Rapport de projet

Mustapha Haouassi Olivier Provost

1 Introduction

L'objectif du TP est de proposer des implémentations de deux algorithmes qui permettent de résoudre le problème de TSP-symétrique : L'algorithme de Rosenkrantz, Stearns et Lewis(RSL) et l'algorithme de Held et Karp(HK).

Les directives principales du TP sont les suivantes :
- Implémenter l'algorithme de Rosenkrantz, Stearns et Lewis(RSL).
- Implémenter l'algorithme de Held et Karp(HK).
- Tester les différentes implémentations et exhiber la meilleure tournée à chaque fois.
- Illustrer graphiquement la solution obtenue et exprimer son erreur à par rapport à la solution optimale.

2 Algorithme de Rosenkrantz, Stearns, Lewis

Soit $\mathcal{G} = (\mathcal{V}, \mathcal{E}, p)$ un graphe connexe valué avec pour chaque arête $e \in \mathcal{E}$, p_e le poids de l'arête e . Soit $root \in \mathcal{V}$ un noeud arbitraire. L'algorithme de RSL commence par construire un arbre de recouvrement de coût minimum T de racine $root$. Ensuite, il parcourt l'arbre T obtenu en profondeur. L'ordre de visite des sommets dans la solution de l'algorithme de RSL est la même que celle retournée par le parcours en profondeur (en enlevant les sommets déjà visités à chaque fois).

3 Implémentation de l'algorithme RSL

L'implémentation de l'algorithme de KSL fait appel à un algorithme de recherche d'arbre de recouvrement minimum ainsi qu'un algorithme de parcours en profondeur.

3.1 Algorithme de recherche d'arbre de recouvrement minimum

Pour trouver l'arbre de recouvrement minimum, on a fait appel à l'algorithme de Kruskal et l'algorithme de Prim. La comparaison entre les temps de calcul sont donné dans le tableau suivant :

3.2 Algorithme de parcours en profondeur

Implémentation de dfs_visit() : Pour implémenter le parcours en profondeur, on a modifié l'implémentation de la fonction $dfs\_visit()$ du fichier $dfs.py$ qui permet de faire un parcours en profondeur, en ajoutant à chaque fois un sommet qui est visité pour la première fois. Quand un sommet *parent* a plusieurs $fils$, on a décidé d'aller visiter celui qui a le plus petit indice (On a décidé d'utiliser cette règle pour permettre la répétabilité de nos résultats).

Implémentation de _visited : Pour la fonction dfs_visit, l ?attribut $\_visited$ a été donc ajouté à la classe *primnode* et à la classe $unionfind$ pour permettre son utilisation. L ?attribut $\_visited$ prend la valeur $False$ par défaut. Lorsque le n ?ud est visité, la valeur est changée à $True$ par le setter $set\_visited()$. Il est aussi possible de savoir si le n ?ud a été visité à l ?aide du getter $get\_visited()$.

Implémentation de get_neighbors(node) : Pour la fonction dfs_visit, nous avions aussi besoin de la fonction $get\_neighbors(node)$ qui est implémenté dans la classe *graphe* qui permet de retourner tous les n ?uds voisins d ?un n ?ud choisi dans le graphe en question.

Implémentation de la fonction l'algorithme de RSL : À l ?aide des anciens TP et de la fonction dfs_visit, nous avions tout pour faire cette fonction. L ?implémentation de l ?algorithme nécessite d ?importer *Prim*, *Kruskal*, dfs_visit, *Graph*, *Edge*. Dans le fichier *rsl.py*, on a implémenté la fonction $def\ Rsl(G,\ root = None,\ choice =' Prim')$. La fonction donne le choix à l ?utilisateur entre l ?algorithme de Prim et l ?algorithme de Kruskal. Si aucun n ?est choisi, l ?algorithme de Prim sera choisi par défaut. Suite à cela, un arbre de recouvrement minimal est créé à partir du graphe G et du choix de l ?algorithme. Avec cet arbre de recouvrement minimal, un parcours en pré-ordre est exécuté sur celui-ci, si aucune racine n ?est indiquée, la racine sera le premier n ?ud de la liste de n ?uds du graphe G. Le parcours est donc exécuté à partir de la racine et une suite de n ?uds est retournée. À l ?aide de cette suite de n ?uds un nouveau graphe qui correspond à la tournée est créé ce qui permet d ?obtenir le coût de la tournée en question et ce qui permet aussi de le tracer de manière graphique.

4 Algorithme de Held et Karp (HK)

Avant d'exposer l'algorithme de Held et Karp, on aura besoin d'introduire les définitions suivantes :

Définition 1. *Étant donné $\mathcal{G}$ graphe connexe et $v_0 \in \mathcal{G}$ un sommet du graphe. Soit $\mathcal{T} = (\mathcal{V}, \mathcal{E}_T)$ avec $\mathcal{E}_T \subset \mathcal{E}$. $\mathcal{T}$ est un $1 - tree$ si :*

- *$T - v_0$ est un arbre*
- *il existe deux arêtes distinctes $e_1, e_2 \in E_T$ tel que v_0 est incident à e_1 et e_2*

De plus, $\mathcal{T}_{opt}$ est de poids minimum si pour tout $\mathcal{T}$ $1 - tree$ $p(\mathcal{T}_{opt}) \leq p(\mathcal{T})$

Remarque 1. *Un tour est un $1 - tree$ dans lequel tous les sommets sont de degrés 2. De plus, un tour de longueur minimum est un $1 - tree$ de poids minimum dans lequel tous les sommets sont de degrés 2*

Proposition 1. *Si un $1 - tree$ de longueur minimum a tous ces sommets de degrés 2. Alors, c'est une solution optimale pour le problème de TSP-symétrique.*

Proposition 2. *Pour construire un $1 - tree$ $\mathcal{T}$ de longueur minimum, il suffit de :*

- *construire un arbre de recouvrement minimum sur les sommet $\mathcal{V} - v_0$ et l'ajouter à $\mathcal{T}$*
- *Ajouter à $\mathcal{T}$ v_0 et les deux arêtes distinctes incidentes à v_0 de poids minimum.*

Proposition 3. *Soit p une valuation des poids des arêtes d'un graphe $\mathcal{G}$ (i.e. $p_{i,j}$ $\forall (i,j) \in \mathcal{E}$). Soit π_i $(\forall v_i \in \mathcal{V})$ une suite de scalaire. Le problème de TSP avec les poids $p_{i,j}$ et le problème de TSP avec les poids $p_{i,j} + \pi_i + \pi_j$ ont des solutions optimales équivalentes.*

Proposition 4. *Le problème de $1 - tree$ avec les poids $p_{i,j}$ et le problème de $1 - tree$ avec les poids $p_{i,j} + \pi_i + \pi_j$ n'ont pas de solutions équivalentes.*

Définition 2. *On appelle le vecteur π le vecteur des pénalités. la valeur π_i correspond donc à la pénalité qu'on donne au sommet v_i selon un critère donné.*

L'idée de la procédure de descente de Held et Karp est de trouver une valuation des pénalités $\pi_i \; \forall v_i \in \mathcal{V}$ de telle sorte à ce que la solution optimale du problème de $1-tree$ donne tous les sommets de degrés 2. Pour ceux, on applique une méthode de descente où on met à jour à chaque itération le vecteur des pénalités π jusqu'à ce que tous les sommets du $1-tree$ calculé avec ce vecteur soit de degrès 2. Les instruction de l'algorithme de Held et Karp sont les suivantes :

Algorithm 1 Procédure de descente de Held et Karp

Require: $k = 0$, $\pi^0 = 0$, $W = -\infty$ (k un compteur, π^k le vecteur des pénalités à chaque itération, W le vrai poids du $1-tree$ courant)
Ensure: $\mathcal{T}$
1: Trouver un minimum $1-tree$ T_k avec les poids modifiée $(p_(i,j) + \pi_i^k + \pi_j^k)$
2: Calculer $w(\pi^k)$ le poids du $1-tree$ dans le graphe de départ
3: Mettre $W = max(W, w(\pi^k))$
4: Mettre $v_k = d_k - 2$ où d_k sont le vecteur des degrès de chaque sommet dans $\mathcal{T}_k$.
5: **if** $(v_k == 0)$ **then**
6:     On est dans un tour optimal $(\mathcal{T} = T_k)$. STOP
7: **end if**
8: Choisir un pas de temps t_k
9: mettre à jour le vecteur π avec $\pi^{k+1} = \pi^k + t_k \cdot v_k$
10: $k = k + 1$ et retourner à 1

Remarque 2. *En pratique, la méthode de descente de Held et Karp prend beaucoup de temps à converger (explosion du nombre d'itérations pour le trouver). L'idée est de se rapprocher le plus possible d'un* $1-tree$ *où la plus part des sommets sont de degrès* 2 *en mettant un critère d'arrêt. Une procédure permettra ensuite de transformer le* $1-tree$ *retourné par l'algo de Held et Karp en un cycle (Solution pas forcément optimale).*

Dans la section suivante, on donnera les détails de notre implémentation

5 Implémentation de l'algorithme

5.1 Methode de descente de Held et Karp

Pour l ?algorithme de HK plusieurs attributs ou fonctions ont été ajoutés dans les classes et plusieurs fonctions ont été créées. Nous commencerons par les attributs, suivi par les fonctions dans les classes et suivi des fonctions.

Implémentation de _pi et _olddeg Pour l ?algorithme de HK nous avons besoin des π des n ?uds pour modifier les coûts des arrêtes du graphe. L ?attribut *_pi* est donc rajouté à *primnode* et à $unionfind$. Un setter a été ajouté pour changer cette valeur et un getter a été ajouté pour obtenir la valeur de cet attribut. L ?attribut *_olddeg* est l ?ancien degré du n ?ud, car entre chaque itération de 1-tree les dégrées des n ?uds peuvent changer. Cet attribut sert à une variante de l ?algorithme de HK qui utilise aussi l ?ancien dégrées du n ?ud pour effectuer la descente.

Implémentation de get_vcost Cette fonction permet de retourner le coût d ?une arrête modifié selon l ?algorithme de HK (*i.e.* $p_ij + \pi_i + \pi_j$). Cela permet d ?obtenir les 1-tree avec le plus petit coût modifié. Ce getter a aussi été implémenté sur les graphes pour retourner leur coût modifié.

Modification de la classe graph Pour la classe *graph*, deux attributs ont étés ajouté qui servent à l ?algorithme de HK. Le premier est l ?attribut *_pi* qui renvoie un vecteur contenant tous les π des n ?uds contenues dans le graphe. Un attribut *_v* a aussi été ajouté, cet attribut contient un vecteur et chaque élément de ce vecteur correspond au degré moins 2 d ?un n ?ud. Cela correspond à un des critères d ?arrêt de l ?algorithme de HK.

La fonction *Delete_node*(*node*) sert à la création des 1-tree. La fonction va donc, pour un n ?ud donné par l ?utilisateur, effacé toutes les arêtes lui étant incidente et va effacer le n ?ud lui-même. À partir du graphe obtenu suite à cette modification, un arbre de recouvrement pourra être fait pour le 1-tree.

La fonction *delete_edge*(*edge*) est similaire à la fonction précédente, mais pour deux n ?uds donnés par l ?utilisation, la fonction effacera l ?arête du graphe si elle existe.

La fonctione *get_edge_copy*() sert à retrouver une arête dans la copie d ?un graphe. Si un graphe est attribué un graphe a une variable, si on modifie la variable cela modifiera le graphe original et vice-versa. Des copies sont donc effectuées pour garder le graphe original sans changement. Mais lors de la copie des graphes, les numéros de références des n ?uds par exemple se voient modifier. Les deux objets ne sont donc plus les mêmes selon python. La fonction *get_edge_copy*() retournera l ?arrête correspondante à l ?aide des numéros d ?identification des n ?uds choisis par l ?utilisateur.

Finalement, on a définie dans un fichier *hkcycle* les fonctions suivantes :

— la fonction $def\ Otree(graph_ini,\ position = 0,\ choice =' Prim')$ qui prend en entrée un graphe, la *position* dans laquelle se trouve le sommet *root* dans la liste de sommets du graphe ainsi que le choix de l'algorithme d'arbre de recouvrement minimum utilisé. Elle retourne le $1 - tree$ de coût minimum.

— la fonction $def\ Hkcycle(graph_ini,\ position,\ t,\ period,\ choice)$ prend en entrée le graphe initial de notre problème, la position du *root* des $1-tree$ qui seront calculés, un pas de temps t, le nombre d'itérations maximum de l'algorithme et le choix de l'algorithme utilisé pour la recherche de arbre de recouvrement de poids minimum utilisé. L'algorithme commence par calculer un $1 - tree$ et met à jour W avant d'entrer dans la boucle principale. Ensuite, il répète les instructions de l'algorithme jusqu'à ce qu'on tombe sur une solution optimale de notre problème ou que le nombre d'itérations maximum soit atteint.

Remarque 3. *Deux choix importants de l'algorithme sont le critère d'arrêt et le pas de temps t. On a choisi de faire comme critère d'arrêt un nombre d'itération maximum pour assurer une borne supérieur sur le temps de notre algorithme. En ce qui concerne le pas de temps, on a pris des pas de temps fixes entrée en entrée. La stratégie qui motivait cette décision est que nous voulions une consommation de temps modéré pour cette étape pour pouvoir faire de la post-optimisation après. Ceci est du au fait qu'on a constaté que pour certaines instances, l'amélioration de la solution par l'algorithme de Held et Karp était relativement médiocre par rapport au temps consommé.*

6 Algorithme de construction du cycle

A la sortie de l'algorithme de Held et Kapr, on dispose d'un $1 - tree$ $\mathcal{T}$ avec des sommets potentiellement de degré différent de 2. On cherche alors à construire un cycle à partir de $\mathcal{T}$. L'algorithme démarre d'un cycle vide et d'un sommet $v_initial$ en entée et ajoute des sommets successivement au cycle jusqu'à ce qu'il contienne tous les sommets du graphe. A chaque fois qu'un sommet est ajouté, il sera automatiquement marqué dans $\mathcal{T}$. L'algorithme à chaque itération cherche à partir d'un sommet $v_{courant}$ un sommet $v_{candidat}$. Au début de l'algorithme le sommet $v_{courant}$ correspond à $v_initial$. la démarche pour trouver le sommet $v_{candidat}$ à partir de $v_{courant}$, marquer les sommets et les ajouter au cycle est la suivant :

1. On marque $v_{courant}$ dans $\mathcal{T}$ et on l'ajoute au cycle. Ensuite à partir de $v_{courant}$, on cherche dans ses voisins non marqué dans $\mathcal{T}$ celui qui le relie à lui avec l'arête de poids le plus faible. deux situations se présentent :
   (a) Il existe au moins un voisin non marqué de $v_{courant}$ dans $\mathcal{T}$. Soit $v_{candidate}$ celui qui le relie à lui avec l'arête de plus faible poids. On met à jour $v_{courant} = v_{candidate}$ et on retourne à 1.
   (b) tous les voisins de $v_courant$ dans $\mathcal{T}$ sont marqué. Dans ce cas, on saute vers un sommet non marqué dans $\mathcal{T}$ (pas forcément voisin de $v_{courant}$ dans $\mathcal{T}$) qui le relie à lui avec l'arête de poids plus faible. en d'autre terme, on cherche dans le graphe initial $\mathcal{G}$, un sommet qui ne soit pas marqué dans $\mathcal{T}$ et qui est relié à $v_{courant}$ avec l'arête de poids plus faible. Soit $v_{candidate}$ ce sommet. On met à jour $v_{courant} = v_{candidate}$ et on retourne à 1.

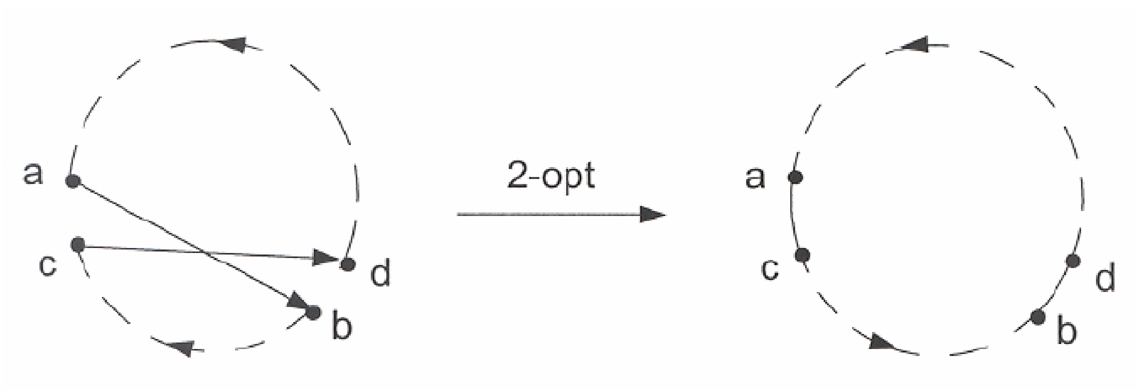
6.1 Implémentation

Pour implémenter notre algorithme de correction de cycle, on a ajouter dans le fichier *hkcycle.py* la fonction $def\ Cycle\_creation(graph,\ graph_ini)$. Elle prend en entrée le one-tree *graph* ainsi que le graphe initial $graph_ini$ et effectue les instructions explicités en haut.

7 Post_optimisation

Il arrive que pour certaines instances de notre problème, l'algorithme retourne des solutions relativement loin de la solution optimale (particulièrement pour l'instance qui ne satisfait pas l'inégalité triangulaire). Pour cela, on a décidé de faire de la post-optimisation.

La post-optimisation a été faite en appliquant l'opérateur $2 - opt$. Il prend en entrée deux arêtes (x, v) et (y, u) dans le cycle (avec $x \neq u \neq y \neq v$) et compare le coût $p_{x,u} + p_{y,v}$ et $p_{x,v} + p_{y,u}$. Si $(p_{x,u} + p_{y,v} < p_{x,v} + p_{y,u})$, alors il échangera les arêtes (x, v) et (y, u) par les arêtes (x, u) et (y, v) si le graphe résultant de cette transformation reste un cycle hamiltonien.



7.1 Implémentation

Pour implémenter la post-opt, on a jouté dans le fichier *hkcycke.py* deux fonction : la fonction $def\ Post_op(graph,\ g_ini)$ prend en entrée un cycle $graph$ et le graphe initial. Pour chaque pair de sommets et chaque pair de voisins de ces sommets, il utilise la fonction $def\ decision(n1,\ n2,\ nei1,\ nei2,\ G,\ g_ini)$ pour tester si l'application d'un $2 - opt$ donne une solution réalisable (cycle hamiltonien à l'aide de dfs) et améliore la solution courante (le poids du cycle). Il met à jour le cycle courant selon le retour de la fonction $def\ decision$.

8 Résultats et commentaire

Dans cette section, nous exposerons des tableaux récapitulatifs des résultats des différentes variantes qu'on a implémenté :

1. Tableau comparant les temps de calculs de RSL avec Prim et Kruskal.

Temps de calcul en s (root = 0)	RSL avec Prim	RSL avec Kruskal
bayg29	0.0230000019073	0.0160000324249
bays29	0.0309998989105	0.0320000648499
brazil58	0.0629999637604	0.0769999027252
brg180	1.06100010872	0.661000013351
dantzig42	0.0460000038147	0.02001211355
fri26	0.0269999504089	0.00000000001
gr17	0.00699996948242	0.00600004196167
gr21	0.0159997940063	0.000000001
gr24	0.0160000324249	0.000651255
gr48	0.135610002333	0.1854420004555
gr120	0.384000062943	0.3952476254319
hk48	0.0469999313354	0.031588800002
pa561	20.75	8.95500016212
swiss42	0.0620000362396	0.0712450000006

2. Tableau donnant la meilleure valeur obtenue par RSL et HK après plusieurs tests : pour RSL, est donné entre parenthèse le sommet racine et l'algorithme pris qui donne cette solution. Pour HK est donné entre parenthèse la racine du 1-tree, le nombre d'itérations de l'algo de HK, le pas choisi et l'algorithme qui donne cette solution.

Meilleur solution obtenu avec	RSL (root, algo)	HK(root ,nb itération, pas)
bayg29	2004(6, prim)	**1610 (1, 1500, 1.5, prim)**
bays29	2471(22, kruskal)	**2020(1, 1500, 0.5, prim)**
brazil58	30521(39, prim)	**25395(16, 750, 0.5, prim)**
brg180	149360(77, kruskal)	**1950(63, 500, 0.5, prim)**
dantzig42	798(20, kruskal)	**699(3, 750, 0.5, prim)**
fri26	1168(0, prim)	**937(0, 1500, 0.5, prim)**
gr17	2316 (14, prim)	**2085 (14, 2000, 0.5, prim)**
gr21	3375(1, prim)	**2707(0, 2000, 0.5, prim)**
gr24	1580(17, prim)	**1272(1, 2000, 0.5, prim)**
Gr48	6476(37, prim)	**5092(44, 750, 1.5, prim)**
Gr120	9593(32, prim)	**7008(117, 500, 1.5, prim)**
hk48	13691(19, kruskal)	**11461(4, 750, 1, prim)**
pa561	3775(557, kruskal)	**2902(10 ,25, 1, prim)**
swiss42	1504(22, kruskal)	**1273(2, 750, 1.5, prim)**

3. Tableau donnant l'erreur relative entre les solutions retournées par chacun des algorithmes dans le tableau précédent et la solution optimale.

Meilleur solution obtenu avec	Erreur relative RSL	Erreur relative HK
bayg29	25%	**0%**
bays29	22%	**0%**
brazil58	20%	**0%**
brg180	76%	**0%**
dantzig42	14%	**0%**
fri26	25%	**0%**
gr17	11%	**0%**
gr21	25%	**0%**
gr24	24%	**0%**
Gr48	28%	**0.9%**
Gr120	38%	**1%**
hk48	19%	**0%**
pa561	37%	5%
swiss42	18%	**0%**

4. Tableau récapitulant les valeurs renvoyées par HK avant et après la post-optimisation avec le temps de calcul dans les deux cas (dans le deuxième cas, seul le temps de la post-optimisation est donné. Pour avoir le temps total, il faut additionner le temps de la post-optimisation avec le temps de HK avant la post-optimisation.

coût et temps de calcul avec HK	Sans post-opt	Avec post-opt
b ayg29	Coût::1987.0 temps:55.8350000381	Coût:1610.0 temps:0.395999908447
bays29	Coût::2102.0 temps:90.8280000687	Coût:2020.0 temps:0.34500002861
brazil58	Coût::29615.0 temps:109.13499999	Coût:25395.0 temps:3.87900018692
Brg180	Coût::17870.0 temps: 978.203999996	Coût::1950.0 temps: 72.376999855
dantzig42	Coût::890.0 temps:59.478000164	Coût:699.0 temps:1.61299991608
fri26	Coût::937.0 temps:40.5380001068	Coût:937.0 temps:0.12299990654
gr17	Coût::2085.0 temps:12.3159999847	Coût:2085.0 temps:0.0369999408722
gr21	Coût:: 2707.0 temps: 10.992000103	Coût:: 2707.0 temps: 0.0629999637604
gr24	Coût::1499.0 temps: 51.7890000343	Coût::1272.0 temps: 0.368000030518
Gr48	Coût:: 6191.0 temps: 72.3580000401	Coût:: 5092.0 temps: 1.55700016022
Gr120	Coût:: 8041.0 temps: 366.867000103	Coût:: 7008.0 temps: 22.268999815
Hk48	Coût:: 13475.0 temps: 73.5260000229	Coût:: 11461.0 temps: 1.4960000515
pa561	Coût::3258.0 temps:982.162000179	Coût:2902.0 temps:4502.84299994
swiss42	Coût::1508.0 temps:103.025000095	Coût:1273.0 temps:1.05499982834

Commentaires

- On remarque que la méthode de RSL en appliquant l'algorithme de Kruskal est globalement plus rapide que celle en appliquant de Prim prend globalement le même temps de calcul.
- On remarque que l'algorithme de RSL demande moins d'efforts de calcul que l'algorithme de HK. Cependant, l'algorithme de HK donne de solutions de meilleurs qualité.
- L'algorithme de HK avec la post-optimisation atteint la valeur optimale pour toutes les instances sauf pa561.tsp
- le dernier tableau nous démontre l'apport de la post optimisation. On voit bien que la post-optimisation améliore considérablement la solution de HK avec un temps de calcul relativement faible (sauf pour pa561.tsp)

9 représentation d'une tournée

Pour représenter les tournée on a utilisé ajouté des points dans le plan selon les coordonnée cartésiennes données des sommets du graphe. On ajoute ensuite une ligne entre chaque pair de sommets reliés par une arête dans le cycle. Et le tout en se basant sur la fonction plot_graph déjà existante dans le fichier python *read_stsp*. Une fonction *plot_graph*2 a donc été crée dans le fichier *read_stsp* pour pouvoir faire des graphes à partir d'instances n'ayant pas de données pour leur noeud.

Pour les instances qui n'ont pas de coordonnée cartésiennes, on a utilisé la méthode statistique de positionnement multidimensionnel (multidimensional scaling) qui permet à l'aide d'une matrice de distance et d'une dimension de l'espace (2 dans notre cas), de donner les coordonnée de chaque point de la matrice de tel sorte à minimiser en valeur absolue la différence entre les distances de la matrice et les vrai distances euclidiennes entre les points. la fonction *cmdscale* sur python permet justement d'appliquer cette méthode. Dans la figure suivante est donné la représentation des tournées de l'instance bayg29 retournées par l'algorithme de KSL et de HK.

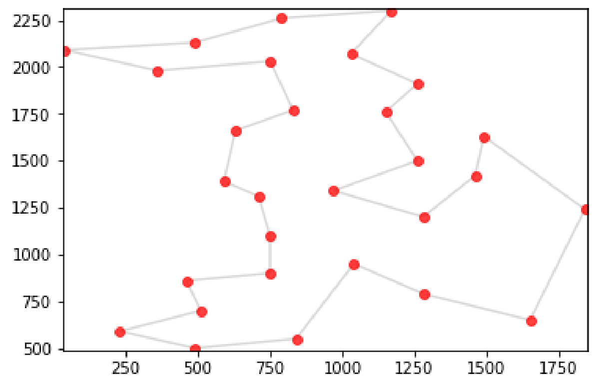


FIGURE 1 – Solution avec HK (optimale) pour l'instance bayg29

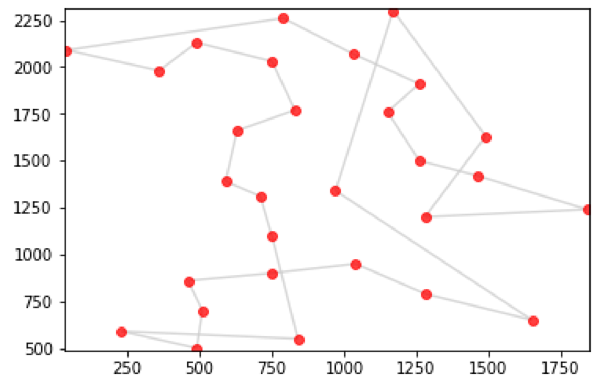


FIGURE 2 – Solution avec RSL pour l'instance bayg29

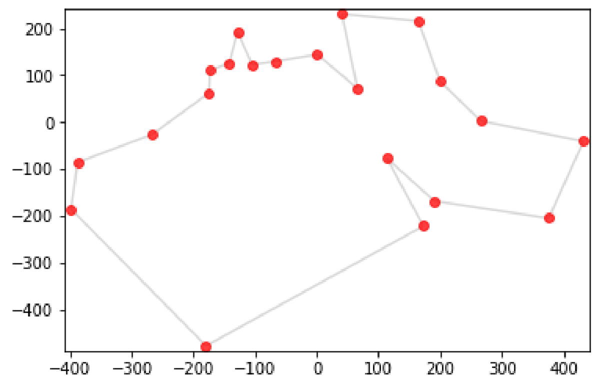


FIGURE 3 – Solution avec HK (optimale) pour l'instance gr21

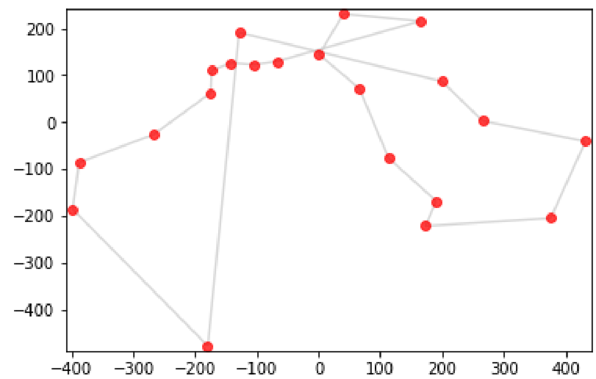


FIGURE 4 – Solution avec RSL pour l'instance gr21

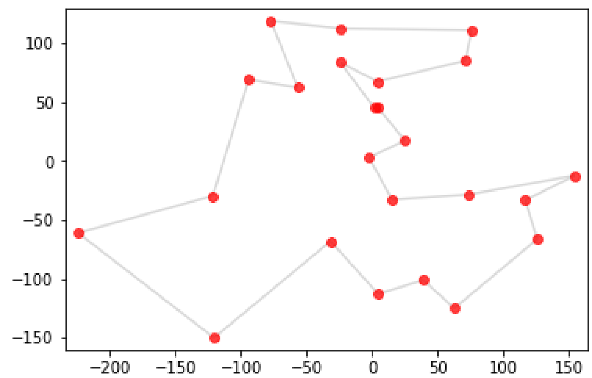


FIGURE 5 – Solution avec HK (optimale) pour l'instance gr24

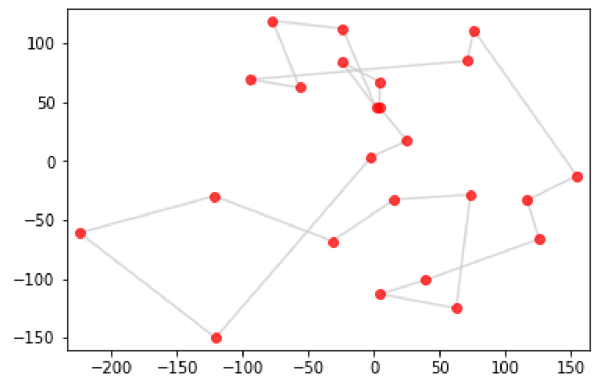


FIGURE 6 – Solution avec RSL pour l'instance gr24

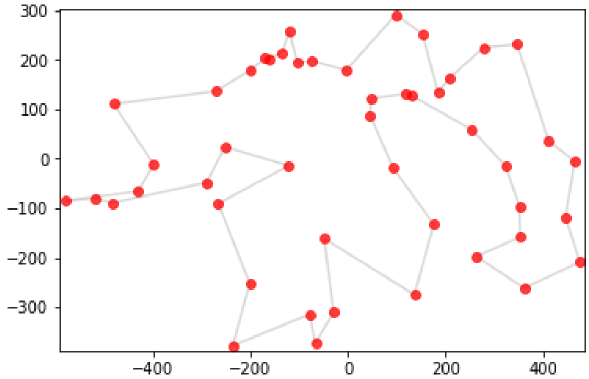


FIGURE 7 – Solution avec HK pour l'instance gr48

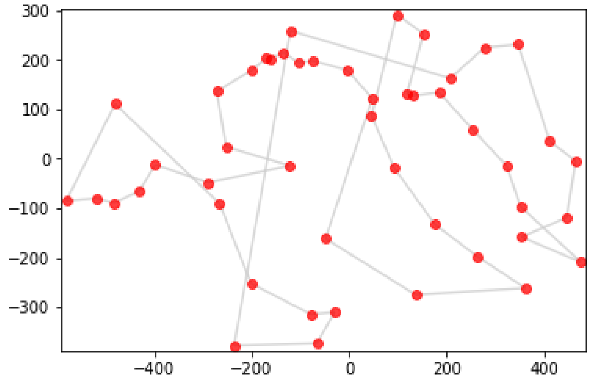


FIGURE 8 – Solution avec RSL pour l'instance gr48

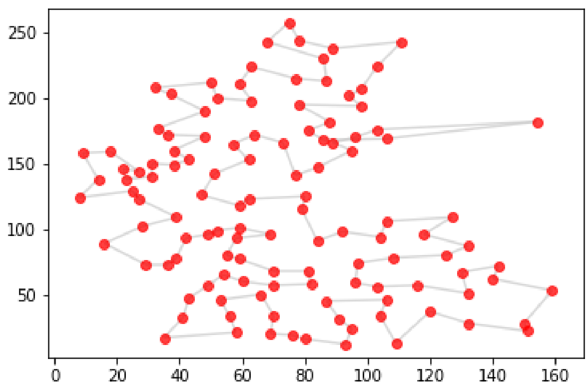


FIGURE 9 – Solution avec HK pour l'instance gr120

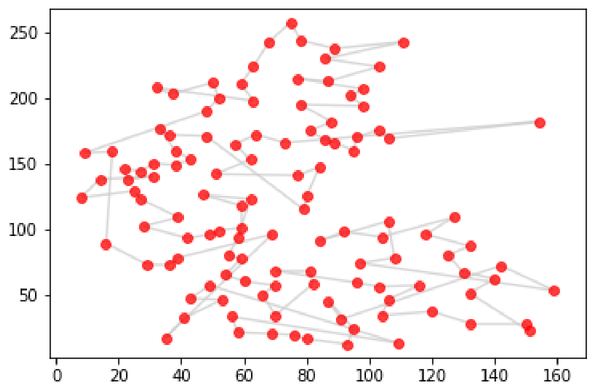


FIGURE 10 – Solution avec RSL pour l'instance gr120

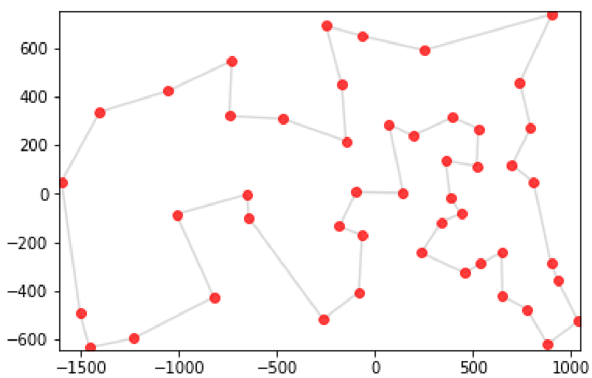


FIGURE 11 – Solution avec HK (optimale) pour l'instance hk48

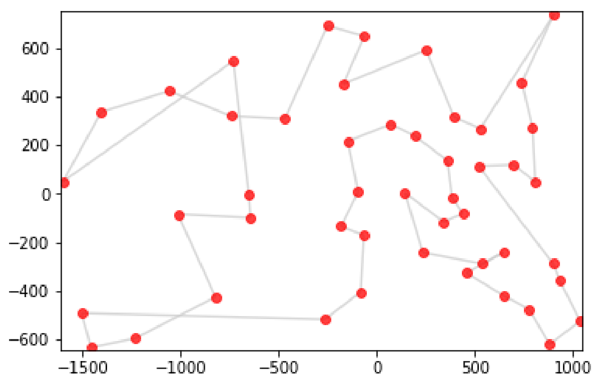
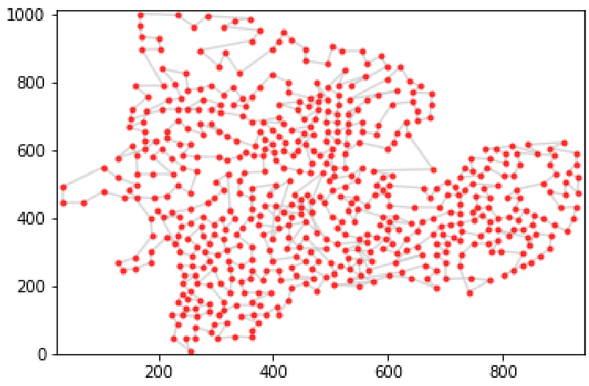


FIGURE 12 – Solution avec RSL pour l'instance hk48



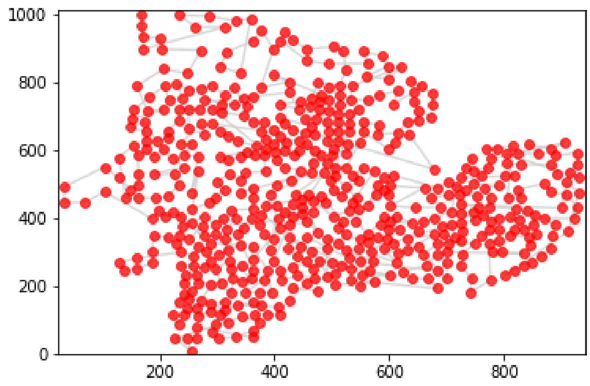
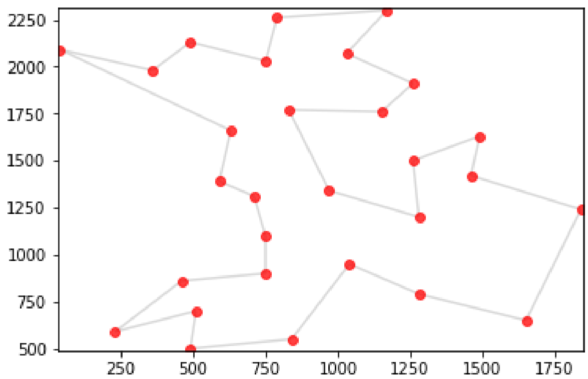


FIGURE 13 – Solution avec HK pour l'instance pa561

FIGURE 14 – Solution avec RSL pour l'instance pa561



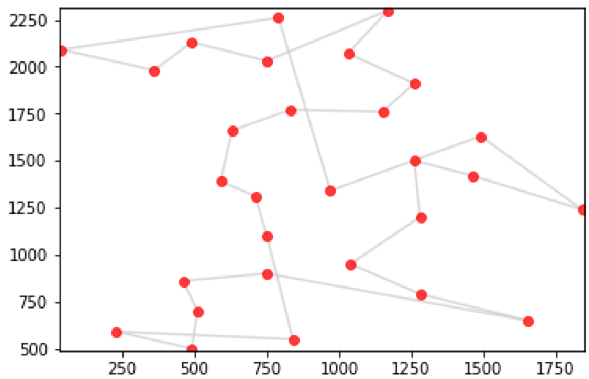
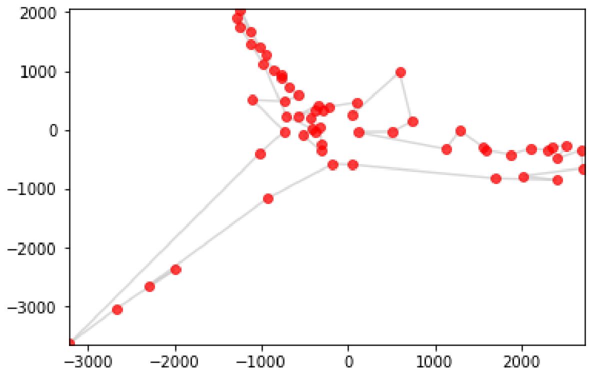


FIGURE 15 – Solution avec HK (optimale) pour l'instance bays29

FIGURE 16 – Solution avec RSL pour l'instance bays29



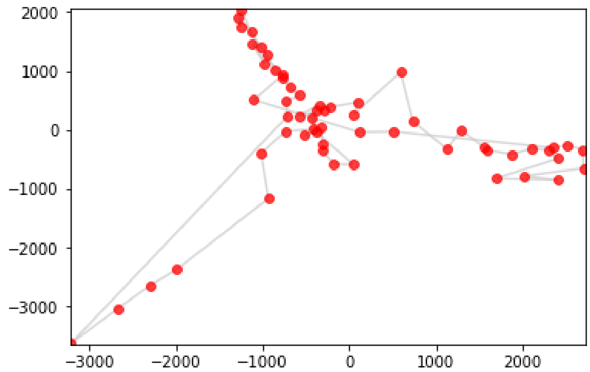


FIGURE 17 – Solution avec HK (optimale) pour l'instance brazil58

FIGURE 18 – Solution avec RSL pour l'instance brazil58

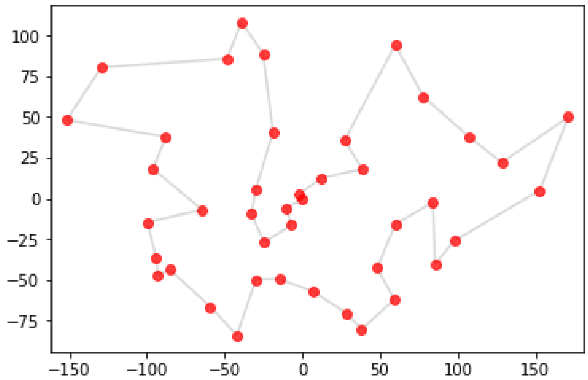


FIGURE 19 – Solution avec HK (optimale) pour l'instance swiss42

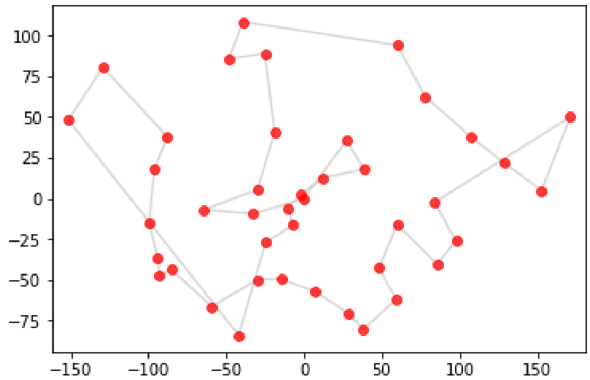


FIGURE 20 – Solution avec RSL pour l'instance swiss42

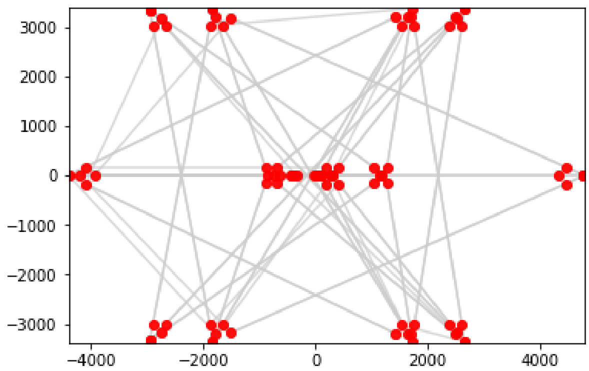


FIGURE 21 – Solution avec HK pour l'instance brg180

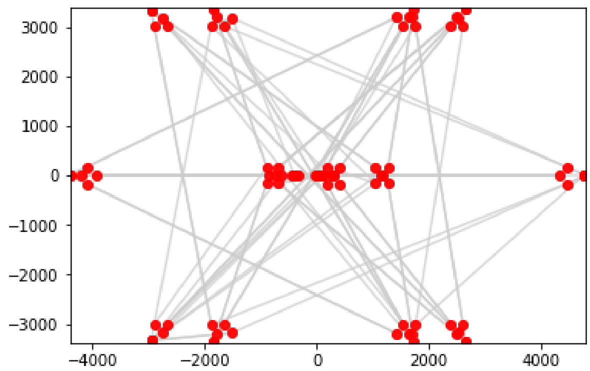


FIGURE 22 – Solution avec RSL pour l'instance brg180

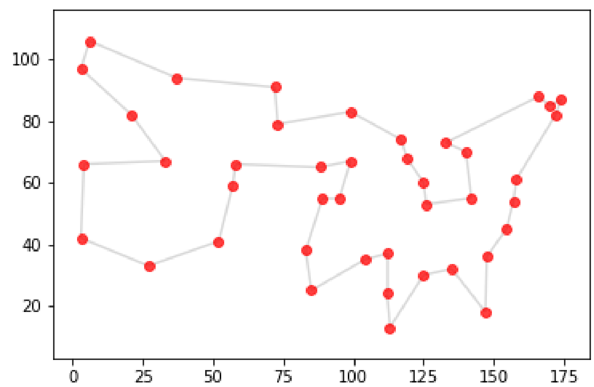


FIGURE 23 – Solution avec HK (optimale) pour l'instance dantzig42

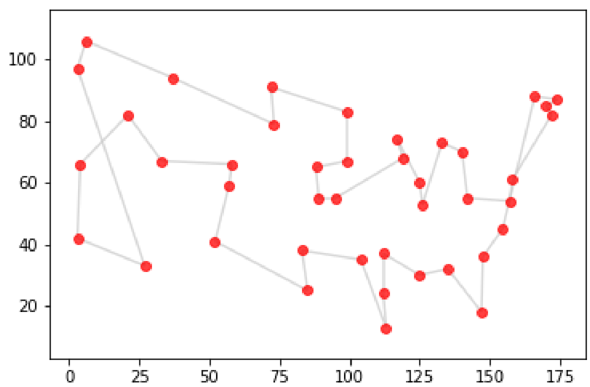


FIGURE 24 – Solution avec RSL pour l'instance dantzig42

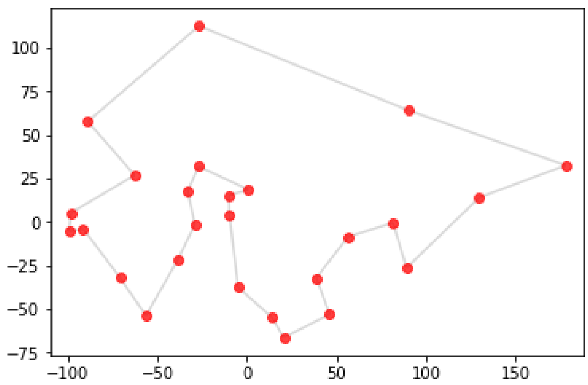


FIGURE 25 – Solution avec HK (optimale) pour l'instance fri26

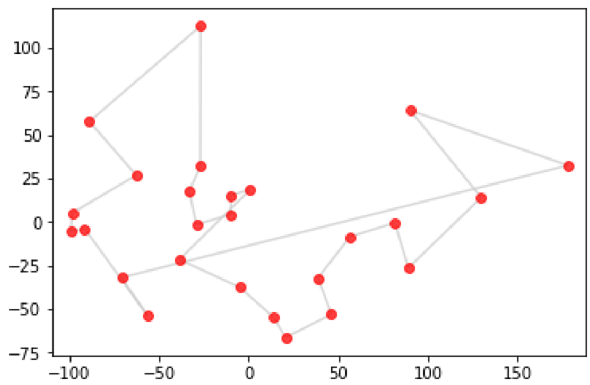


FIGURE 26 – Solution avec RSL pour l'instance fri26

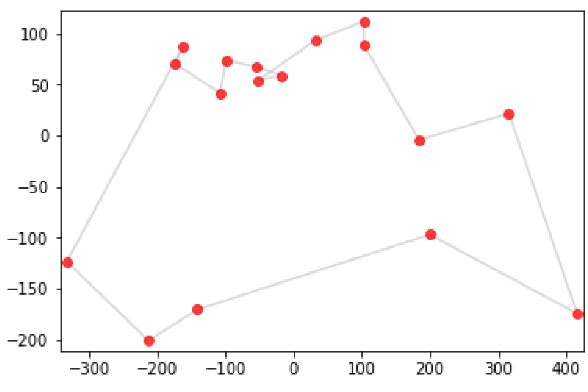


FIGURE 27 – Solution avec HK (optimale) pour l'instance gr17

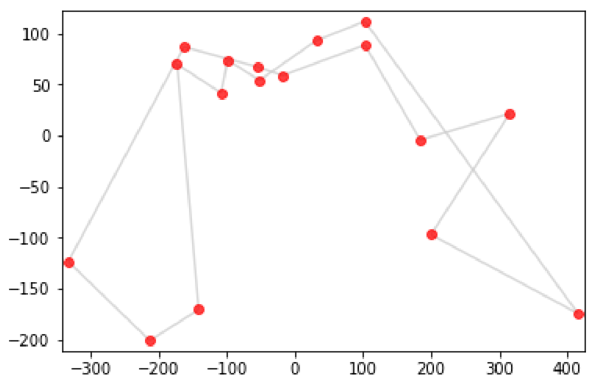


FIGURE 28 – Solution avec RSL pour l'instance gr17