Algorithmique et Programmation
Niveau 2

Frédéric Goualard

Laboratoire des Sciences du Numérique de Nantes, UMR CNRS 6004
Office #112-11

Algorithmes

Comment trier ses cartes ?

Notions de complexité des algorithmes

Comment trier ses cartes ?

Complexité temporelle

- Tri par insertion.

Notions de complexité des algorithmes

Comment trier ses cartes ?

Complexité temporelle

- Tri par insertion. $\mathcal{O}(n^2)$

Notions de complexité des algorithmes

Comment trier ses cartes ?

Complexité temporelle

- Tri par insertion. $\mathcal{O}(n^2)$
- Tri rapide.

Notions de complexité des algorithmes

Comment trier ses cartes ?

Complexité temporelle

- Tri par insertion. $\mathcal{O}(n^2)$
- Tri rapide. $\mathcal{O}(n \log n)$

Notions de complexité des algorithmes

Comment trier ses cartes ?

Complexité temporelle

- Tri par insertion. $\mathcal{O}(n^2)$
- Tri rapide. $\mathcal{O}(n \log n)$
- Tri comptage.

Notions de complexité des algorithmes

Comment trier ses cartes ?

Complexité temporelle

- Tri par insertion. $\mathcal{O}(n^2)$
- Tri rapide. $\mathcal{O}(n \log n)$
- Tri comptage. $\mathcal{O}(n)$

Détermination d'une fonction majorant le temps d'exécution ou l'occupation spatiale en fonction de la longueur des entrées

La complexité en pratique

Complexité	Exemple
$\mathcal{O}(1)$	Accès à une case de tableau
$\mathcal{O}(n)$	Traversée d'une liste chaînée
$\mathcal{O}(n^2)$	Tri par insertion
$\mathcal{O}(n^3)$	Multiplication naïve de matrices
$\mathcal{O}(n^5)$	—
$\mathcal{O}(2^n)$	Calcul récursif naïf de la suite de Fibonacci

Complexité temporelle	Taille n					
	10	20	30	40	50	60
$\mathcal{O}(1)$	1×10^{-6}s	1×10^{-6}s	1×10^{-6}s	1×10^{-6}s	1×10^{-6}s	1×10^{-6}s
$\mathcal{O}(n)$	1×10^{-5}s	2×10^{-5}s	3×10^{-5}s	4×10^{-5}s	5×10^{-5}s	6×10^{-5}s
$\mathcal{O}(n^2)$	1×10^{-4}s	2×10^{-4}s	3×10^{-4}s	4×10^{-4}s	5×10^{-4}s	6×10^{-4}s
$\mathcal{O}(n^3)$	1×10^{-3}s	2×10^{-3}s	3×10^{-3}s	4×10^{-3}s	5×10^{-3}s	6×10^{-3}s
$\mathcal{O}(n^5)$	0.1s	3.2s	24.3s	1.7min	5.2min	13min
$\mathcal{O}(2^n)$	1×10^{-3}s	1s	17.9min	12.7 jours	35.7années	366 siècles

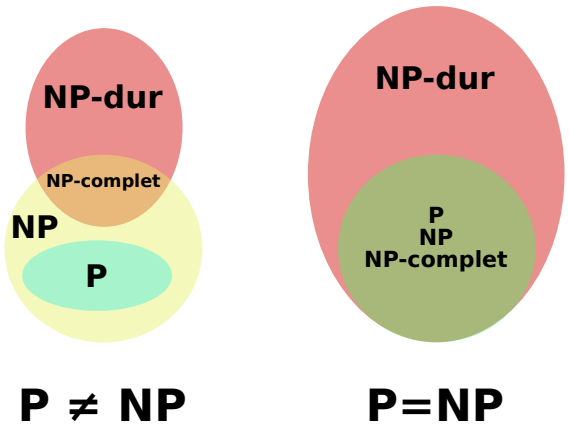
D'après : *Computers and Intractability, A Guide to the Theory of NP-Completeness*, M.R. Garey et D.S. Johnson, W.H. Freeman & Company, 1979.

Attention : la complexité théorique ne préjuge pas des performances sur des problèmes réels :

- Méthode du simplexe de complexité exponentielle
- Multiplication de matrices (n^3 pour l'algorithme naïf et $n^{2.37}$ pour l'algorithme de Coppersmith-Winograd—mais gros coefficient constant)

$P = NP$?

- **P** : problème de décision (oui/non) dont on peut trouver la solution avec une complexité polynomiale (une chaîne est-elle un palindrome ?)
- **FP** : problème non de décision dont on peut trouver la solution avec une complexité polynomiale (multiplication de deux entiers)
- **NP** : problème de décision dont on peut vérifier la réponse positive avec une complexité polynomiale *sur une machine déterministe* — une solution positive peut être trouvée en temps polynomial *sur une machine non déterministe* (existence d'un chemin hamiltonien)
- **NP-complet** : problème de décision en lequel tous les problèmes NP peuvent être recodés en temps polynomial
- **NP-dur** : problème dont on peut trouver la solution avec une complexité polynomiale *sur une machine non déterministe* — pas nécessairement un problème de décision (TSP)

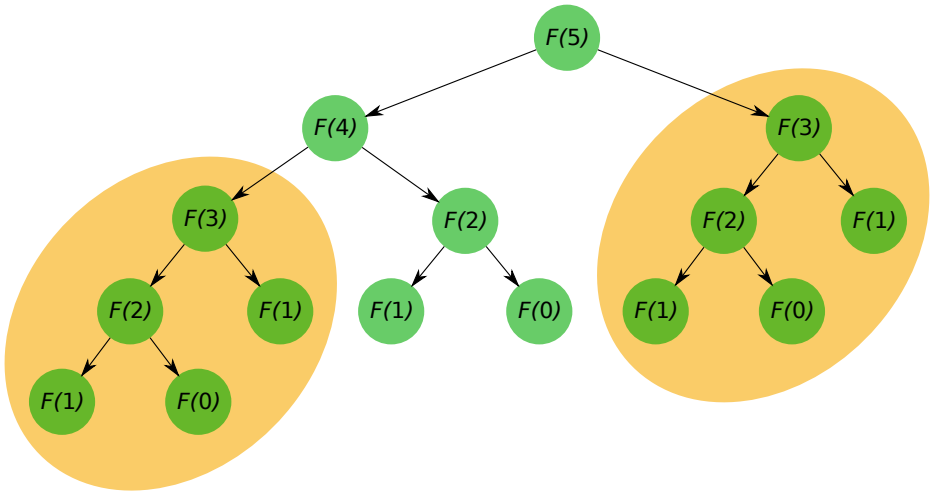


Tractabilité/Intractabilité

- Faire la différence entre la complexité d'un problème et la complexité d'un algorithme particulier pour résoudre ce problème :

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

```
uint32_t F(uint32_t n)
{
  switch (n) {
  case 0: return 0;
  case 1: return 1;
  default: return F(n-1) + F(n-2);
  }
}
```



Complexité temporelle : $\mathcal{O}(2^n)$
Complexité spatiale : $\mathcal{O}(n)$

Tractabilité/Intractabilité

- Faire la différence entre la complexité d'un problème et la complexité d'un algorithme particulier pour résoudre ce problème :

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

```
uint32_t F(uint32_t n)
{
  if (n < 2) {
        return n;
  } else {
        uint32_t F2 = 0;
        uint32_t F1 = 1;
        uint32_t fib;
        for (uint32_t i = 2; i <= n; ++i) {
              fib = F1 + F2;
              F2 = F1;
              F1 = fib;
        }
        return fib;
  }
}
```

Complexité temporelle : $\mathcal{O}(n)$
Complexité spatiale : $\mathcal{O}(1)$

Tractabilité/Intractabilité

- Faire la différence entre la complexité d'un problème et la complexité d'un algorithme particulier pour résoudre ce problème :

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

```
uint32_t F(uint32_t n)
{
  const double phi = (1+sqrt(5))/2; // Nombre d'or
  return (uint32_t)round(pow(phi,n)/sqrt(5));
}
```

Complexité temporelle : $\mathcal{O}(\log n)$
Complexité spatiale : $\mathcal{O}(1)$

Tractabilité/Intractabilité

- Faire la différence entre la complexité d'un problème et la complexité d'un algorithme particulier pour résoudre ce problème :

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

```
uint32_t F(uint32_t n)
{
  const array<uint32_t,17>  FT {
            0,   1,   1,   2,  3,   5, 8,
                 13,  21,  34,  55, 89, 144,
                 233, 377, 610, 987
  };
  return FT[n];
}
```

Complexité temporelle : $\mathcal{O}(1)$
Complexité spatiale : $\mathcal{O}(n)$

Mais :

- $F(48) > 2^{32}$
- $F(94) > 2^{64}$

Donc l'espace mémoire est borné en pratique et peut être considéré en $\mathcal{O}(1)$!

Tractabilité/Intractabilité

- Faire la différence entre la complexité d'un problème et la complexité d'un algorithme particulier pour résoudre ce problème :

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

```
uint32_t F(uint32_t n)
{
  const array<uint32_t,17>  FT {
            0,   1,   1,   2,  3,   5, 8,
                 13,  21,  34,  55, 89, 144,
                 233, 377, 610, 987
  };
  return FT[n];
}
```

Complexité temporelle : $\mathcal{O}(1)$
Complexité spatiale : $\mathcal{O}(n)$

Mais :

- $F(48) > 2^{32}$
- $F(94) > 2^{64}$

Donc l'espace mémoire est borné en pratique et peut être considéré en $\mathcal{O}(1)$!

Le calcul de $F(n)$ est un problème dans **FP**.

Choisir un algorithme

Que faire si un problème n'est pas dans **P** (ou dans **P** mais avec un algorithme coûteux) ?

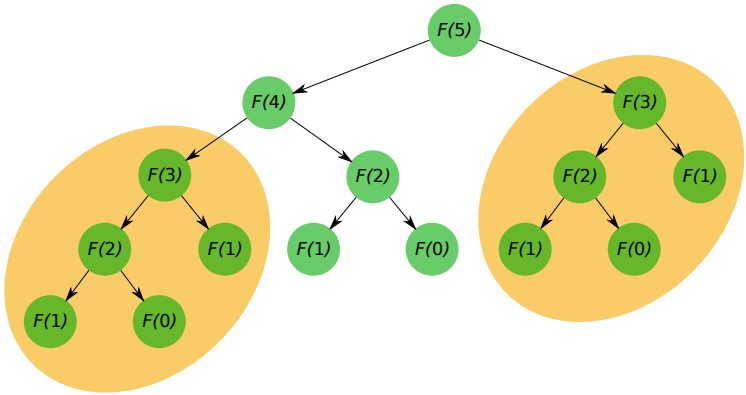
- Utiliser un algorithme exact si les instances sont petites
- Utiliser un algorithm approché (« heuristique ») offrant un résultat proche de la solution (ou la solution avec une certaine probabilité)
- Adapter l'algorithme aux cas particuliers d'instances considérées

Exemple : Traveling Salesman Problem (TSP) — circuit hamiltonien de coût minimum

- Algorithme de Held-Karp ($\mathcal{O}(n^2 2^n)$)
  Programmation dynamique (découpage du problème en plus petits problèmes)
- *Algorithme glouton* du plus proche voisin ($\mathcal{O}(n^2)$)
- *Branch and Bound*

Programmation dynamique

Calcul de Fibonacci :



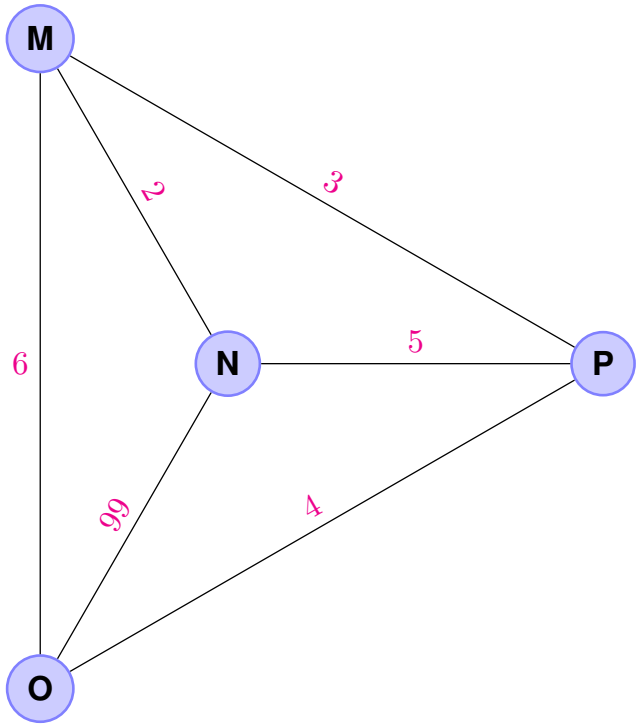

```
uint32_t F_aux(uint32_t n, vector<uint32_t>& FT);

const uint32_t UNUSED = 4; // F(n) != 4 \forall n
uint32_t F(uint32_t n)
{ // Condition: n >= 1
  vector<uint32_t> FT(n+1,UNUSED);
  FT[0] = 0;
  FT[1] = 1;
  return F_aux(n,FT);
}

uint32_t F_aux(uint32_t n, vector<uint32_t>& FT)
{
  if (FT[n] != UNUSED) {
        return FT[n];
  } else {
        FT[n] = F_aux(n-1,FT) + F_aux(n-2,FT);
        return FT[n];
  }
}
```

- Découpage du problème en sous-problème
- Mémorisation du travail déjà effectué pour le réutiliser

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :

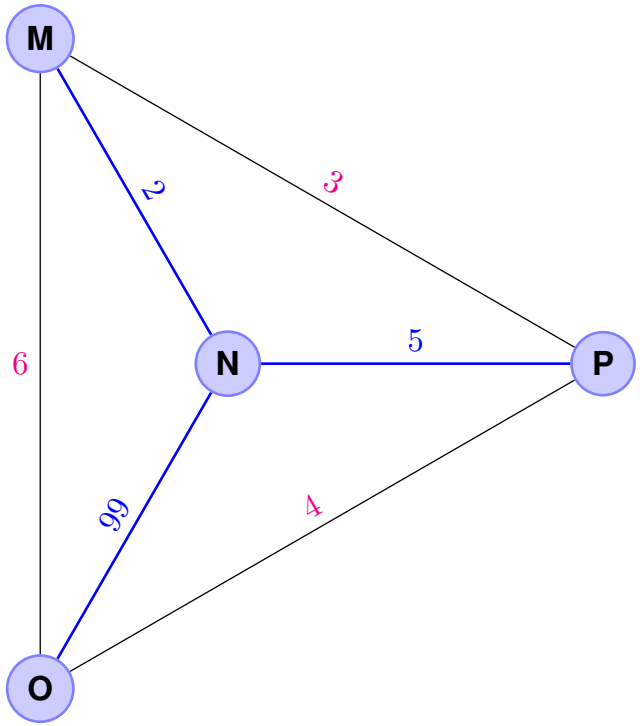


D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :

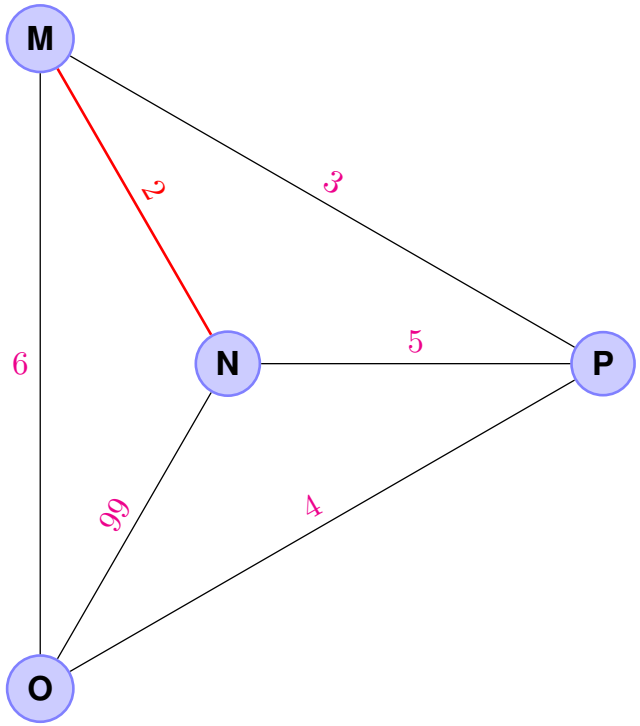


D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :

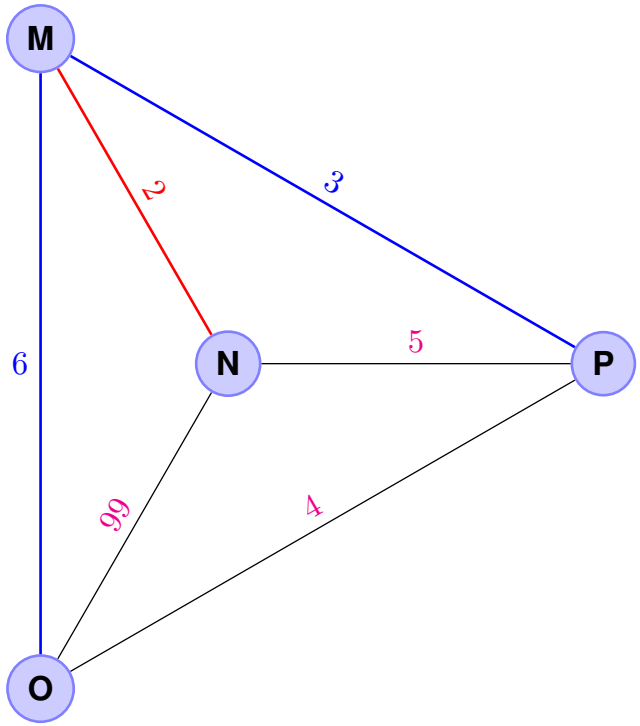


D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :

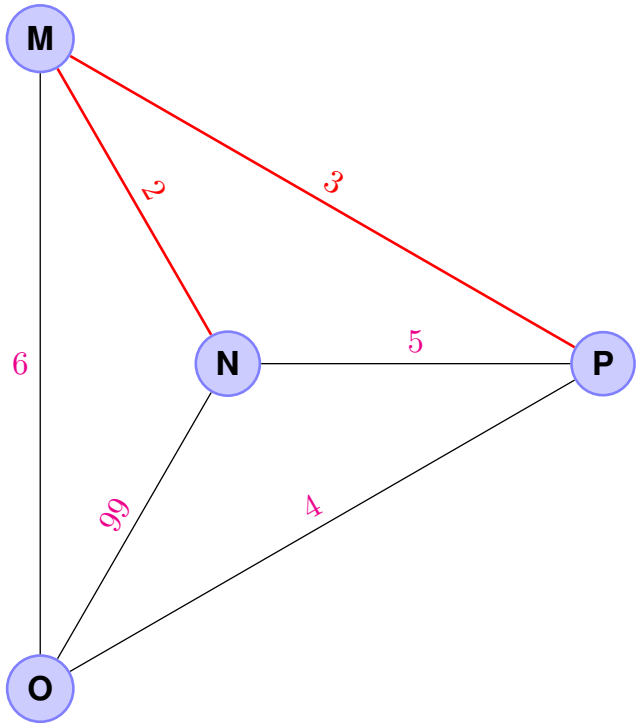


D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :

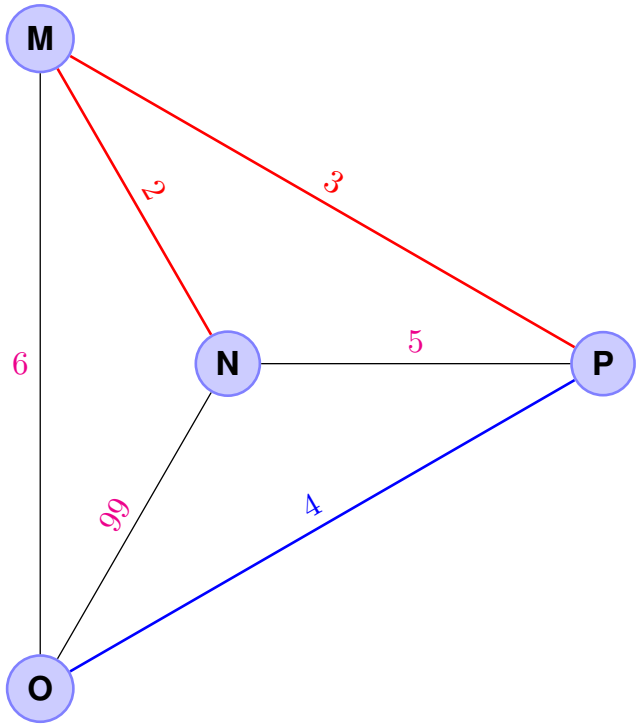


D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :

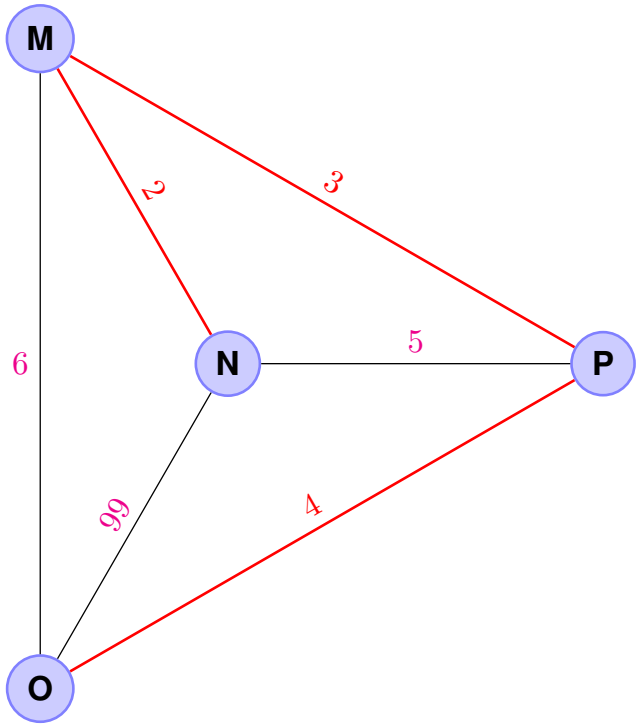


D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :

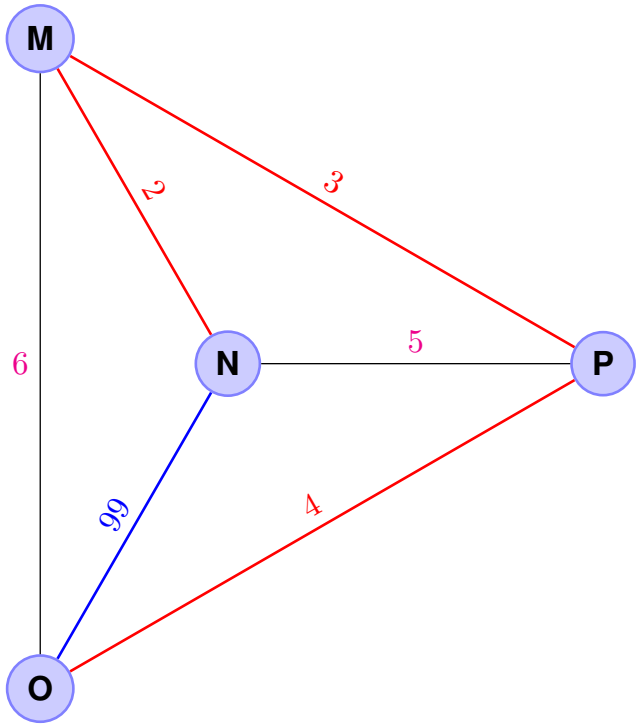


D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :

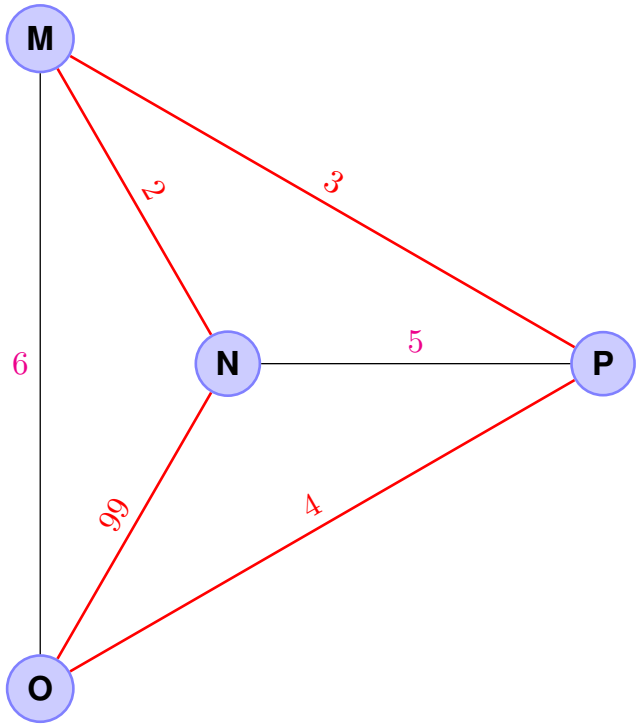


D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :



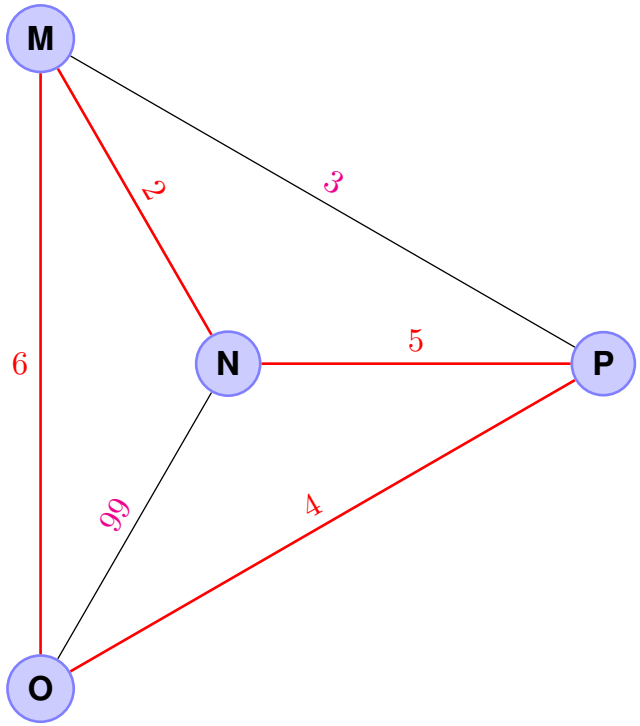
D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

- Coût : $2 + 3 + 4 + 99 = 108$

Algorithmes gloutons

- Résolution d'un problème en faisant le meilleur choix local à chaque pas
- Choix d'une suite d'optimums locaux pour atteindre l'optimum global (pas garanti)

TSP (cycle hamiltonien de coût minimal) :



D'après http://www.cim.mcgill.ca/~sveta/COMP102/Lecture17.pdf

- Coût : $2 + 3 + 4 + 99 = 108$
- **Mais** coût minimal : $2 + 6 + 4 + 5 = 17$!

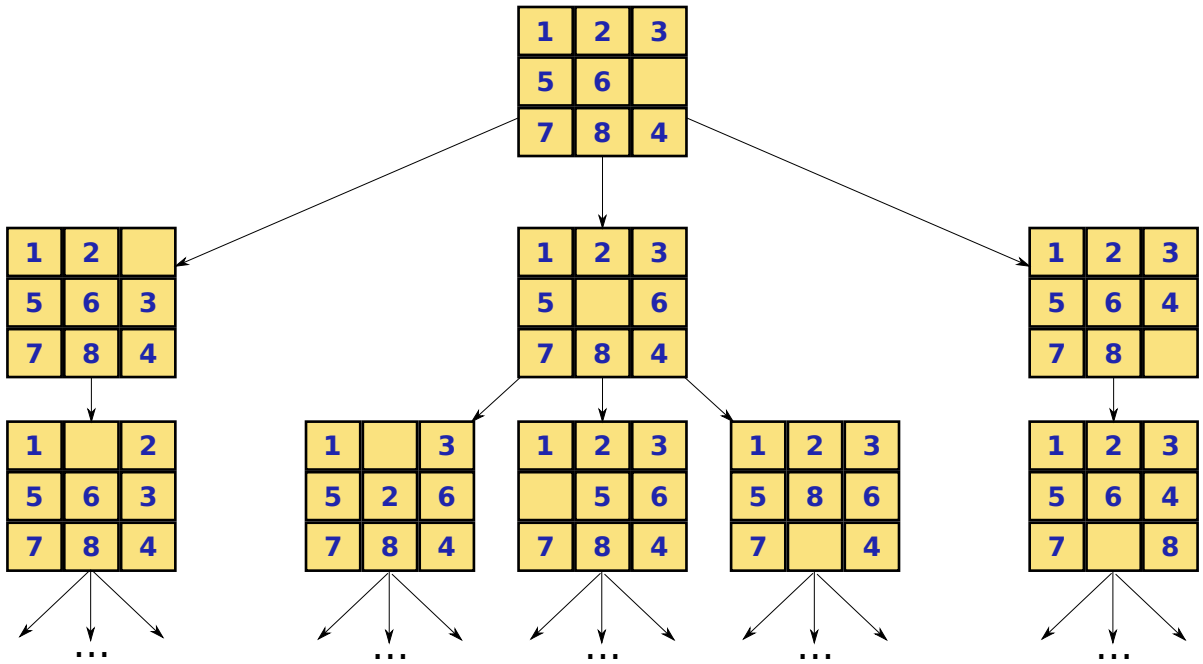
Branch-and-bound

D'après : https://www.geeksforgeeks.org/branch-bound-set-3-8-puzzle-problem/

Branch-and-bound

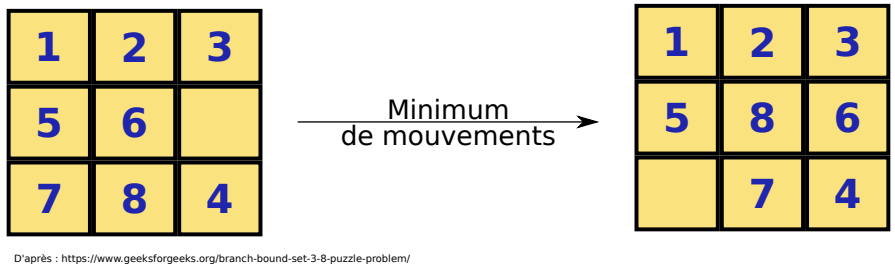
D'après : https://www.geeksforgeeks.org/branch-bound-set-3-8-puzzle-problem/

brute force



D'après : https://courses.cs.washington.edu/courses/cse473/12au/slides/lect3.pdf

Branch-and-bound



D'après : https://www.geeksforgeeks.org/branch-bound-set-3-8-puzzle-problem/

Branch and bound

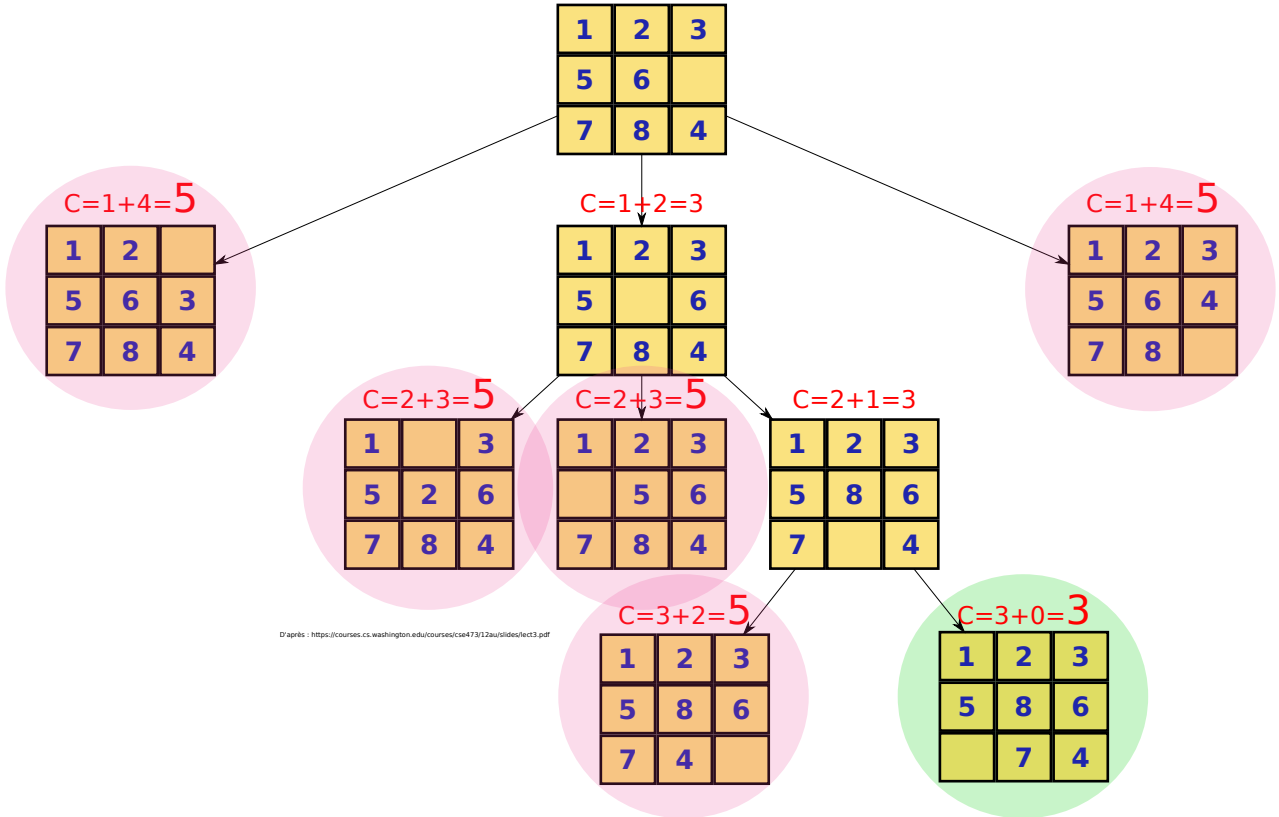
- Déterminer une fonction de coût donnant une borne inférieure du nombre de coups nécessaires

$$C(\text{nœud}) = \text{profondeur} + \text{nbre. d'éléments pas en place}$$

- Explorer l'arbre de recherche en privilégiant le noeud avec C minimum
- Sauver C lorsque l'on arrive à une feuille solution $\rightarrow C_S$
- Ne pas explorer une branche si $C > C_S$

Branch-and-bound

D'après : https://www.geeksforgeeks.org/branch-bound-set-3-8-puzzle-problem/



D'après : https://courses.cs.washington.edu/courses/cse473/12au/slides/lect3.pdf

Fin du cours