Rapport du Voyageur de Commerce

Jonah TURNER
TPA11 IAFA

Recherche Métaheuristique

La recherche métaheuristique a été mise en œuvre dans traveling_salesperson.py en Python. J'ai utilisé la bibliothèque externe Numpy pour générer les séquences aléatoires des chemins initiaux. Toute version récente de Numpy utilisant Python 3.8 ou ultérieur sera capable d'exécuter le code.

Le script Python implémente l'escalade la plus raide et la recherche avec liste taboue. Le script effectue d'abord une itération de l'escalade la plus raide puis itère sur le même algorithme max_essais fois, qui peut être passé en tant qu'argument de ligne de commande. L'algorithme d'escalade la plus raide utilise à la fois 2-opt et 2-swap pour trouver des voisins, tandis que la liste taboue utilise seulement 2-swap.

Le temps d'exécution peut varier considérablement en fonction du nombre d'itérations effectuées, mais des itérations accrues sur les fichiers de données plus importants produisent souvent de meilleurs résultats. La recherche taboue est moins coûteuse à exécuter que d'itérer sur l'escalade la plus raide, donc le paramètre d'itération maximale par défaut est beaucoup plus élevé.

Programme ZIMPL

L'ensemble V représente les villes dans le TSP. Les arêtes, notées par E, symbolisent les chemins potentiels entre ces villes et sont définies pour les paires de villes où la première est inférieure à la seconde. Le programme utilise en outre un ensemble de puissance P[] de V, ainsi qu'un ensemble d'indices K, pour faciliter la définition des contraintes. Les coordonnées de chaque ville sont stockées dans les paramètres px[V] et py[V]. Les variables de décision x[E] sont binaires, indiquant la présence ou l'absence d'une arête dans le tour. La fonction objectif cherche à minimiser la distance totale du tour, calculée comme la somme des distances euclidiennes pour chaque arête sélectionnée. Deux contraintes critiques sont imposées : two_connected, qui assure que chaque ville est connectée exactement à deux autres, formant un tour continu, et no_subtour, qui empêche la formation de tours plus petits et isolés en limitant le nombre d'arêtes dans tout sous-ensemble de villes. Ce modèle vise à identifier le tour le plus court qui visite chaque ville exactement une fois et retourne au point de départ, en adhérant aux contraintes stipulées pour assurer une solution de tour valide et complète.

La solution aurait dû inclure des contraintes supplémentaires pour limiter la complexité de la recherche comme la stratégie d'élimination subtile Miller-Tucker-Zemlin (MTZ), mais une mise en œuvre fonctionnelle n'a pas été réalisée.

Resultats

Le programme ZIMPL n'a malheureusement pas pu lire les données tsp50 sans manquer de mémoire, donc les meilleurs résultats ont été obtenus avec la solution Python.

Comme une graine spécifique n'a pas été définie, les résultats sont quelque peu aléatoires, en particulier pour les instances avec plus de villes. Une exécution performante produit les résultats suivants pour tsp50.txt en utilisant la recherche Tabou :

Chemin - [21, 23, 25, 48, 18, 19, 49, 20, 22, 24, 9, 13, 15, 16, 47, 17, 14, 12, 11, 10, 0, 33, 30, 32, 28, 26, 27, 29, 31, 34, 41, 39, 38, 37, 35, 36, 40, 43, 44, 42, 1, 3, 5, 45, 4, 46, 2, 6, 7, 8]

Distance - 473.32

Resultats ZIMPL

	Optimum Reached	Value Obtained	Time
Tsp5	Yes	194.04	0.0s
Tsp25	No	OOM	~
Tsp50	No	OOM	~
Tsp101	No	OOM	~

Résultats Python Metaheuristics (max_essais = 10, tabou_iters = 200)

	Value Obtained	Best Method	Total Time(s)
Tsp5	194.04	SHC	0.002
Tsp25	279.99	Iterated SHC	1.27
Tsp50	446.47	Tabou	10.05
Tsp101	777.822	Tabou	40.53

Distance signalée pour Tsp25 inférieure à l'optimal suggéré sur Moodle, le chemin trouvé est le suivant (indexé à 0):
[21 23 18 19 24 22 20 0 2 3 5 1 4 8 7 6 14 12 17 11 10 9 13 15 16]

Version française traduite avec l'aide de ChatGPT

Référence ZIMPL: https://zimpl.zib.de/download/zimpl.pdf