

# Algorithme de Dijkstra

En théorie des graphes, l'**algorithme de Dijkstra** (prononcé [dɛikstra]) sert à résoudre le problème du plus court chemin. Il permet, par exemple, de déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. Plus précisément, il calcule des plus courts chemins à partir d'une source vers tous les autres sommets dans un graphe orienté pondéré par des réels positifs. On peut aussi l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée.

L'algorithme porte le nom de son inventeur, l'informaticien néerlandais Edsger Dijkstra, et a été publié en 1959<sup>2</sup>.

Cet algorithme est de complexité polynomiale. Plus précisément, pour *n* sommets et *a* arcs, le temps est en *O*((*a* + *n*) log *n*), voire en *O*(*a* + *n* log *n*).

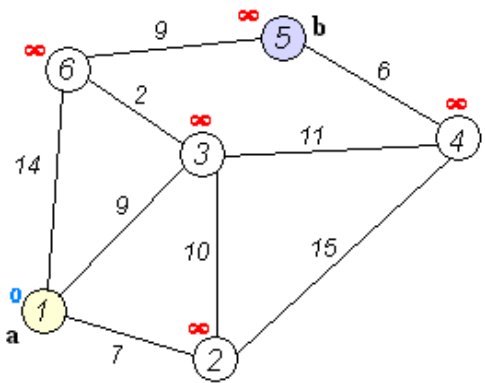
## Problème du plus court chemin

L'algorithme de Dijkstra permet de résoudre un problème algorithmique : le problème du plus court chemin. Ce problème a plusieurs variantes. La plus simple est la suivante : étant donné un graphe non-orienté, dont les arêtes sont munies de poids, et deux sommets de ce graphe, trouver un chemin entre les deux sommets dans le graphe, de poids minimum. L'algorithme de Dijkstra permet de résoudre un problème plus général : le graphe peut être orienté, et l'on peut désigner un unique sommet, et demander d'avoir la liste des plus courts chemins pour tous les autres nœuds du graphe.

## Principe sur un exemple

L'algorithme prend en entrée un graphe orienté pondéré par des réels positifs et un sommet source. Il s'agit de construire progressivement un sous-graphe dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet de départ. La distance correspond à la somme des poids des arcs empruntés.

### Algorithme de Dijkstra



L'algorithme de Dijkstra pour trouver le chemin le plus court entre a et b. Il choisit le sommet non visité avec la distance la plus faible, calcule la distance à travers lui à chaque voisin non visité, et met à jour la distance du voisin si elle est plus petite. Il marque le sommet visité (en rouge) lorsque il a terminé avec les voisins.

Découvreur ou inventeur	<u>Edsger Dijkstra</u>
Date de découverte	<u>1959</u> <sup>1</sup>
Problèmes liés	<u>Algorithme de recherche de chemin (d)</u> , <u>algorithme de la théorie des graphes (d)</u> , <u>algorithme glouton</u> , <u>algorithme</u>
Structure des données	<u>Graphe</u>
Basé sur	<u>Algorithme de parcours en largeur</u>
À l'origine de	<u>Algorithme A*</u> , <u>link-state routing protocol (en)</u> , <u>Open Shortest Path First</u> , <u>IS-IS</u>
<b>Complexité en temps</b>	
Pire cas	Pour un graphe à <i>n</i> sommets et <i>m</i> arcs :  <i>O</i> (( <i>n</i> + <i>m</i> ) log( <i>n</i> )) pour l'implémentation avec un <u>tas binaire</u> ,

Au départ, on considère que les distances de chaque sommet au sommet de départ sont infinies, sauf pour le sommet de départ pour lequel la distance est nulle. Le sous-graphe de départ est l'ensemble vide.

$O(m + n \log(n))$  pour l'implémentation avec un tas de Fibonacci

Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet de distance minimale et on l'ajoute au sous-graphe. Ensuite, on met à jour les distances des sommets voisins de celui ajouté. La mise à jour s'opère comme suit : la nouvelle distance du sommet voisin est le minimum entre la distance existante et celle obtenue en ajoutant le poids de l'arc entre sommet voisin et sommet ajouté à la distance du sommet ajouté.

On continue ainsi jusqu'à épuisement des sommets (ou jusqu'à sélection du sommet d'arrivée).

Distance entre la ville A et la ville J

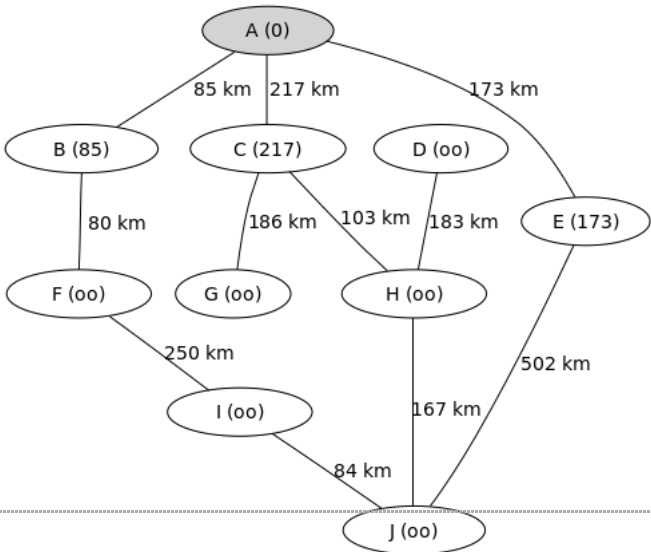
L'algorithme de Dijkstra fonctionne aussi sur un graphe non orienté. L'exemple ci-contre montre les étapes successives dans la résolution du chemin le plus court dans un graphe. Les nœuds symbolisent des villes identifiées par une lettre et les arêtes indiquent la distance entre ces villes. On cherche à déterminer le plus court trajet pour aller de la ville A à la ville J.

En neuf étapes, on peut déterminer le chemin le plus court menant de A à J, il passe par C et H et mesure 487 km.

Présentation sous forme de tableau

On peut aussi résumer l'exécution de l'algorithme de Dijkstra avec un tableau. Chaque étape correspond à une ligne. Une ligne donne les distances courantes des sommets depuis le sommet de départ. Une colonne donne l'évolution des distances d'un sommet donné depuis le sommet de départ au cours de l'algorithme. La distance d'un sommet choisi (car minimale) est soulignée. Les distances mises à jour sont barrées si elles sont supérieures à des distances déjà calculées.

Animation d'un algorithme de Dijkstra



Étape 1 : on choisit la ville A. On met à jour les villes voisines de A qui sont B, C, et E. Leurs distances deviennent respectivement 85, 217, 173, tandis que les autres villes restent à une distance infinie.

▶

⏏

■

◀◀

▶▶

1/9 ▼

Algorithme de Dijkstra

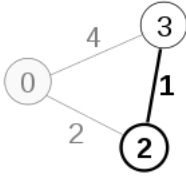
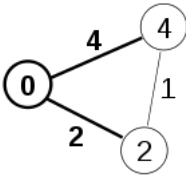
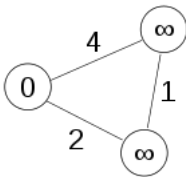
	à A	à B	à C	à D	à E	à F	à G	à H	à I	à J
étape initiale	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
A(0)		<u>85</u>	217	∞	173	∞	∞	∞	∞	∞
B(85 <sub>A</sub> )		-	217	∞	173	<u>165</u>	∞	∞	∞	∞
F(165 <sub>B</sub> )		-	217	∞	<u>173</u>	-	∞	∞	415	∞
E(173 <sub>A</sub> )		-	<u>217</u>	∞	-	-	∞	∞	415	675
C(217 <sub>A</sub> )		-	-	∞	-	-	403	<u>320</u>	415	675
H(320 <sub>C</sub> )		-	-	503	-	-	<u>403</u>	-	415	<del>675</del> 487
G(403 <sub>C</sub> )		-	-	503	-	-	-	-	<u>415</u>	487
I(415 <sub>F</sub> )		-	-	503	-	-	-	-	-	<u>487</u>
J(487 <sub>H</sub> )		-	-	<u>503</u>	-	-	-	-	-	-
D(503 <sub>H</sub> )		-	-	-	-	-	-	-	-	-

Le tableau donne non seulement la distance minimale de la ville A à la ville J (487) mais aussi le chemin à rebours (J - H - C - A) pour aller de A à J ainsi que toutes les distances minimales de la ville A aux autres villes rangées par ordre croissant.

## Schéma de l'algorithme

Le graphe est noté  $G = (S, A)$  où :

- l'ensemble  $S$  est l'ensemble fini des sommets du graphe  $G$  ;
- l'ensemble  $A$  est l'ensemble des arcs de  $G$  tel que : si  $(s_1, s_2)$  est dans  $A$ , alors il existe un arc depuis le nœud  $s_1$  vers le nœud  $s_2$  ;
- on définit la fonction *poids* définie sur  $S \times S$  dans  $\mathbb{R}^+ \cup \{+\infty\}$  qui à un couple  $(s_1, s_2)$  associe le poids positif *poids*( $s_1, s_2$ ) de l'arc reliant  $s_1$  à  $s_2$  (et  $+\infty$  s'il n'y a pas d'arc reliant  $s_1$  à  $s_2$ ).



Le poids du chemin entre deux sommets est la somme des poids des arcs qui le composent. Pour une paire donnée de sommets  $s_{deb}$  (le sommet du départ)  $s_{fin}$  (le sommet d'arrivée) appartenant à  $S$ , l'algorithme trouve un chemin depuis  $s_{deb}$  vers  $s_{fin}$  de moindre poids (autrement dit un chemin le plus léger ou encore le plus court).

L'algorithme fonctionne en construisant un sous-graphe  $P$  de manière que la distance entre un sommet  $s$  de  $P$  depuis  $s_{deb}$  soit connue et soit un minimum dans  $G$ . Initialement,  $P$  contient simplement le nœud  $s_{deb}$  isolé, et la distance de  $s_{deb}$  à lui-même vaut zéro. Des arcs sont ajoutés à  $P$  à chaque étape :

1. en identifiant tous les arcs  $a_i = (s_{i1}, s_{i2})$  dans  $P \times G$ ;
2. en choisissant l'arc  $a_j = (s_{j1}, s_{j2})$  dans  $P \times G$  qui donne la distance minimum depuis  $s_{deb}$  à  $s_{j2}$  en passant tous les chemins créés menant à ce nœud.

L'algorithme se termine soit quand  $P$  devient un arbre couvrant de  $G$ , soit quand tous les nœuds d'intérêt<sup>3</sup> sont dans  $P$ .

On peut donc écrire l'algorithme de la façon suivante :

Entrées :  $G = (S, A)$  un graphe avec une pondération positive *poids* des arcs,  $s_{deb}$  un sommet de  $S$

```

P := ∅
d[a] := +∞ pour chaque sommet a
d[sdeb] = 0
Tant qu'il existe un sommet hors de P
  Choisir un sommet a hors de P de plus petite distance d[a]
  Mettre a dans P
  Pour chaque sommet b hors de P voisin de a
    Si d[b] > d[a] + poids(a, b)
      d[b] = d[a] + poids(a, b)
      prédécesseur[b] := a
  Fin Pour
Fin Tant Que

```

## Implémentation de l'algorithme

### Fonctions annexes

L'algorithme utilise les fonctions annexes suivantes.

#### Initialisation de l'algorithme

```

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

```

#### Recherche d'un nœud de distance minimale

- On recherche un nœud de distance minimale (relié par l'arc de poids le plus faible) de  $s_{deb}$  parmi les nœuds situés hors de  $P$ . Le complémentaire de  $P$  est noté  $Q$ . On implémente pour cela une fonction `Trouve_min(Q)` qui choisit un nœud de  $Q$  de distance minimale.

```

Trouve_min(Q)
1 mini := infini
2 sommet := -1
3 pour chaque sommet s de Q
4   si d[s] < mini
5     alors
6       mini := d[s]
7       sommet := s
8 renvoyer sommet

```

#### Mise à jour des distances

- On met à jour les distances entre  $s_{deb}$  et  $s_2$  en se posant la question : vaut-il mieux passer par  $s_1$  ou pas ?

```

maj_distances(s1, s2)
1 si d[s2] > d[s1] + Poids(s1, s2) /* Si la distance de sdeb à s2 est plus grande que */
2                                 /* celle de sdeb à S1 plus celle de S1 à S2 */
3   alors

```

```

4      d[s2] := d[s1] + Poids(s1,s2) /* On prend ce nouveau chemin qui est plus court */
5      prédécesseur[s2] := s1      /* En notant par où on passe */

```

## Fonction principale

Voici la fonction principale utilisant les précédentes fonctions annexes :

```

Dijkstra(G,Poids,sdeb)
1 Initialisation(G,sdeb)
2 Q := ensemble de tous les nœuds
3 tant que Q n'est pas un ensemble vide faire
4     s1 := Trouve_min(Q)
5     Q := Q privé de s1
6     pour chaque nœud s2 voisin de s1 faire
7         maj_distances(s1,s2)
8     fin pour
9 fin tant que

```

Le plus court chemin de  $s_{deb}$  à  $s_{fin}$  peut ensuite se calculer itérativement selon l'algorithme suivant, avec  $A$  la liste représentant le plus court chemin de  $s_{deb}$  à  $s_{fin}$  :

```

1 A := suite vide
2 s := sfin
3 tant que s != sdeb faire
4     A := cons(s, A) /* on ajoute s en tête de la liste A */
5     s := prédécesseur[s] /* on continue de suivre le chemin */
6 fin tant que
7 A := cons(sdeb, A) /* on ajoute le nœud de départ */

```

Attention : s'il n'y a pas de chemin de  $s_{deb}$  à  $s_{fin}$  cette partie de l'algorithme fait une boucle infinie ou une erreur selon votre implémentation.

## Spécialisation de l'algorithme

Il est possible de spécialiser l'algorithme en arrêtant la recherche lorsque l'égalité  $s_1 = s_{fin}$  est vérifiée, dans le cas où on ne cherche que la distance minimale entre  $s_{deb}$  et  $s_{fin}$ .

## Complexité de l'algorithme

L'efficacité de l'algorithme de Dijkstra repose sur une mise en œuvre efficace de `Trouve_min`. L'ensemble  $Q$  est implémenté par une file à priorités. Si le graphe possède  $|A|$  arcs et  $|S|$  nœuds, qu'il est représenté par des listes d'adjacence et si on implémente la file à priorités par un tas binaire (en supposant que les comparaisons des poids d'arcs soient en temps constant), alors la complexité en temps de l'algorithme est  $O((|A| + |S|) \times \log(|S|))$ . En revanche, si on implémente la file à priorités avec un tas de Fibonacci, l'algorithme est en  $O(|A| + |S| \times \log(|S|))$ <sup>5</sup>.

## Correction de l'algorithme

La démonstration de correction est une récurrence sur  $\text{Card}(P)$  (qui augmente de 1 à chaque itération) et repose sur l'invariant suivant :

$$\begin{cases} \forall c \in P, d(c) = \text{mini}(c) \\ \forall c \notin P, d(c) = \text{mini}P(c) \end{cases}$$

où :

- $\text{mini}(c)$  est le poids d'un plus court chemin menant à  $c$  ;
- $\text{miniP}(c)$  est le poids d'un plus court chemin dont tous les sommets intermédiaires sont dans  $P$  menant à  $c$ .

Si  $n = \text{Card}(P)$ , la preuve est la suivante :

- pour  $n = 0$  et  $1$  : immédiat ;
- pour  $n$  un entier non nul, supposons l'invariant vrai.

L'algorithme sélectionne un pivot  $a \notin P$  tel que  $d(a) = \underset{x \notin P}{\text{Min}} d(x)$ , et l'ajoute dans  $P$ . Il faut donc montrer que  $d(a) = \text{mini}(a)$  après modification de  $P$  :

Par hypothèse,  $d(a) = \text{miniP}(a)$  avant modification de  $P$ , donc s'il existe un chemin  $C$  tel que  $\text{poids}(C) < d(a)$  alors ce chemin contient au moins un sommet  $b \neq a$  tel que  $b \notin P$ .

Soit donc un tel sommet  $b$  tel que tous ses prédécesseurs dans  $C$  soient dans  $P$  (il existe car le premier sommet de  $C$  est l'origine qui est dans  $P$ ). Décomposons  $C$  en  $Cb^-$  et  $Cb^+$  où  $Cb^-$  est la première partie de  $C$  dont le dernier sommet est  $b$ , et  $Cb^+$  la suite de  $C$ . Alors  $\text{poids}(C) = \text{poids}(Cb^-) + \text{poids}(Cb^+) \geq \text{poids}(Cb^-) = d(b) \geq d(a)$  : contradiction.

Il n'existe donc aucun chemin  $C$  tel que  $\text{poids}(C) < d(a)$  d'où  $d(a) = \text{mini}(a)$ . L'égalité est toujours vraie pour les autres éléments de  $P$ .

Enfin, l'algorithme met à jour la fonction  $d$  (et prédécesseur) pour les successeurs  $b$  du pivot  $a$  :  $d[b] = \min(d[b], d[a] + \text{poids}(a, b))$ .

Montrons qu'alors,  $\forall c \notin P, d(c) = \text{miniP}(c)$  :

- si  $c$  n'est pas un successeur du pivot  $a$ , alors il n'existe aucun nouveau chemin  $C$  menant à  $c$  tel que  $\text{poids}(C) < d(c)$  et dont tous les sommets intermédiaires sont dans  $P$  après l'ajout de  $a$  dans  $P$  puisqu'il faudrait alors passer par  $a$  avant de passer par un autre sommet  $d \in P$ , mais ce ne serait pas le plus court chemin jusqu'à  $d$  puisque le poids du plus court chemin jusqu'à  $d$  a déjà été calculé avant l'ajout de  $a$  dans  $P$  par hypothèse, et ce chemin ne contient donc que des sommets ayant été dans  $P$  avant l'ajout de  $a$  dans  $P$  ;
- sinon, il existe de tels chemins, en notant  $C$  le meilleur d'entre eux, alors  $C$  sera un des nouveaux chemins engendrés par l'ingestion du pivot  $a$  par  $P$ , donc  $a \in C$  et d'autre part le pivot  $a$  est le dernier sommet intermédiaire de  $C$  puisque le plus court chemin menant aux autres sommets de  $P$  ne passe par  $a$  comme expliqué plus haut.  $C$  est donc la réunion du plus court chemin menant à  $a$  avec l'arc  $(a, c)$  : d'où  $\text{poids}(C) = d(c) = \text{poids}((a, c)) + d(a) = \text{miniP}(c)$ .

## Applications

L'algorithme de Dijkstra trouve une utilité dans le calcul des itinéraires routiers. Le poids des arcs pouvant être la distance (pour le trajet le plus court), le temps estimé (pour le trajet le plus rapide), la consommation de carburant et le prix des péages (pour le trajet le plus économique). <sup>[réf. nécessaire]</sup>

Une application courante de l'algorithme de Dijkstra apparaît dans les protocoles de routage interne « à état de liens », tels que *Open Shortest Path First* (OSPF)<sup>6</sup> ou *IS-IS*<sup>7</sup> – ou encore *PNNI* (en) sur les réseaux *ATM* –, qui permettent un routage internet très efficace des informations en cherchant le parcours le plus efficace. [réf. nécessaire]

## Comparaison avec d'autres algorithmes

- L'algorithme de Dijkstra est fondé sur un parcours en largeur.
- La spécialisation de l'algorithme de Dijkstra qui calcule un plus court chemin d'une source à une destination est une instance de l'algorithme A\* dans lequel la fonction heuristique est la fonction nulle. L'algorithme A\* qui utiliserait une heuristique minorante et monotone (par exemple la distance à vol d'oiseau) peut être plus efficace [réf. souhaitée].
- L'algorithme ne s'applique pas aux graphes avec des poids négatifs. Mais l'algorithme de Bellman-Ford permet de résoudre le problème des plus courts chemins depuis une source avec des poids négatifs (mais sans cycle négatif).
- L'algorithme de Floyd-Warshall calcule des plus courts chemins entre tous les sommets dans un graphe où les poids peuvent être négatifs.

## Notes et références

- (en) E. W. Dijkstra, « A note on two problems in connexion with graphs », *Numerische Mathematik*, Springer Science+Business Media, vol. 1, n<sup>o</sup> 1, décembre 1959, p. 269-271 (ISSN 0029-599X (<https://portal.issn.org/resource/issn/0029-599X>) et 0945-3245 (<https://portal.issn.org/resource/issn/0945-3245>), OCLC 1760917 (<https://worldcat.org/fr/title/1760917>), DOI 10.1007/BF01386390 (<https://dx.doi.org/10.1007/BF01386390>), lire en ligne (<https://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf>)
- Dijkstra, E. W., « A note on two problems in connexion with graphs », *Numerische Mathematik*, vol. 1, 1959, p. 269–271 (DOI 10.1007/BF01386390. (<https://dx.doi.org/10.1007/BF01386390>)).
- Par exemple, les nœuds n'ayant pas d'arêtes autres que celle que l'on a parcourue pour arriver à eux, ne sont pas considérés comme des nœuds d'intérêt.
- La suite de caractères /\* … \*/ est un commentaire.
- Cormen *et al.* 2010, p. 610
- (en) John Moy, « RFC2328 OSPF Version 2 » (<https://tools.ietf.org/html/rfc2328#page-161>), avril 1998 (consulté le 19 mai 2016) : « *Using the Dijkstra algorithm, a tree is formed from this subset of the link state database.* », p. 161
- (en) David R. Oran, « RFC1142 : OSI IS-IS Intra-domain Routing Protocol » (<https://tools.ietf.org/html/rfc1142>), février 1990 (consulté le 19 mai 2016) : « *An algorithm invented by Dijkstra (see references) known as shortest path first (SPF), is used as the basis for the route calculation.* »

## Annexes

Sur les autres projets Wikimedia :

*algorithme de Dijkstra*, sur le Wiktionnaire

## Bibliographie

- (en) « *A short introduction to the art of programming* » de Edsger W. Dijkstra (<http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD316.PDF>), 1971, contenant l'article original (1959) décrivant l'algorithme de Dijkstra (pages 67 à 73).

- (en)  Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein, *Introduction to Algorithms*, MIT Press et McGraw-Hill, 2001, 2<sup>e</sup> éd. [détail de l'édition], section 24.3, « *Dijkstra's algorithm* », pages 595–601.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein (trad. de l'anglais), *Algorithmique : Cours avec 957 exercices et 158 problèmes*, Dunod, 2010 [détail de l'édition]

## Articles connexes

---

- Lexique de la théorie des graphes
- Recherche de chemin et Problèmes de cheminement
- Algorithme de parcours en largeur
- Algorithme A\*
- Article sur l'algorithme de Ford-Bellman qui permet de calculer le plus court chemin avec des poids d'arcs négatifs

## Liens externes

---

- Explication détaillée de l'algorithme de Dijkstra et implémentation complète en C++ (http://www.nimbustier.net/publications/dijkstra/index.html)
- Explication détaillée de l'algorithme de Dijkstra et implémentation complète en Python (http://www.gilles-bertrand.com/2014/03/dijkstra-algorithm-python-example-source-code-shortest-path.html)
- (en) Applet Java montrant l'algorithme de Dijkstra étape par étape (http://students.ceid.upatras.gr/~papagel/project/kef5\_7\_1.htm)
- (en) Applets Java utilisant l'algorithme (http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml)

---

Ce document provient de « https://fr.wikipedia.org/w/index.php?title=Algorithme\_de\_Dijkstra&oldid=205855471 ».

▪