INF 431

F. Morain

Graphes III: introduction à l'optimisation combinatoire

21 mars 2007

Où en est-on ?

Amphi 1: introduction.

Amphi 2: génie logiciel avec Java.

Amphi 3: analyse lexicale.

Amphi 4: analyse syntaxique.

Amphi 5: graphes I.

Amphi 6: graphes II (parcours).

Amphi 7: graphes III (optimisation combinatoire).

Amphi 8: graphes IV (topologie).

Plan

I. Plus courts chemins.

II. L'algorithme de Dijkstra.

III. Arbre couvrant de poids minimal.

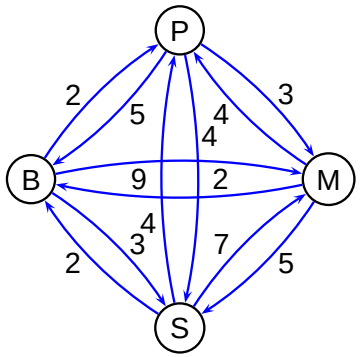
IV. Exploration.

I. Plus courts chemins

Pb. on cherche le plus court chemin entre tous les sommets i et j d'un graphe $\mathcal{G}$ quand on dispose d'une fonction d donnant la distance entre i et j (avec $d(i,j) \geqslant 0$).
On représente le graphe par :

$$D_{i,j} = \begin{cases} d(i,j) & \text{si } (i,j) \in A, \\ 0 & \text{si } i = j, \\ \infty & \text{sinon.} \end{cases}$$

Ex.



Premier algorithme : on définit $D_{i,j}^{(k)}$ comme étant la longueur du plus court chemin de i à j ayant au plus k flèches. On a :

$$D_{i,j}^{(k)} = \min\left(D_{i,j}^{(k-1)}, \min_{1\leqslant\ell\leqslant n}(D_{i,\ell}^{(k-1)} + D_{\ell,j}^{(k-1)})\right),$$

d'où un algorithme en $O(n^4)$.

L'algorithme de Floyd :
On considère $F_{i,j}^{(k)}$ comme étant la longueur du plus court chemin de i à j ne passant par aucun sommet d'indice $> k$. On a :

$$F_{i,j}^{(k)} = \min\left(F_{i,j}^{(k-1)}, F_{i,k}^{(k-1)} + F_{k,j}^{(k-1)}\right).$$

Le coût de l'algorithme est en $O(n^3)$.

Rem. On peut modifier le programme pour stocker le plus court chemin, en mémorisant l'indice k fournissant le chemin minimum de i à j.

II. L'algorithme de Dijkstra

Pb: trouver la distance d'**un sommet** s à tous les autres sommets d'un graphe valué $\mathcal{G} = (\mathcal{S}, \mathcal{A})$, avec $d(i,j) = 0$ pour $i = j$; $d(i,j) > 0$ pour $i \neq j$; $d(i,j) = +\infty$ si $(i,j) \notin \mathcal{A}$.

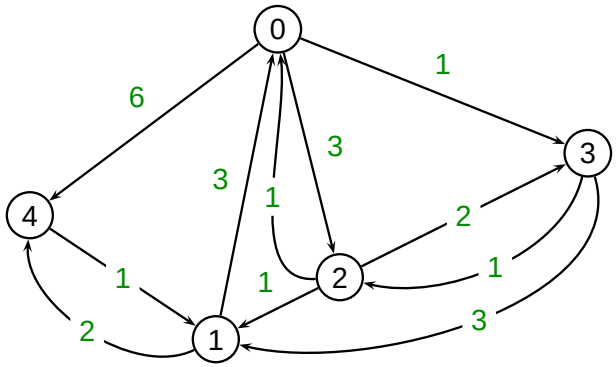
Principes:

- On calcule la distance de s à tous les autres sommets de $\mathcal{G}$ (approche gloutonne).
- À chaque sommet $t \in \mathcal{S}$, on associe une étiquette $L(t)$, initialisée à $d(s,t)$; à la fin de l'algorithme, cette valeur contient la valeur du plus petit chemin de s à t.
- L'algorithme construit un ensemble $T \subset \mathcal{S}$ tel que pour tout $t \in T$, tout chemin minimal de s à t ne passe que par des éléments de T. L'ensemble T grossit incrémentalement de $\emptyset$ à $\mathcal{S}$.

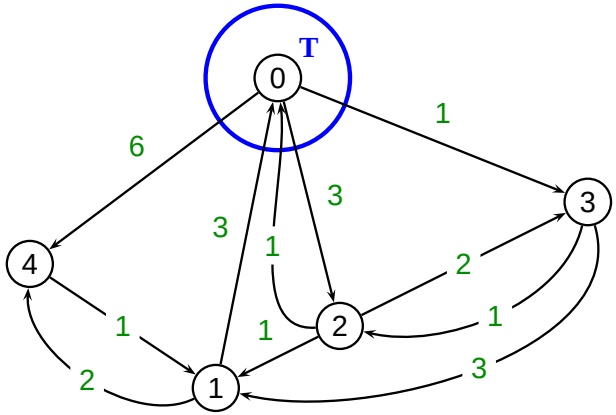
L'algorithme en pseudocode

```
1. pourtout t ≠ s faire
     L(t) <- d(s, t);
2. L(s) <- 0; T <- {s};
3. tantque T ≠ S
4.   trouver v ∉ T telque L(v) = min {L(t), t ∉ T};
5.   T <- T ∪ {v};
6.   pourtout t ∉ T faire
7.     L(t) <- min (L(t), L(v) + d(v, t));
```


Un exemple numérique



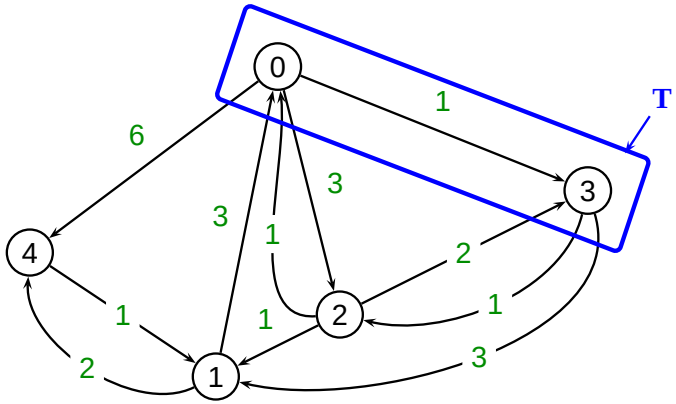
Initialisation



t	1	2	3	4
$L(t)$	$+\infty$	3	**1**	6

$\Rightarrow$ on sélectionne le sommet 3.

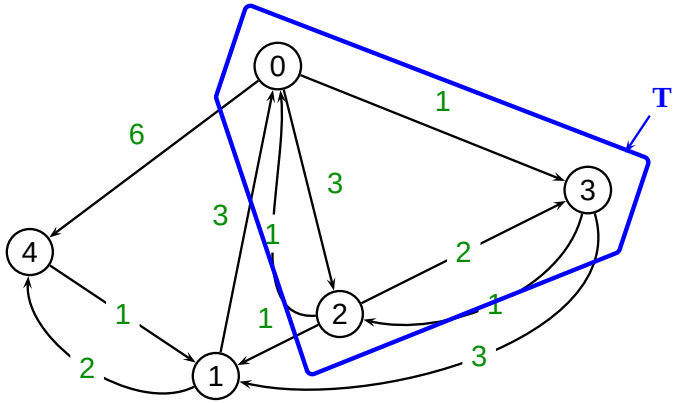
Étape 1



t	1	2	3	4
$L(t)$	**4**	**2**	**1**	6

$\Rightarrow$ on sélectionne 2.

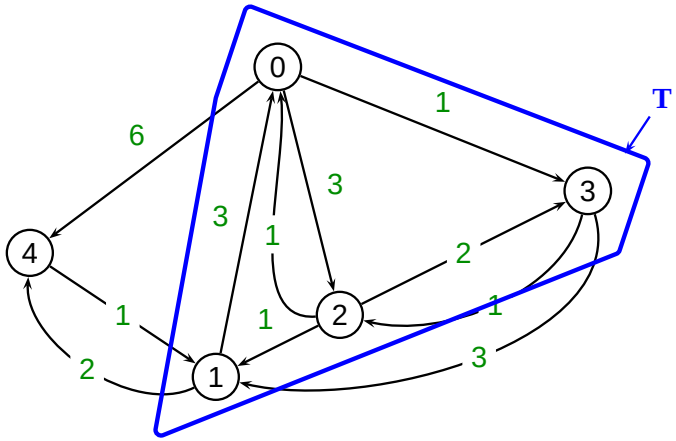
Étape 2



t	1	2	3	4
$L(t)$	**3**	**2**	**1**	6

$\Rightarrow$ on sélectionne 1.

Étape 3



t	1	2	3	4
$L(t)$	**3**	**2**	**1**	**5**

Validité de l'algorithme

Prop. À l'entrée de la boucle 3:

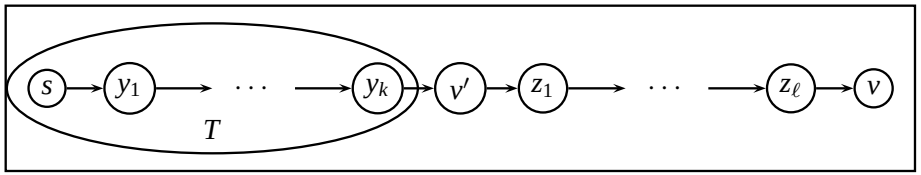
(a) pour tout $t \in T$, $L(t)$ est la longueur d'un chemin minimal de s à t;

(b) pour tout $v \notin T$, $L(v)$ est la longueur d'un chemin minimal de s à v ne passant par aucun sommet de $\mathcal{S} - T$ (en dehors de v).

$|T| = 1$: (a) et (b) sont vraies par construction (cf. ligne 2.).

$|T| > 1$: juste avant la ligne 5, d'après l'hypothèse de récurrence, $L(v)$ est la distance d'un chemin minimal de s à v ne passant que par des sommets de T.

On raisonne par l'absurde: quand on ajoute v, ce n'est pas la longueur d'un chemin minimal de s à v (dans $\mathcal{G}$).
$\Rightarrow$ un plus court chemin doit contenir au moins un autre sommet qui ne soit pas dans T. Soit v' le premier que l'on rencontre en partant de s.



Le chemin $c = (s, y_1, \ldots, y_k, v', z_1, \ldots, z_\ell, v)$ est tel que $y_i \in T$.
Fait 1: $c' = (s, y_1, \ldots, y_k, v')$ est un plus court chemin entre s et v' (sinon, on aurait un chemin encore plus court de s à v).
Fait 2: par construction, la longueur δ du chemin c' est $< L(v)$.
Conclusion: par hypothèse de récurrence sur (b), $\delta = L(v') < L(v)$, ce qui contredit la propriété à la ligne 5.
La propriété (a) est bien encore valide; (b) s'en déduit grâce aux lignes 6-7. □

Implantations

Choix:

- Principe: tout doit être programmé dans la classe abstraite;
- Input: le graphe;
- Output: L, un tableau indexé par les sommets (table de hachage).

Il est plus facile de gérer l'ensemble $U = S - T$:

```
1. pourtout t ≠ s faire
     L(t) <- d(s, t);
2. L(s) <- 0; U <- S - {s};
3. tantque U ≠ ∅
4.   trouver v ∈ U telque L(v) = min {L(t), t ∈ U};
5.   U <- U - {v};
6.   pourtout t ∈ U faire
7.     L(t) <- min (L(t), L(v) + d(v, t));
```


Utilisation d'un ensemble (`HashSet`)

```
Hashtable<Sommet,Integer> Dijkstra(Sommet s){
  int INFINITY = 1000000; // majorant sur les L[]
  Hashtable<Sommet,Integer> L =
    new Hashtable<Sommet,Integer>();

  // 1. initialisation
  for(Sommet t : sommets())
    if(! t.equals(s))
      if(existeArc(s, t))
        L.put(t, valeurArc(s, t));
      else
        L.put(t, INFINITY);
  // 2. copie prudente
  HashSet<Sommet> U
    = new HashSet<Sommet>(sommets());
```

```
    L.put(s, 0); // L[s] <- 0
    U.remove(s);
    // 3.
    while(! U.isEmpty()){
      // 4. recherche du minimum
      Sommet v = null;
      int Lv = INFINITY;
      for(Sommet t : U)
        if(L.get(t) < Lv){
          v = t;
          Lv = L.get(t);
        }
      // 5.
      U.remove(v);
      // 6.
      for(Arc a : voisins(v)){
        Sommet t = a.destination();
        int tmp = Lv + a.valeur();
        if(tmp < L.get(t)) // [3]
          L.put(t, tmp);
      }
    }
    return L;   }
```


Analyse

Trouver le minimum des **`L(t)`** pour $t \in U$ coûte $O(|U|) = O(n - |T|)$.

La mise à jour coûte $O(n_v)$ si n_v est le nombre de voisins de v, d'où un coût total:

$$\sum_{|T|=1}^{n} (O(n - |T|) + O(n_v)) = O(n^2) + O(|\mathcal{A}|) = O(n^2).$$

Améliorations

Première amélioration: on peut utiliser une file de priorité qui permet de trouver le minimum d'un ensemble en temps $O(\log n)$, avec un coût de mise à jour en $O(\log n)$.
Le temps de calcul de notre algorithme serait alors:

$$\sum_{|T|=1}^{n} (O(\log n) + O(n_v \log n)) = O((|\mathcal{A}| + n) \log n),$$

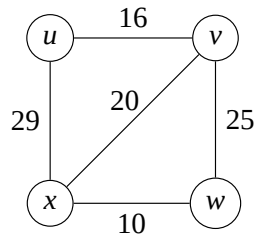
ce qui est meilleur que $O(n^2)$ si $|\mathcal{A}| \ll n^2/\log n$. Cf. poly pour l'implantation avec les **`TreeSet`**.

Seconde amélioration: $O(n \log n) + O(|A|)$ en utilisant un tas de Fibonnaci.

III. Arbre couvrant de poids minimal

Pb typique: étant donné un réseau de canaux de capacité connue devant irriguer un nombre donné de villes, minimiser la somme totale des capacités.

Modélisation: graphe (non orienté) dont les sommets sont les villes et les arêtes les canaux; $val(a) =$ capacité de l'arc a.

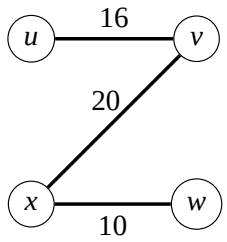


Principe

On note $\varpi(\mathcal{G})$ le poids d'un graphe, défini comme la somme des valeurs de ses arcs.

Nous cherchons un sous-graphe couvrant connexe de $\mathcal{G}$, qui passe par tous les sommets de $\mathcal{G}$, et de poids minimum.

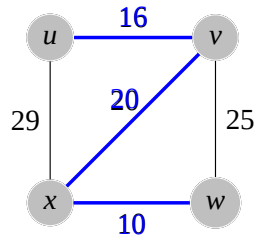
Rem. Si ce sous-graphe possédait un cycle, il suffirait de l'enlever pour diminuer son poids. $\Rightarrow$ on cherche en fait un arbre couvrant de poids minimum.



A) L'algorithme de Prim

```
1. Choisir un sommet initial s;
2. tantque c'est possible
3.   Trouver l'arête de poids minimal joignant
     un sommet déjà sélectionné à un sommet
     non encore sélectionné.
```

Ex.



Implantation

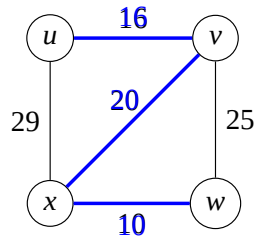
Avec une file de priorité d'arcs classés par ordre croissant de valuation $\Rightarrow O(m \log m)$.

```
Prim(G = (S, A))
X <- S;
tantque X ≠ ∅ faire
  u <- élément suivant de X;
  X <- X - {u};
  T[u] <- ∅; // arbre couvrant de racine u
  F <- file de priorité avec les arêtes (u, x);
  Y <- {u}; // ensemble des sommets de T[u]
  tantque F ≠ ∅ faire
    a = (s, t) <- arête minimale dans F;
    si t ∈ Y alors
      // (s, t) ferme un cycle
    sinon
      Y <- Y ∪ {t}; X <- X - {t};
      T[u] <- T[u] ∪ {(s, t)};
      pour v voisin de t
        ajouter (t, v) dans F;
```


B) L'algorithme de Kruskal

```
1. Trier les arêtes par ordre croissant:
   B = (a0, a1, ..., am);
2. T <- ∅;
3. pour i <- 0 à m-1 faire
     si bi ne crée pas de cycle
       ajouter ai à T;
```

Ex.



Implantation

Principe: union-find pour gérer les cycles.
$C[s] = \{t \in \mathcal{S}$, il existe un chemin entre t et s dans la forêt en construction $\}$.

```
insérer(C, a)
a = (s, t)
si C[s] == NIL et C[t] == NIL alors
  // l'arête (s, t) est nouvelle
  C[s] <- {s, t};
  C[t] <- C[s]; // partage
sinon si C[s] == NIL alors
  // t est déjà connu
  C[t] <- C[t] ∪ {s};
  C[s] <- C[t]; // partage
sinon si C[t] == NIL alors
  // s est déjà connu
  C[s] <- C[s] ∪ {t};
  C[t] <- C[s];
sinon // s et t sont déjà connus, on ne fait rien
```


Ex. On commence par wx:

$$C[w] = C[x] = \{w, x\}.$$

La deuxième arête est uv, ce qui crée

$$C[u] = C[v] = \{u, v\}.$$

Quand on veut insérer xv, on assiste à la fusion des composantes:

$$C[w] = C[x] = C[u] = C[v] = \{w, x, u, v\}.$$

Les deux dernières arêtes ne sont pas insérées, puisque les sommets sont déjà dans C.

Prop. Le coût de Kruskal est $O(m(\log m + \alpha(m)))$ avec $\alpha(m)$ l'inverse de la fonction d'Ackermann $Ack(m, m)$.

C) Correction des algorithmes de Prim et Kruskal

On construit T avec la liste $(a_1, a_2, \ldots, a_{n-1})$ telle que

$$val(a_1) \leqslant val(a_2) \leqslant \cdots \leqslant val(a_{n-1}).$$

Supposons qu'il existe un arbre couvrant S tq $\varpi(S) < \varpi(T)$.

Soit a une arête de poids minimum présente dans T mais pas dans S:

$$(a_1, a_2, \ldots, a_{r-1}, a_r = a, a_{r+1}, \ldots, a_{n-1})$$

avec $a_i \in S$ pour $i < r$.

$S + a$ contient un cycle (tous les sommets de $\mathcal{S}$ sont dans S).

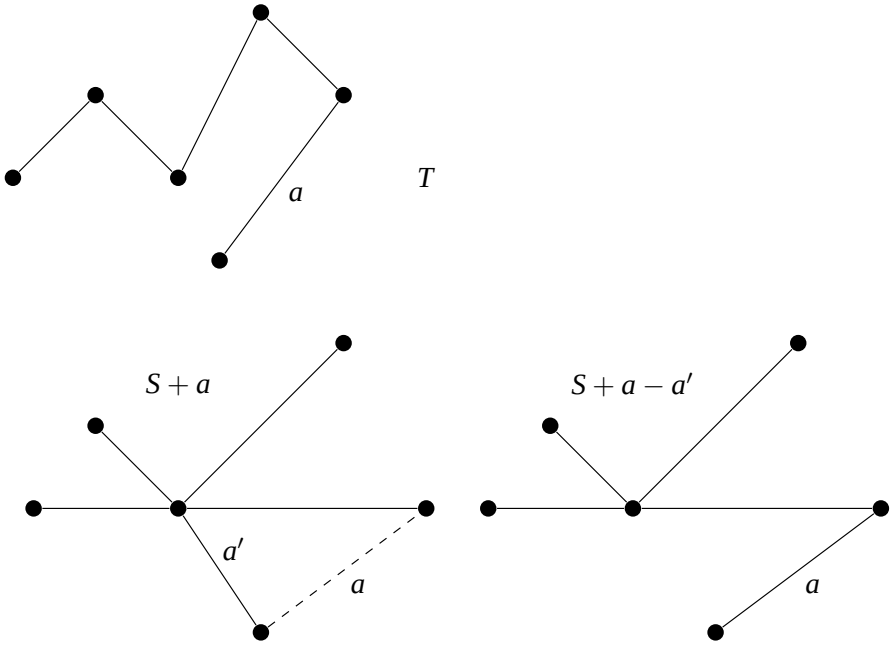
Ce cycle contient une autre arête a' non présente dans T: si toutes les arêtes étaient dans T, a n'aurait pu être choisi par l'algorithme (car aurait créé ce cycle).

$S' = S + a - a'$ est encore un arbre couvrant.

Si $val(a') < val(a)$: comme $a' \notin T$, il n'était pas éligible par l'algorithme, donc il y aurait un cycle dans $\{a_1, a_2, \ldots, a', \ldots, a_{r-1}\}$ et donc un cycle dans S.

D'où $\varpi(S') = \varpi(S) + val(a) - val(a') \leqslant \varpi(S) < \varpi(T)$.

De proche en proche, on peut donc passer de T à S, alors que le poids de T est plus grand que celui de S, contradiction.



IV. Exploration

Prototype: labyrinthe.

Principes:

- On parcourt une grille $n \times m$. On part du début $(0, 0)$ pour aller en $(n-1, m-1)$.
- À chaque case libre (i, j), on cherche la prochaine case libre en testant chaque direction. Si on en trouve une, on continue, si non, on retourne en arrière (backtrack).

Quelques exemples: n reines, jeux.

A) Sudoku

1				2	4	5		
		8						
	7	2	3				4	6
								9
			4			3		
	1			9	6			2
		6						
				3		1		8
				4		2		

Programme super naïf: si (i, j) est libre, on essaie toutes les possibilités et on avance.

Codage: `char[9][9] S`.

Test d'une possibilité

```
static boolean estPossible(char[][] S, int i,
                           int j, char c){
  // sur la ligne i ?
  for(int k = 0; k < m; k++)
    if(S[i][k] == c)
      return false;
  // sur la colonne j ?
  for(int k = 0; k < n; k++)
    if(S[k][j] == c)
      return false;
  // dans le carré ?
  int ci = i/3, cj = j/3;
  for(int l = 0; l < 3; l++)
    for(int k = 0; k < 3; k++)
      if(S[3*ci+l][3*cj+k] == c)
        return false;
  return true; // tout va bien
}
```

Fonction principale

```
static boolean resoudre(char[][] S, int i, int j){
   if(j == m){ // on a fini une ligne
     i++;
     if(i == n){ // fini!
       afficher(S);
       return true;
     }
     else // on commence la nouvelle ligne
       j = 0;
  }
  if(S[i][j] != '-') // déjà occupé
    return resoudre(S, i, j+1);
  for(int k = 0; k < n; k++){
    // on teste toutes les possibilités
    char c = chiffres[k];
    if(estPossible(S, i, j, c)){
      char[][] T = copier(S);
      T[i][j] = c;
      if(resoudre(T, i, j+1))
        return true;
    } }
  return false;
}
```


Quelques idées pour aller plus vite

- Maintenir dynamiquement une liste de lignes/colonnes/carrés triée par ordre décroissant de remplissage.
- Cf. aussi le projet de F. Pottier.

B) Battons Kasparov

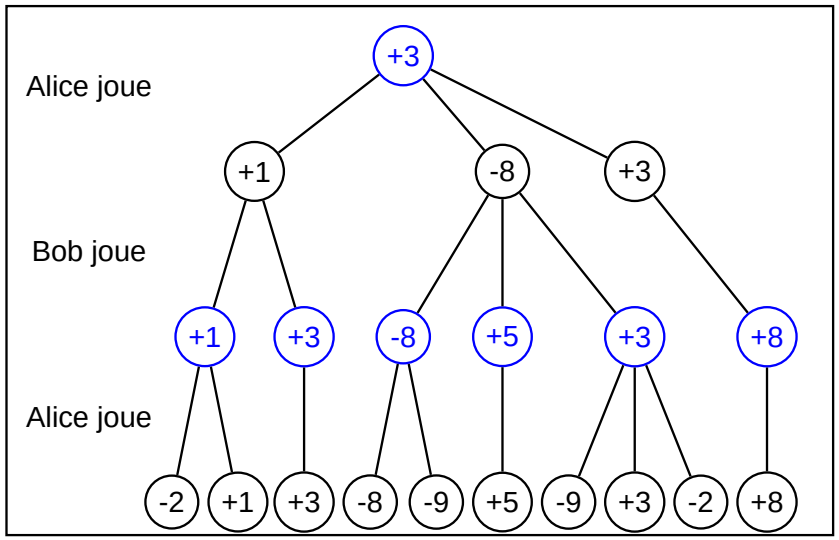
Comment faire pour gagner aux échecs :

- Imiter les grands-maîtres : programmer l'ordinateur pour lui faire utiliser la démarche humaine (à la Botvinik).
- Utiliser les capacités d'un ordinateur : un ordinateur peut examiner de nombreuses données en un temps court.

Principes généraux d'un programme de jeu :

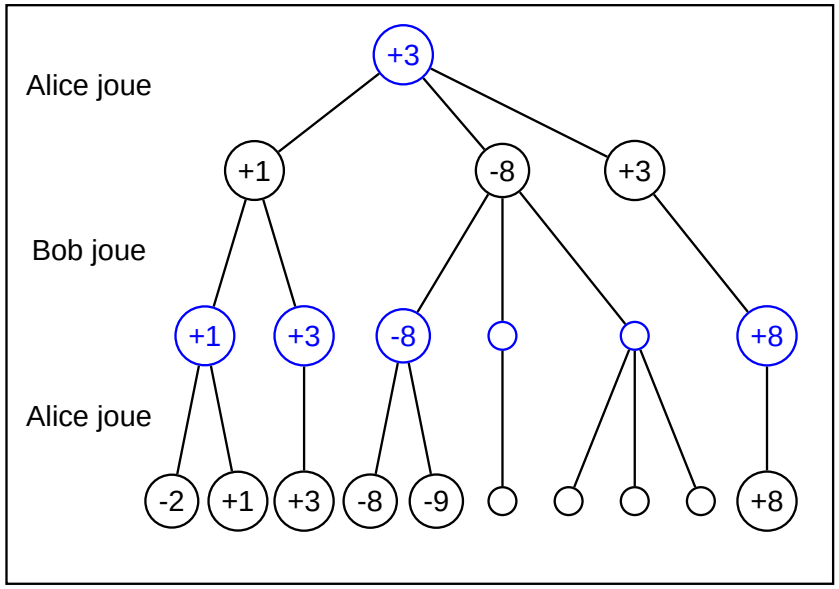
- À partir d'une position donnée, on énumère les coups valides et on crée la liste des nouvelles positions.
- On estime la valeur de chaque nouvelle position.
- On garde le meilleur chemin.
- On itère jusqu'à la profondeur la plus grande possible.

Algorithme minimax



Complexité : $O(b^{\text{prof}})$.

Principe de l'élagage $\alpha\beta$



Codage d'une position

Une matrice 8×8 est une très mauvaise idée.
Il faut une structure de données astucieuse permettant le calcul rapide des nouvelles positions. Par exemple (thèse de J. C. Weill) :

- les cases sont numérotées de 0 à 63 ;
- les pièces sont numérotées de 0 à 11 : pion blanc $= 0$, cavalier blanc $= 1$, ..., pion noir $= 6$, ..., roi noir $= 11$.

On stocke la position dans le vecteur de bits

$$(c_1, c_2, \ldots, c_{768})$$

tq $c_{64i+j+1} = 1$ ssi la pièce i est sur la case j.
Stockage des positions : table de hachage la plus grande possible qui permet de reconnaître une position déjà vue ; incorpore une bibliothèque d'ouvertures.

Fonction d'évaluation

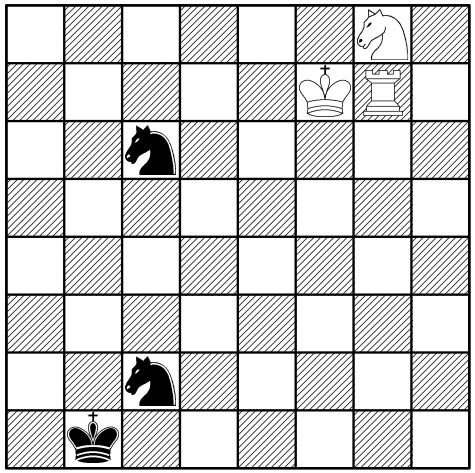
Le secret de tout bon programme !

Idée de base : évaluation de la force d'une position par une combinaison linéaire mettant en œuvre le **poids** d'une pièce (reine $= 900$, tour$= 500$, etc.).

Autres idées : coder la stratégie (position forte du roi, etc.).

Fins de partie

Idée : faire le catalogue exhaustif des finales avec peu de pièces (analyse rétrograde).



Il faut 243 coups à Blanc pour capturer une pièce sans danger, avant de gagner (Stiller, 1998) – Blanc joue.

Les matchs

Deep blue contre Kasparov (1997)
Projet démarré en 1989.
L'équipe : C. J. Tan, Murray Campbell (fonction d'évaluation), Feng-hsiung Hsu, A. Joseph Hoane, Jr., Jerry Brody, Joel Benjamin.
La machine :

- peut engendrer $200,000,000$ positions par seconde (ou 60×10^9 en 3 minutes, le temps alloué) ;
- C, AIX, IBM SP Parallel System (MPI) ;
- 32 nœuds avec des RS/6000 SP (chip P2SC) ; chaque nœud contient 8 processeurs spécialisés pour les échecs.

Deep Fritz contre Kramnik (2002): 8 processeurs à 2.4 GHz et 256 Mo, qui peuvent calculer 3 millions de coups à la seconde. Le programmeur F. Morsch a soigné la partie algorithmique. Kramnik ne fait que match nul (deux victoires chacun, quatre nulles), sans doute épuisé par la tension du match.

Résumé du cours

- Plus court(s) chemin(s): Dijkstra.
- Arbre couvrant de poids minimum: Prim, Kruskal.
- Exploration.

Prochains rendez-vous: TD cet après-midi; amphi 08 mercredi prochain 28/03.