ALEKSANYAN Volodya
COURNIL-RABEUX Clément
LDD3 IM

K-coloriage

Nous avons suivi l'algorithme heuristique vu en cours de k-coloriage et nous avons décidé de l'implémenter en python .

Structure du graphe

La structure d'un graphe est représentée comme un dictionnaire avec en clefs les noeuds et en valeur la liste des noeuds liés Un coloriage est ici représenté comme un dictionnaire associant à chaque noeud une couleur (un nombre entier ici)

Algorithme

Dans l'algorithme, on colorie le noeud de plus haut degré, puis on colorie de la même couleur le + de noeuds possibles n'y étant pas connectés Puis on choisit un nouveau noeud de plus haut degré possible non coloré, et on recommence . Il faut donc garder en mémoire les noeuds déjà coloriés, avoir accès au noeud de plus haut degré et pouvoir vérifier les couleur de ses voisins. Il faut de + pouvoir vérifier que tout les noeuds ont bien été attribué d'une couleur (si le nombre de couleurs utilisés ne dépasse pas le seuil donné), et pouvoir énumérer les noeuds n'étant pas nos voisins . C'est pourquoi nous avons crée les fonctions respectives max_degre,neighbours_colors,is_colored et get_not_connected . L'algorithme est implémenté dans la fonction naive_coloration.

Problème du Voyageur du Commerce

Dans un premier temps, nous avons implémenté l'algorithme naïf vu en cours, qui essaie toutes les permutations améliorant localement le résultat.

Structure du graphe

Le graphe est représenté comme un dictionnaire qui, pour chaque nœud, associe un dictionnaire de nœuds avec leur poids. Les arcs sont supposés être symétriques.
Un chemin commence par un nœud, passe par tous les autres nœuds exactement une fois et finit sur lui-même. Il est représenté par un tableau d'entiers.

Algorithme

Nous avons implémenté un code pour générer un graphe aléatoire ainsi que pour trouver un cycle hamiltonien.
Pour l'algorithme du cours, après avoir déroulé à la main, on trouve le même résultat.

```
def test():
    graph = {1:{      2:12, 3:10,                7:12},
         2:{1:12,       3:8, 4:12                },
         3:{1:10, 2:8,       4:11, 5:3,       7:9 },
         4:{      2:12, 3:11,      5:11, 6:10     },
         5:{            3:3, 4:11,       6:6, 7:7 },
         6:{                 4:10, 5:6,       7:9 },
         7:{1:12,       3:9,       5:7,  6:9      }
         }

    print(([1, 2, 4, 3, 5, 6, 7, 1], 65) == tsp.best_path(graph, [1,2,3,4,5,6,7,1]))
    print(([1, 2, 4, 6, 5, 3, 7, 1], 64) == tsp.loop_over_paths(graph, [1,2,3,4,5,6,7,1]))

    graph = {1 : {2: 1, 4: 7, 5: 13, 6: 17},
             2 : {1: 1, 4: 10, 5: 4, 6: 16},
             3 : {4: 4, 5: 9, 6: 15},
             4 : {1: 7, 2: 10, 3: 4, 5: 1, 6: 4},
             5 : {1: 13, 2: 4, 3: 9, 4: 1},
             6 : {1: 17, 2: 16, 3: 15, 4: 4}
             }

    path = [1, 2, 4, 5, 3, 6, 1]
    print(([1, 2, 5, 4, 3, 6, 1], 42) == tsp.best_path(graph, path))
    print(([1, 2, 5, 3, 4, 6, 1], 39) == tsp.loop_over_paths(graph, path))
```

Nous avons également implémenté l'algorithme connu **2-OPT**, encore une fois en se basant sur le cours, et nous avons remarqué que pour des graphes pas très grands, la différence est minime. De plus, souvent, le résultat est pire, comme le montre l'image ci-dessous :

```
([1, 2, 4, 3, 5, 6, 7, 1], 65)
([1, 2, 4, 6, 5, 3, 7, 1], 64)
([1, 3, 2, 4, 6, 5, 7, 1], 65)
```

Néanmoins, après avoir généré un graphe plus grand, de taille 15, on remarque qu'à chaque fois, en utilisant l'heuristique, on arrive à un meilleur résultat. Ce résultat est bien plus prometteur.

```
1 : {2: 2, 3: 20, 4: 3, 6: 8, 8: 20, 9: 20, 10: 16, 11: 15, 12: 7, 14: 2, 15: 19}
2 : {1: 2, 3: 9, 4: 15, 5: 13, 6: 5, 7: 1, 8: 13, 9: 8, 10: 2, 12: 11, 13: 18, 14: 5}
3 : {1: 20, 2: 9, 6: 9, 7: 14, 8: 3, 9: 7, 10: 5, 11: 17, 12: 11, 13: 15, 15: 10}
4 : {1: 3, 2: 15, 5: 8, 6: 16, 8: 18, 9: 16, 10: 19, 12: 18, 15: 5}
5 : {2: 13, 4: 8, 6: 15, 7: 5, 8: 17, 11: 8, 13: 6, 14: 9, 15: 9}
6 : {1: 8, 2: 5, 3: 9, 4: 16, 5: 15, 7: 20, 8: 6, 10: 3, 11: 14, 12: 4, 14: 16, 15: 15}
7 : {2: 1, 3: 14, 5: 5, 6: 20, 9: 10, 11: 8, 12: 15}
8 : {1: 20, 2: 13, 3: 3, 4: 18, 5: 17, 6: 6, 11: 5, 12: 2, 13: 20}
9 : {1: 20, 2: 8, 3: 7, 4: 16, 7: 10, 11: 6, 12: 11, 13: 9, 14: 3}
10 : {1: 16, 2: 2, 3: 5, 4: 19, 6: 3, 11: 18, 12: 2, 13: 18, 14: 10, 15: 16}
11 : {1: 15, 3: 17, 5: 8, 6: 14, 7: 8, 8: 5, 9: 6, 10: 18, 14: 17, 15: 14}
12 : {1: 7, 2: 11, 3: 11, 4: 18, 6: 4, 7: 15, 8: 2, 9: 11, 10: 2, 13: 8}
13 : {2: 18, 3: 15, 5: 6, 8: 20, 9: 9, 10: 18, 12: 8, 15: 19}
14 : {1: 2, 2: 5, 5: 9, 6: 16, 9: 3, 10: 10, 11: 17}
15 : {1: 19, 3: 10, 4: 5, 5: 9, 6: 15, 10: 16, 11: 14, 13: 19}
path = [1, 2, 3, 6, 4, 5, 7, 9, 11, 8, 12, 13, 15, 10, 14, 1]
1 step:  ([1, 2, 3, 6, 4, 5, 7, 9, 11, 8, 12, 13, 15, 10, 14, 1], 127)
normal:  ([1, 2, 3, 6, 4, 5, 7, 9, 11, 8, 12, 13, 15, 10, 14, 1], 127)
2-OPT:  ([1, 4, 15, 5, 13, 9, 11, 7, 2, 6, 12, 8, 3, 10, 14, 1], 78)
```

Améliorations ?

Nous n'avons malheureusement pas eu le temps de tester d'autres algorithmes en raison des contraintes de temps, même si nous avions pensé à explorer les heuristiques présentées en cours et à utiliser la programmation dynamique. Nous avons perdu du temps à coder un algorithme pour vérifier si le graphe est **k-coloriable**, car l'énoncé l'a suggéré alors qu'en réalité, ce n'était pas nécessaire.