

Recherche sur le problème du voyageur de commerce

Abdallah Meebed

15 juin 2022

Table des matières

1 Définitions et motivations	3
1.1 Définitions	3
1.2 Classification des problèmes algorithmiques	5
1.3 Problème $P \stackrel{?}{=} NP$	6
2 Approximation des problèmes NP-complet	7
2.1 Problème du voyageur de commerce	7
2.2 Conclusion	10
2.3 Classification du voyageur de commerce	11
3 Résoudre le problème du voyageur de commerce	12
3.1 Algorithme de Held-Karp	12
3.1.1 Complexité en espace	13
3.1.2 Complexité en temps	14

1 Définitions et motivations

La théorie de la complexité est le domaine des mathématiques qui étudie le temps de calcul et la mémoire nécessaire par un ordinateur pour résoudre un problème algorithmique (un problème qui peut être résolu avec un algorithme). Les algorithmes qui résolvent le problème sont évalués selon des critères pour trouver celui l'algorithme le plus optimale (et approprié) avec les ressources disponibles. Des critères possibles sont la vitesse de résolution ou l'espace de mémoire nécessaire pour résoudre le problème. Dans un premier temps les définitions de ces critères sera présentée. Dans un autre temps, la notion de classification des problèmes sera introduite ; les problèmes sont classés selon leur « difficulté » à résoudre à l'aide des algorithmes actuels les plus optimales.

1.1 Définitions

Définition 1.1. La complexité en temps représente le temps mis par un algorithme pour résoudre un problème. C'est une des notions les plus importantes dans la théorie de la complexité. Les notations grand O de Landau sont utilisées pour évaluer le temps nécessaire pour résoudre un problème. On définit la notation $O(f(n))$ avec $f(n)$ comme variable discrète et n la taille des entrées pour voir la croissance du nombre d'opérations maximum (et par la suite, le temps maximum pris pour résoudre le problème) quand $n \rightarrow \infty$. Seulement le terme le plus important (le terme qui croit le plus rapidement) est pris en considération, sans son coefficient. Un algorithme $f(n)$ est résolu dans un temps polynomial quand $f(n) = O(n^k)$ pour $k \in \mathbb{Z}^+$ et quand $n \rightarrow \infty$. Quand un algorithme résout le problème dans un temps polynomial ou moins (logarithmique), il est considéré comme un algorithme rapide¹.

Par exemple, 2 algorithmes sont utilisés pour chercher un mot dans un dictionnaire de taille n mots. Le premier compare les mots par ordre alphabétique jusqu'à trouver le mot désiré. L'autre est l'algorithme de recherche dichotomique (*binary search* en anglais) : il divise l'ensemble des mots en 2 parties égales et évalue dans quelle sous-ensemble le mot appartient (selon l'ordre alphabétique) et recommence la première étape avec le sous-ensemble (redivise le sous-ensemble en 2) jusqu'à trouver le mot dans le dictionnaire. Le premier prends n étapes au pire des cas (le mot cherché est le dernier mot dans le dictionnaire), il est donc d'ordre $O(n)$. Il est facile de trouver l'ordre de complexité de la recherche dichotomique. Le nombre maximal de recherche est $\lceil \log_2 n \rceil$ et $\log_2 n = \log n / \log 2$ (changement de base) alors la recherche dichotomique est d'ordre $O(\log n)$ (le

1. A la limite que l'ordre du polynomial soit raisonnablement solvable par un ordinateur actuel

coefficient $1/\log 2$ n'est pas pris en considération). Par la suite la recherche dichotomique est plus optimale pour chercher un mot dans un dictionnaire.

Les ordres de grandeur les plus communs sont les suivants : $O(1)$ pour ordre constant, $O(\log n)$ pour ordre logarithmique, $O(n^k)$ avec $k \in \mathbb{Z}^+$ pour ordre polynomial (cas spécial pour $k = 1$, ordre linéaire), $O(k^n)$ pour ordre exponentiel et $O(n!)$ pour ordre factoriel. La difficulté d'un problème algorithmique dépend de l'ordre de grandeur de l'algorithme qui le résolve le plus efficacement.

Définition 1.2. Une machine du Turing est un concept abstrait qui représente un ordinateur. Cette machine joue le rôle d'une personne capable de suivre des instructions simples (voir [vidéo par Computerphile](#)² pour aller plus loin, ce n'est pas nécessaire). Simplement dit, une machine de Turing déterministe fait tout ce qu'un ordinateur moderne peut faire³.

Il existe aussi une machine de Turing non déterministe qui peut choisir entre plusieurs étapes à exécuter (c'est une manière de formaliser mathématiquement une recherche exhaustive de toutes les combinaisons ensuite les évaluer). Il est important de noter que c'est un concept abstrait.

Définition 1.3. Un problème de décision est un problème pour lequel la solution est soit « oui » ou « non ». Par exemple est-ce que l'entier positif n est un nombre premier ?

Définition 1.4. Un problème de fonction est un problème pour lequel la solution n'est pas simplement « oui » ou « non ». Par exemple quels sont les diviseurs non-triviaux de l'entier positif n ? La solution est une liste de diviseurs (ou non si n est premier).

Définition 1.5. Une réduction est un algorithme qui transforme une instance de problème X à un autre problème Y . Si cet algorithme existe, on dit alors que le problème X se réduit au problème Y , s'écrit aussi comme $X \leq Y$ ou bien $X \propto Y$. La réduction est utile dans plusieurs cas :

1. Rapidement résoudre X . Généralement un nouveau problème est réduit à un problème déjà solvable pour le résoudre et utiliser la solution pour résoudre le nouveau problème.
2. Montrer que résoudre X n'est pas plus difficile (parlant en complexité de temps et d'espace) que résoudre Y ⁴.

2. Lien hypertexte pour la version électronique, sinon le titre est *Turing Machines Explained - Computerphile* sur Youtube

3. Les langues de programmations utilisées dans un ordinateur (C, C++, Python, etc.) sont dites langues Turing-complet

4. Si l'algorithme s'exécute dans un temps polynomial

1.2 Classification des problèmes algorithmiques

Les problèmes algorithmiques sont classifiés selon les ressources temporaires et spatiales nécessaires pour les résoudre efficacement. Il est intéressant pour la suite de connaître seulement les classes spécifiés dans Figure 1.1.

Définition 1.6. Classe P est la classe des problèmes de décision résolus dans un temps polynomial (ou moins) par une machine de Turing déterministe *pour tout instance d'entrée*. Les problèmes de classe P sont considérés comme des problèmes qui sont rapide à résoudre, et par la suite faciles. Quelques problèmes dans la classe P : évaluation d'un circuit logique, déterminer si un mot est un palindrome et déterminer si un entier positif n est premier.

Définition 1.7. Classe NP est la classe des problèmes de décision résolus dans un temps polynomial par une machine de Turing non déterministe *pour tout instance d'entrée*. Par contre, une solution donnée peut être vérifiée dans un temps polynomial par une machine de Turing déterministe. En pratique, les ordinateurs actuels sont équivalents à des machines de Turing déterministe. Dans ce contexte, un problème de classe NP ne peut pas être résolu par une machine de Turing déterministe dans un temps polynomial (actuellement, voir Section 1.3).

Le problème le plus connu dans la classe NP est la factorisation d'un entier en nombres premiers. Les facteurs premiers ne peuvent pas être trouvés dans un temps polynomial (actuellement) mais si les facteurs sont donnés, il est rapide de vérifier si leur produit est bien égale au nombre. C'est la base de la cryptographie moderne.

Définition 1.8. Classe NP-complet est la classe des problèmes NP mais avec une propriété supplémentaire : tout autre problème en classe NP peut être réduit dans un temps polynomial à un problème NP-complet. Il est important de noter aussi que n'importe quel problème dans NP-complet peut être réduit dans un temps polynomial à un problème NP-difficile. Les 21 problèmes NP-complet de Karp [3] sont les exemples les plus connus de problèmes NP-complet. Par exemple, le problème suivant fait partie de la classe NP-complet : Soit $G(V, A)$ un graphe quelconque, est-ce qu'il existe un cycle Hamiltonien (un cycle qui passe par tout les sommets une seule fois et revient au sommet initiale) ?

Définition 1.9. Classe NP-difficile est la classe des problèmes (pas forcément des problèmes de décision) résolus dans un temps polynomial par une machine de Turing non déterministe et une solution donnée ne peut pas être vérifiée dans un temps polynomial par une machine de Turing déterministe. Autrement dit, il n'y a pas de méthode pour

rapidement vérifier la solution. Le problème du voyageur de commerce est un problème classé dans NP-difficile.

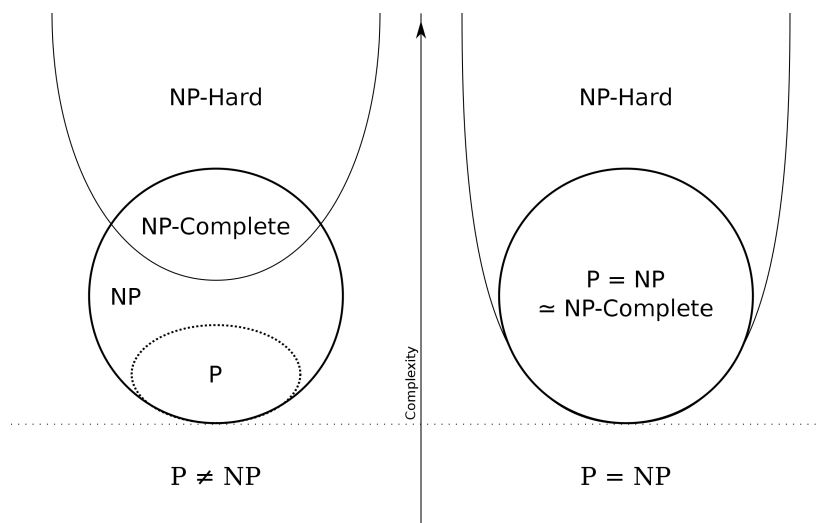


FIGURE 1.1 – Diagramme d'Euler sur les classes de complexité

1.3 Problème $P \stackrel{?}{=} NP$

Vérifier l'égalité $P \stackrel{?}{=} NP$ est une question essentielle dans la théorie de la complexité mais qui n'est pas encore résolue. Pour l'instant, tant qu'il n'y a pas de preuve d'algorithme qui résout un problème NP dans un temps polynomial, on suppose $P \neq NP$. Dans ce cas, certaines problèmes très importantes (factorisation des entiers en cryptographie, la conception de protéines en médecine, etc.) dans beaucoup de domaines resteront non-solvable d'une manière exacte à partir d'une taille d'entrée assez grande. Sinon, une preuve que $P = NP$ peut se présenter sous forme d'un algorithme qui s'exécute dans un temps polynomial. Dans ce cas, le reste des problèmes de classe NP peuvent être résolus dans un temps polynomial car par définition ils peuvent être réduits à un problème NP-complet.

2 Approximation des problèmes NP-complet

Un approche commune pour résoudre rapidement (dans un temps polynomial) un problème de classe NP est d'obtenir une approximation qui est prouvé d'être proche de la solution, c'est une ϵ -approximation.

Définition 2.1. Un algorithme est ϵ -approximé pour un problème P_1 ssi soit (i) P_1 est un problème de maximisation et pour tout instance de P_1

$$|(F^* - \hat{F})/F^*| \leq \epsilon, \quad 0 < \epsilon < 1,$$

ou bien (ii) P_1 est un problème de minimisation et pour tout instance de P_1

$$|(F^* - \hat{F})/F^*| \leq \epsilon, \quad \epsilon > 0,$$

avec F^* la solution optimale (supposé strictement positive) et \hat{F} la solution approximative obtenue.

Ce document présente la partie du théorème de [4] concernant le problème du voyageur de commerce.

2.1 Problème du voyageur de commerce

Dans cette section, le problème du voyageur de commerce (TSP pour *Traveling salesman problem*) et la preuve du théorème de [4] sont présentés. Il est important de noter que si $P=NP$, l'approximation resterait un problème dans la classe NP-complet (les deux classes sont équivalents). Pour la suite il est supposé que $P \neq NP$, qui impliquera que n'importe quelle algorithme pour résoudre le problème dans un temps polynomial doit produire des mauvaises approximations dans au moins un instance.

Problème de voyageur de commerce : Soit un graphe $G(N, A)$ un graphe complet avec une fonction poids $\omega : A \rightarrow Z$, trouver le cycle Hamiltonien (le cycle qui passe par tout les sommets exactement une fois) le plus optimal selon les critères suivantes :

1. Minimiser la longueur du cycle Hamiltonien.
2. Minimiser le temps d'arrivé moyen aux sommets. Le temps d'arrivé est mesuré selon le premier sommet et le poids des arêtes étant le temps pour aller d'un sommet à un autre. Soit $i_1, i_2, \dots, i_n, i_{n+1} = i_1$ un cycle Hamiltonien, alors le temps d'arrivé Y_k au sommet i_k est :

$$Y_k = \sum_{j=1}^{k-1} \omega(i_j, i_{j+1}), \quad 1 < k \leq n+1$$

Le temps d'arrivé moyen (à minimiser) est alors

$$\bar{Y} = \frac{1}{n} \sum_{k=2}^{n+1} Y_k = \frac{1}{n} \sum_{j=1}^n (n+1-j) \omega(i_j, i_{j+1})$$

3. Minimiser la variance des temps d'arrivés, qui est défini comme

$$\sigma = \frac{1}{n} \sum_{k=2}^{n+1} (Y_k - \bar{Y})^2$$

Afin de prouver que n'importe quelle approximation de ce problème est de classe au moins NP-complet, ce problème sera réduit à un problème prouvé être NP-complet selon [3] : Vérifier si un graphe contient un cycle Hamiltonien.

Théorème 2.1. L' ϵ -approximation du problème de voyageur des commerces est NP-complète.

Démonstration. Soit $G(N, A)$ un graphe quelconque. Chaque critère d'optimisation sera traité séparément :

1. Trouver un cycle Hamiltonien $\propto \epsilon$ -approximation du voyageur de commerce (minimiser la longueur du cycle) : Soit $G_1(V, E)$ un graphe complet ($E = \{(u, v) \mid u \neq v, u, v \in V\}$) avec $V = N$ et $n = |N|$. La fonction de poids $\omega : E \rightarrow \mathbb{Z}$ est définie :

$$\omega\{u, v\} = \begin{cases} 1 & \text{si } (u, v) \in A, \\ k & \text{sinon} \end{cases}$$

k est une valeur à choisir ultérieurement. Pour $k > 1$, la solution du TSP sur G_1 aura une longueur n ssi G contient un cycle Hamiltonien, sinon la solution aura une longueur au moins $k + n - 1$. Si on choisi $k \geq (1 + \epsilon)n$, il suffit juste d'évaluer la solution approximative : si elle est inférieure ou égale à $(1 + \epsilon)n$ (ne pas oublier l'erreur commise par l'approximation) alors G contient un cycle Hamiltonien. Sinon, G n'en contient pas. Cela revient à résoudre le problème de trouver si un graphe contient un cycle Hamiltonien (un problème NP-complet).

2. Trouver un cycle Hamiltonien $\propto \epsilon$ -approximation du voyageur de commerce (minimiser le temps d'arrivé moyen) : Soit $G_1(V, E)$ comme défini ci-dessus. Le temps d'arrivé moyen de G_1 est au maximum $(n + 1)/2$ ssi G contient un cycle Hamiltonien. Sinon, $\bar{Y} \geq k/n + (n - 1)/2$. Si on choisi $k > (1 + \epsilon)n(n + 1)/2$, il suffit juste d'évaluer la solution approximative : si elle est inférieure ou égale à $(1 + \epsilon)(n + 1)/2$, alors la solution exacte sera $(n + 1)/2$ et donc G contient un cycle Hamiltonien. Sinon, G n'en contient pas.

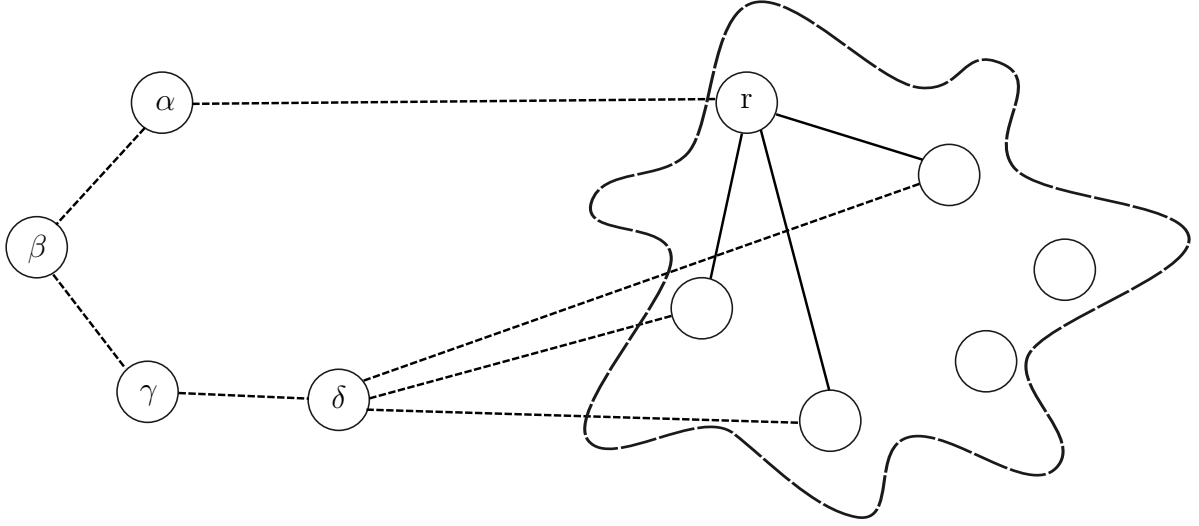


FIGURE 2.1 – Construction de G_1 , les lignes pointillées représentent les arêtes supplémentaires, exclusives à G_1

3. Trouver un cycle Hamiltonien $\propto \epsilon$ -approximation du voyageur de commerce (minimiser la variance du temps d'arrivée) : Utilisant, $G(N, A)$, on construit un graphe $G_1(N_1, A_1)$ (voir Figure 2.1) avec :

$$N_1 = N \cup \{\alpha, \beta, \gamma, \delta\}, \quad A_1 = A \cup \{(r, \alpha), (\alpha, \beta), (\beta, \gamma), (\gamma, \delta)\} \cup \{(\delta, z) \mid (r, z) \in A\}$$

pour r comme sommet quelconque dans G . Il est évident que G_1 contient un cycle Hamiltonien ssi G en contient. Le TSP est obtenu avec le graphe $G_2(N_2, A_2)$ avec $N_2 = N_1$ et $A_2 = \{(u, v) \mid u \neq v, u, v \in V\}$ (graphe complet) avec la fonction de poids $\omega : A_2 \rightarrow \mathbb{Z}$ est définie :

$$\omega\{u, v\} = \begin{cases} 1 & \text{si } (u, v) \in A_1, \\ k & \text{sinon} \end{cases}$$

Lemme 2.2 obtient une borne inférieure de σ .

Lemme 2.2. Pour $k > \sqrt{\frac{(1+\epsilon)(n)(n-1)(n+1)}{3}}$ et $\epsilon > 0$, G_2 contient un cycle Hamiltonien de variance $\sigma \leq \frac{(n-1)(n+1)}{12}$ ssi G_1 est Hamiltonien. Sinon, $\sigma > \frac{(1+\epsilon)(n-1)(n+1)}{12}$

Démonstration. Si G_1 est Hamiltonien alors ce cycle est aussi dans G_2 avec le poids de chaque arêtes = 1. Alors $\bar{Y} = (n+1)/2$ et

$$\sigma = \frac{1}{n} \sum_{k=2}^{n+1} (Y_k - \bar{Y})^2 = \frac{1}{n} \sum_{k=2}^{n+1} Y_k^2 - \bar{Y}^2 = \frac{2n^2 + 3n + 1}{6} - \frac{(n+1)^2}{4} = \frac{(n+1)(n-1)}{12}$$

Si G_1 n'est pas Hamiltonien, alors chaque cycle dans G_2 doit contenir un arête de poids k . Soit le tour optimal $i_1, i_2, \dots, i_n, i_{n+1}$ avec $i_1 = i_{n+1} = \beta$. Il y a 3 cas :

Cas 1 : $\omega(\beta, i_2) = 1$ et $\omega(i_j, i_{j+1}) = k$ pour $j, 1 < j \leq n$: Dans ce cas $Y_2 = 1$ et $Y_{n+1} \geq k + n - 1$. Si $\bar{Y} \geq k/2 + 1$ alors $|Y_2 - \bar{Y}| \geq k/2$. Si $\bar{Y} < k/2 + 1$ alors $|Y_{n+1} - \bar{Y}| \geq k/2 + 1$. Dans tous les cas pour $k > \sqrt{\frac{(1+\epsilon)(n)(n-1)(n+1)}{3}}$:

$$\sigma \geq \frac{(k/2)^2}{n} = \frac{k^2}{4n} > \frac{(1+\epsilon)(n-1)(n+1)}{12}$$

Cas 2 : $\omega(\beta, i_2) = k$ et tout le reste des arêtes ont un poids 1. Ceci implique que $i_n = \alpha$ ou $i_n = \gamma$ car leurs arêtes incidents sont les seules qui relient à β avec un poids 1. Sans perte de généralité on suppose que $i_n = \alpha$. Puisque γ est un sommet dans le cycle, il doit y avoir deux sommets distincts incidents u et v et $v \neq \beta \neq u$ car β est déjà dans le cycle et $u \neq i_2$ car (β, γ) n'est pas dans le cycle. Selon la construction de G_2 il est évident que (β, γ) et (γ, δ) sont les seules qui ont un poids 1 est incidents à γ . Il n'y a que δ qui disponible alors un cycle avec seulement $\omega(\beta, i_2) = k$ et le reste des arêtes de poids 1 est impossible.

Cas 3 : $\omega(\beta, i_2) = k$ et $\omega(i_j, i_{j+1}) = k$. Dans ce cas $Y_2 = k$ et $Y_{n+1} \geq 2k + n - 2$. Si $\bar{Y} \geq 3k/2$ alors $|Y_2 - \bar{Y}| \geq k/2$. Si $\bar{Y} < 3k/2$ alors $|Y_{n+1} - \bar{Y}| \geq k/2$. Par la suite $\sigma > \frac{(1+\epsilon)(n+1)(n-1)}{12}$ (voir Cas 1).

Ceci prend en considération tous les cas dans lesquels G_1 n'est pas Hamiltonien. □

Il est maintenant facile de voir que pour obtenir une approximation sur la variance revient à une réduction du problème de chercher un cycle Hamiltonien, et alors le problème ϵ -approximatif du voyageur des commerces est NP-complet. □

2.2 Conclusion

Pour résumer, tout algorithme ϵ -approximative du TSP est NP-complet. Ceci à été établi en réduisant le problème à un problème prouvé d'être NP-complet. Autrement dit, si un algorithme peut approximer la solution du TSP pour n'importe quelle graphe, il est au moins assez difficile que résoudre le problème : Est-ce que ce graphe contient un cycle Hamiltonien? La solution de cette question n'est pas obtenue dans un temps polynomial et donc soit l'algorithme approximative ne résolve pas le problème dans un temps polynomial ou bien il n'est pas ϵ -approximative.

2.3 Classification du voyageur de commerce

Le problème du voyageur de commerce peut être considéré sous plusieurs classes de complexité selon le problème posé. La version d'un problème de décision : est-ce qu'il existe une solution pour un graphe $G(V, A)$? revient à demander s'il existe un cycle Hamiltonien, car s'il existe au moins un cycle alors il y aura une solution. Cette question est donc en classe NP-complet (elle peut être réduite à un problème NP-complet). La version de la question avec un seuil : Est-ce qu'il existe une solution avec un coût total inférieur à x ? Cette formulation place le problème aussi dans la classe NP-complet, il est facile de vérifier le poids d'un cycle Hamiltonien (somme des arêtes) et donc la vérification d'une solution est faite rapidement.

Par contre, la version « complète » : Trouver le cycle Hamiltonien le moins lourd, est placé dans NP-difficile. Premièrement, ce problème n'est plus un problème de décision, c'est un problème de fonction. Ensuite, trouver une solution est au moins assez difficile que trouver un cycle Hamiltonien, donc le problème ne peut pas être résolu dans un temps polynomial (supposant que $P \neq NP$). En plus, il n'existe pas de méthode rapide pour vérifier si un cycle Hamiltonien dans le graphe est en fait le cycle le moins lourd ; une solution donnée ne peut pas être vérifiée dans un temps polynomial.

3 Résoudre le problème du voyageur de commerce

Dans cette section différents algorithmes pour résoudre exactement ou approximativement le TSP sont présentés.

3.1 Algorithme de Held-Karp

L'algorithme Held-Karp est un algorithme de programmation dynamique développé par Held et Karp [2] et aussi indépendamment par Bellman [1] en 1962 qui résout exactement le TSP avec un ordre de complexité en temps de $\Theta(2^n n^2)$ et en espace de $\Theta(2^n n)$. Pour référence, une recherche exhaustive d'une solution dans un graphe complet est d'ordre de temps $O(n!)$ est d'espace $O(n^2)$.

Programmation dynamique

Le principe de la programmation dynamique est de transformer un problème en sous-problèmes et de les résoudre récursivement en utilisant la mémorisation (mémoriser les valeurs de retour les résultats intermédiaires) pour diminuer la complexité en temps (c'est compensé par une augmentation de l'espace nécessaire par contre). Ce domaine a été développé dans les années 1950s par Richard Bellman.

Un exemple simple pour calculer un terme dans la suite de Fibonacci qui est défini comme :

$$F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2) \quad n \geq 2$$

un programme informatique peut tout simplement traduire cela en code. Cette implémentation est inefficace car le temps d'exécution de cet algorithme est exponentiel (ordre $O(2^n)$) car le programme calcul $F(x)$ pour le même x intermédiaire plusieurs fois. Par contre, sauvegarder la valeur de x et la renvoyer si besoin améliore beaucoup le temps d'exécution (ordre $O(n)$, chaque terme de la suite est calculé seulement une fois).

Revenant à l'algorithme, il est divisé en deux parties. Pour référence les sommets sont numérotés arbitrairement entre 1 et n . La première partie est de trouver la chaîne la moins lourde qui commence à 1 et termine au sommet x avec $x \in \{2 \dots n\}$ qui passe par tout le reste des sommets $\{2 \dots n\} \setminus x$. On définit $S \subseteq \{2 \dots n\}$, $d(a, b)$ le poids de l'arête (a, b) , $g(S, e)$ comme le coût minimal de la chaîne qui commence à 1 et termine à e en passant seulement par tout les sommets dans S , $k = |S|$. Il y a alors le cas trivial et le cas récursif dans l'algorithme :

1. Cas trivial : Si $S = \emptyset$ alors $g(S, e) = d(1, e)$

2. Cas récursif : Si $S \neq \emptyset$ alors soit $S_i = S \setminus s_i$ avec s_i un sommet arbitraire dans S . Alors :

$$g(S, e) = \min_{1 \leq i \leq k} g(S_i, s_i) + d(s_i, e)$$

Il est évident que s_i est l'avant dernier sommet dans la chaîne. Simplement dit, l'algorithme évalue le coût de la chaîne la plus courte qui se termine par s_i et ajoute $d(s_i, e)$, pour chaque sommet s_i différent. Ensuite, il choisit le plus optimal et sauvegarde 2 données : s_i de la chaîne la plus courte et le prix total $g(S, e)$. Sauvegarder s_i est utile pour reconstruire la chaîne : il suffit de retracer les sommets avant derniers qui donneront alors la chaîne la plus courte.

L'algorithme exécute ces étapes par ordre k croissant (calculer $g(S, e)$ pour tout les S de taille 1, ensuite de taille 2, etc.). Chaque calcul de $g(S, e)$ se fait en k étapes (la valeur de chaque $g(S_i, s_i)$ est déjà sauvegardée et il ne suffit que de reprendre cette donnée de la mémoire car $|S_i| < |S|$).

Après avoir calculé tout les $g(S, e)$ avec S de taille $n - 2$ (dans ce cas l'algorithme a trouvé la chaîne la moins lourde qui commence à 1 et qui passe par tout les sommets et termine à e pour chaque $e \neq 1$), la solution est de poids G avec $S = \{2 \dots n\}$:

$$G = \min_{e \in S} g(S \setminus e, e) + d(e, 1)$$

3.1.1 Complexité en espace

L'algorithme a besoin de l'espace pour stocker chaque $g(S, e)$ (et éventuellement s_i mais c'est juste un facteur de 2). Donc il suffit de calculer le nombre de paire $g(S, e)$ possible pour estimer la complexité en espace. Pour les S de taille k , il existe $\binom{n-1}{k}$ combinaisons différentes et pour chaque il y a $n - 1 - k$ choix différents pour e . Alors le nombre de paire $g(S, e)$ sont (pour k variant entre 0 et $n - 2$, il faut au moins 1 sommet comme choix pour e) :

$$\begin{aligned} \sum_{k=0}^{n-2} (n-1-k) \binom{n-1}{k} &= \sum_{k=0}^{n-2} (n-1-k) \times \frac{(n-1)!}{k!(n-1-k)!} \\ &= \sum_{k=0}^{n-2} (n-1) \times \frac{(n-2)!}{k!(n-2-k)!} \\ &= \sum_{k=0}^{n-2} (n-1) \binom{n-2}{k} = (n-1) \sum_{k=0}^{n-2} \binom{n-2}{k} \\ &= (n-1) 2^{n-2} \end{aligned}$$

L'algorithme a donc une complexité d'espace de $\Theta(n2^n)$. Par contre, si seulement le coût minimal est souhaité et pas le cycle, il est pratique de réutiliser les espaces mémoire pour

chaque k croissant, car il ne faut que savoir $g(S, e)$ pour S de taille k pour calculer $g(S, e)$ de taille $k + 1$.

3.1.2 Complexité en temps

L'algorithme prend du temps pour calculer chaque $g(S, e)$ pour un S de taille k . Ce temps est proportionnel à la taille de S car il évalue tout les $g(S', e')$ pour chaque $e' \in S$ et donc de même façon que le calcul de la complexité d'espace :

$$\begin{aligned}
\sum_{k=1}^{n-2} k(n-1-k) \binom{n-1}{k} &= \sum_{k=1}^{n-2} k(n-1-k) \times \frac{(n-1)!}{k!(n-1-k)!} \\
&= \sum_{k=1}^{n-2} (n-1)(n-2) \times \frac{(n-3)!}{(k-1)!(n-3-(k-1))!} \\
&= \sum_{k=1}^{n-2} (n-1)(n-2) \binom{n-3}{k-1} = (n-1)(n-2) \sum_{k=1}^{n-2} \binom{n-3}{k-1} \\
&= (n-1)(n-2) 2^{n-3}
\end{aligned}$$

L'algorithme a donc une complexité de temps de $\Theta(n^2 2^n)$.

Références

- [1] Richard BELLMAN. “Dynamic Programming Treatment of the Travelling Salesman Problem”. In : *J. ACM* 9.1 (1962), 61–63. ISSN : 0004-5411. DOI : [10.1145/321105.321111](https://doi.org/10.1145/321105.321111). URL : <https://doi.org/10.1145/321105.321111>.
- [2] Michael HELD et Richard M. KARP. “A Dynamic Programming Approach to Sequencing Problems”. In : *Journal of the Society for Industrial and Applied Mathematics* 10.1 (1962), p. 196-210. DOI : [10.1137/0110015](https://doi.org/10.1137/0110015). eprint : <https://doi.org/10.1137/0110015>. URL : <https://doi.org/10.1137/0110015>.
- [3] Richard M. KARP. “Reducibility among Combinatorial Problems”. In : Boston, MA : Springer US, 1972, p. 85-103. DOI : [10.1007/978-1-4684-2001-2_9](https://doi.org/10.1007/978-1-4684-2001-2_9).
- [4] Sartaj SAHNI et Teofilo GONZALEZ. “P-Complete Approximation Problems”. In : *J. ACM* 23.3 (1976), 555–565. ISSN : 0004-5411. DOI : [10.1145/321958.321975](https://doi.org/10.1145/321958.321975). URL : <https://doi.org/10.1145/321958.321975>.