ALGO1 – Parcours en largeur

François Schwarzentruber

8 janvier 2021

1 Plus court chemin

Définition 1 (chemin élémentaire) Soit $G = (S, A)$ un graphe orienté pondéré.
Un chemin $c = c_0 \to c_1 \to \cdots \to c_\ell$ est élémentaire si les c_i sont distincts deux à deux.

Définition 2 (poids d'un chemin) Soit $G = (S, A)$ un graphe orienté pondéré. Soit $c = c_0 \to c_1 \to \cdots \to c_\ell$ un chemin. Son poids $poids(c)$ est $\sum_{i=0}^{\ell-1} poids(c_i \to c_{i+1})$.

Définition 3 (Plus court chemin)
Soit $G = (S, A)$ un graphe orienté pondéré. Soit $s, t \in S$. Supposons qu'il existe un chemin de s à t. Un plus court chemin de s à t est un chemin est de poids $min\{poids(c) \mid c \text{ chemin de } s \text{ à } t\}$.

Proposition 4 La notion de plus court chemin est bien définie.

Définition 5 (Problème du plus court chemin depuis une source)
— Entrée : un graphe orienté G, un sommet s de G ;
— Sortie : pour tout sommet t, un plus court chemin de s à t, (s'il en existe un).

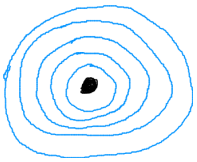
2 Pseudo-codes du parcours en largeur et de l'algorithme de Dijktsra

pre : graphe orienté G, un sommet s de G
post : tableau d où, pour tout sommet u, $d[u]$ est le poids d'un plus court chemin de s à u

Les arcs sont de poids 1.
fonction parcoursLargeur(G, s)
    pour tout sommet v de G, $d[v] := +\infty$
    $d[s] := 0$
    $F :=$ file vide
    enfiler s dans F

    $pred[s] := []$
    tant que F non vide **faire**
        $u := F$.défiler
        pour tout successeur v de u **faire**
            si $d[v] = +\infty$ **alors**
                $d[v] := d[u] + 1$
                F.enfiler(v)
                $pred[v] := u$
    retourner $d, pred$

Les arcs sont pondérés positivement.
fonction dijkstra(G, s)
    pour tout sommet v de G, $d[v] := +\infty$
    $d[s] := 0$
    $F :=$ file de priorité vide
    enfiler les sommets de G dans F
                avec priorité donné par d
    $pred[s] := []$
    tant que F non vide **faire**
        $u :=$ défiler élément prioritaire (min) de F
        pour tout successeur v de u **faire**
            si $d[v] > d[u] + poids(u, v)$ **alors**
                $d[v] := d[u] + poids(u, v)$
                F.mettreAJourDiminution(v)
                $pred[v] := u$
    retourner $d, pred$



3 Parcours en largeur

3.1 Motivation

On s'intéresse à calculer un plus court chemin dans un graphe : trouver un itinéraire pour aller de la maison à la boulangerie, réaliser mat en le moins de coups possibles aux échecs, etc.

La notion de plus court chemin est bien définie. S'il existe un chemin du sommet s à t alors

$$L_{s,t} = \{\text{longueur de } c \mid c \text{ chemin de } s \text{ à } t\} \subseteq \mathbb{N}$$

est non vide et admet un minimum $\ell_{s,t}$. Sinon, $\ell_{s,t} = +\infty$. Un plus court chemin de s à t est un chemin c de s à t telle que la longueur de c est égale à $\ell_{s,t}$.

3.2 Algorithme

Le parcours en largeur calcule la plus courte distance depuis une source s à tout nœud t. Le calcul est stocké dans $d[t]$. *pred[t]* est le prédécesseur dans un plus court chemin depuis s à t. On utilise une file F pour stocker les nœuds à traiter.

3.3 Analyse de l'algorithme

Théorème 6 Chaque sommet est enfilé au plus une fois.

Démonstration. Invariant : un sommet t qui a été dans la file est tel que $d[t]$ est fini. ■

Théorème 7 Le parcours en largeur est en $O(|S| + |A|)$.

Démonstration. Soit E l'ensemble des sommets qui ont été dans la file et qui n'y sont plus. Au début $E = \emptyset$. A chaque passage de boucle « tant que », E gagne un élément et $|E|$ croit strictement de 1. $|E|$ est majoré par $|S|$. Donc il y a $O(|S|)$ passages dans la boucle « tant que ».

La boucle « pour » coûte $|t \mid s' \to t|$. Les s' sont tous différents à chaque fois (c'est l'élément qui entre dans E) donc tout le travail effectué dans tous les passages de la boucle « pour » coûte $\Sigma_{s' \in S}|t \mid s' \to t| = O(|A|)$. ■

Théorème 8 À la fin de l'algorithme, pour tout sommet t, $d[t]$ est la longueur d'un plus court chemin de s à t s'il en existe un et $+\infty$ sinon.

Démonstration.
Notons $\delta(t)$ la longueur d'un plus court chemin de s à t s'il en existe un et $\delta(t) = +\infty$ sinon.
1) On a l'invariant suivant : $\delta \leqslant d$ (on montre l'initialisation et la conservation comme d'habitude).
2) Posons $\mathcal{P}(n)$:

> « Il y a un moment dans l'exécution tel que, pour tout $t \in S$,
>
> 1. $\delta(t) \leqslant n \Rightarrow d[t] = \delta(t)$
> 2. $\delta(t) > n \Rightarrow d[t] = +\infty$
> 3. La file F, notée F_n, contient exactement les sommets t tels que $\delta(t) = n$ ».

Montrons $\mathcal{P}(n)$ pour tout n par récurrence sur n.

— Pour $n = 0$, on considère le moment juste avant de commencer l'exécution de la boucle « tant que ». D'où $\mathcal{P}(0)$.

— Supposons $\mathcal{P}(n)$ vraie. Et montrons $\mathcal{P}(n+1)$. Au moment correspondant à $\mathcal{P}(n)$, la file F_n contient exactement les sommets à distance n. Soit $k = |F_n|$. Pour $\mathcal{P}(n+1)$, on considère le moment où l'on a exécuter k tour de boucle « tant que » supplémentaire. La file F_{n+1} contient alors des successeurs des k sommets à distance n.

1. Soit $t \in S$ tel que $\delta(t) \leqslant n+1$.
   — Si $\delta(t) \leqslant n$, $d[t] = \delta(t)$ par $\mathcal{P}(n)$, 1. et $d[t]$ est inchangé.
   — Sinon $\delta(t) = n+1$. Notons c un plus court chemin de s à t et u le prédécesseur de t dans c. Le sous-chemin $s \to u$ de c reste un plus court chemin et $\delta(u) = n$. Par $\mathcal{P}(n)$ 3., u est F_n. Par $\mathcal{P}(n)$ 1., on a $d[u] = \delta(u)$ .Par $\mathcal{P}(n)$ 2., on avait bien $d[t] = +\infty$. Donc, la mise à jour de $d[t]$ a lieu : $d[t] := d[u] + 1 = \delta(u) + 1 = n + 1$.
2. Soit $t \in S$ tel que $\delta(t) > n+1$. Montrons que $d[t] = +\infty$. en montrant $t \notin F_{n+1}$ et ainsi $d[t]$ n'a pas été mis à jour. Par l'absurde, si $t \in F_{n+1}$, la mise à jour $d[t] := d[u] + 1$ a eu lieu, avec $u \in F_n$. Par $\mathcal{P}(n)$ 1. et 3., $d[u] = \delta(u) = n$. On aurait donc $d[t] = n+1$ et donc $\delta(t) \leq n+1$. Contradiction avec l'invariant $\delta \leq d$. Donc, $t \notin F_{n+1}$.

3. Montrons $\delta(t) = n+1$ ssi t est dans F_{n+1}.

   $\boxed{\Leftarrow}$ $\delta(t) = n+1$ implique par le point 1 que l'on vient de montrer (deuxième tiret) que t est mis à jour entre le moment $\mathcal{P}(n)$ et $\mathcal{P}(n+1)$. Il est donc enfilé et est donc dans F_{n+1}.

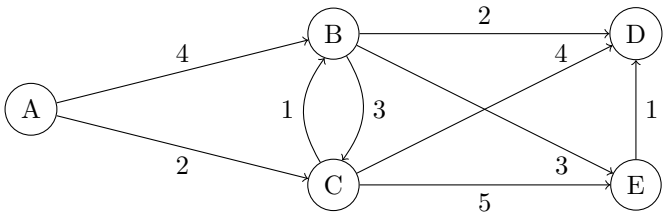
   $\boxed{\Rightarrow}$ Réciproquement, si t est mis à jour entre le moment $\mathcal{P}(n)$ et $\mathcal{P}(n+1)$, alors la mise à jour était $d[t] := d[u] + 1$ avec $u \in F_n$. Mais alors, par $\mathcal{P}(n)1.3$, $d[u] = \delta(u) = n$. Donc $d[t] = n+1$ implique $\delta(t) = n+1$ par le point 1.

D'où $\mathcal{P}(n+1)$.

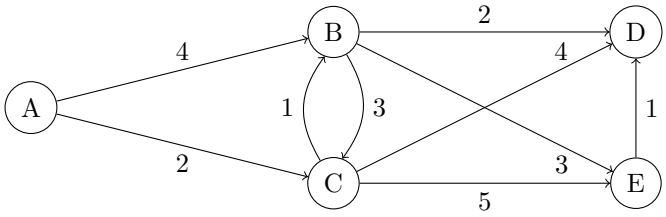
Conclusion On a montré par récurrence $\mathcal{P}(n)$ pour tout $n \in \mathbb{N}$. Soit $t \in S$.
— Si $d[t] < +\infty$, on applique $\mathcal{P}(\delta[t])$ et on $d[t] = \delta(t)$.
— Si $\delta[t] = +\infty$, on applique l'invariant $\delta \leq d$ et on a $d[t] = +\infty$.

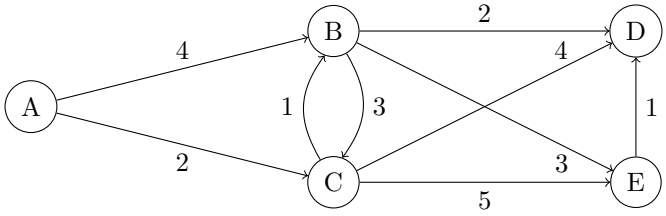
■

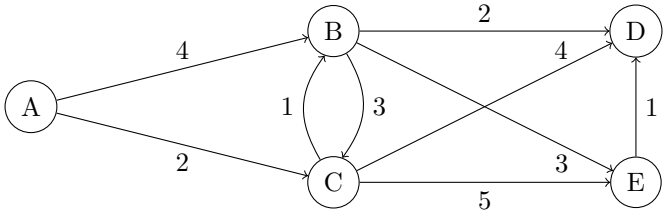
4 Explication de l'algorithme de Dijkstra

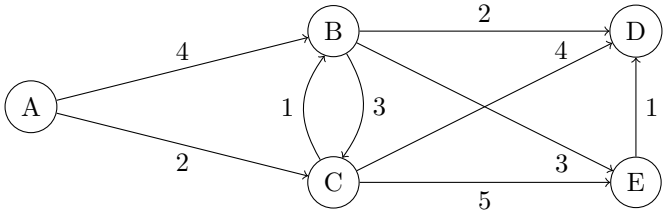


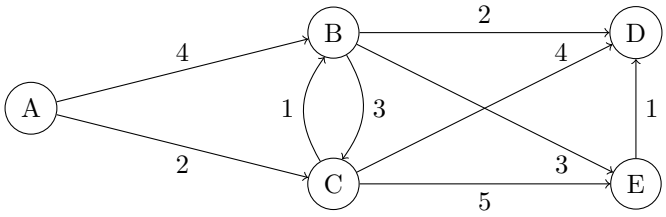
5 Exemple d'exécution de l'algorithme de Dijkstra



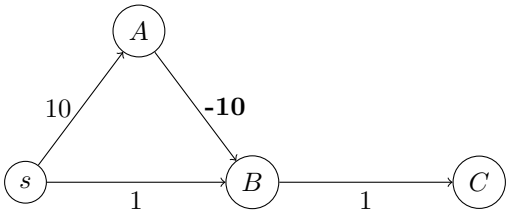








6 Contre-exemple avec des poids négatifs

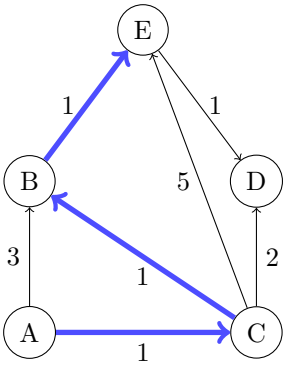


7 Algorithme de Dijkstra

Dans les jeux, on s'intéresse au nombre minimum de coup à jouer. Mais il arrive que les coups aient des coûts (en euros par exemple). Ou lorsque l'on calcule un itinéraire, nous avons les distances entre les villes. Nous allons alors modéliser le problème avec un graphe pondéré, c'est-à-dire dont les arcs sont étiquetés par des distances.

7.1 Adaptation du parcours en largeur

Un graphe pondéré $G = (S, A)$ est un graphe muni d'une fonction de **pondération positive** $w : A \to \mathbb{R}^+$. $w(s \to t)$ s'appelle le poids de l'arc $s \to t$. Le poids d'un chemin est le somme des poids des arcs par lesquels il passe. Le dessin suivant montre un plus court chemin de l'origine A vers E.

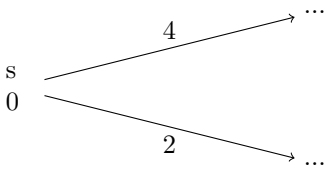


Supposons un instant que les poids sont des entiers. Nous sommes ici dans une situation typique où l'algorithme du parcours en largeur fonctionne : il suffit d'ajouter des sommets intermédiaires.

Nous sommes dans une situation où beaucoup d'instants dans l'algorithme sont des instants d'attente. Rien ne sert d'attendre, il suffit.... de mettre des 'alarmes' sur les noeuds. Bref, on utilise une file de priorité. Les priorités sont les valeurs $d[t]$. Ici, on peut mettre à jour *plusieurs fois* la valeur $d[t]$ pour un sommet t donné (on rerègle une alarme) alors que pour le parcours en largeur dès que $d[t]$ est affecté, c'est la bonne valeur.

7.2 Algorithme

Commençons par dire que pour aller de la source s à la source s, la distance est nulle.



Puis à chaque étape, on considère le sommet non colorié (non traité) dont la distance courante est la plus courte. On le colorie. Puis on met à jour les sommets directement accessibles à partir de celui-ci si le résultat est meilleur.

On a terminé lorsque toute les sommets sont coloriés (traités).

On a besoin d'adapter la file de priorité traditionnelle avec une nouvelle opération : la mise à jour en cas de diminution.

7.3 Analyse de l'algorithme

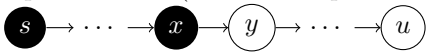
Théorème 9 L'algorithme est correct.

Démonstration. Comme d'habitude, on appelle $\delta(t)$ la distance minimale de s à t et $+\infty$ s'il n'y a pas de chemin. Comme pour le parcours en largeur, on a le même invariant qui dit que d surestime δ : $\delta \leqslant d$.

On a aussi l'invariant suivant : toutes les distances calculées pour les noeuds déjà traités sont exactement les distances minimales. Formellement, $\forall t \notin F$, $d[t] = \delta(t)$. Montrons le.

— Initialisation : Au début tous les éléments sont dans la file de priorité. Donc la propriété est trivialement vraie.

— Conservation : Soit u le sommet qui va être défilé. Si u =s, $d[s] = 0 = \delta(s)$. Maintenant, regardons le cas $u \neq s$. S'il n'y a pas de chemin de s à u, alors $+\infty = \delta(u) \leqslant d[u] = +\infty$.
Sinon, on a un plus court chemin c de s à u. Soit y le premier sommet de c qui est dans F (il existe, au pire il s'agit de u lui-même) et $x \notin F$ son prédécesseur (il a bien un précécesseur car $s \notin F$).



Voici les faits :

— Montrons $d[y] = \delta(y)$. D'après l'invariant, on a $d[x] = \delta(x)$. C'était le cas dès que x a été retiré de la file et lorsque $d[y]$ a été mis à jour. A ce moment là, l'arrête $x \to y$ a été considéré dans l'algorithme. On a donc $d[y] \leq d[x] + poids(x, y) = \delta(x) + poids(x, y)$. Le sous-chemin de s à x est un plus court chemin : son poids est donc $\delta(x)$. Le sous-chemin de s à y est un plus court chemin : son poids est donc $\delta(y)$, mais aussi $\delta[x] + poids(x, y)$. Donc $d[y] \leq \delta(y)$ (et égalité car $\delta \leq d$).

— On a $\delta(y) \leq \delta(u)$ car il n'y a que des **poids positifs**.

— On a $\delta(u) \leq d[u]$ car d surestime δ.

— De plus, comme u est sur le sommet qui vient d'être défilé, c'est le minimum de la file donc $d[u] \leqslant d[y]$.

En résumé :

$$d[y] = \delta(y) \leqslant \delta(u) \leqslant d[u] \leqslant d[y]$$

Donc $d[u] = \delta(u)$.

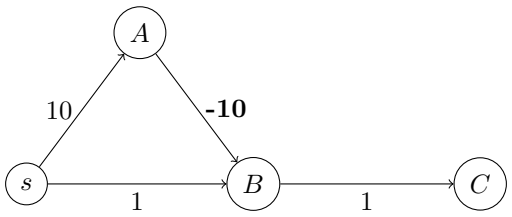
■

Théorème 10 L'algorithme effectue $O(|S| + |\text{file créer}| + |S||\text{défiler min}| + |A||\text{mise à jour}|)$.

	tableau	tas	tas de Fibonacci (complexité amortie)
file créer	$O(S)$	$O(S)$	$O(S)$
défiler min	$O(S)$	$O(\log S)$	$O(\log S)$
mise à jour	$O(1)$	$O(\log S)$	$O(1)$
en tout	$O(\|S\|^2)$	$O((\|S\| + \|A\|) \log(\|S\|))$	$O(\|S\| \log(\|S\|) + \|A\|)$

7.4 Contre-exemple avec des poids négatifs

Considérons le graphe suivant :



Si on exécute l'algorithme de Djikstra à partir du sommet s, alors la valeur $d[C]$ calculé est 2 alors que le chemin $s \to A \to B \to C$ est de longueur 1.

8 Optimisations de l'algorithme de Dijkstra

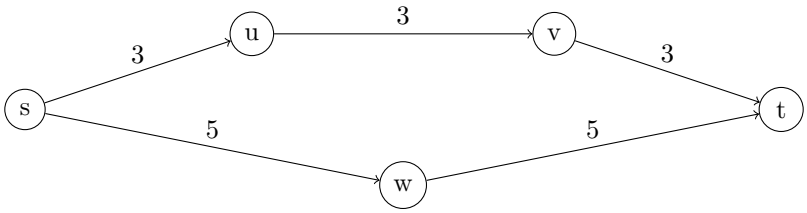
Définition 11 (Problème du plus court chemin depuis une source vers une destination)
- Entrée : un graphe orienté G, un sommet s de G, un sommet t de G ;
- Sortie : un plus court chemin de s à t (s'il en existe un).

Première optimisation simple : s'arrêter quand t est défilé

8.1 Recherche bidirectionnelle

Recherche bidirectionnelle
Alterner Dijkstra depuis s, et Dijkstra depuis t mais avec les arcs inverses
S'arrêter quand il y a un sommet w qui a été traité par les deux Dijkstra
Construire un chemin en prenant comme sommet intermédiaire un sommet x avec $d_1[x] + d_2[x]$ minimum

Exemple 1 Dans l'exemple ci-dessous, la recherche bidirectionnelle s'arrête avec w. Pourtant, $s \to w \to t$ n'est pas un plus court chemin.



Théorème 12 L'algorithme de recherche bidirectionnelle est correct.

Démonstration.
Considérons un plus court chemin de s à t.
S'il contient un x traité par Dijkstra1 et Dijkstra2 alors c'est bon.
S'il contient un x qui est ni traité par Dijkstra1, ni par Dijkstra2, alors par l'absurde... c'est pas possible.
S'il contient que des éléments traités par l'un ou l'autre, la frontière ne se touche pas directement mais à travers un arc, ... on montre que on a bien un x avec $d_1[x] + d_2[x]$ = poids de s à t. ■

8.2 Recherche orientée but ou A^*

Définition 13 (fonction potentielle, heuristique) $\lambda : S \to \mathbb{R}^+$.

Exemple 14 $\lambda(u) :=$ distance vol d'oiseau de u à t.

Algorithme A^*

Reponderer le graphe : $\overline{poids}(u, v) := poids(u, v) - \lambda(u) + \lambda(v)$.
Exécuter Dijkstra sur ce nouveau graphe

Théorème 15 L'algorithme A^* est correct dès que $\overline{poids}(u, v) \geq 0$ pour tout sommet u, v.

9 Discussion

9.1 Recherche non informée

Le parcours en profondeur utilise peu de mémoire (seule une branche est en mémoire à un instant donné). Bien pour les grands graphes. Mais il ne donne pas les plus courts chemins.

Le parcours en largeur donne le plus court chemin mais utilise beaucoup de mémoire. Dijkstra était le parcours en largeur avec des poids.

Un compromis est Iterative deepening search : on fait parcours en profondeur depuis s en bornant par une hauteur 1, puis par une hauteur 2, puis par une hauteur 3... ça utilise peu de mémoire mais c'est plus lent car on visite plusieurs fois les nœuds.

9.2 Recherche informée = avec une heuristique

Recherche best-first va là où c'est le mieux, mais ne tient pas compte du chemin déjà parcouru. Ce n'est pas optimal. Recursive best-first search pallie à ça, est optimal et utilise peu de mémoire. $A*$ pallie à ça, ressemble à Dijkstra avec une heuristique. A^* est optimal sur les graphes si l'heuristique est consistante et sur les arbres si admissible.

IDA^* est un Iterative deepening search, mais en bornant avec des valeurs successives intéressantes pour $d + h$. SMA^* est A^* où on jette le pire nœud quand plus de mémoire.

9.3 Autres problématiques

— recherche bidirectionnelle
— pas de graphe mais un environnement géométrique (Any-Angle)
— multi-agent path finding
— planification classique (le graphe de recherche est décrit implicitement avec une description des actions)
— plus court chemin pour pour un objet en 2D ou en 3D

10 Arbre implicite

Théorème 16 Soit a l'arité et h la hauteur maximale de recherche dans un arbre.
— La complexité temporelle du parcours en largeur est en $O(a^h)$.
— La complexité spatiale du parcours en largeur est en $O(a^h)$.

Parcours en profondeur itéré (iterative deepening depth-first search)
  pour $i := 0$ à h **faire**
    parcours en profondeur depuis la racine jusqu'à la profondeur i

Théorème 17 Le parcours en profondeur itéré retourne un plus court chemin s'il est de longueur plus petite que h.

Théorème 18 Soit a l'arité et h la hauteur maximale de recherche dans un arbre.
— La complexité temporelle du parcours en profondeur itéré est en $O(a^h)$.
— La complexité spatiale du parcours en profondeur itéré est en $O(a)$.

11 Notes bibliographiques

L'algorithme de Dijkstra a été inventé en 1959 mais il ne mentionne pas l'utilisation des tas comme structure de données [Dij59]. En intelligence artificielle, l'algorithme de Dijkstra s'appelle *uniform-cost search* [RN10]. L'algorithme A^* a été proposé en 1968 [HNR68]. Vous trouvez une belle discussion dans [RN10]. L'algorithme A^* est aussi utilisé dans des graphes/arbres implicites.

Iterative deepening depth-first search est étudié dans un article de 1985 [Kor85]. Il existe d'ailleurs une version du parcours en profondeur itéré avec poids et heuristique : IDA^* (iterative-deepening A^*) ([RN10], toujours). IDA^* est un *iterative deepening search*, mais en bornant avec des valeurs successives intéressantes pour $d + \lambda$. Une autre manière de faire pour utiliser moins de mémoire : oublier les sommets les moins bons dans la file de priorité : citons SMA^* (pour simplified memory-bounded A^*) [Rus92]. Vous trouverez de superbes démonstrations ici : `https://www.movingai.com/` et `https://www.redblobgames.com/pathfinding/`

Références

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 :269–271, 1959.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2) :100–107, 1968.

[Kor85] Richard E. Korf. Depth-first iterative-deepening : An optimal admissible tree search. *Artif. Intell.*, 27(1) :97–109, 1985.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition.* Pearson Education, 2010.

[Rus92] Stuart J. Russell. Efficient memory-bounded search methods. In Bernd Neumann, editor, *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, pages 1–5. John Wiley and Sons, 1992.