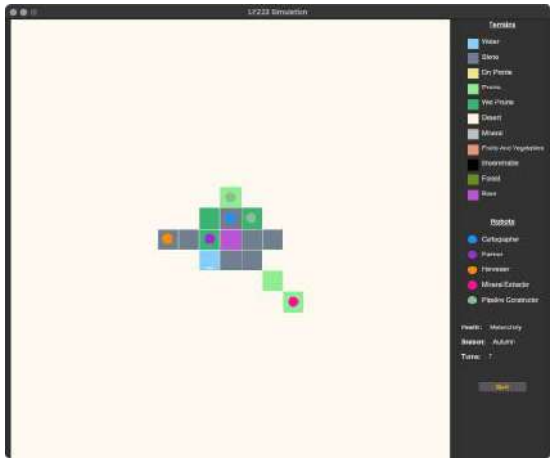
À propos du projet

Vue de LV-223 au début de la simulation



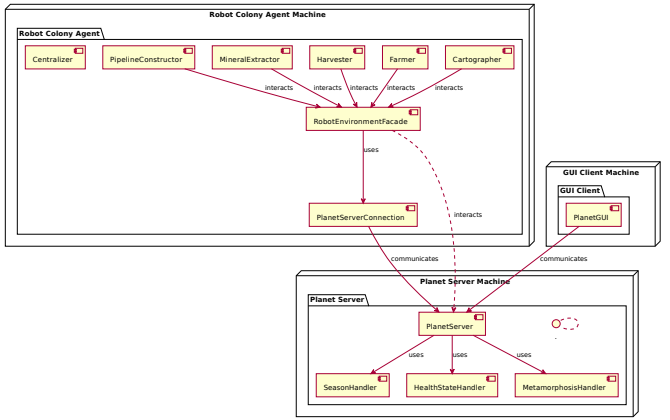
À propos du projet

Vue de LV-223 au bout de quelques tours



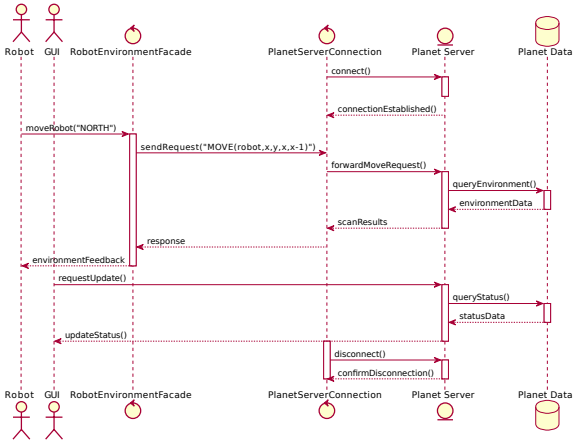
À propos du projet

Diagramme de déploiement



À propos du projet

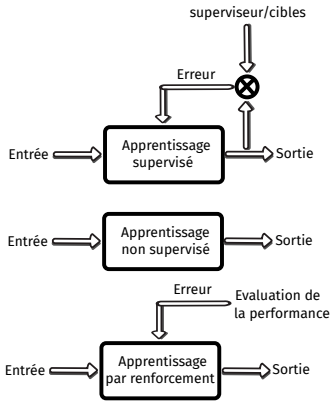
Exemple de séquence pour l'action "navigate(NORTH)"



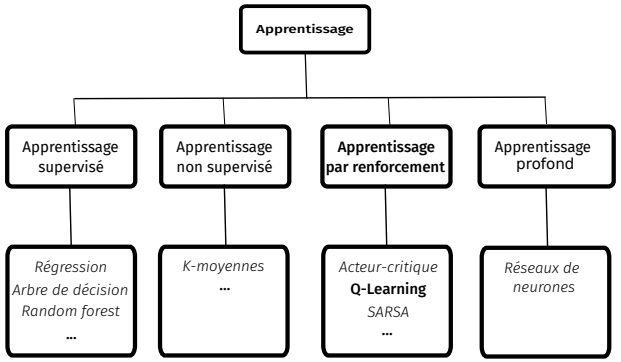
Systèmes multi-agents

Apprentissage

Classification



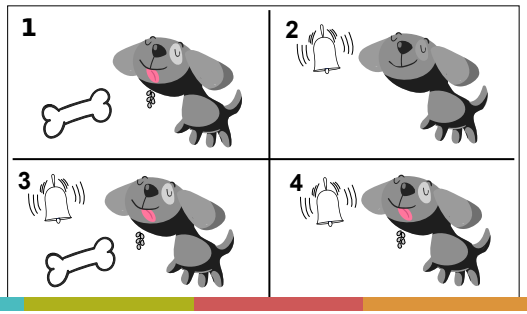
Techniques d'apprentissage



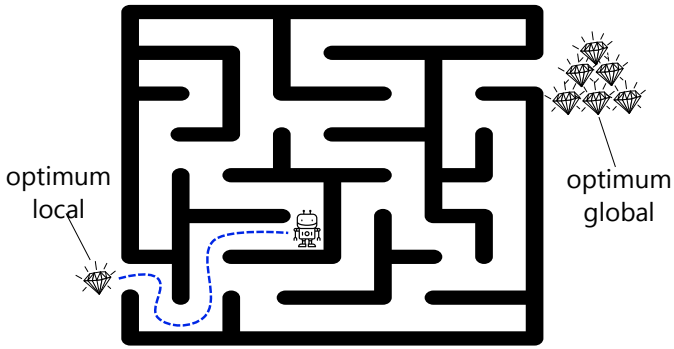
Apprentissage par renforcement

Apprendre à un agent comment se comporter dans un environnement

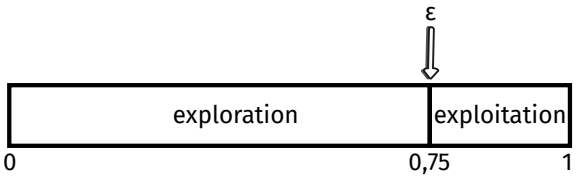
- Environnement réel ou simulé
- Environnement $\longrightarrow$ **récompenses**
- Environnement $\longrightarrow$ données d'entraînement
- **Objectif** : **maximiser** le nombre de récompenses fournies par l'environnement



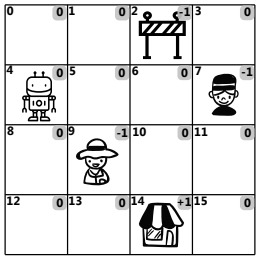
Apprentissage par renforcement : exploration / exploitation



Apprentissage par renforcement : ε-*greedy*



Apprentissage par renforcement : Q-*learning*



Récompenses de l'environnement

- État : **numéro de case**
- Action : **Nord**, **Sud**, **Est**, **Ouest**

Apprentissage par renforcement : Q-*learning*

Fonction d'utilité ou Q-*function* en anglais (Q pour *Quality*)

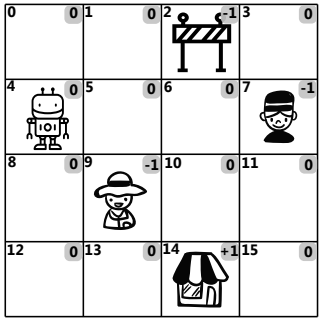
- $Q : E \times A \rightarrow \mathbb{R}$
  Mesure la "qualité" d'une combinaison état-action

Apprentissage par renforcement : Q-*learning*

$$Q(e_t, a_t)^\pi = \mathbb{E}(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | e_t, a_t)$$

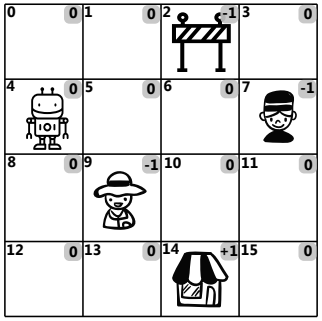
- e_t : état à l'instant t
- a_t : action à l'instant t
- R_i : récompense à l'instant i (*reward* en anglais)
- γ : facteur d'actualisation qui détermine l'importance des récompenses futures ($0 \leq \gamma \leq 1$) – par exemple $\gamma = 0,9$
  - $\gamma = 0 \rightarrow$ l'agent ne considère que les récompenses actuelles
  - $\gamma = 1 \rightarrow$ l'agent vise une récompense élevée à long terme
- π : politique de choix des suites d'actions associées à des états (*policy* en anglais) $\rightarrow$ exploration ou exploitation à l'aide de ϵ-greedy

Apprentissage par renforcement : Q-*learning*



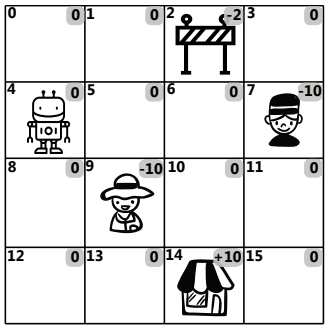
$$R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4}$$
$$= 0 + (-1) + 0 + (+1)$$
$$= 0$$

Apprentissage par renforcement : Q-*learning*



$$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4}$$
$$= 0 + 0,9 \times (-1) + 0,81 \times 0 + 0,729 \times (+1)$$
$$= -0,171$$

Apprentissage par renforcement : Q-*learning*



$$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4}$$
$$= 0 + 0,9 \times (-10) + 0,81 \times 0 + 0,729 \times (+10)$$
$$= -1,71$$

Apprentissage par renforcement : Q-*learning*

Mise à jour de la fonction d'utilité

$$Q(e_t, a_t)_{\text{nouveau}} =$$
$$Q(e_t, a_t)_{\text{ancien}} + \alpha[r_t + \gamma \max_{a_{t+1}} Q(e_{t+1}, a_{t+1}) - Q(e_t, a_t)_{\text{ancien}}]$$

- r_t : récompense reçue en passant de l'état e_t à e_{t+1}
- α : taux d'apprentissage qui détermine dans quelle mesure les informations nouvellement acquises remplaceront les anciennes informations ($0 \leq \alpha \leq 1$) – par exemple $\alpha = 0,1$
  - $\alpha = 0 \rightarrow$ l'agent n'apprend rien
  - $\alpha = 1 \rightarrow$ l'agent ne considère que les informations les plus récentes

Apprentissage par renforcement : Q-*learning*

Algorithme de la mise à jour de la fonction d'utilité

```
POUR chaque cycles d'apprentissage
  Sélectionner un état aléatoire

  TANTQUE état final non atteint FAIRE
    Sélectionner une des actions possibles pour l'état
        courant
    Utiliser cette action et considérer l'état suivant
    Récupérer la valeur maximale Q pour ce nouvel état en
        fonctions de toutes les actions possibles
    Mettre à jour Q : Q(état, action) = Q(état, action) +
        alpha * (R(état, action) + gamma * Max(état
        suivant, toutes les actions) - Q(état, action))
    L'état suivant devient l'état courant
  FIN TANTQUE
FIN POUR
```


Apprentissage par renforcement : Q-*learning*

- Implémentation par Q-table
- Dans notre cas, une table à 3 dimensions permettant de stocker les actions en fonction de l'état précédent et suivant

```
private final double[][][] qValues;

stateValues= new double[grid.width][grid.height];
for (double[] doubles: stateValues) Arrays.fill(doubles, 0);

qValues = new
    double[grid.width][grid.height][Action.NUM_ACTIONS];
for (double[][] doubles: qValues)
    for (double[] aDouble: doubles) Arrays.fill(aDouble, 0);
```

- Solution plus générique :

```
private Map<Pair<S, A>, Double> qValues = new HashMap<>();
```


Apprentissage par renforcement : Q-*learning*

```
État:    0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
 0 : [  -1   0  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1]
 1 : [   0  -1  -2  -1  -1   0  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1]
 2 : [  -1   0  -1   0  -1  -1   0  -1  -1  -1  -1  -1  -1  -1  -1  -1]
 3 : [  -1  -1  -2  -1  -1  -1  -1 -10  -1  -1  -1  -1  -1  -1  -1  -1]
 4 : [   0  -1  -1  -1  -1   0  -1  -1   0  -1  -1  -1  -1  -1  -1  -1]
 5 : [  -1   0  -1  -1  -1  -1   0  -1  -1 -10  -1  -1  -1  -1  -1  -1]
 6 : [  -1  -1  -2  -1  -1   0  -1 -10  -1  -1   0  -1  -1  -1  -1  -1]
 7 : [  -1  -1  -1   0  -1  -1   0  -1  -1  -1  -1   0  -1  -1  -1  -1]
 8 : [  -1  -1  -1  -1  -1  -1  -1  -1  -1 -10  -1  -1   0  -1  -1  -1]
 9 : [  -1  -1  -1  -1  -1   0  -1  -1   0  -1   0  -1  -1   0  -1  -1]
10 : [  -1  -1  -1  -1  -1  -1   0  -1  -1 -10  -1   0  -1  -1  10  -1]
11 : [  -1  -1  -1  -1  -1  -1  -1 -10  -1  -1   0  -1  -1  -1  -1   0]
12 : [  -1  -1  -1  -1  -1  -1  -1  -1   0  -1  -1  -1  -1   0  -1  -1]
13 : [  -1  -1  -1  -1  -1  -1  -1  -1  -1 -10  -1  -1   0  -1  10  -1]
14 : [  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1]
15 : [  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1   0  -1  -1  10  -1]
```


Systèmes multi-agents

Résumé

Agents et systèmes multi-agents : résumé

Agents

- Un agent perçoit et interagit avec son environnement de manière intelligente et autonome
- 2 types de base : réactif ou cognitif

Système multi-agents

- Un système multi-agents se décrit selon :
  - ses agents
  - l'organisation entre agents
  - les interactions entre agents
  - l'environnement dans lequel ils évoluent
- Les SMA servent à résoudre des problèmes complexes ou à simuler des phénomènes complexes

Bibliothèques

Quelques bibliothèques ou environnements multi-agents

- MadKit (Java) : http://www.madkit.net/madkit/
- Jade (Java) : https://jade.tilab.com
- Repast Symphony : https://repast.github.io

Autres liens

- Apprentissage par renforcement en Java
  - Deeplearning4j : https://github.com/deeplearning4j/deeplearning4j
  - java-reinforcement-learning : https://github.com/chen0040/java-reinforcement-learning

Logiques

Logiques

Introduction aux logiques

Introduction

- De nombreux problèmes peuvent être résolus à l'aide d'**algorithmes de recherche** en IA

Toutefois

- Un humain ne résoud pas tous les problèmes de cette manière
- Un humain fait aussi des déductions à partir de connaissances et selon un modèle **SI ... ALORS ...*** (ex. diagnostic)
  - SI j'observe une erreur "segmentation fault"
  - ALORS je dois vérifier mes allocations dynamiques

Introduction

2 types de connaissances :

procédurales

- **"savoir faire quelque chose"**
- Ex. : un algorithme s'intéresse à une succession de tâches à réaliser pour obtenir un résultat

déclaratives

- **"savoir que quelque chose est vrai ou faux"**
- Ex. : connaissance de faits et de relations entre faits

Logique $\leftrightarrow$ connaissances déclaratives

Introduction

La logique s'intéresse aux connaissances déclaratives

- Elle représente des connaissances à l'aide de **symboles** et formalise la notion d'**inférence**
- Elle permet **manipulation et raisonnement** de l'information sémantique contenue dans une phrase

La logique joue un rôle central car elle peut à la fois

- Décrire des faits et des lois $\rightarrow$ **connaissances déclaratives**
- Traiter ces faits pour extraire de la connaissance $\rightarrow$ **inférence**

Introduction

Un système utilisant des connaissances se basent sur

- Une **base de connaissances** + des **axiomes**
  - Ensemble de phrases décrivant des faits et des lois dans un langage formel
  - Spécifique à un domaine
- Un **moteur d'inférence**
  - Ensemble de méthodes permettent de produire de nouvelles connaissances à partir de connaissances existantes dans la base
  - Indépendant du domaine

Exemple

- **Base de connaissances**
  - "Socrate est un homme" ; "Tous les hommes sont mortels"
- **Résultat de l'inférence**
  - "Socrate est mortel"

Introduction

Différents types de logiques

- Logique classique
  - Logique **propositionnelle** : $A \Rightarrow B$ et $B \Rightarrow C$ alors $A \Rightarrow C$
  - Logique des **prédicats** $\forall x$, homme$(x) \Rightarrow$ mortel(x), homme(Socrate) $\Rightarrow$ mortel(Socrate)
- Logique de l'**incertain**
  - Floue $v(A) = 0.60$
  - Probabiliste
- Logique multivaluée
  - discrète $A \in \{\text{vrai}, \text{faux}, \text{inconnu}\}$
  - etc.

Logiques

Logique classique propositionnelle

Logique propositionnelle

C'est un **langage formel** qui permet d'exprimer des connaissances (logique d'ordre 0)

- Une **proposition** est un fait qui peut être soit **Vrai**, soit **Faux**
- Exemples de propositions :
  - 7 est un nombre premier (Vrai)
  - 2 + 2 = 3 (Faux)
  - Il pleut aujourd'hui (Vrai ou Faux)

Logique propositionnelle

- **Symboles de propositions** : P, Q, R, S, "il pleut demain", etc.
- **Constantes** : Vrai, Faux
- **Opérateurs** (ou *connecteurs*)
  - $\wedge$ (et)
  - $\vee$ (ou)
  - $\neg$ (non)
  - $\Rightarrow$ (implique)
  - $\Leftrightarrow$ (équivalent)

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
Faux	Faux	Vrai	Faux	Faux	Vrai	Vrai
Faux	Vrai	Vrai	Faux	Vrai	Vrai	Faux
Vrai	Faux	Faux	Faux	Vrai	Faux	Faux
Vrai	Vrai	Faux	Vrai	Vrai	Vrai	Vrai

($P \Rightarrow Q$ équivalent à $\neg P \vee Q$)

Logique propositionnelle : formule

Une formule peut être :

- **Simple** (ou atomique) : contient une seule proposition
  - P, "7 est un nombre premier", Vrai, etc.
- **Composée** : construite à partir de formules et d'opérateurs
  - $\neg P$, $(P \wedge Q)$, $\neg(P \vee Q) \Rightarrow R$, etc.

Exemple

- P = "Il fait beau aujourd'hui"
- Q = "Il fait plus froid qu'hier"
- R = "On va à la plage"

On peut représenter :

- "Il ne fait pas beau aujourd'hui et plus froid qu'hier" par $\neg P \wedge Q$
- "Nous n'irons à la plage que s'il fait beau" par $R \Rightarrow P$

Logique propositionnelle : lexique

- **Valuation** : valeur de vérité (Vrai ou Faux) associée à une proposition ou à une formule
- **Interprétation** : donne une valuation à chaque symbole d'une formule
- **Modèle** : interprétation qui rend une formule vraie
- Formule **consistante** : admet au moins un modèle
- Formule **valide** : formule toujours vraie (*tautologie*)

Formule consitante — Formule valide

	P	Q	P ∨ Q	(P ∨ Q) ∧ ¬Q	((P ∨ Q) ∧ ¬Q) ⇒ P
Une valuation	Vrai	Vrai	Vrai	Faux	Vrai
	Vrai	Faux	Vrai	Vrai	Vrai
Un modèle	Faux	Vrai	Vrai	Faux	Vrai
Une interprétation	Faux	Faux	Faux	Faux	Vrai

Logique propositionnelle : base de connaissances

- **Base de connaissances** BC : ensemble de formules qui admettent au moins un modèle commun

Une BC peut-elle produire une nouvelle proposition α ?

- On nomme α un théorème que l'on cherche à prouver
- Pour tous les modèles de BC, α est-il vrai également ?
- Notation : $BC \models \alpha$

Plusieurs méthodes de résolution

- Table de vérité
- Inférence
- Conversion en un problème de satisfaction de contrainte
- etc.

Logique propositionnelle : base de connaissances

Résolution avec une table de vérité

- BC : $P \vee Q$, $P \Leftrightarrow Q$
- Théorème α : $(P \vee \neg Q) \wedge Q$?

Atomes — Base de connaissances — Théorème

P	Q	P ∨ Q	P ⇔ Q	(P ∨ Q) ∧ Q
Vrai	Vrai	Vrai	Vrai	Vrai
Vrai	Faux	Vrai	Faux	Faux
Faux	Vrai	Vrai	Faux	Faux
Faux	Faux	Faux	Vrai	Faux

- Tous les modèles de BC sont vrais (un seul ici), et rendent également vrai le théorème
- Donc $BC \models \alpha$ (lire "de BC on déduit α")

Logique propositionnelle : base de connaissances

Résolution avec une table de vérité

- Long à mettre en oeuvre pour un grand nombre d'atomes (2^{atomes})
- Il est plus efficace d'utiliser des **équivalences** et des **règles d'inférence** pour prouver le théorème

Equivalence

- A est équivalente à B : $A \equiv B$
- Tout modèle de A est un modèle de B et tout modèle de B est un modèle de A

Inférence

- Si A et B sont vraies, alors C est vraie : $A, B \vDash C$
- Tout modèle de A et B est un modèle de C
- A et B sont appelées **prémisses**, et C est appelée **conclusion**

Logique propositionnelle : équivalences

- **Double négation** : $\neg\neg A \equiv A$
- **Contradiction** : $A \wedge \neg A \equiv$ Faux et $A \vee \neg A \equiv$ Vrai
- **Lois de De Morgan** : $\neg(A \vee B) \equiv \neg A \wedge \neg B$ et $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- **Constantes** : $A \vee$ Vrai $\equiv$ Vrai ; $A \wedge$ Vrai $\equiv A$; $A \Rightarrow$ Vrai $\equiv$ Vrai et Vrai $\Rightarrow A \equiv A$
- **Idempotence** : $A \vee A \equiv A$ et $A \wedge A \equiv A$
- **Tiers exclu** : $A \vee$ Faux $\equiv A$; $A \wedge$ Faux $\equiv$ Faux ; $A \Rightarrow$ Faux $\equiv \neg A$; Faux $\Rightarrow A \equiv$ Vrai
- **Implication et équivalence** : $A \Rightarrow B \equiv \neg A \vee B$; $A \Rightarrow$ Faux $\equiv \neg A$; $A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$ et $A \Leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$

Logique propositionnelle : règles d'inférence

- **Élimination et introduction de la conjonction** : $\frac{A \wedge B}{A}$ $\frac{A \wedge B}{B}$ $\frac{A, B}{A \wedge B}$
- **Introduction de la disjonction** : $\frac{A}{A \vee B}$ $\frac{B}{A \vee B}$
- **Résolution** : $\frac{A \vee B,\ \neg A}{B}$ $\frac{A \vee B,\ \neg B \vee C}{A \vee C}$
- **Élimination de l'implication** (*modus ponens*, *modus tolens*) : $\frac{A \Rightarrow B,\ A}{B}$ $\frac{A \Rightarrow B,\ \neg B}{\neg A}$
- **Introduction de l'implication** : $\frac{A,\ B}{A \Rightarrow B}$ $\frac{A,\ B}{B \Rightarrow A}$ $\frac{A}{B \Rightarrow A}$ $\frac{A \Rightarrow B,\ B \Rightarrow C}{A \Rightarrow C}$

Remarque : $\frac{X,\ Y}{X \Rightarrow Y}$ est équivalent à $X,\ Y \vDash X \Rightarrow Y$

Logique propositionnelle : preuve par inférence

Comment utiliser ces règles pour prouver un théorème ?

Chaînage avant (recherche dirigée par les faits)

- On choisit une règle dont les prémisses sont dans la base de connaissances
- On ajoute la conclusion à la base de connaissances
- On a prouvé le théorème quand celui-ci est dans la base de connaissances

Par l'absurde

- On suppose que $\neg\alpha$ est dans la base de connaissances
- On procède par chaînage avant jusqu'à prouver que le théorème est Faux

Logique propositionnelle : preuve par inférence

Chaînage arrière (recherche dirigée par les buts)

- On part d'un but α et on cherche une règle qui peut conclure à α
- On prouve alors les prémisses de cette règle récursivement
- On a prouvé le théorème quand les prémisses sont dans la base de connaissances

Suivant ce que l'on cherche à démontrer, l'une des trois méthodes sera plus efficace

Logique propositionnelle : preuve par inférence (exemple)

Exemple

- On dispose de quatre atomes possibles :
  - P : la pile est chargée
  - A : l'antenne est sortie
  - S : la station est réglée
  - R : la radio fonctionne
- Base de connaissances
  - 1 : $R \Leftrightarrow (P \wedge A \wedge S)$ – la radio marche SSI la pile, l'antenne et la station sont opérationnelles
  - 2 : P – la pile est chargée
  - 3 : $\neg R$ La radio ne marche pas
- Vérifier que la station n'est pas réglée ou que l'antenne n'est pas sortie
  - Théorème à prouver : $\neg A \vee \neg S$

Logique propositionnelle : preuve par inférence (exemple)

Résolution par chaînage avant

- Base de connaissances
  - 1 : $R \Leftrightarrow (P \wedge A \wedge S)$
  - 2 : P
  - 3 : $\neg R$
- Inférence
  - 4 : $1 \equiv (R \Rightarrow (P \wedge A \wedge S)) \wedge ((P \wedge A \wedge S) \Rightarrow R)$ – équivalence de l'équivalence
  - 5 : $4 \vDash (P \wedge A \wedge S) \Rightarrow R$ – élimination de la conjonction
  - 6 : $5,\ 3 \vDash \neg(P \wedge A \wedge S)$ – élimination de l'implication (*modus tolens*)
  - 7 : $6 \equiv \neg P \vee \neg A \vee \neg S$ – De Morgan
  - 8 : $7,\ 2 \vDash \neg A \vee \neg S$ – résolution $\rightarrow$ preuve

Logique propositionnelle : preuve par inférence (exemple)

Résolution par l'absurde

- Base de connaissances
  - 1 : $R \Leftrightarrow (P \wedge A \wedge S)$
  - 2 : P
  - 3 : $\neg R$
  - $\neg\alpha$: $\neg(\neg A \vee \neg S)$
- Inférence
  - 4 : $1 \equiv (R \Rightarrow (P \wedge A \wedge S)) \wedge ((P \wedge A \wedge S) \Rightarrow R)$ – équivalence de l'équivalence
  - 5 : $4 \vDash (P \wedge A \wedge S) \Rightarrow R$ – élimination de la conjonction
  - 6 : $5,\ 3 \vDash \neg(P \wedge A \wedge S)$ – élimination de l'implication (*modus tolens*)
  - 7 : $\neg\alpha \equiv \neg\neg A \wedge \neg\neg S$ – De Morgan
  - 8 : $7 \equiv A \wedge S$ – double négation
  - 9 : $2,\ 8 \vDash P \wedge A \wedge S$ – introduction de la conjonction
  - 10 : $6,\ 9 \vDash (P \wedge A \wedge S) \wedge \neg(P \wedge A \wedge S)$ – introduction de la

Logique propositionnelle : preuve par inférence (exemple)

Résolution par table de vérité

P	A	S	R	$R \Leftrightarrow (P \wedge A \wedge S)$	P	$\neg R$	$\neg A \vee \neg S$
F	F	F	F	V	F	V	V
F	F	F	V	F	F	F	V
F	F	V	F	V	F	V	V
F	F	V	V	F	F	F	V
F	V	F	F	V	F	V	V
F	V	F	V	F	F	F	V
F	V	V	F	V	F	V	F
F	V	V	V	F	F	F	F
V	**F**	**F**	**F**	**V**	**V**	**V**	**V**
V	F	F	V	F	V	F	V
V	**F**	**V**	**F**	**V**	**V**	**V**	**V**
V	F	V	V	F	V	F	V
V	**V**	**F**	**F**	**V**	**V**	**V**	**V**
V	V	F	V	F	V	F	V
V	V	V	F	F	V	V	F

Logiques

Logique classique des prédicats

Logique des prédicats : introduction

Les phrases suivantes ne peuvent être représentées en logique propositionnelle :

- “Marcus était pompéien”
- “Tous les pompéiens étaient Romains”
- “Tous les Romains étaient soit loyaux à César, soit ils le haïssaient”
- “Les gens tentent d’assassiner les souverains envers lesquels ils ne sont pas loyaux”
- “Marcus a tenté d’assassiner César”

Pourtant, il est possible de déduire de nouveaux faits à partir de ces phrases, par exemple : “Marcus haïssait César”

Il est impossible de représenter

- Des faits concernant des groupes d’objets
- Des relations entre des groupes d’objets

Logique des prédicats : introduction

La logique des prédicats est une **logique d’ordre 1**, plus expressive que la logique propositionnelle

- Un **Prédicat** est une “fonction” appliquée à un ou plusieurs termes (généralisation de la notion de proposition)
  - exemple : homme(x) signifie x est un homme
- La logique est munie des quantificateurs : $\exists$ (existence), $\forall$ (universalité)

Exemple

- “Tout homme est mortel. Socrate est un homme, donc Socrate est mortel”
- Traduction sous la forme de prédicats
  - $\forall x$ homme$(x) \Rightarrow$ mortel(x)
  - homme(Socrate)
  - donc mortel(Socrate)

Logique des prédicats

Que peut être un prédicat ?

- Un **atome** : une constante du domaine – exemples : Socrate, 3, Faux
- Une **fonction** avec *n* paramètre(s) – exemples : jeune(Agnès), frère(Joseph, René), parent(x, y)
- Une **variable** : emplacement d'une constante dans une fonction

Opérateurs

- $\wedge$ (et), $\vee$ (ou), $\neg$ (non), $\Rightarrow$ (implique), $\Leftrightarrow$ (équivalent)
- $=$ (égalité), $\forall$ (quel que soit), $\exists$ (il existe)

Formules

- atomique (sans opérateurs) – exemple : frère(Jean, Richard)
- complexes – exemple : frère(Jean, Richard) $\Rightarrow$ frère(Richard, Jean)

Logique des prédicats

Quantification existentielle : $\exists$

- Exemple
  - "Il y a quelqu'un d'intelligent à l'ENSICAEN"
  - $\exists x$ estA$(x,$ ENSICAEN$) \wedge$ intelligent(x)
- $\exists x\ P$ est équivalent à une disjonction d'instanciation de P
  estA(Pierre, ENSICAEN) $\wedge$ intelligent(Pierre) $\vee$
  estA(Paul, ENSICAEN) $\wedge$ intelligent(Paul) $\vee$
  estA(Jacques, ENSICAEN) $\wedge$ intelligent(Jacques) $\cdots$
- $\exists$ s'utilise essentiellement avec $\wedge$

Logique des prédicats

Quantification universelle : $\forall$

- Exemple
  - "Toute personne à l'ENSICAEN est intelligente"
  - $\forall x$ estA$(x,$ ENSICAEN$) \Rightarrow$ intelligent(x)
- $\forall x\ P$, est équivalent à une conjonction d'instanciation de P
  estA(Pierre, ENSICAEN) $\Rightarrow$
  intelligent(Pierre) $\wedge$ estA(Paul, ENSICAEN) $\Rightarrow$ intelligent(Paul) $\wedge$
  estA(Jacques, ENSICAEN) $\Rightarrow$ intelligent(Jacques) $\cdots$
- $\forall$ s'utilise essentiellement avec $\Rightarrow$

Logique des prédicats

Il peut être nécessaire d'utiliser plusieurs quantificateurs pour exprimer une formule

- aime(x, y) : x aime y
- $\forall x\ \exists y$ aime(x, y) : tout le monde aime quelqu'un

Ordre important lorsque les quantificateurs sont de types différents :

- $\exists y\ \forall x$ aime(x, y) : il y a quelqu'un que tout le monde aime

On peut passer de l'un à l'autre (double négation) :

- $\exists x$ aime$(x,$ glaces$)$: quelqu'un aime les glaces
- $\neg\ \forall x\ \neg$aime$(x,$ glaces$)$: pas tout le monde n'aime pas les glaces

Logique des prédicats

BC permet-elle de générer le théorème α ?

- Impossibilité d'utiliser une table de vérité
- $\Rightarrow$ **inférence obligatoire**

Inférence en logique des prédicats

- Similaire à la logique des propositions
- Possibilité de substituer les variables dans les prédicats

Exemple

- Base de connaissances :
  - 1 : $\forall x\ \forall y$ parent$(x, y) \Rightarrow$ plusAgée(x, y)
  - 2 : $\forall x\ \forall y$ mère$(x, y) \Rightarrow$ parent(x, y)
  - 3 : mère(Jeanne, Henri) (Jeanne d'Albret et Henri IV)
- Théorème ?
  - α : plusAgée(Jeanne, Henri)

Logique des prédicats : preuves

Inférence en chaînage avant

- 4 : $2 \vDash$ mère(Jeanne, Henri) $\Rightarrow$ parent(Jeanne, Henri)
  - Substitutions x/Jeanne y/Henri
- 5 : $3, 4 \vDash$ parent(Jeanne, Henri)
  - Élimination de l'implication (*modus ponens*)
- 6 : $1 \vDash$ parent(Jeanne, Henri) $\Rightarrow$ plusAgée(Jeanne, Henri)
  - Substitutions x/Jeanne y/Henri
- 7 : $6, 5 \vDash$ plusAgée(Jeanne, Henri)
  - Elimination de l'implication (*modus ponens*) $\rightarrow$ preuve

Logique des prédicats : preuves

Inférence en chaînage arrière

- 4 : $1 \vDash$ parent(Jeanne, Henri) $\Rightarrow$ plusAgée(Jeanne, Henri)
  - Substitutions x/Jeanne y/Henri
  - On cherche maintenant à prouver $\alpha' =$ parent(Jeanne, Henri)
- 5 : $2 \vDash$ mère(Jeanne, Henri) $\Rightarrow$ parent(Jeanne, Henri)
  - Substitutions x/Jeanne y/Henri
  - On cherche maintenant à prouver $\alpha'' =$ mère(Jeanne, Henri)
  - mère(Jeanne, Henri) est dans la base de connaissances $\rightarrow$ preuve

Logique des prédicats : résumé

Avantages

- Support du raisonnement en vue de sa mécanisation
- Possibilité d'exprimer que des propriétés sont vérifiées pour un ou tous les objets sans les nommer $\rightarrow$ preuve automatique de théorème

Limitations

- Représentation des connaissances de nature incertaine, imprécises et sujettes à révision $\rightarrow$ logique de l'incertain
- Impossible de qualifier les prédicats (tous les prédicats ont un seul argument) $\rightarrow$ logique d'ordre supérieur

Exemple simple d'un moteur d'inférence

```
public class InferenceEngine {
  private Map<String, Boolean> knowledgeBase;

  public InferenceEngine() {
    this.knowledgeBase = new HashMap<>();
  }

  public void addToKnowledgeBase(String statement, boolean
      value) {
    this.knowledgeBase.put(statement, value);
  }

  public boolean evaluate(String statement) {
    return this.knowledgeBase.get(statement);
  }

  public static void main(String[] args) {
    InferenceEngine engine = new InferenceEngine();
    engine.addToKnowledgeBase("rain", true);
    engine.addToKnowledgeBase("sun", false);
    engine.addToKnowledgeBase("clouds", true);
```


Logique des prédicats : exercice

Base de connaissances

- Faits
  - père(Henri, Louis)
  - père(Henri, Élisabeth)
  - mère(Marie, Louis)
  - homme(Louis)
  - femme(Élisabeth)
  - aime(Henri, Marie)
- Règles
  - $\forall X, \forall Y$ père$(X, Y) \Rightarrow$ parent(X, Y)
  - $\forall X, \forall Y$ mère$(X, Y) \Rightarrow$ parent(X, Y)
  - $\forall X, \forall Y$ aime$(X, Y) \Rightarrow$ aime(Y, X)
  - $\forall X, \forall Y, \forall Z$ parent$(Z, X) \wedge$ parent$(Z, Y) \wedge$ homme$(X) \Rightarrow$ frère(X, Y)
  - $\forall X \neg$homme$(X) \Rightarrow$ femme(X)
  - $\forall X, \forall Y$ mère$(X, Y) \Rightarrow$ femme(X)

Logique des prédicats : exercice

Questions

1. Définir grandPère(X, Y)
2. Définir tante(X, Y)
3. Montrer frère(Louis, Élisabeth) par chaînage arrière
4. Peut-on déduire femme(Marie) ?
5. Peut-on déduire homme(Henri) ?
6. Faire un chaînage avant

Logiques

Logique de l'incertain

Logique de l'incertain

En général une connaissance est souvent imprécise, voire incertaine

D'où vient cette incertitude ?

- Phénomène physique ou non (capteur, mesure, fait inexact ou imparfait)
- Connaissances exprimées par une sémantique (non quantifiable)

Statut de la connaissance : Jean est grand

- précise : hauteur(Jean) $= 1,83m$
- imprécise (intervalle) : hauteur(Jean) $\in [1,80; 1,90]$
- vague (ensemble flou) : valeur de vérité de 0,85

Théories du raisonnement avec incertitude

- les probabilités classiques
- les probabilités bayésiennes

Logiques

Logique de l'incertain : logique floue

Pourquoi la logique floue ?

Nous raisonnons avec des concepts linguistiques "flous" ou approximatifs

Concepts imprécis

- Âge, poids, température, ...

Dépendances imprécises

- Si la température est basse et que l'électricité n'est pas trop chère alors je monte suffisamment le chauffage

Pourquoi la logique floue ?

Exemple de règles floues

- Règles de conduite automobile à l'approche d'un carrefour contrôlé par des feux tricolores

Prémisse 1	Prémisse 2	Prémisse 3	Conclusion
le feu est rouge	ma vitesse est faible	le feu est proche	je freine fort
le feu est rouge	ma vitesse est élevée	le feu est loin	je freine doucement
le feu est orange	ma vitesse est moyenne	le feu est loin	je maintiens ma vitesse
le feu est vert	ma vitesse est faible	le feu est proche	j'accélère

A comparer avec : "Si le feu est rouge, si ma vitesse dépasse 55,6 km/h et si le feu est à moins de 12,3m, alors j'appuie sur le frein avec une force de 33,2 Newtons"

Historique

- **1965** : concept d'**ensemble flou** introduit par Zadeh (Berkeley)
- **1970** : premières applications (systèmes experts - aide à la décision en médecine, etc.)
- **1974** : première application industrielle (régulation floue d'une chaudière à vapeur réalisée par Mamdani)
- **1985** : produits japonais de type "Fuzzy Logic Inside" (machines à laver, appareils photos, etc.)
- **1987** : premier train piloté avec la logique floue (Sendaï au Japon)
- **depuis 1990** : généralisation
  - appareils électroménagers (machines à laver, etc.)
  - systèmes audiovisuels (appareils photo, etc.)
  - systèmes automobiles embarqués (ABS, climatisation, etc.)
  - systèmes autonomes mobiles
  - systèmes de décision, diagnostic, reconnaissance
  - systèmes de contrôle/commande

Représentation en logique classique

En logique classique, il n'y a que deux états possibles

- **nulle** : vitesse = 0
- **non nulle** : vitesse = 1

```
{java} boolean vitesse; lire(vitesse); if (vitesse == 0) { /* vitesse nulle */
} else { /* vitesse non nulle */ }
```


Représentation en logique floue

En logique floue, tout problème est représenté par des ensembles flous

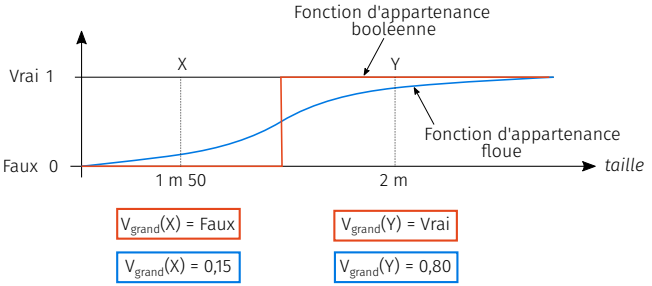
- **très lent** [0, 0.25]
- **lent** [0.25, 0.5]
- **rapide** [0.5, 0.75]
- **très rapide** [0.75, 1]

```
{java} float vitesse; lire(vitesse); if ((vitesse >= 0) && (vitesse < 0.25)) {
/* vitesse : très lent */ } else if ((vitesse >= 0.25) && (vitesse < 0.5)) { /*
vitesse : lent */ } else if ((vitesse >= 0.5) && (vitesse < 0.75)) { /* vitesse
: rapide */ } else /* vitesse >= 0.75 && vitesse <= 1.0 */ { /* vitesse : très
rapide */ }
```


Comparaison logiques classique et floue

La logique floue remplace l'opérateur booléen {Faux, Vrai} par l'intervalle de vérité [0, 1] pour exprimer l'appartenance

- "*grand*" (variable linguistique) dépend de "*taille*" (variable réelle)



Comparaison logiques classique et floue

Schéma général de la logique classique

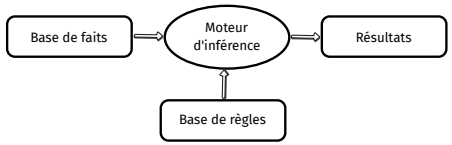
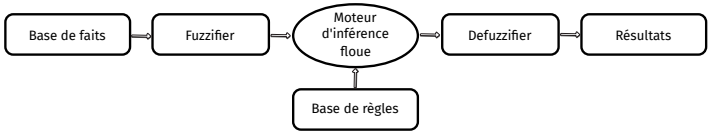
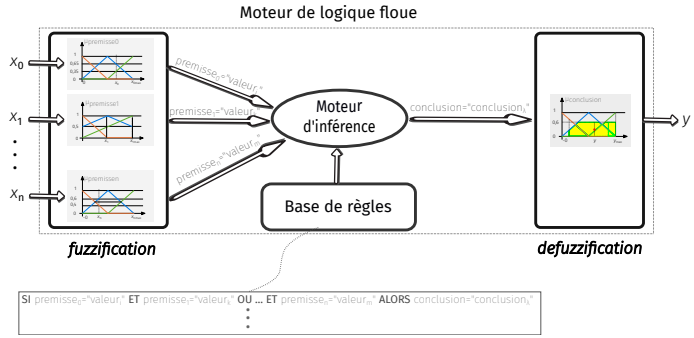


Schéma général de la logique floue



Un peu plus précisément

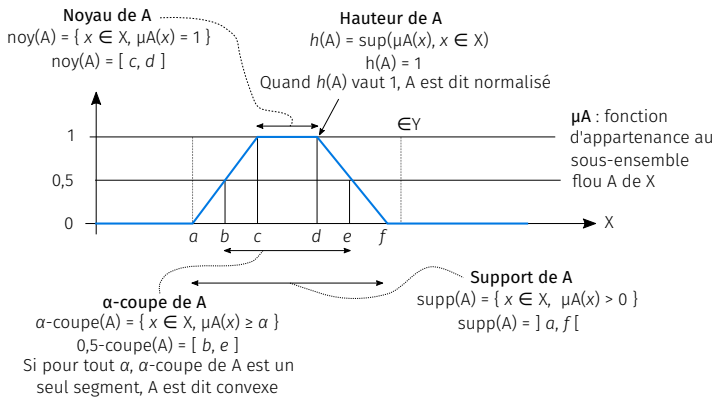


Fuzzification, inférence et défuzzification

Il faut choisir les méthodes de **fuzzification**, d'**inférence** et de **défuzzification**

- En logique floue : **infinité** de définitions des opérateurs ET, OU, NON et "SI ... ALORS ..."
- Équivalences** et règles d'inférences sont conservées exceptée la contradiction et le tiers exclu

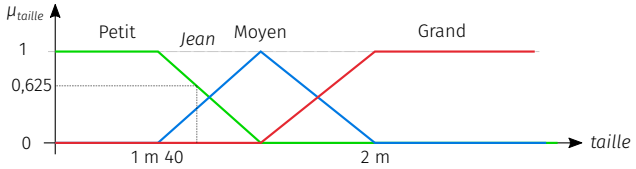
Vocabulaire



Fuzzification

- La **fuzzification** permet d'exprimer à quel point une règle doit être appliquée en fonction des valeurs numériques mesurées

La fuzzification est gérée par les fonctions d'appartenance



- Soit une règle "SI Jean est Petit ET ... ALORS ..."
- Supposons un capteur retournant $1m50$ pour la taille de Jean, alors Jean est considéré comme "Petit" à $62,5\%$ et "Moyen" à $37,5\%$ pour la fonction d'appartenance ci-dessus

Opérateurs $\wedge$, $\vee$, $\neg$ (ET, OU, NON)

Opérateurs de Zadeh

- $\mu A \wedge B(x,y) = \min(\mu A(x), \mu B(y))$
- $\mu A \vee B(x,y) = \max(\mu A(x), \mu B(y))$
- $\mu \neg A(x) = 1 - \mu A(x)$

Opérateurs de Larsen

- $\mu A \wedge B(x,y) = \mu A(x)\mu B(y)$
- $\mu A \vee B(x,y) = \mu A(x) + \mu B(y) - \mu A(x)\mu B(y)$
- $\mu \neg A(x) = 1 - \mu A(x)$

Opérateur $\Rightarrow$ (ALORS)

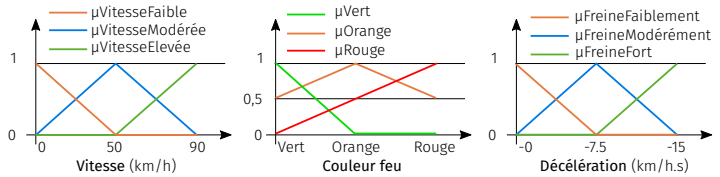
Réalisation avec un ET flou

- $\mu A \Rightarrow B(x,y) = \min(\mu A(x), \mu B(y))$ (Mamdani)
- $\mu A \Rightarrow B(x,y) = \mu A(x)\mu B(y)$ (Larsen)
- etc.

Réalisation par une implication floue

- $\mu A \Rightarrow B(x,y) = \max(1 - \mu A(x), \mu B(y))$ (Zadeh and Lee)
- $\mu A \Rightarrow B(x,y) = \min(1, 1 - \mu A(x) + \mu B(y))$ (Lukasiewicz)
- $\mu A \Rightarrow B(x,y) = \min(1, \mu B(y)/\mu A(x))$ (Goguen)
- etc.

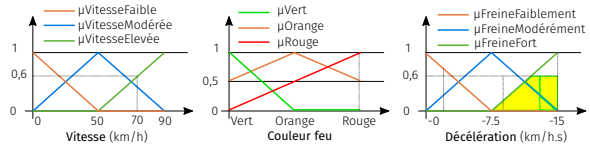
Exemple



Prémisse 1	Prémisse 2	Conclusion
le feu est rouge	ma vitesse est faible	je freine doucement
le feu est rouge	ma vitesse est élevée	je freine fort
le feu est rouge	ma vitesse est moyenne	je freine modéremment
le feu est orange	ma vitesse est moyenne	je freine doucement
...	...	...

Exemple

- Soit la règle : “SI le feu est rouge ET ma vitesse est élevée ALORS je freine fort”
- Cette règle s’applique-t-elle pour une vitesse de 70 km/h ?



- La vitesse est élevée (60 %) et le feu est rouge (100 %)
- La règle entière est vraie à 60 % (opérateur ET de Zadeh)
- L’implication de Mamdani $\rightarrow$ zone jaune pour cette règle
- On itère sur toutes les règles qui concernent les valeurs “70 km/h” et “Rouge” $\rightarrow$ ensemble flou
- on somme les aires (union)

Défuzzification

Pour de la commande $\rightarrow$ renvoie une valeur

- **Centre de gravité** : décision $= \frac{\int_S x\mu(x)dx}{\int_S \mu(x)dx}$ avec $S = \{x \in X\}$
- **Moyenne des *maxima*** : décision $= \frac{\int_S xdx}{\int_S dx}$ avec $S = \{x_m \in X, \mu(x_m) = sup(\mu(x), x \in X)\}$

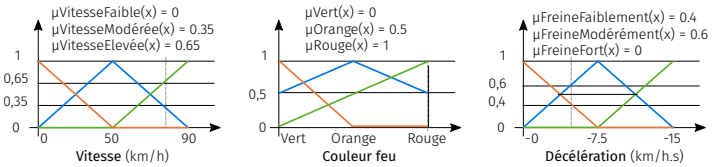
Pour détecter des cas anormaux $\rightarrow$ renvoie vrai ou faux

- On fixe un seuil
- Puis on retourne une alerte si $\mu(x)$ est au-dessus/dessous

Exemple de détection d’anomalies (freinage anormal)

Règles floues

- Si la vitesse est modérée et le feu est orange
  - Alors, il faut freiner modérément
- Si la vitesse est élevée et le feu est rouge
  - Alors, il faut freiner fort



Exemple de détection d’anomalies (freinage anormal)

On utilisera les opérateurs de Zadeh et l’inférence de Lee

- $\mu A \wedge B(x, y) = \min(\mu A(x), \mu B(y))$
- $\mu A \Rightarrow B(x, y) = \max(1 - \mu A(x), \mu B(y))$

On placera le seuil de détection à 0,5

- Il faut lever une alerte si la valeur d’une règle est inférieure au seuil

Exemple de détection d'anomalies (freinage anormal)

- Si la vitesse est modérée et le feu est orange
  - Alors, il faut freiner modérément
  - μVitesseModérée $\wedge$ Orange $\Rightarrow$ FreineModérément
  - $= \max(1 - \mu\text{VitesseModérée} \wedge \text{Orange}, \mu\text{FreineModérément})$
  - $= \max(1 - \min(\mu\text{VitesseModérée}, \mu\text{Orange}), \mu\text{FreineModérément})$
  - $= \max(1 - \min(0.35, 0.5), 0.6) = 0.65$
  - La règle est respectée $\Rightarrow$ Pas d'alerte conducteur
- Si la vitesse est élevée et le feu est rouge
  - Alors, il faut freiner fort
  - μVitesseElevée $\wedge$ Rouge $\Rightarrow$ FreineFort
  - $= \max(1 - \mu\text{VitesseElevée} \wedge \text{Rouge}, \mu\text{FreineFort})$
  - $= \max(1 - \min(\mu\text{VitesseElevée}, \mu\text{Rouge}), \mu\text{FreineFort})$
  - $= \max(1 - \min(0.65, 1), 0) = 0.35$
  - La règle n'est pas respectée $\Rightarrow$ Alerte conducteur

Bibliothèques

Quelques bibliothèques :

- fuzzylite (Java, C++) : https://fuzzylite.com
- jfuzzylogic (Java) : http://jfuzzylogic.sourceforge.net
- eFLL (C/C++) : https://github.com/zerokol/eFLL

Recherches

Recherches

Formulation du problème

Certains problèmes sont faciles à résoudre

Trouver la(les) valeur(s) de x pour

$$ax^2 + bx + c = 0$$

Solution

$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, si $b^2 - 4ac > 0$

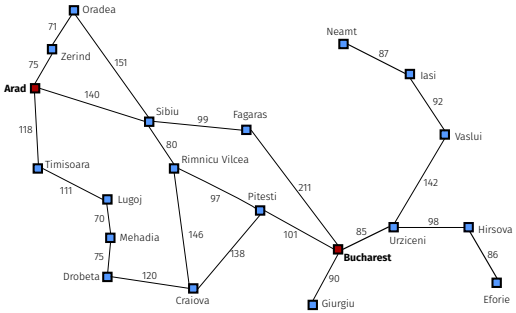
Ces problèmes ont une solution directe

- Il suffit d'appliquer une formule connue pour les résoudre
- Aucun signe d'intelligence dans l'application de la solution
  Tout problème pour lequel il n'existe pas d'algorithme connu ou raisonnable permettant de le résoudre relève a priori de l'IA

Certains problèmes nécessitent d'effectuer une recherche

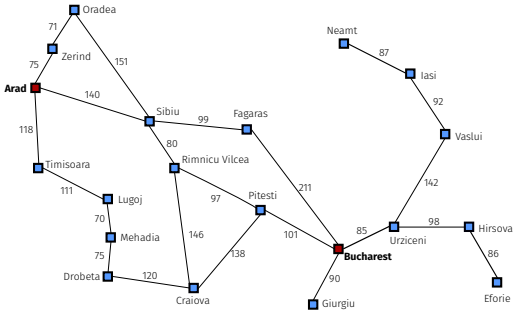
- Plusieurs possibilités sont à explorer avant de trouver la solution
- Le nombre de possibilités peut être très grand, voire infini

Exemple : trouver une route d'Arad à Bucarest en Roumanie



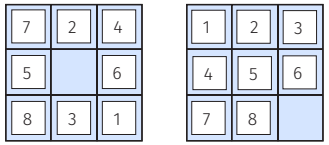
Certains problèmes nécessitent d'effectuer une recherche

Autre exemple : trouver la route la plus courte d'Arad à Bucharest



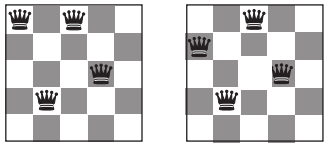
Certains problèmes nécessitent d'effectuer une recherche

Trouver la suite de mouvements pour aller à la solution



Départ Solution

Trouver un état satisfaisant



Non satisfaisant Satisfaisant

Espace de recherche

Un problème de recherche est défini par :

- Un **espace de recherche** : ensemble des états parmi lesquels on recherche une solution
  - Ex. : ensemble des routes entre deux villes, ensemble des villes, ensemble des successions de déplacements valides, ensemble de configurations de reines, etc.
- Un **état initial** : état de départ de la recherche
  - Ex. : la ville d'Arad
- Une **condition objectif** : "Quelle est la condition d'arrêt de la recherche ?"
  - Ex. : une route d'Arad à Bucarest, la route la plus courte d'Arad à Bucarest, le taquin terminé, une configuration des reines ne s'attaquant pas, etc.

Étapes d'une recherche

- La recherche parcourt l'espace de recherche jusqu'à trouver un état remplissant la condition objectif
- L'efficacité de la recherche dépend de :
  - Le type d'espace de recherche
  - La méthode de parcours
  - La condition d'arrêt

Important

- On peut influencer l'efficacité de la recherche en définissant correctement le problème de recherche, la méthode de parcours et la condition d'arrêt
- D'où l'importance de la **formulation** de la recherche

Recherche et graphes

- De nombreux problèmes peuvent être assimilés à la **recherche dans un graphe**
  - Espace de recherche : ensemble des noeuds du graphe
  - Arêtes entre les noeuds : déplacement autorisé d'un noeud à un autre
  - État initial : point de départ dans le graphe
  - État objectif : point d'arrivée dans le graphe
- Naturellement : recherche d'une route entre deux villes

Recherche dans un graphe

Plus spécifiquement, la recherche sur graphe est décrite par :

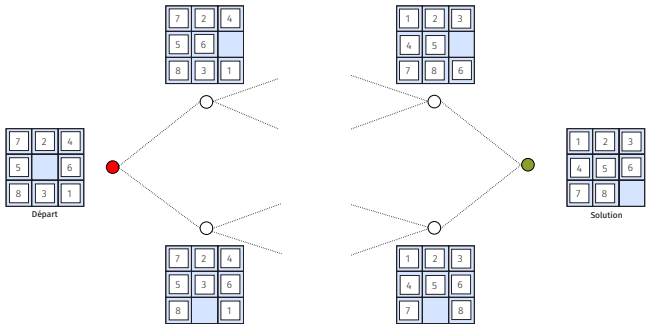
- Les **états du graphe** : les états d'un jeu, la ville visitée, sont représentés par les noeuds du graphe
- Les **actions**/**arêtes** du graphe : pour passer d'un état à un autre, les actions correspondent aux arêtes du graphe
- Un **état initial** (état de départ) : ville de départ, taquin dans son état initial
- Un **état objectif** (état cible) : ville d'arrivée, taquin résolu, etc.

Remarque

- Il n'est pas nécessaire de construire l'intégralité du graphe pour faire la recherche : il se construit au fur et à mesure qu'on le parcourt

Conversion en un graphe

- La conversion n'est pas toujours évidente



Conversion en un graphe

La conversion est parfois compliquée : exemple du problème des n-reines

- On recherche un état et non un chemin $\rightarrow$ recherche "sous contrainte"
- Il n'y a *a priori* pas d'état initial, ni d'actions à effectuer
- Il faut convertir la recherche d'états en recherche sur graphe avec :
  - des états / un espace de recherche
  - des actions
  - un état initial
  - un état final

Conversion en un graphe

Solution 1

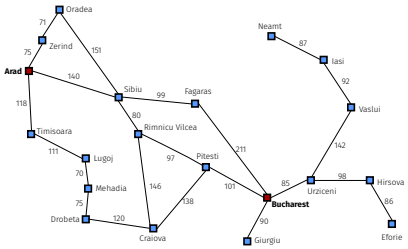
- États : l'ensemble des états à n-reines
- État initial : un état quelconque
- Action : déplacer une reine
- État final : n-reine sans attaque

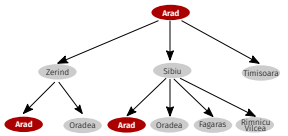
Solution 2

- États : l'ensemble des états à 0, 1, 2, ..., n-reines
- État initial : aucune reine
- Action : ajouter une reine
- État final : n-reine sans attaque

Du graphe à l'arbre de recherche

- La recherche réalise le parcours du graphe/espace de recherche
- Cela permet de définir un arbre de recherche
  - Trace de l'exploration du graphe/espace de recherche qui se construit petit à petit
  - La branche allant jusqu'au noeud qui répond à l'objectif est la solution de la recherche





Recherche générique

- La liste des noeuds permet de définir dans quel ordre l'arbre de recherche est construit
  - Contient la liste des futurs noeuds à explorer
  - L'évaluation du noeud se fait lors de son défilement !
- **objectif()** – [dépend du problème] – vérifie si l'état courant remplit l'objectif $\longrightarrow$ stoppe la construction de l'arbre
- **étendre()** – [dépend du problème] – génère l'ensemble des noeuds successeurs du noeud courant $\longrightarrow$ crée les noeuds fils du noeud courant
- **enfiler()** – [dépend de la stratégie de recherche] – utilisée pour garder "à jour" une liste des prochains noeuds à explorer

Pseudo-code d'une recherche générique

```
Solution fonctionRecherche() {
  noeudsATraiter = File(étatInitial)
  TANTQUE nonVide(noeudsATraiter) {
    noeudCourant = défiler(noeudsATraiter)
    SI objectif(noeudCourant) ALORS
      retourne chemin(noeudCourant)
    SINON
      listeFils = étendre(noeudCourant) // fils du noeud
          courant
      POUR chaque noeudFils dans listeFils
        enfiler(noeudsATraiter, noeudFils)
  }
  retourne ECHEC
}
```


Recherche générique

- Les différentes méthodes de recherche se différencient par leur stratégie de recherche : la fonction enfiler()

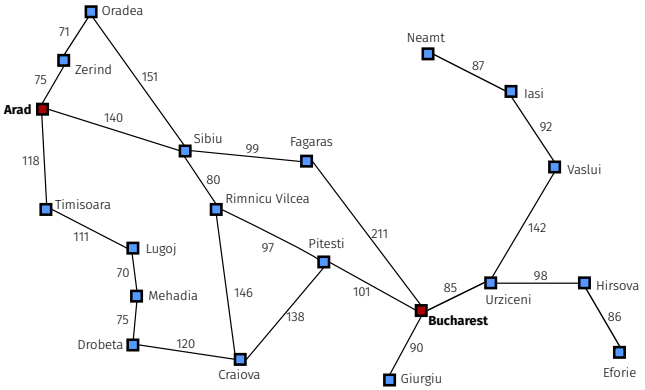
Deux façons principales d'aborder les recherches

- Recherche **non informée / aveugle**
  - Utilisation des informations du problème uniquement
- Recherche **informée**
  - Utilisation d'informations additionnelles permettant de réduire le temps de recherche

Recherches

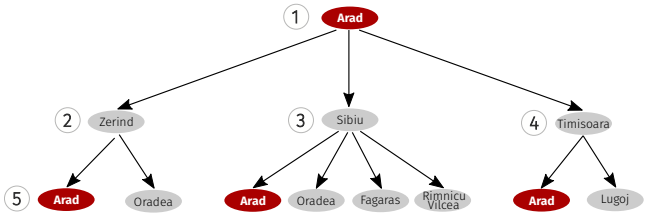
Recherche non informée

Recherche d'un chemin d'Arad à Bucarest



Recherche en largeur / *Breadth-first search* (BFS)

- Parcourt l'ensemble des noeuds voisins du noeud courant avant de visiter la suite



- Implémentation : **file d'attente FIFO**

Recherche en largeur / *Breadth-first search* (BFS)

Propriétés

- **Complétude** ?
  - Oui, tous les noeuds sont examinés
- **Optimalité** ?
  - Oui, pour le plus court chemin
- Complexité en **temps** ?
  - ?
- Complexité **mémoire** ?
  - ?

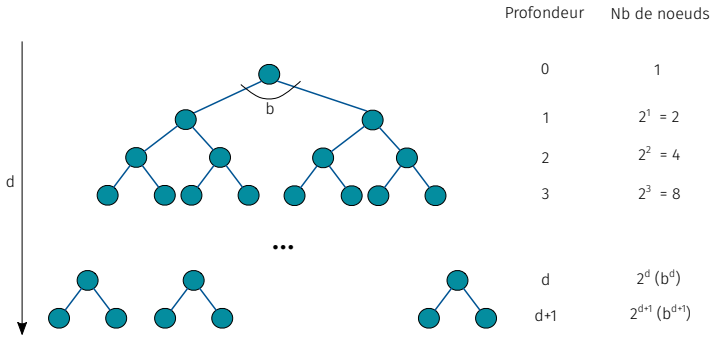
La complexité dépendant de :

- b : facteur de branchement
- d : profondeur de la solution optimale
- m : profondeur maximale

Recherche en largeur / *Breadth-first search* (BFS)

Complexité en temps

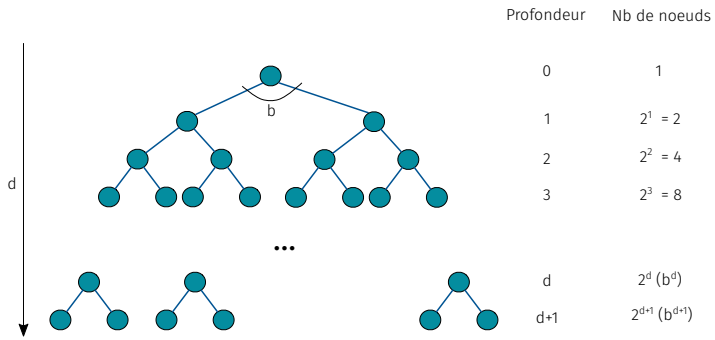
- Nombre de noeuds à visiter avant de trouver la solution



Recherche en largeur / *Breadth-first search* (BFS)

Complexité en mémoire

- Nombre de noeuds à garder en mémoire avant de trouver la solution



Profondeur	Nb de noeuds
0	1
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
d	2^d (b^d)
d+1	2^{d+1} (b^{d+1})

Recherche en largeur / *Breadth-first search* (BFS)

Propriétés

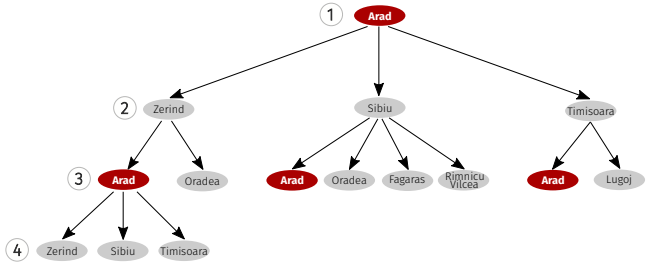
- **Complétude** ?
  - Oui, tous les noeuds sont examinés
- **Optimalité** ?
  - Oui, pour le plus court chemin
- Complexité en **temps** ?
  - $\mathcal{O}(b^d)$ (exponentielle en d)
- Complexité **mémoire** ?
  - $\mathcal{O}(b^d)$ (exponentielle en d)

Conclusion

Trouve une solution optimale, mais coûts très importants

Recherche en profondeur / *Depth-first search* (DFS)

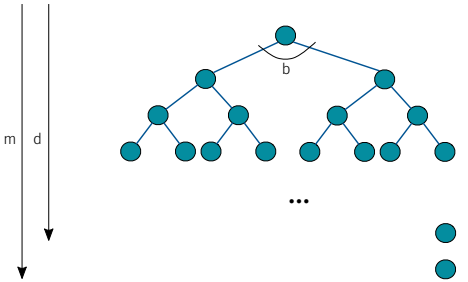
- Parcourt les noeuds fils avant les noeuds voisins, avec retour en arrière si extension impossible



- Implémentation : **pile LIFO**

Recherche en profondeur / *Depth-first search* (DFS)

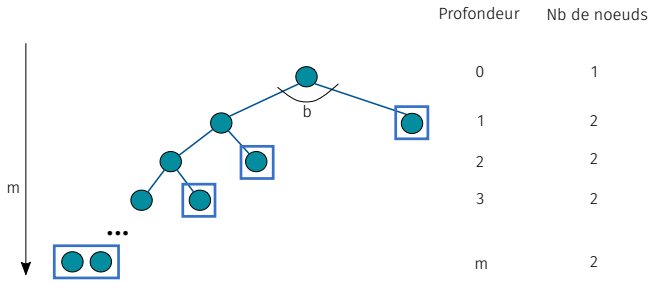
Complexité en temps



Profondeur	Nb de noeuds
0	1
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
d	2^d
m	2^m

Recherche en profondeur / *Depth-first search* (DFS)

Complexité en mémoire



Recherche en profondeur / *Depth-first search* (DFS)

Propriétés

- **Complétude** ?
  - Non ! Peut entrer dans une boucle infinie
  - Oui, pour un graphe complet acyclique OU si on gère les répétitions
- **Optimalité** ?
  - Non ! La première solution trouvée n'est pas forcément la meilleure
- Complexité en **temps** ?
  - $\mathcal{O}(b^m)$ (exponentielle en m)
- Complexité **mémoire** ?
  - $\mathcal{O}(mb)$ (linéaire)

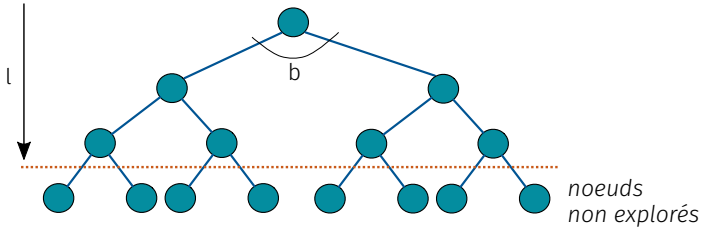
Conclusion

- Économe en mémoire

Recherche en profondeur / *Depth-first search* (DFS)

Comment limiter les boucles ?

- Recherche en profondeur limitée

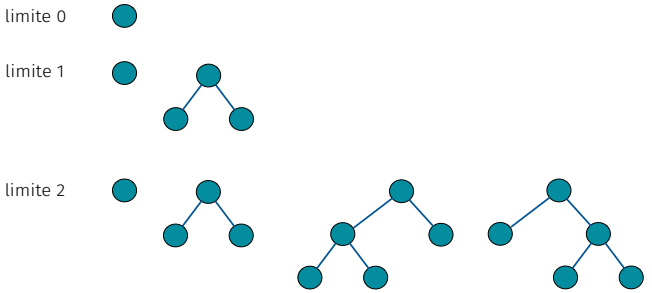


Problème

- Comment trouver la profondeur limite ?
- Elle dépend du problème : 20 villes $\rightarrow l = 20$

Recherche en profondeur itérative / *Iterative depth depth-first search* (ID-DFS)

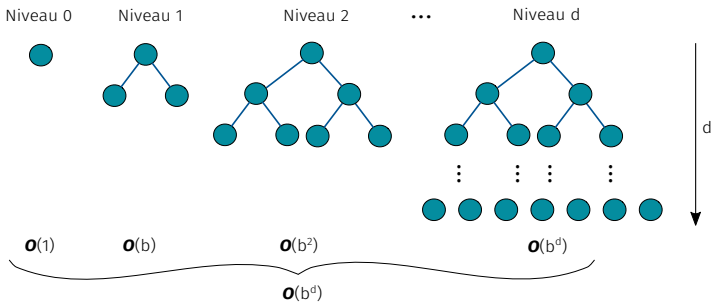
- Procède à une recherche en profondeur limitée, avec limite à 0, puis 1, 2, 3, etc. jusqu'à obtention de la solution



- Combine les avantages de la recherche en profondeur et de la recherche en largeur

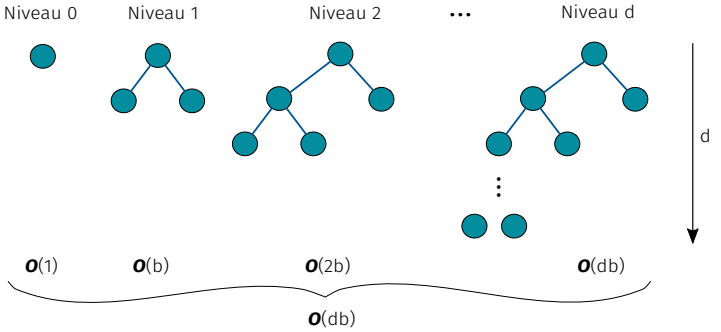
Recherche en profondeur itérative / *Iterative depth depth-first search* (ID-DFS)

Complexité en temps



Recherche en profondeur itérative / *Iterative depth depth-first search* (ID-DFS)

Complexité en mémoire



Recherche en profondeur itérative / *Iterative depth depth-first search* (ID-DFS)

Propriétés

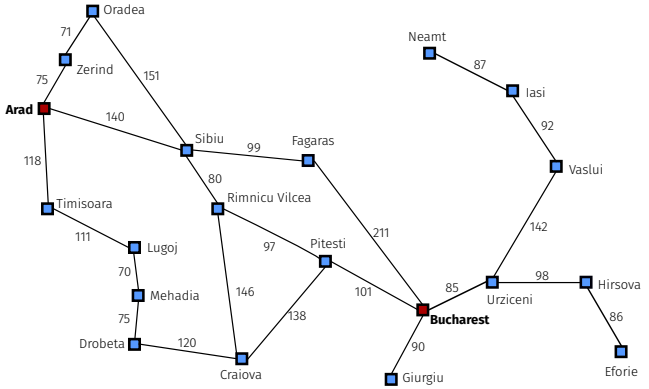
- **Complétude** ?
  - Oui, comme la recherche en largeur si on incrémente la profondeur limite de un à chaque itération
- **Optimalité** ?
  - Oui, comme la recherche en largeur
- Complexité en **temps** ?
  - $\mathcal{O}(b^d)$ (exponentielle en d)
- Complexité **mémoire** ?
  - $\mathcal{O}(db)$ (linéaire)

Conclusion

- Trouve une solution optimale, économe en mémoire, mais coûts en temps important

Problème de recherche du plus court chemin

Les actions ont un coût



Recherche non informée avec des coûts

- On applique un **coût** aux actions
- Les noeuds se voient appliquer une **fonction de coût**
  - $g(n)$: coûts cumulés pour aller de l'état initial à l'état n

Stratégie de recherche

- On étend "intelligemment" : d'abord les noeuds ayant la fonction de coût $g(n)$ minimum (équivalent à une recherche en largeur si les coûts valent 1)
- Exemple : algorithme de **Dijkstra** du plus court chemin qui trouve tous les plus courts chemins depuis un état initial
- Cas particulier : recherche en **coût uniforme**

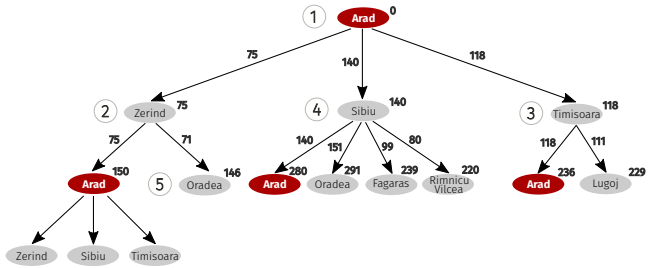
Recherche en coût uniforme / *Uniform cost search*

```
Solution fonctionRechercheCoutUniforme() {
  noeudsATraiter = FilePriorité(étatInitial)
  TANTQUE (nonVide(noeudsATraiter)) {
    noeudCourant = défiler(noeudsATraiter)
    SI objectif(noeudCourant) ALORS
      retourne chemin(noeudCourant)
    SINON
      listeFils = étendre(noeudCourant) // fils du noeud
          courant
      POUR chaque noeudFils dans listeFils
        enfilerPriorité(noeudsATraiter, noeudFils,
            évaluer(noeudFils))
  }
  retourne ECHEC
}
```

- enfilerPriorité () correspond à l'ajout dans une file de priorité
- évaluer() correspond à la fonction de coût $g(n)$

Recherche en coût uniforme / *Uniform cost search*

- On étend d'abord les noeuds ayant la fonction $g(n)$ minimum



Recherche en coût uniforme / *Uniform cost search*

Propriétés

- **Complétude** ?
  - Oui, si les fonctions de coût sont uniformes
- **Optimalité** ?
  - Oui, pour le chemin de moindre coût
- Complexité en **temps** ?
  - $\mathcal{O}(b^d)$ dans le pire des cas, et nombre de noeuds dont le coût est inférieur au coût optimal dans la pratique
- Complexité **mémoire** ?
  - $\mathcal{O}(b^d)$ dans le pire des cas, et nombre de noeuds dont le coût est inférieur au coût optimal dans la pratique

Conclusion

- Trouve une solution optimale, coûts importants dans le pire des cas

Problème et suppression des répétitions

- Il arrive souvent que des noeuds soient rencontrés plusieurs fois
- Perte de temps en prenant en compte plusieurs fois le même noeud

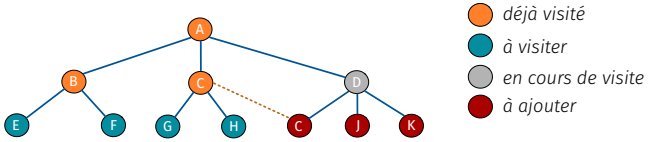
Deux cas possibles

- Noeud déjà visité (rencontre d'un noeud pour lequel on connaît le chemin optimal depuis le noeud initial)
- Non visité (rencontre d'un noeud dans la liste des noeuds à visiter, dont on ne connaît pas encore le chemin optimal depuis le noeud initial)

Problème et suppression des répétitions

Noeud déjà visité

- Ne peut pas appartenir à une solution optimale



- Il ne faut pas étendre le noeud s'il est déjà visité

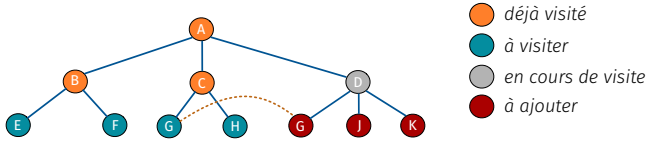
Problème

- Il faut garder les états en mémoire

Problème et suppression des répétitions

Noeud non visité

- Correspond à un noeud pour lequel un noeud identique (ou d'état identique) est présent dans une autre branche
- Il faut garder l'un des noeuds (le plus adapté pour trouver le plus court chemin)



- Il ne faut pas étendre un noeud s'il y a une meilleure solution
- Il faut supprimer l'ancien noeud si le nouveau noeud est meilleur

Problème et suppression des répétitions

Suppression des non visités dans une recherche en largeur

- Ne pas étendre si un noeud a déjà été visité
- Le meilleur noeud aura évidemment été visité en premier

Suppression des non visités dans une recherche en profondeur (itérative)

- L'ordre de visite ne garantit pas la possibilité de supprimer des noeuds
- Il faut garder une trace des noeuds déjà visités avec leur coût/profondeur associés

Suppression des non visités dans une recherche en coût uniforme

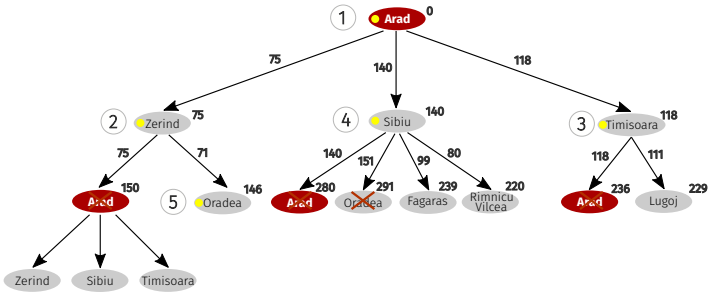
- Supprimer les noeuds dont la fonction de coût est la plus grande

Utilisation de deux listes pour gérer les noeuds : liste fermée (noeuds déjà visités) et liste ouverte (noeuds à visiter)

Pseudo-code de la recherche en coût uniforme, sans répétitions

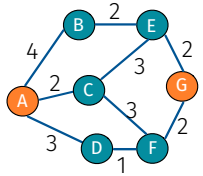
```
Solution fonctionRechercheCoutUniforme() {
  listeOuverte = FilePriorité(étatInitial)
  listeFermée = Liste()
  TANTQUE (!vide(listeOuverte)) {
    noeudCourant = défiler(listeOuverte)
    enfiler(listeFermée, noeudCourant)
    SI objectif(noeudCourant) ALORS
      retourne chemin(noeudCourant)
    SINON
      listeFils = étendre(noeud)
      POUR chaque noeudFils dans listeFils ET
           !dans listeFermée {
        SI noeudFils dans listeOuverte ALORS
          mettreAJour(listeOuverte, noeudsFils,
              g(noeudsFils))
        SINON
          enfilerPriorité(listeOuverte, noeudFils,
              g(noeudFils))
      }
  }
  retourne ECHEC;
```


Suppression des répétitions dans une recherche en coût uniforme



Exercice

Trouver le chemin le plus court de A à G ?



- Tracer l'arbre de recherche :
  - Pour une recherche en largeur
  - Pour une recherche en profondeur
  - Pour une recherche en coût uniforme
- On prendra les noeuds dans l'ordre alphabétique lorsqu'on étendra le problème
- On gérera la répétition de noeuds avec des listes "ouvertes" et "fermées"

Recherches

Recherche informée

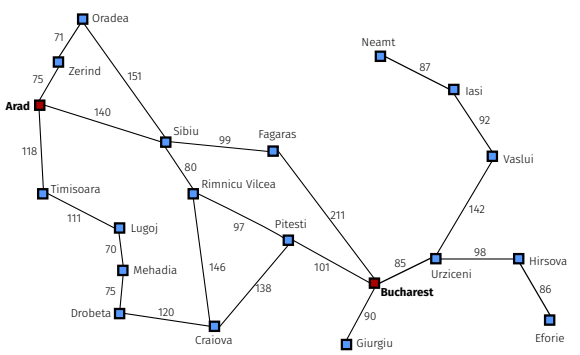
Recherches informées

Les recherches non informées (*à l'aveugle*) utilisent uniquement les informations du problème

- Les **recherches informées** sont plus efficaces
- Elles utilisent une fonction supplémentaire pour estimer le potentiel d'un noeud
- Cette fonction permettant d'estimer le potentiel, notée $h(n)$, s'appelle **fonction heuristique**

- Ce type de recherche est une recherche *best-first* (meilleur potentiel visité en premier)
- L'évaluation d'un noeud prend en compte la fonction heuristique
- C'est une extension de la recherche en coût uniforme

Recherches informées : exemple



Heuristique
"Distance à vol d'oiseau vers Bucharest"

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Recherches informées : fonction d'évaluation

La fonction d'évaluation d'un noeud n, notée $f(n)$, dépend de :

- $g(n)$: coût réel pour aller de l'état initial jusqu'à l'état n par le plus court chemin
- $h(n)$: estimation du coût minimal pour aller de l'état n à l'état objectif

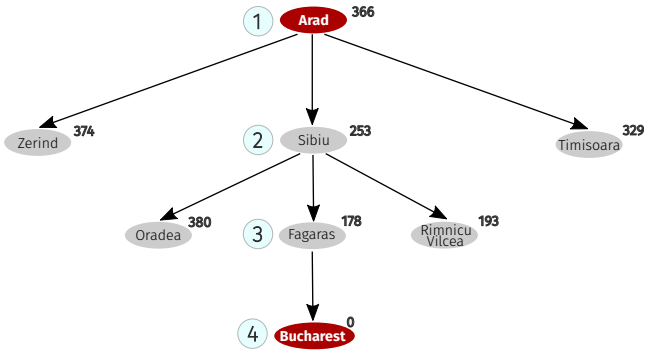
Plusieurs cas

- Algorithme **glouton** : $f(n) = h(n)$
- Algorithme A^* : $f(n) = g(n) + h(n)$
- Variantes de A^*

La qualité de la recherche dépend de la qualité de l'heuristique

Recherche gloutonne *best-first* / *greedy best-first*

- Application de la recherche *best-first* avec $f(n) = h(n)$
- $h(n)$: fonction heuristique
- Par exemple : distance à vol d'oiseau vers Bucharest



Recherche gloutonne *best-first* / *greedy best-first*

Propriétés

- **Complétude** ?
  - Non ! Si on ne gère pas les répétitions, oui sinon
- **Optimalité** ?
  - Non ! Le chemin optimal au sens de l'heuristique n'est pas le chemin optimal au sens des coûts
- Complexité en **temps** ?
  - $\mathcal{O}(b^m)$ dans le pire des cas (souvent meilleure)
- Complexité **mémoire** ?
  - $\mathcal{O}(b^m)$ dans le pire des cas (souvent meilleure)

Conclusion

- Peut fournir une solution acceptable en temps raisonnable, selon la qualité de l'heuristique

Inconvénients des recherches à coût uniforme et gloutonne

- La recherche en coût uniforme n'utilise pas d'information *a priori* sur la solution
  - $f(n) = g(n)$
  - Explore toutes les solutions
- La recherche gloutonne ne tient pas compte de la fonction de coûts
  - $f(n) = h(n)$
  - Peut partir dans des branches à coûts élevés !

Solution

- L'algorithme A^* permet de prendre en compte les deux
  - $f(n) = g(n) + h(n)$
  - Fonctionne bien si la fonction heuristique est **admissible** et **monotone**

Fonction heuristique admissible et monotone

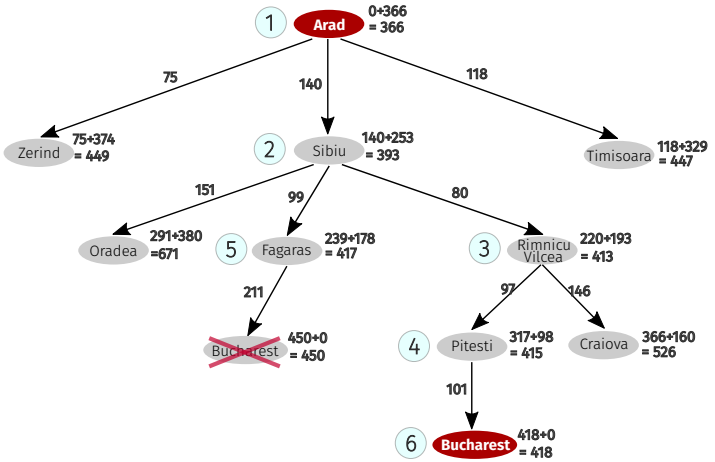
- Une fonction heuristique est dite **admissible**
  - Si pour un noeud n, $h(n) \leq h^*(n)$, avec $h^*(n)$ le coût réel pour atteindre le noeud destination depuis le noeud courant n

Une fonction heuristique admissible ***ne surestime jamais le coût****, elle doit être optimiste !*

- Une fonction heuristique est dite **monotone**
  - Si pour tous noeuds x et y, on a $h(x) \leq c(x, y) + h(y)$

- Dans notre exemple, la distance à vol d'oiseau est inférieure ou égale à la distance réelle à parcourir, donc, c'est une fonction heuristique admissible

Recherche A^*



Recherche A^*

On prononce A étoile, ou A star (à l'anglaise)

Propriétés

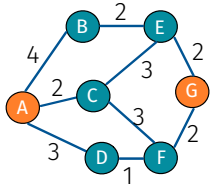
- **Complétude** ?
  - Oui
- **Optimalité** ?
  - Oui (avec une fonction heuristique admissible)
- Complexité en **temps** ?
  - $\mathcal{O}(b^d)$ dans le pire des cas (souvent meilleure $\mathcal{O}(db)$)
- Complexité **mémoire** ?
  - $\mathcal{O}(b^d)$ dans le pire des cas (souvent meilleure $\mathcal{O}(db)$)

Recherche A^*

L'algorithme A^* est couramment utilisé dans les jeux pour trouver un chemin entre les obstacles (jeux de stratégie $\rightarrow$ permet le déplacement des unités)

Exercice

Trouver le chemin le plus court de A à G ?



Noeud	Distance
A	4
B	3
C	2
D	3
E	1
F	1
G	0

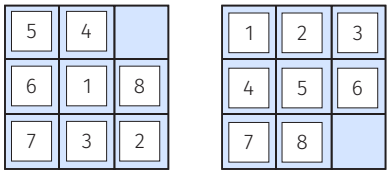
- Tracer l'arbre de recherche
  - Pour une recherche gloutonne
  - Pour une recherche A^*
- On prendra les noeuds dans l'ordre alphabétique s'ils sont au même niveau

Choix des heuristiques

- Les heuristiques doivent être **admissibles** pour que la recherche fonctionne correctement, c.-à-d. : $h(n) \leq h^*(n)$
- Comment choisir une bonne heuristique ?
  - $h1$: nombre de pièces mal placées
  - $h2$: somme des distances de chaque pièce à leur destination (distance de Manhattan)

Distance de Manhattan

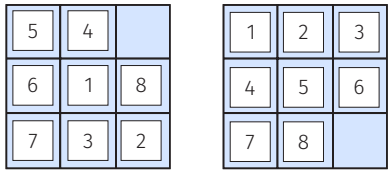
- Nombre de déplacements nécessaires s'il n'y a aucune autre pièce sur le jeu



Départ Solution

Choix des heuristiques

Exemple



- $h1(\text{départ})$ = 7
- $h2(\text{départ})$ = 16 = 2 + 3 + 3 + 2 + 2 + 2 + 0 + 2

Choix des heuristiques

Pour deux fonctions $h1$ et $h2$ admissibles

- Si $h1(n) \leq h2(n)$, pour tout état n de l'espace de recherche
- Alors $h2$ domine $h1$
- Et $h2$ est plus efficace

Exemple : nombre de noeuds explorés en moyenne pour le problème du taquin

- Solution en 14 niveaux de profondeur (peu mélangé)
  - parcours en profondeur $\rightarrow 3\,473\,941$ noeuds
  - $A^*(h1) \rightarrow 539$ noeuds
  - $A^*(h2) \rightarrow 119$ noeuds
- Solution en 24 niveaux de profondeur (très mélangé)
  - IDDFS $\rightarrow$ échec car trop de noeuds (3^{24})
  - $A^*(h1) \rightarrow 39\,135$ noeuds
  - $A^*(h2) \rightarrow 1\,641$ noeuds

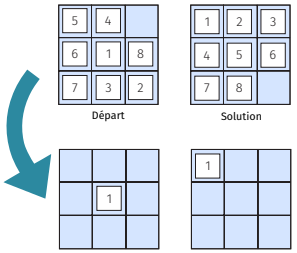
Choix des heuristiques

Comment créer une bonne fonction heuristique ?

- En résolvant des problèmes "relaxés" (avec moins de contraintes)

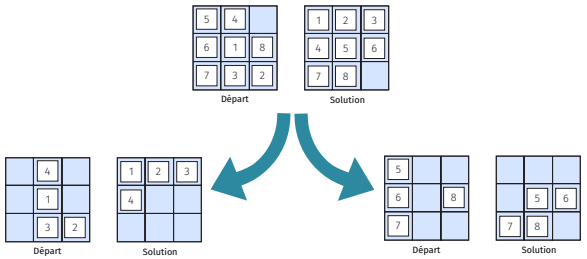
Exemple pour le taquin

- Résoudre indépendamment les 8 "sous-taquins"
- Correspond à $h2$
- Ignorer les effets négatifs entre les pièces permet d'être optimiste !



Choix des heuristiques

- On peut améliorer la fonction $h2$ en considérant deux sous-problèmes :
  - $h(n) = d1234 + d5678$, avec les coûts exacts des deux solutions



- Ces distances peuvent être stockées dans une base de données (soit précalculées, soit par programmation dynamique)

Recherches

Recherche sous contraintes

Les recherches sous contraintes

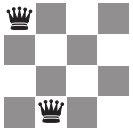
- Il peut arriver que l'on recherche un état et non un chemin $\rightarrow$ recherche **sous contraintes** / *Constraint satisfaction problem* (CSP)

Un problème de recherche sous contraintes définit :

- Un **état** qui correspond à un ensemble de variables : $X_1, X_2, \cdots, X_n$
- Chaque variable X_i prend sa valeur dans un **domaine** D_i non vide ($\rightarrow$ le domaine peut être différent pour chaque variable)
- Un ensemble de **contraintes** : $C_1, C_2, \cdots, C_m$
- Une **assignation complète** correspond à l'affection d'une valeur pour chaque variable X_i
- Une **solution** correspond à une assignation complète qui satisfait l'ensemble des contraintes C_i

Exemple : problème de n-reines

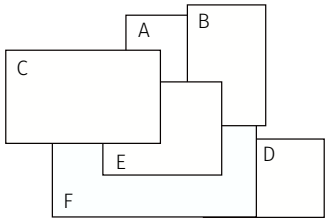
- **Objectif** : trouver un état tel que les n-reines ne s'attaquent pas
- **Variables** : R_1, R_2, R_3, R_4 (R_i : reine dans la colonne i) et valeur de R_i : ligne de la reine
- **Domaine** : $D_1 \times D_2 \times D_3 \times D_4$ (D_i : ensemble des valeurs possibles pour R_i)
- **Contraintes**
  - $R_i \neq R_j$: deux reines ne doivent pas être sur la même ligne
  - $|R_i - R_j| \neq |i - j|$: deux reines ne doivent pas être sur la même diagonale



Assignation : R_1=1, R_2=4

Exemple : coloration de cartes

- **Objectif** : colorier une carte avec k-couleurs sans que deux pays adjacents n'aient la même couleur
- **Variables** : A, B, C, D, E (les pays)
- **Domaine** : $D_A = \{R, V, B\}$ $D_A = D_B = D_C = D_D = D_E$
- **Contraintes** : $A \neq B$, $A \neq C$, $\cdots$



Contraintes

Une contrainte peut être :

- **Explicite** : la variable doit prendre sa valeur dans un ensemble limité de valeurs
- **Implicite** : la valeur doit répondre à une fonction test
- **Unaire** : une seule variable est impliquée – $A \neq V, R_1 > 2$
- **Binaire** : une paire de variables est impliquée – $R_1 \neq R_2$
- D'ordre plus élevé quand plus de deux variables sont impliquées – fonction "toutes différentes"

Contraintes

La recherche sous contraintes peut être vue comme une recherche classique

- **Espace de recherche** : ensemble des différentes assignations (partielle ou complète) d'une valeur aux variables
- **État initial** : aucune variable assignée
- **Actions** : assigner une valeur à une variable
- **État objectif** : assignation complète répondant à toutes les contraintes

Contraintes

- Les actions ont toutes le même coût (coût = 1) car on recherche un état et non un chemin $\rightarrow$ pas de chemin/solution optimale
- La profondeur maximale d'une solution est connue d'avance
  - Nombre de variables à assigner
  - Profondeur limitée par le nombre de variables à assigner
  - Pas de boucle infinie
- Il n'est pas nécessaire de garder en mémoire les noeuds (recherche en profondeur)

Pseudo-code de la recherche en profondeur sous contraintes

L'appel initial de la fonction se fait avec une assignation vide

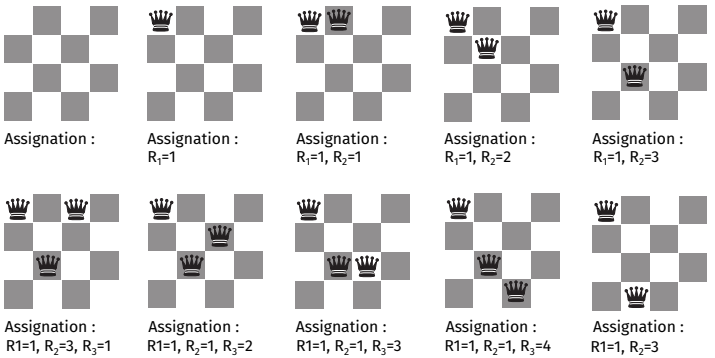
- sélectionne() : on sélectionne une variable qui n'a pas encore de valeur dans l'assignation
- consistant() : on vérifie que toutes les valeurs assignées répondent aux contraintes
- complet() : on vérifie si l'ensemble des variables ont une valeur, si oui, fin de la recherche

Pseudo-code de la recherche en profondeur sous contraintes

```
Solution fonctionBacktrackingRecursive(assignation) {
  var = sélectionne(variables, assignation)
  POUR chaque valeur dans domaine(var) {
    SI valeur consistant(contraintes, assignation) ALORS {
      insère(assignation, var, valeur)
      SI complet(assignation) ALORS
        retourne assignation
      résultat = fonctionBacktrackingRecursive(assignation)
      SI résultat != ECHEC ALORS
        retourne´resultat
      SINON
        supprime(assignation, var)
    }
  }
  retourne ECHEC
}
```

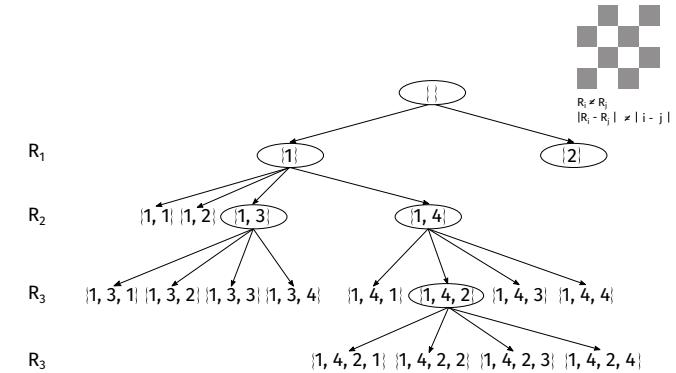

Algorithme backtraking

On assigne dans l'ordre R_1, R_2, R_3 et R_4



Algorithme backtraking

On assigne dans l'ordre R_1, R_2, R_3 et R_4

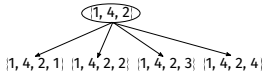


Algorithme backtraking

```
boolean solveNQUtil(int board[][], int col) {
  /* base case: If all queens are placed then return true */
  if (col >= N)
    return true;

  /* Consider this column and try placing this queen in all
     rows one by one */
  for (int i = 0; i < N; i++) {
    /* Check if the queen can be placed on board[i][col] */
    if (isSafe(board, i, col)) {
      /* Place this queen in board[i][col] */
      board[i][col] = 1;

      /* recur to place rest of the queens */
      if (solveNQUtil(board, col + 1) == true)
        return true;

      /* If placing queen in board[i][col] doesn't lead to
         a solution then remove queen from board[i][col] */
      board[i][col] = 0; // BACKTRACK
    }
```

Algorithme backtraking : optimisation

On peut améliorer l'algorithme en remarquant que certaines branches n'ont plus aucune solution



Le fait d'assigner une valeur à une variable crée de nouvelles contraintes que l'on peut propager aux noeuds fils $\longrightarrow$ Réduction du facteur de branchement

Propagation des contraintes

La propagation de contraintes permet de réduire le temps de la recherche

- Plusieurs techniques permettent de propager les contraintes
  - Le *forward checking* : simple à mettre en oeuvre
  - La consistance des arcs (*arc consistency*) : plus efficace

Propagation des contraintes : *forward checking*

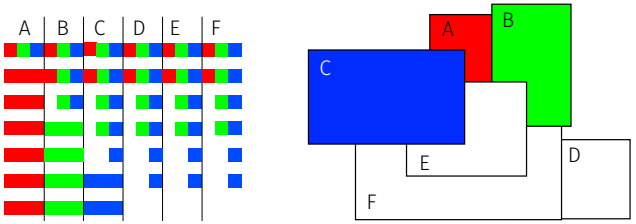
La procédure de *forward checking* consiste à :

- Récupérer l'ensemble des valeurs autorisées pour chaque variable au début de l'algorithme
- Dès que l'on assigne une valeur à une variable, on supprime pour toute autre variable les valeurs qui ne sont plus autorisées
- Si une variable n'a plus de valeurs autorisées, on remonte dans l'arbre de recherche

$\longrightarrow$ Mise à jour de domaine(var)

Propagation des contraintes : *forward checking*

Exemple de coloration de cartes en *forward checking*



Les ensembles des valeurs possibles pour les variables D, E et F sont vides, donc on remonte dans l'arbre de recherche

Propagation des contraintes : arc et consistance d'arc

Un arc correspond à la contrainte d'une variable sur une autre

- L'arc $X \to Y$ est **consistant** SSI pour chaque valeur possible de la variable X, il y a une valeur possible pour la variable Y

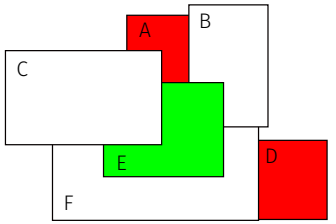
Vérifier la consistance des arcs consiste à

- Récupérer l'ensemble des valeurs autorisées pour chaque variable au début de l'algorithme
- Dès qu'une variable X perd une valeur possible, on vérifie la consistance de tous les arcs $X \to Y$
- Si un arc n'est pas consistant, on remonte dans l'arbre de recherche

Propagation des contraintes : arc et consistance d'arc

Exemple de coloration de cartes

Couleur	R	V	B
A	✓	✗	
B	✗	✗	
C	✗	✗	
D	✓		
E	✗	✓	
F	✗	✗	



Les arcs $F \to B$ et $F \to C$ ne sont plus consistants, donc on remonte dans l'arbre de recherche

Amélioration de la recherche sous contraintes : heuristiques

On peut améliorer l'efficacité des recherches sous contraintes à l'aide d'**heuristiques**

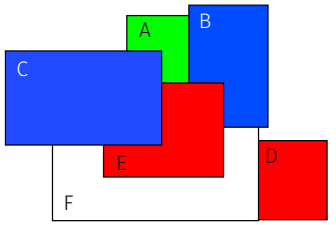
Il existe 2/3 d'heuristiques principales, indépendantes du problème, qui visent à :

- Choisir la prochaine variable à fixer
  - La variable la plus contrainte (avec le moins de valeurs possible)
  - La variable la plus contraignante (qui enlève le plus de valeurs possible aux autres variables)
- Choisir la valeur à affecter
  - La valeur la moins contraignante (qui enlève le moins de valeurs aux autres variables)

Amélioration de la recherche sous contraintes : heuristiques

Exemple de coloration de cartes

Couleur	R	V	B
A	✗	✓	✗
B	✗	✗	✓
C	✗	✗	✓
D	✓	✗	
E	✓	✗	✗
F	✗	✓	✗



Recherche sous contraintes : résumé

Les problèmes de recherche sous contraintes sont des problèmes particuliers

- États définis par un ensemble de valeurs attribuées à des variables
- L'état objectif doit répondre à des contraintes
- On recherche un **état** et pas un chemin

L'algorithme de recherche (*backtracking search*) est une recherche en profondeur

- On part d'une assignation vide
- On affecte une variable à chaque fois que l'on descend d'un niveau dans l'arbre de recherche
- La propagation de contraintes (*forward checking*, *arc consistency*) permet d'éviter de descendre dans des branches vouées à l'échec
- Des heuristiques permettent de réduire significativement le temps de recherche

Recherches

Recherche concurrentielle

Les recherches concurrentielles

- La **recherche concurrentielle** concerne les recherches du meilleur coup à jouer dans un jeu à plusieurs joueurs
  - Théorie des jeux
- Implique plusieurs joueurs avec des intérêts différents
  - Plusieurs centres de décision
- Les jeux étant une version simplifiée du monde réel, la recherche concurrentielle s'applique aussi à différents domaines :
  - économie, politique, diplomatie, guerre, etc.

Les recherches concurrentielles : types de jeux

- Coopératifs / non-coopératifs
  - On gagne ensemble / la victoire de l'un est la défaite de l'autre
- Information complète / incomplète
  - Les joueurs ont accès à l'ensemble des états du jeu / ont une ou des informations secrètes
  - Ex. : échecs, go, domino / poker, tarot, Monopoly
- Déterministe / non-déterministe
  - N'implique pas de chance ou hasard
  - Ex. : échecs / poker
- À somme nulle / somme positive
  - La perte d'un joueur correspond exactement au gain de l'autre joueur
  - Ex. : échecs / dilemme du prisonnier
- Simultané / séquentiel
  - Les joueurs jouent en même temps / chacun leur tour
  - Ex. : pierre feuille ciseau / échecs

Les recherches concurrentielles : types de jeux

On s'intéresse dans ce cours aux jeux :

- à 2 joueurs
- non coopératifs
- à information complète
- déterministe
- à somme nulle
- séquentiel

Exemples

- morpion
- puissance 4
- dames
- échecs
- go

Les recherches concurrentielles : principe

- La recherche concurrentielle consiste à trouver le meilleur coup à jouer pour gagner la partie
  - on ne cherche plus un chemin
  - on ne cherche plus un état
- Deux joueurs s'affrontent
  - **MAX** : le joueur (IA) qui cherche à jouer le meilleur coup maximise une fonction d'évaluation (utilité à jouer le coup)
  - **MIN** : le joueur adversaire (IA ou humain) minimise une fonction d'évaluation (utilité à jouer le coup)
  - **MAX** et **MIN** jouent l'un après l'autre
- On a besoin d'une **fonction d'évaluation** ou d'**utilité** d'un état

Les recherches concurrentielles : algorithme Minimax

- On calcule récursivement la valeur d'un noeud ⇔ On réfléchit plusieurs coups à l'avance
- La valeur à remonter correspond au
  - Maximum des valeurs des noeuds fils, si **MAX** joue
  - Minimum des valeurs des noeuds fils, si **MIN** joue
- Si on est à un noeud terminal ("partie terminée"), la valeur correspond à la fonction d'évaluation/utilité de l'état du noeud

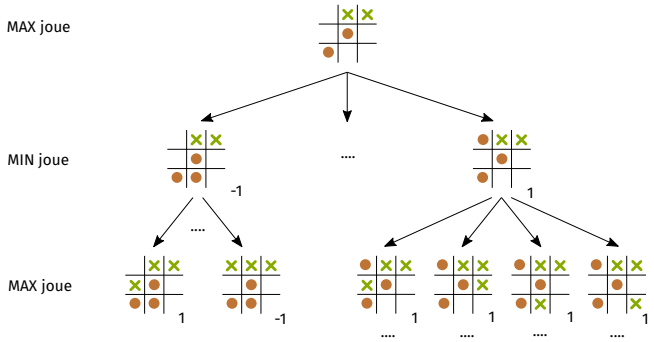
Les recherches concurrentielles : algorithme Minimax

```
Action fonctionMinimaxDecision(état) {
  V = fonctionMaxValeur(état)
  retourne action dans successeurs(état) avec valeur == V
}

Valeur valeurMax(état) {
  SI terminal(état) ALORS
    retourne utilité(état)
  V = INFINI
  POUR chaque successeur(état) FAIRE
    V = max(V, fonctionMinValeur(successeur(état)))
  retourne V
}

Valeur valeurMin(état) {
  SI terminal(état) ALORS
    retourne´utilite(état)
  V = -INFINI
  POUR chaque successeur(état) FAIRE
    V = min(V, fonctionMaxValeur(successeur(état)))
  retourne V
```

Les recherches concurrentielles : algorithme Minimax

Exemple du morpion

- Utilité : perte = -1, égalité = 0, gain = 1, MAX joue les ronds



Les recherches concurrentielles : algorithme Minimax

Propriétés (c'est une recherche en profondeur !)

- **Complétude** ?
  - Oui, dans un arbre fini
- **Optimalité** ?
  - Oui, même contre un adversaire optimal
- Complexité en **temps** ?
  - $\mathcal{O}(b^m)$
- Complexité **mémoire** ?
  - $\mathcal{O}(mb)$

Conclusion

- Algorithme optimal (mais important en temps suivant les jeux !)
- Inconvénient : l'algorithme doit parcourir l'intégralité de l'arbre pour connaître la valeur des feuilles

Les recherches concurrentielles : algorithme Minimax

Propriétés

Valeur de b/m pour quelques jeux

- Échecs
  - Nombre de coups possibles à chaque tour : ~ 35
  - Nombre de tours : $\simeq 75$
- Go
  - Nombre de coups possibles à chaque tour : ~ 300
  - Nombre de tours : $\simeq 250$

Minimax ne peut pas être appliqué sur ces jeux en temps raisonnable

- Besoin de limiter
  - la profondeur de recherche
  - le nombre de coups à considérer

Les recherches concurrentielles : algorithme Minimax

Coupure Minimax (rech. en profondeur + limite de profondeur)

- Le test terminal devient un test en milieu de partie
- Parcours l'arbre de recherche jusqu'à une certaine profondeur
- Remplace la fonction d'utilité par une fonction d'évaluation

Fonction d'évaluation

- Associe un score à un état (estime l'utilité de l'état terminal) $\rightarrow$ heuristique

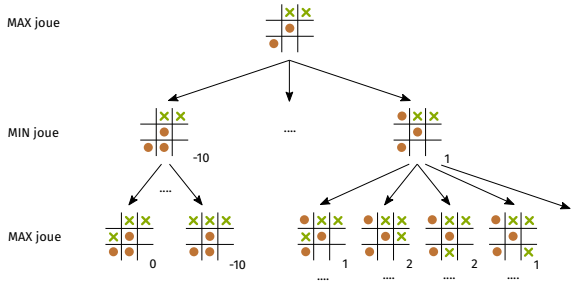
Exemple

- Morpion
  - Eval(état) $=$ # lignes possibles joueur $-$ # lignes possibles adversaire

Les recherches concurrentielles : algorithme Minimax

Exemple du morpion

- Profondeur = 2 $\longrightarrow$ on réfléchit deux coups en avance
- Eval(état) = +10 si victoire, -10 si défaite
- Sinon # lignes possibles pour MAX— # lignes possibles pour MIN



Les recherches concurrentielles : élagage alpha-beta

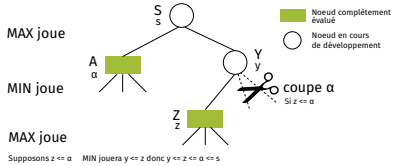
L'**élagage alpha-beta** permet de réduire le facteur de branchement en évitant d'explorer des branches sans intérêt

- **alpha** et **beta** valent respectivement $-\infty$ et $+\infty$ au départ
- **alpha** correspond au meilleur coup de MAX, et MAX se met à jour (resp. **beta**/MIN)
- Les valeurs de **alpha** et **beta** sont passées aux noeuds fils lors de leurs évaluations $-\infty, +\infty$

Les recherches concurrentielles : élagage alpha-beta

alpha correspond au meilleur coup de MAX

- MAX a la possibilité de jouer un coup qui va amener à un état avec la valeur **alpha**
- Si dans le prochain coup exploré par MAX, MIN peut jouer un coup moins bon que **alpha**, MAX ne le jouera pas
- Alors, il n'est pas nécessaire de poursuivre l'exploration des autres coups possibles pour MIN
  - Coupure **alpha** lors de l'exploration des coups de MIN
- De même avec **beta**, le meilleur coup de MIN



Les recherches concurrentielles : élagage alpha-beta

```
Action fonctionAlphaBeta(état, profondeur) {
  V = fonctionMaxValeur(état, profondeur, -INFINI, INFINI)
  retourne action dans successeurs(état) avec valeur == V
}

Valeur fonctionMaxValeur(état, profondeur, alpha, beta) {
  SI terminal(état) OU profondeur == 0 ALORS
    retourne évaluation(état)
  V = -INFINI
  POUR chaque E dans successeurs(état) FAIRE
    V = max(V, fonctionMinValeur(E, profondeur-1, alpha,
        beta))
    SI V > beta ALORS
      retourne V /* coupure beta*/
    alpha = max(alpha, V)
  retourne V
}
```

Les recherches concurrentielles : élagage alpha-beta

```
Valeur fonctionMinValeur(état, profondeur, alpha, beta) {
  SI terminal(état) OU profondeur == 0 ALORS
    retourne évaluation(état)
  V = +INFINI
  POUR chaque E dans successeurs(état) FAIRE
    V = min(V, fonctionMaxValeur(E, profondeur-1, alpha,
        beta))
    SI V < alpha ALORS
      retourne V /* coupure alpha */
    beta = min(beta, V)
    retourne V
}
```


Les recherches concurrentielles : remarques sur l'élagage

- **alpha** est toujours inférieur à **beta**
- L'ordre dans lequel les coups sont explorés influe sur les performances de l'élagage **alpha-beta**
  - Si on trouve le meilleur coup à jouer dans la première branche, il y aura beaucoup de coupures
- La qualité de l'heuristique influe sur la performance de l'élagage
  - Si on a deux coups à jouer : a et b, et deux heuristiques h_1 et h_2
  - Si $h_1(a) = h_1(b)$ et $h_2(a) < h_2(b)$
  - Alors a et b sont identiques au sens de h_1 (peu de possibilités de coupure)
  - Alors b est un meilleur coup au sens de h_2 (possibilité de coupure plus importante)
- **Complexité en temps**
  - Dans le pire des cas, identique à minimax : $\mathcal{O}(b^m)$
  - Bien meilleur dans le meilleur des cas : $\mathcal{O}(b^{m/2})$

Les recherches concurrentielles : résumé

- L'algorithme minimax permet de gagner à tous les coups, mais il n'est pas applicable aux jeux trop complexes
- Remplacer la fonction d'utilité par une fonction d'évaluation permet de limiter la profondeur de recherche
- L'élagage **alpha-beta** permet d'éviter l'exploration de certaines branches inutiles

Les recherches concurrentielles : mais encore ?

AlphaGo

- Victoire au jeu de Go d'une IA sur l'humain en 2016 (Google)

Variante de alpha/beta : arbre de recherche de Monte Carlo

- Exploration d'une partie des branches

Utilisation de méthodes d'apprentissage (*deep Learning*)

- Choix des branches
- Fonction d'évaluation d'un état du jeu

Métaheuristiques

Métaheuristiques

Introduction

Introduction

Exemple et explosion combinatoire

- Imaginons le problème du chemin optimal permettant de desservir 14 clients
- Combien existe-t-il de chemins possibles ?
  - $14! = 8,718 \times 10^{10} = 88$ milliards de solutions
- Si un ordinateur peut vérifier environ 1 million de solutions par seconde, il faudra $8,718 \times 10^{10}/10^6 = 88\,000$ secondes, soit environ 24 heures pour toutes les vérifier et trouver le meilleur !

Introduction

Heuristique

- "chercher" (du grec $\epsilon\upsilon\rho\iota\sigma\kappa\epsilon\iota\nu$")
- Les heuristiques sont souvent des algorithmes spécialisés (TSP, remplissage de bacs, etc.)

Meta-

- abstraction d'un autre concept
  - au-delà, dans un niveau supérieur
  - utilisé pour compléter
- Exemple : métadonnées = "données à propos des données"

Métaheuristique

- "Heuristique autour des heuristiques"
- Les **métaheuristiques** constituent des techniques d'optimisation par recherche de minima locaux

Introduction

Une métaheuristique

- Peut résoudre des problèmes d'optimisation des domaines discret et continu
- Est une stratégie qui "guide" le processus de recherche
- Explore efficacement l'espace de recherche pour trouver de bonnes solutions (presque optimales)
- N'offre aucune garantie d'optimalité globale ou locale
- Manque d'un indicateur de "qualité" de la solution (s'arrête souvent en raison d'une limite de temps ou d'itération externe)
  I. Boussaïd, J. Lepagnot, P. Siarry. "A survey on optimization metaheuristics". Information Sciences, 237, 82-117, 2013.

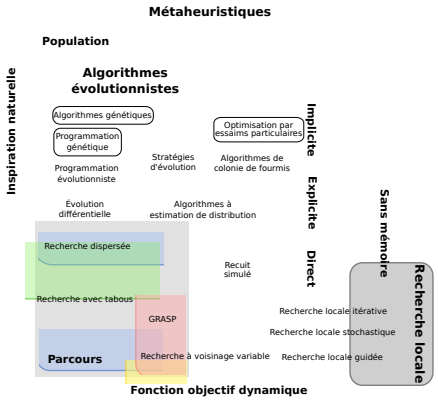
https://www.sciencedirect.com/science/article/pii/S0020025513001588

Introduction

Applications par domaines

- Ingénierie et conception industrielle
- Logistique (ordonnancement, routage)
- Finance (gestion de portefeuille, optimisation d'investissements)
- Biologie et biochimie (ex. : conformation moléculaire)
- Jeux vidéos (ex. : comportements, création de niveaux, personnages, etc.)

Introduction : classification



Source : Wikimedia (Johann "nojhan" Dréo, Caner Candan – Metaheuristics classification), licence CC-BY-SA 3.0

Métaheuristiques

Métaheuristiques basées parcours

Métaheuristiques basées parcours

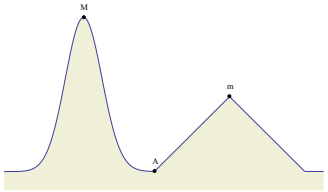
- Recherche locale basique (3192 av. J.C.)
- Recuit simulé (*simulated annealing*) (1983)
- Recherche tabou (1986)
- Algorithme glouton / GRASB *Greedy randomized adaptive search procedure* (1989)
- Recherche locale guidée (1997)
- Recherche de voisinage variable (1999)
- Recherche locale itérative (1999)

Algorithme glouton

Principe

- Algorithme le plus simple : à chaque itération on ajoute l'élément le **plus prometteur**
- Cet algorithme ne construit qu'une seule solution, mais de manière itérative

Inconvénient

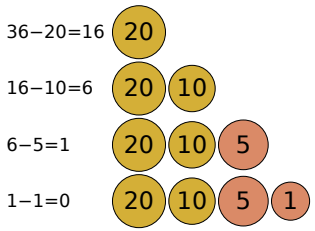


À partir de A, l'algorithme glouton favorise la solution m à la place de la solution optimale M.

Algorithme glouton

Applications

- Allocation de ressources : ordonnanceur sur un système d'exploitation, rendu de monnaie, réservation dans un train, création d'un horaire, etc.



Algorithme glouton

```
/* Algorithme GRASP */
FAIRE {
   solution = useRandomizedGreedyAlgorithm();
   meilleure_solution = localSearchFrom(solution);
} TANTQUE (conditionArret() == FALSE)
Retourne meilleure_solution
```

Recuit simulé

Principe

- Inspirée du recuit des lames en métallurgie
- $P = e^{-\frac{\Delta E}{kT}}$ avec $k = 1,380510^{-23} J/K$ (constante de Boltzmann) et T la température absolue

S. Kirkpatrick, C. D. Gelatt Jr, M. P. Vecchi. "Optimization by Simulated Annealing". Science. 220(4598): 671–680, 1983.

Applications

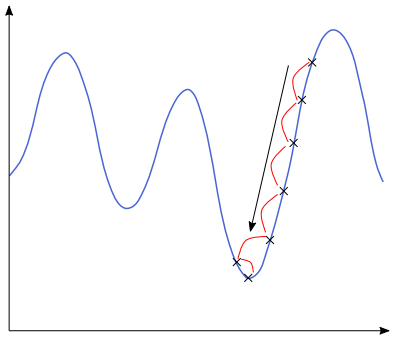
- Problèmes courants : voyageur de commerce, coloration de graphes, ordonnancement, etc.
- Ingénierie : conception VLSI, placement, routage, traitement d'images, etc.

Recuit simulé

```
Crée une solution initiale aléatoire s
E_ancien = cout(s);
POUR (t = tmax; t >= tmin; t = next_temp(t)) {
  POUR (i = 0; i < imax; i++) {
    successeur(s); // choix aléatoire d'une autre solution
    E_nouveau = cout(s);
    delta = E_nouveau - E_ancien;
    SI (delta > 0)
      SI (random() >= exp(-delta/K*t);
        undo(s); // mauvais choix rejeté
      SINON
        E_ancien = E_nouveau  // mauvais choix accepté
    SINON
    E_ancien = E_nouveau; // toujours accepter les bons
        choix
  }
}
```


Recherche tabou

Descente de gradient



Recherche tabou

Principe

- Descente de gradient améliorée
- À chaque itération, déplacement vers le meilleur voisin même s'il est moins bon que la solution actuelle
- On mémorise dans une FIFO la liste des positions déjà visitées qui deviennent taboues
- Ne bloque pas dans le premier optimum rencontré
  F. Glover, M. Laguna. "Tabu search". Ed Kluwer Academic, 1997.

Applications

- Problèmes courants : voyageur de commerce, etc.

Recherche tabou

```
meilleureSolution = s0
meilleurCandidat = s0
listeTaboue = []
listeTaboue.push(s0)
TANTQUE (!conditionArret()) {
  voisins = getNeighbors(meilleurCandidat)
  POUR (candidat DANS voisins) {
    SI ((! listeTaboue.contains(candidat)) ET
        (fitness(candidat) > fitness(meilleurCandidat)))
      meilleurCandidat = candidat
    }
  }
  SI (fitness(meilleurCandidat) >
      fitness(meilleureSolution)) {
    meilleureSolution = meilleurCandidat
  }
  listeTaboue.push(meilleurCandidat)
  SI (listeTaboue.size > maxTabuSize) {
    listeTaboue.removeFirst()
  }
}
```


Métaheuristiques

Métaheuristiques basées population

Métaheuristiques basées population

Algorithmes évolutionnaires

- Programmation évolutionnaire (1962)
- Algorithmes génétiques (1975)
- Algorithme d'estimation de distribution (1996)

Intelligence par essaims

- Optimisation par colonie de fourmis (1992)
- Optimisation par essaim particulaire (1995)
- Accouplement d'abeilles (2005)

Évolution différentielle (1995)

Programmation par mémoire adaptative (1997)

Métaheuristiques basées population

- Les populations ne constituent pas nécessairement un ensemble de solutions complètes
- Deux stratégies de sélection des individus d'une nouvelle population
  - sélectionner tous les nouveaux individus
  - sélectionner les plus aptes

Il est important de débuter avec une population diversifiée

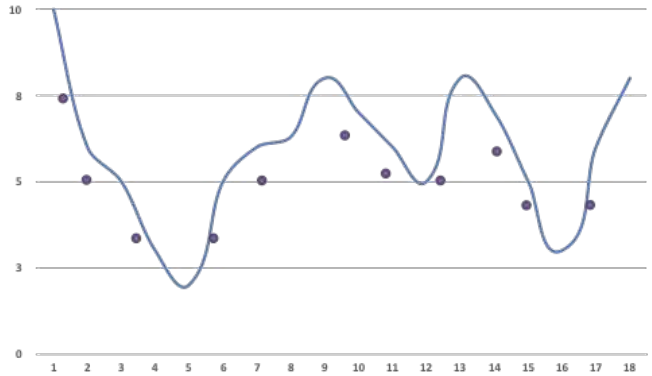
Métaheuristiques basées population

Programmation évolutionnaire

- Les populations évoluent en “générations”
- La taille d’une population est (presque toujours) fixe
- Les nouveaux individus sont créés en combinant les caractéristiques des individus parents (généralement deux) – caractère aléatoire important
- Les individus évoluent en utilisant des opérateurs de variation (par exemple, “mutation”, “entrecroisement”) agissant directement sur leurs représentations
- La population suivante comprend un mélange d’enfants et de parents en fonction de la stratégie de “sélection des survivants”

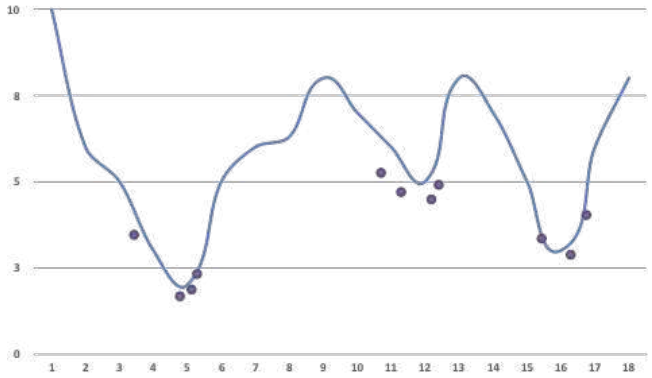
Métaheuristiques basées population

Programmation évolutionnaire (population initiale)



Métaheuristiques basées population

Programmation évolutionnaire (nième population)



Métaheuristiques

Algorithmes génétiques (*ga*)

ga : concept d'évolution biologique

- Évolution biologique étudiée depuis la fin du 18e siècle
- Darwin (1809-1882) et Lamarck (1744-1829) se sont opposés sur les raisons des modifications entre les parents et les descendants, appelées **mutations**
- Darwin : un descendant est la moyenne de ses parents avec aléatoirement des différences – seules les mutations qui apportent un avantage sélectif sont conservées et se propagent
- Lamarck : les variations sont une réponse à un besoin physiologique interne (cou de la girafe)
- La thèse de Darwin semble en partie confirmée : lors de la reproduction, des mutations aléatoires se produisent de temps en temps

ga : concept d'évolution artificielle

- Algorithmes de recherche supervisés basés sur le concept d'évolution biologique
- Développée par John Holland à l'université du Michigan (années 1970) pour
  - comprendre le processus adaptatif des systèmes naturels
  - concevoir des logiciels qui conservent la robustesse des systèmes naturels
  - fournir des techniques efficaces pour l'optimisation et les applications d'apprentissage automatique

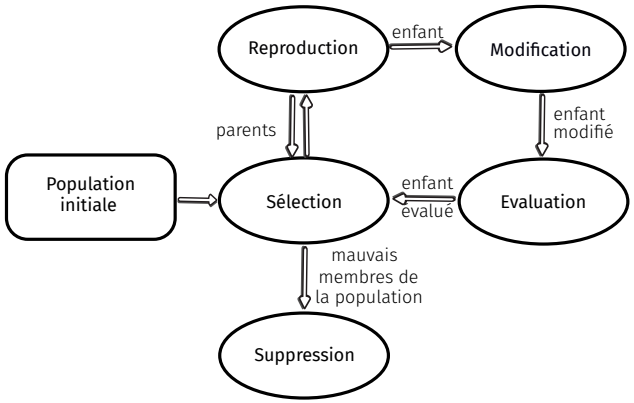
Applications

- Largement utilisé aujourd'hui dans les milieux d'affaires, scientifiques et d'ingénierie :
  - gestion du trafic aérien,
  - positionnement des antennes relais mobiles
  - etc.

ga : caractéristiques d'un algorithme génétique

- Algorithme **stochastique** (aléatoire) – résultat différent à chaque exécution
- Utilisé pour résoudre des problèmes difficiles
- Gère une **population** de solutions (*individus*)
- Les solutions sont codées à l'aide des **gènes**
- La **reproduction** crée de nouveaux membres dans la population
- **Entrecroisement** et **mutation** interviennent pendant la reproduction
- Il y a *survie* des plus **aptes** : les meilleurs individus auront plus de chance de se reproduire

ga : cycle d'un algorithme génétique



ga : algorithme génétique simple

```
initialiser(population);
évaluer(population);
TANTQUE (Condition de terminaison non satisfaite) {
   pairesDeParents = sélectionnerParents(population);
   nouveauxIndividus = recombiner(pairesDeParents);
   muter(nouveauIndividus);
   évaluer(nouveauIndividus);
   population = sélectionnerIndividus();
}
```

- Le critère de terminaison peut être :
  - nombre maximal de générations
  - aptitude maximale pour les individus

ga : population

- La population consiste en un ensemble d'**individus**
- Les individus possèdent une liste de **gènes** qui encodent les solutions du problème
- À chaque individu dans la population est associée une évaluation de son **aptitude** (*fitness*)
- Les **gènes** peuvent être représentés par :
  - Des chaînes de bits (0101 ... 1110)
  - Des réels (3.14 78.5 ... -12.3 0.0)
  - Des permutations d'éléments
  - Toute structure de données

ga : exemples de gènes et d'individu

Le "travelling salesman problem"

"'{java} class TspGene implements Gene { int townIndex; }

class TspIndividual extends Individual { ArrayList genome; }

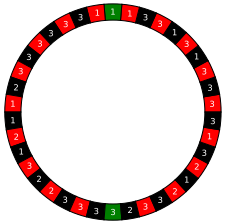
```
### Le labyrinthe

'''{java}
class MazeGene implements Gene {
   int direction; // North, South, East, West
}

class MazeIndividual extends Individual {
   ArrayList<Gene> genome;
}
```


ga : reproduction

- Des parents sont sélectionnés aléatoirement pour se reproduire
- Les meilleurs parents ont de meilleures chances de se reproduire
- Les parents sont replacés dans la population après reproduction

Exemple de sélection des parents

- Parents 1, 2 et 3 avec des aptitudes respectives de 10, 7 et 21
- Utilisation d'une roulette "biaisée"



ga : modification

Une modification est aléatoire

Deux types de modifications

- **Mutation** : appliquée à un seul génome
- **Entrecroisement** ou enjambement (*crossover*) : appliqué à deux génomes

Entrecroisement et mutation sont des **opérateurs génétiques**

ga : mutation

Une mutation cause une modification locale

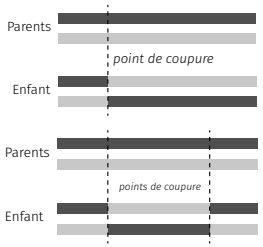
- Les mutations ont une probabilité faible ($\sim 5\%$ au départ)
- Un algorithme génétique utilise des mutations aléatoires
- Les mutations sont parfois contraintes en fonction de l'environnement

Exemples

- $(0110110) \to (0010110)$
- $(3.1478.5 \cdots - 12.30.0) \to (3.1478.5 \cdots - 13.30.0)$
- $(12345) \to (23145)$

ga : entrecroisement

L'entrecroisement recombine le matériel génétique des parents



Remarque

- L'entrecroisement produit presque toujours des mauvais descendants :
  - Si les gènes liés sémantiquement sont éloignés
  - Si les gènes sont contraints par le reste du génome

ga : évaluation

Un individu est "évalué" par une fonction d'évaluation ou fonction de *fitness*

- La fonction peut être calculée à partir des données contenues dans les gènes
- Elle peut aussi être donnée par :
  - un humain
  - un test réel
  - etc.

Exemples de fonctions de *fitness*

- "travelling salesman problem" : calcul du nombre total de kilomètres en calculant les distances entre villes (deux à deux)
- Labyrinthe : distance de Manhattan entre la sortie et la case courante

ga : survie des individus

- La taille de la population doit être constante
- La solution la plus simple consiste à remplacer totalement les adultes par les enfants
- Des tournois entre des individus des deux populations peuvent être organisés
  - en opposant systématiquement un individu de chaque génération
  - en les tirant au sort dans la population

Métaheuristiques

Optimisation par colonies de fourmis (*aco*)

aco : optimisation par colonie de fourmis

- Également basée sur une approche population
- Contrairement aux algorithmes évolutionnaires où les individus évoluent séparément, toute la population participe à la construction d'une **mémoire partagée**
- La mémoire est utilisée lors de la phase de recombinaison afin de générer la nouvelle population
- La mémoire partagée est représentée par la "matrice de phéromone"
  M. Dorigo, V. Maniezzo et A. Colorni. "Ant system: optimization by a colony of cooperating agents." IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 26(1):29-41, 1996

aco : phéromone

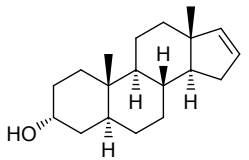
Phéromone

- du grec $\phi\epsilon\rho\omega$ (porter) et $o\rho\mu\eta$ (désir)

Phéromone (biologie)

- Substance chimique produite par un animal ou un insecte pour attirer d'autres animaux ou insectes, en particulier un partenaire

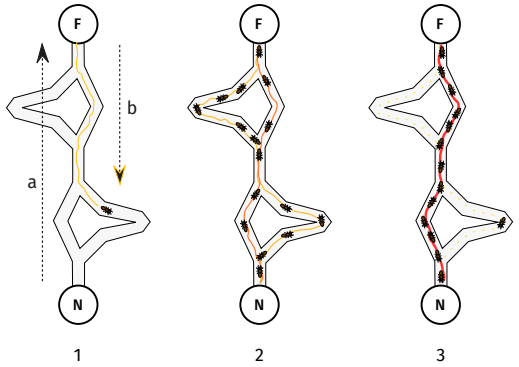
Phéromone (chimie)



aco : population de fourmis

- Les fourmis se déplacent dans un territoire familier où la sécurité est reconnue
- Elles secrètent de la phéromone partout où elles passent
- Elles détectent la phéromone sur le sol qui a tendance à atteindre des niveaux de concentration élevés
- La phéromone s'évapore avec le temps

aco : optimisation



- Chaque solution possible est associée à un niveau de phéromone
- Ce niveau augmente ou diminue en fonction de la probabilité que la solution fasse partie de la solution finale

aco : règle de déplacement

$$p_{ij}^k(t) = \begin{cases} \frac{\gamma+\tau_{ij}(t)^\alpha \cdot \eta_{ij}^\beta}{\sum_{l \in J_i^k}(\gamma+\tau_{il}(t)^\alpha \cdot \eta_{il}^\beta)} & \text{si } j \in J_i^k \\ 0 & \text{si } j \notin J_i^k \end{cases} \quad (1)$$

où :

- J_i^k : liste des déplacements possibles pour une fourmi k lorsqu'elle se trouve sur un noeud i
- η_{ij} : visibilité, qui est égale à l'inverse de la distance de deux noeuds i et j $(1/d_{ij})$
- $\tau_{ij}(t)$: intensité de la piste à une itération donnée t

Les trois principaux paramètres contrôlant l'algorithme sont α, β et γ qui vérifient l'importance relative de l'intensité, de la visibilité d'une arête, et du fait qu'un noeud n'a pas encore été visité

aco : dépôt de phéromone

- Une fourmi k dépose une quantité $\Delta\tau_{ij}^k$ de phéromone sur chaque arête de son parcours

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{L^k(t)} & \text{si } (i,j) \in T^k(t) \\ 0 & \text{si } (i,j) \notin T^k(t) \end{cases} \quad (2)$$

où :

- $T^k(t)$ est la tournée faite par la fourmi k à l'itération t
- $L^k(t)$ est la longueur du trajet
- Q est un paramètre de réglage

aco : niveau de phéromone à l'itération $t+1$

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) \qquad (3)$$

où :

- $\rho\tau_{ij}(t)$ est la quantité de phéromones évaporées à l'itération t
- m est le nombre de fourmis utilisées pour l'itération t
- ρ un paramètre de réglage

Bibliothèques

Java

- JAMES : http://www.jamesframework.org
- jMetal : https://github.com/jMetal/jMetalSP

C++

- jMetalCpp : http://jmetalcpp.sourceforge.net