TP4

Auteurs : Karam ELNASORY · Erkin Tunc BOYA· Semih DOYNUK

Code dans la repositoire GitHub : https://github.com/ErkinTunc/ARP-Recherche-Non-Inform-

Def : Chemin Hamilton

chemin Hamiltonien est un chemin passant par chaque sommet de V exactement une fois

Le problème de décision du chemin Hamiltonien consiste à déterminer si un tel chemin existe dans G.

Le problème d'optimisation

associé consiste à trouver un tel chemin de coût minimum sur un graphe pondéré

> Le problème du chemin Hamiltonien est un problème NP-complet, ce qui signifie qu'aucun algorithme polynomial n'est connu pour le résoudre dans le cas général.

Ex1 Un sacré voyage

Vous souhaitez réaliser un roadtrip aux USA en visitant toutes les capitales des états américains.
Vous pouvez choisir votre ville de départ et votre ville d'arrivée, mais vous ne pouvez visiter chaque ville qu'une seule fois.

> fichier *us_capitals.txt* → contenant les coordonnées géographiques (x, y) de chaque capitale d'état américain

id_ville	latitude (x)	longtitude (y)
1	6734	1453
2	2233	10
3	5530	1424

Par simplicité, nous allons considérer que la distance entre deux villes est la distance euclidienne entre leurs coordonnées géographiques, c'est-à-dire

$$d(u,v) = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}.$$

Q1: Modélisation Constructive

- Etape 1: Espace d'Etat
  - $s : (u, v)$

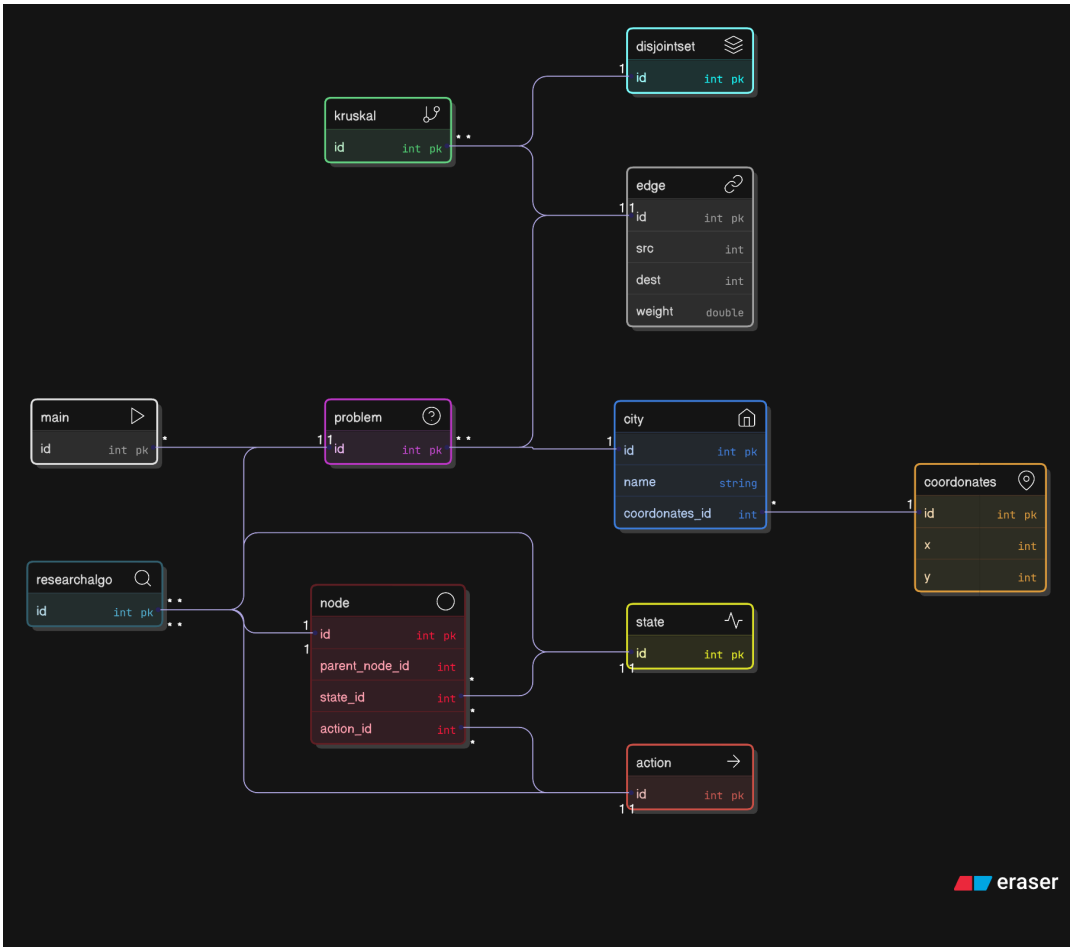
    - u → la ville actuelle
    - v → l'ensemble de ville qui sont deja visité
- Etape 2: Etat-Initial
    - $s_0(v_{depart}, [v_{depart}])$ ← état initial
- Etape 3: Actions

  Action → Se deplacer dans une ville v_i tel que $v_i \in v$
- Etape 4: Fonction de Succeseurs
    - $succ((u, v), v_i) = (v_i, v \cup v_i)$
- Etape 5: Ensemble d'états but $T \subseteq S$
    - S = $s_n(\epsilon, [v_{depart}])$ ← état final ou $[v_{depart}]$ contient tous les villes
- Etape 6: Fonction de coût
    - $cout(u, v_i) = \sqrt{(x_u - x_{vi})^2 + (y_u - y_{vi})^2}$
        - x_u, y_u represante le ville courante
        - x_{vi}, y_{vi} represante le ville prochainS

Q2:

- Un algorithme de recherche non-informée sera difficilement efficace pour résoudre ce problème
- Le parcours en largeur demandera beaucoup trop de mémoire
- Le parcours en profondeur risque de s'égarer dans des solutions non-optimales.
- Le parcours en profondeur itératif n'aucun intêret ici, car (comme pour le problème du cavalier) $d = D$
- Parcours en profondeur modifié

EX2



sur le git

EX3

```
Class noeud:
   ville curr
   [villes visitées]
   g :cout du départ jusqu'a ici
      h :cout heuristique
   f : g+h
   parent :

Fnctn HeuristiqueMST(villes_non_visités):
   Si estvide(villes_non_visités) :
      retourner 0
```

```
    edges = liste vide
    Pour chaque u dans villes_non_visites:
        Pour chaque v dans villes_non visites (u≠v):
        edges.ajouter(Edge(u,v,distance(u,v)))
    Retourner Kruskal.run(taille(villes_non_visites),edges)

Fctn A_etoile(ville_d):
    OpenList: File_prioritaire //ordonné par f croissant
    ClosedList: Ens_vide

    h_start = HeuristiqueMST(villes-{ville_d})
    start_node = noeud(ville_d,{ville_d}, 0, h_start, h_start, null)

    OpenList.add(start_node)
    tant que OpenList pas vide:
        current = OpenList.pop()
        Si taille(current.list_visite) == nbVilles:
            Retourner Chemin(current) // chemin opti trouvé
        ClosedList.add(current.etat)

        Pour chaque voisins dans ville:
            si voisins pas dans current.liste_visite=
                new_g = current.g + dist(current.ville-curr, voisin)
                restantes = ville - (current.list_visite U voisin)

                new_h = HeuristiqueMST(restantes)

                new_node = Noeud(voisin, current.list_visite U voisin, new_g, new_h, new_g + new_h, curren
t)
                Si new_node.etat pas dans closedList:
                    OpenList.add(new_node)
    Retourner "Pas de solution"
```