Algorithmes de recherche de chemins les plus courts

Dans ce petit rapport je listerai les algorithmes de « Chemin le plus court », ainsi que des comparaisons entre eux pour les prochaines fois que j'ai à en choisir un.

Certains algorithmes sont plus limités que d'autres (chemins entre deux points donnés < chemin entre un point donné et les autres < chemin entre tous les points).

Chemin entre deux points donnés 2

A* 2

Jump Point Search (JPS et JPS+) 3

Chemin entre un point donné et les autres 4

Djikstra 4

Bellman-Ford 5

Shortest Path Faster Algorithm (SPFA) 6

Chemin entre tous les points 8

Floyd-Warshall 8

Johnson 9

Résumé 10

Expériences 10

Conclusion 11

Chemin entre deux points donnés

A*

P. E. Hart, N. J. Nilsson et B. Raphael, « A Formal Basis for the Heuristic Determination of Minimum Cost Paths », IEEE Transactions on Systems Science and Cybernetics SSC4, vol. 4, no 2, 1968, p. 100–107 (DOI 10.1109/TSSC.1968.300136)

Algo très rapide grâce à l'utilisation d'heuristique, complexité $O(|E|)$ en temps et en mémoire.

Intuition

L'algorithme effectue une recherche en profondeur sur le graphe, en choisissant toujours le nœud ayant la meilleure heuristique. Si on marche dans Paris dans le but d'atteindre la Tour Eiffel, et qu'on peut la voir au loin, on emprunte les rues qui vont en direction de cette Tour. Cela nous amène parfois dans un cul de sac, mais c'est généralement une bonne stratégie.

Pseudo-code

```
Structure nœud = {
      index : Entier
      cout, heuristique: Nombre
   }

depart = Nœud(x=_, y=_, cout=0, heuristique=0)

Fonction compareParHeuristique(n1:Nœud, n2:Nœud)
       si n1.heuristique < n2.heuristique
           retourner 1
       ou si n1.heuristique == n2.heuristique
           retourner 0
       sinon
           retourner -1

Fonction cheminPlusCourt(g:Graphe, objectif:Nœud, depart:Nœud)
       closedList = File()
       openList = FilePrioritaire(comparateur = compareParHeuristique)
       openList.ajouter(depart)
       tant que openList n'est pas vide
           u = openList.defiler()
           si u == objectif
               reconstituerChemin(u)
               terminer le programme
           pour chaque voisin v de u dans g
               si non(   v existe dans closedList
                            ou v existe dans openList avec un coût inférieur)
                    v.cout = u.cout + 1
                    v.heuristique = v.cout + distance(v, objectif)
                    openList.ajouter(v)
           closedList.ajouter(u)
       terminer le programme (avec erreur)
```

Variantes

- WA* (Weighted A*) : Judea Pearl, Heuristics : Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 1984, p.382 (ISBN 978-0-201-05594-8)
- Dynamic Weighting: Ira Pohl, The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving (1973) Proceedings of the Third International Joint Conference on Artificial Intelligence (IJCAI-73) 3: p.11–17
- Samples Dynamic Weighting : Andreas Köll (1992). « A new approach to dynamic weighting » Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92): p.16–17
- AlphaA* : Bjørn Reese, « AlphA*: An ε-admissible heuristic search algorithm », Systematic Software Engineering A/S, 1999

Jump Point Search (JPS et JPS+)

Harabor, D. D., & Grastien, A. (2012, July). The JPS Pathfinding System. In SOCS.

Algorithme adapté aux grilles à poids uniformes, dérivé de A*.

Intuition

Un chemin, généralement vu comme la suite des nœuds qui relient le début à la fin, est ici remplacé par une suite de vecteurs $\vec{d}$ de longueur 1 si le déplacement est horizontal/vertical et $\sqrt{2}$ si le déplacement est en diagonal. Cet algorithme privilégie de prendre des diagonales dès que possible, car si deux chemins sont semblables, celui ayant fait une diagonale en premier est conservé.

On visite les points itérativement en utilisant ce vecteur $\vec{d}$.

L'algorithme se décrit avec 2 nouvelles opérations : l'élagage (pruning) et le saut (jumping).

Si nous sommes au point P et nous déplaçons vers le point voisin X. Si le déplacement est horizontal/vertical, on sait qu'aucun voisins de X ne pourra réduire ce chemin. On peut alors élaguer notre champ de recherche.

To be continued…

Chemin entre un point donné et les autres

Djikstra

Dijkstra, E. W., « A note on two problems in connexion with graphs », Numerische Mathematik, vol. 1, 1959, p. 269–271 (DOI 10.1007/BF01386390.).

Possiblement l'algorithme ayant la meilleure complexité dans le domaine, $O(|E| + |V| \log|V|)$.

Il est néanmoins limité par des graphes dont les poids sont positifs.

L'algorithme est basé sur un parcours en largeur du graphe (BFS, Breadth First Search).

Intuition

On cherche à se déplacer de la ville A à la ville Z sans avoir aucune idée de la localisation de Z. Dans la ville A, vous voyez des panneaux de signalisation vers les villes alentour (ville B et ville C) avec leur distance en kilomètres ($d(A,B) = 1km$ et $d(A,C) = 8km$). Vous partez vers la ville B car elle est plus proche de vous, et si vous devez rentrer vite, vous savez que c'est le plus rapide. Depuis la ville B, vous avez à nouveau des panneaux pour les villes C et D et leur distances respectives $d(B,C) = 4km$ et $d(B,D) = 3km$. On peut calculer la nouvelle distance de A à C : $d(A,C) = d(A,B) + d(B,C) = 5km$ car elle est plus petite qu'avant. On note aussi que la prochaine fois qu'on est dans la ville C, on a meilleur temps de passer par B pour rentrer. On ne connait pas encore la ville de C, mais on va plutôt aller à la ville D car elle est plus courte si on doit rentrer rapidement ($d(A,D) = d(A,B) + d(B,D) = 4km$. Et ainsi de suite jusqu'à arriver à la ville Z si c'est le but. Si on voulait juste se promener, on continue jusqu'à visiter toutes les villes. On connait maintenant toutes les distances depuis A, mais aussi, on a établi les chemins à prendre pour y arriver !

Pseudo-code

```
Initialisation(G,sdeb)
  pour chaque point s de G faire
     d[s] := infini /* on initialise les sommets autres que sdeb à infini */5
  fin pour
  d[sdeb] := 0      /* la distance au sommet de départ sdeb est nulle */

Trouve_min(Q)
  mini := infini
  sommet := -1
  pour chaque sommet s de Q
     si d[s] < mini
     alors
         mini := d[s]
         sommet := s
  renvoyer sommet

maj_distances(s1,s2)
  si d[s2] > d[s1] + Poids(s1,s2)      /* Si la distance de sdeb à s2 est plus grande que */
                                       /* celle de sdeb à S1 plus celle de S1 à S2 */
     alors
         d[s2] := d[s1] + Poids(s1,s2) /* On prend ce nouveau chemin qui est plus court */
         prédécesseur[s2] := s1        /* En notant par où on passe */

Dijkstra(G,Poids,sdeb)
  Initialisation(G,sdeb)
  Q := ensemble de tous les nœuds
  tant que Q n'est pas un ensemble vide faire
        s1 := Trouve_min(Q)
        Q := Q privé de s1
        pour chaque nœud s2 voisin de s1 faire
            maj_distances(s1,s2)
        fin pour
  fin tant que

// Si on avait défini un point de fin (ville Z)
  A := suite vide
  s := sfin
  tant que s != sdeb faire
    A := cons(s, A)            /* on ajoute s en tête de la liste A */
```

```
    s := prédécesseur[s]       /* on continue de suivre le chemin */
  fin tant que
  A := cons(sdeb, A)           /* on ajoute le nœud de départ */
```

Améliorations possibles

Il existe plusieurs articles décrivant une amélioration de l'algorithme, notamment adaptés aux graphes à faible densité.

Huang, Y., Yi, Q., & Shi, M. (2013, March). An improved Dijkstra shortest path algorithm. In Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (pp. 226-229). Atlantis Press.

Xu, M. H., Liu, Y. Q., Huang, Q. L., Zhang, Y. X., & Luan, G. F. (2007). An improved Dijkstra's shortest path algorithm for sparse network. Applied Mathematics and Computation, 185(1), 247-254.

Bellman-Ford

Edward F. Moore (1959). « The shortest path through a maze » dans Proc. Internat. Sympos. Switching Theory 1957, Part II : 285–292 p., Cambridge, Mass.: Harvard Univ. Press.

Possiblement un peu plus long que l'algorithme de Djikstra, mais pouvant être utilisé sur des graphes ayant des poids négatifs. Permet aussi la détection de circuits absorbants (circuit entre 2 nœuds ou plus dont le poids cumulé est négatif). Complexité $O(|V||E|)$ mais borné à $\theta(|E|)$.

Le pseudo-code est vraiment rapide.

Intuition

L'algorithme fonctionne par récurrence. On a un nœud source s.

On se base sur quelques formules simples :

- La distance de la source s au point t sachant qu'on peut passer au plus par k nœuds est noté $d[t,k]$.
- On peut dire $d[t,0] = +\infty$ (sauf si $t = s$, car ce cas $d[s,0] = 0$)
- Par récurrence, $d[t,k] = \min\left(d[t,k-1]; d[u,k-1] + dist(u,t)\right)$. La distance de la source s à t en passant par k nœuds est soit la même qu'en passant par $k-1$ nœuds, soit la distance de s à u plus la distance de u à t.

Pseudo-code

```
fonction Bellman-Ford(G = (S, A), poids, s)
   pour u dans S faire
      d[u] = +∞
      pred[u] = null
   d[s] = 1
   //Boucle principale
   pour k = 1 jusqu'à taille(S) - 1 faire
      pour chaque arc (u, v) du graphe faire
         si d[v] > d[u] + poids(u, v) alors
            d[v] := d[u] + poids(u, v)
            pred[v]:= u
   retourner d, pred
```

On a la carte de distance d ainsi que la carte de précédence $pred$, qui nous permet de reconstituer le chemin de la source s à n'importe quel point.

Pour détecter un circuit absorbant, on a juste à vérifier

```
   pour chaque arc (u, v) du graphe faire
      si d[v] > d[u] + poids(u, v) alors
      afficher "il existe un cycle absorbant"
```

Améliorations possibles

La variable k représente le nombre le nombre d'arcs maximums d'un chemin à prendre en compte pour calculer les chemins les plus courts. Si les distances ne sont pas modifiées en prenant en compte des chemins de k arcs, on sait que les distances ne changeront pas non plus en prenant $k+1$. On peut alors interrompre l'algorithme immédiatement.

```
fonction Bellman-Ford(G = (S, A), poids, s)
   pour u dans S faire
      d[u] = +∞
      pred[u] = null
   d[s] = 1
   //Boucle principale
   pour k = 1 jusqu'à taille(S) - 1 faire
      changement_fait = faux
      pour chaque arc (u, v) du graphe faire
         si d[v] > d[u] + poids(u, v) alors
            d[v] := d[u] + poids(u, v)
            pred[v]:= u
            changement_fait = vrai
      si pas de changement_fait
         quitter la boucle
   retourner d, pred
```

Shortest Path Faster Algorithm (SPFA)

Moore, Edward F. (1959). "The shortest path through a maze". Proceedings of the International Symposium on the Theory of Switching. Harvard University Press. pp. 285–292.

Une amélioration de l'algorithme de Bellman-Ford, mais utilise une liste de nœuds utiles à la réduction des distances calculées. La complexité reste $O(|V||E|)$. Il est néanmoins estimé que le temps d'exécution moyen est de $O(|E|)$.

Intuition

A remplir…

Pseudo-code

```
Fonction ShortestPathFasterAlgorithm (G = (S, A), poids, s)
   Pour u dans S faire
      d[u] := +∞
      prec[u] := -1
   d[s] := 0
   Inserer s dans Q
   Tant que Q n'est pas vide faire
      u := Q[0]
      Q := Q privé de u
      Pour tout arc (u, v) dans A faire
         Si d[u] + w(u, v) < d[v] alors
            d[v] := d[u] + w(u, v)
            prec[v] := u
            Si v n'est pas dans Q alors
               Ajouter v à la fin de Q
   Retourner d et prec
```

Améliorations possibles

Si on connait le type de graphe que nous utilisons, il est possible d'utiliser la liste (FIFO) Q comme une liste de priorités. A vrai dire, je ne saurais pas dire quand il est utile d'appliquer ces améliorations, mais je les insère tout de même ici :

- La technique SLF (Small Label First) : On met la priorité de nos calculs dans les nœuds les plus proches du nœuds source. Au lieu d'ajouter v à la fin de Q, on peut le placer au début si $d[Q[0]] < d[v]$. Le pseudo code serait alors

```
Fonction SLF(Q, d)
   Si d[last(Q)] < d[first(Q)] alors
      u := pop du dernier element de Q
      Inserer Q au début de Q
```

- La technique LLL (Large Label Last) est relativement identique à SLF dans l'idée qu'on manipule l'ordre des éléments de Q. Ici, il est demandé que le premier élément de Q doit être inférieur à la moyenne des distances des nœuds dans Q. Tout élément dont la distance est supérieure à la moyenne est placé à la fin de la liste. Le pseudo code :

```
Fonction LLL(Q, d)
   x := moyenne de d(v) pour tout v dans Q
   Tant que d[first(Q)] > x faire
      u := pop du premier élément de Q
      Inserer u à la fin de Q
```

Le pseudo code de l'algorithme peut alors être réécrit :

```
Fonction ShortestPathFasterAlgorithm (G = (S, A), poids, s)
   Pour u dans S faire
      d[u] := +∞
      prec[u] := -1
   d[s] := 0
   Inserer s dans Q
   Tant que Q n'est pas vide faire
      u := Q[0]
      Q := Q privé de u
      Pour tout arc (u, v) dans A faire
         Si d[u] + w(u, v) < d[v] alors
            d[v] := d[u] + w(u, v)
            prec[v] := u
            Si v n'est pas dans Q alors
               Ajouter v à la fin de Q
      SLF(Q, d) si on applique cette méthode
      LLL(Q, d) si on applique cette méthode
   Retourner d et prec
```

Chemin entre tous les points

Floyd-Warshall

Bernard Roy, « Transitivité et connexité. », C. R. Acad. Sci. Paris, vol. 249, 1959, p. 216–218

Cet algorithme fonctionne sur les graphes orientés à poids négatifs tant qu'il n'existe pas de circuit absorbant. Complexité $\Theta(|V|^3)$.

Intuition

A l'instar de Bellman-Ford, l'algorithme se base sur un principe de récurrence :

Je vais noter $d_k(i,j)$ la distance entre i et j calculée à l'étape k.

Si on considère le graphe G_1 aux sommets $S_1 = \{1\}$, on calcule la distance $d_1(1,1) = 0$. Avec le graphe G_2 avec $S_2 = \{1,2\}$, les distances avec le nœud 2 sont $d_2(1,2)$, $d_2(2,1)$ et $d_2(2,2)$, que l'on peut calculer facilement. On a aussi $d_2(1,1) = d_1(1,1)$.

Maintenant considérons G_3 avec $S_3 = \{1,2,3\}$, la distance $d_3(1,3) = \min\big(d_2(1,2) + cout(2,3); cout(1,3)\big)$, ainsi que $d_3(2,3) = \min\big(d_2(2,1) + cout(1,3); cout(2,3)\big)$. Mais aussi, $d_3(1,2) = \min\big(d_2(1,2); d_3(1,3) + cout(3,2)\big)$.

On peut montrer par récurrence que $d_k(i,j) = \min\big(d_{k-1}(i,j); d_{k-1}(i,k) + d_{k-1}(k,j)\big)$.

(Très mauvaise intuition, à refaire plus tard…)

Pseudo-code

```
Function FloydWarshall(G)
  W := matrice d'adjacence de G de taille N x N
  Pour k allant de 0 à N :
     Pour i allant de 0 à N :
        Pour j allant de 0 à N :
           W[i, j] = min(W[i, j] ; W[i, k] + W[k, j])
  Retourner W
```

Améliorations possibles

Toroslu, I. H. (2021). Improving The Floyd-Warshall All Pairs Shortest Paths Algorithm. arXiv preprint arXiv:2109.01872.

On peut réduire le nombre de relaxations car beaucoup d'arcs ne sont pas modifiés durant la boucle principale de l'algorithme.

Dans le peu d'essais réalisés, cette amélioration divise le temps d'exécution par 10.

```
Fonction FloydWarshallAmeliore(W : Matrice d'adjacence)
   Pour i allant de 0 à N
      Pour j allant de 0 à N
         prec[i, j] := -1
         Si i != j et W[i, j] != ∞ alors
            Ajouter j à out[i]
            Ajouter i à in[j]
   tous_k ← {1, 2, 3, …, N}
   Tant que tous_k n'est pas vide faire
      min_k := ∞
      Pour tous k dans tous_k faire
          min_k = min{min_k; in[k] * out]}
      Retirer min_k de tous_k
      Pour i dans in[k] faire
         Pour j dans out[k] faire
            Si W[i, j] > W[i, k] + W[k, j] alors
               W[i, j] := W[i, k] + W[k, j]
               prec[i, j] := k
               Si W[i, j] = ∞ alors
                  Ajouter j à out[i]
                  Ajouter i à in[j]
   Retourner W
```

Johnson

Donald B. Johnson, « Efficient algorithms for shortest paths in sparse networks », Journal of the ACM, vol. 24, no 1, 1977, p. 1–13 (DOI 10.1145/321992.321993).

Apparemment plus adapté aux graphes de faible densité ($D = \frac{2|E|}{|V|\cdot(|V|-1)}$ pour le calcul de densité), la complexité est de $O(|V|^2 \log|V| + |V||E|)$.

Intuition

On peut expliquer la complexité par le principe simple de cet algorithme qui se base sur Djikstra, mais qui retire le problème de la contrainte des poids négatifs en appliquant un premier algorithme de Bellman-Ford. Cela permet aussi d'interrompre le calcul s'il existe un cycle absorbant.

Pseudo-code

```
Entrée : G un graphe pondéré sans cycle de poids négatif
Sortie : les distances entre toutes paires de sommets

Johnson(G)
          G1 := G où on a ajouté un sommet q et des arcs entre q et tout sommet de G
          h[.] := Bellman-Ford(G1, q)
          G2 := G où pour chaque arc (s, t), le poids de (s, t) est poids(s, t) + h(s) - h(t)
où poids(s, t) est le poids de (s, t) dans G
          pour tout sommet s de G
                    d[s, .] := Dijkstra(G, s)
          pour tout s, t sommets de G
                    d[s, t] := d[s, t] - (  h(s) - h(t) )
          retourner d
```

Résumé

Algorithme	$s \to d$	$s \to \forall d$	$\forall s \to \forall d$	Complexité	Poids négatifs
A*	Oui	Non	Non	$O(\|E\|)$	Oui
JPS	Oui	Non	Non	$O(\|E\|)$	Non
Djikstra	Oui	Oui	Non	$O(\|E\| + \|V\| \log\|V\|)$.	Non
Bellman-Ford	Oui	Oui	Non	$O(\|V\|\|E\|)$	Oui
SPFA	Oui	Oui	Non	$O(\|V\|\|E\|)$	Oui
Floyd-Warshall	Oui	Oui	Oui	$\Theta(\|V\|^3)$.	Oui
Johnson	Oui	Oui	Oui	$O(\|V\|^2 \log\|V\| + \|V\|\|E\|)$.	Oui

Expériences

Ci-dessous, je note les temps d'exécution (en s) enregistrées durant les quelques expériences réalisées.

Les algorithmes sont codés au plus optimal que j'ai pu, tout en restant homogène dans la façon de coder, mais le programme est lancé en mode Debug sans aucun flag d'optimisation du compilateur.

Attention, la structure de données utilisée est la matrice d'adjacence. Il est possible qu'en se basant sur une structure de nœuds avec des pointeurs sur les voisins directs, les temps d'exécution soient très différents.

Il faut prendre en compte que pour comparer un algorithme d'une classe à une autre, il faut diviser ou multiplier le temps d'exécution par $|V|$, voire $|V|^2$ pour passer de « chemin entre 2 points donnés » à « chemin entre tous les points ». Dans le tableau ci-dessous, je note **en gras** les algorithmes ayant les meilleures statistiques d'une même classe.

La durée notée pour l'algorithme SPFA (Shortest Path Faster Algorithm) est la moyenne des durées quand on utilise l'amélioration SPF, LLL, les deux, et aucun des deux.

L'algorithme A* est utilisé avec une heuristique nulle afin d'être comparable avec les autres algorithmes. Les algorithmes JPS et JPS+ ne sont pas implémentés car les graphes utilisés dans nos expériences ne sont pas adaptés à cet algorithme (grille à poids uniformes).

L'algorithme de Bellman-Ford est appliqué avec l'interruption anticipée (voir « Bellman-Ford / Améliorations possibles »). L'algorithme de Djikstra n'est pas amélioré.

Algorithme \ ($\|V\|$; $\|E\|$)	(265 ; 1590) D = 0.0455	(870 ; 5220) D = 0.014	(1767 ; 10602) D = 0.007
A*	**0.0010**	**0.0078**	**0.0402**
JPS	??????	??????	??????
Djikstra	0.0011	0.0141	0.0573
Bellman-Ford	0.0074	0.0708	0.4711
SPFA	**0.0008**	**0.0103**	**0.0418**
Floyd-Warshall	0.9285	30.623	257.50
Floyd-Warshall amélioré	**0.1089**	**4.5063**	**33.466**
Johnson	0.2527	9.4610	97.603

Les expériences sont réalisées en créant des graphes ayant une densité faible (graphes « sparses ») comme en témoigne les densités D présentées.

Conclusion

Dans le cas des graphes que j'utilise, on peut voir que pour trouver le chemin le plus court entre deux points donnés sans heuristique, l'utilisation d'un algorithme de classe supérieur peut faire l'affaire. Pour les recherches de chemin entre deux points donnés ou entre un point donné et le reste il est préférable d'utiliser l'algorithme du Shortest Path Faster Algorithm (SPFA), tandis que si on recherche les chemins les plus courts entre toutes les paires de points il est préférable d'utiliser l'algorithme de Floyd-Warshall amélioré.

Sans trop comprendre comment j'en arrive à ces statistiques ni comment les utiliser, je remarque qu'il est même préférable d'appliquer une seule fois un algorithme de recherche dans toutes les paires plutôt que d'utiliser plusieurs fois un algorithme de classe inférieure. Je remarque qu'il semble y avoir une relation entre le ratio $R_T = \frac{T_{Floyd-Warshall\ amélioré}}{T_{SPFA}}$ et la densité du réseau car $R_T \times D \approx 6.0$. Donc à première vue, si on doit appliquer SPFA plus de $\frac{6}{D}$ fois, il est préférable d'appliquer directement l'algorithme de Floyd-Warshall amélioré. Dans les expériences, cela se rapproche de $\frac{N}{2}$.