Compléments sur les plus courts chemins

Exercice 0.1. Rappeler tous les algorithmes de plus court chemin que vous connaissez et expliquer leur usage et leur complexité.

Dans la suite on se place dans un graphe G qui peut être orienté et où chaque arête est pondéré positivement par une distance d

Définition 0.1. On notera $d_G(x, y)$ la longueur d'un plus court chemin de x à y dans G.

Proposition 0.1.
- *d_G vérifie l'inégalité triangulaire* : $\forall x, y, z\ d_G(x, z) \leq d_G(x, y) + d_G(y, z)$
- *Si d est toujours strictement positive alors $d_G(x, y) = 0 \Leftrightarrow x = y$.*
- *Si G est non orienté alors d_G est symétrique.*

1 Rappels sur l'algorithme de Dijkstra

```
let dijkstra (g: (int * float) list array) (source: int) : float array * int array =
  let n = Array.length g in
  let dist = Array.make n infinity in
  let pqueue = Pqueue.create () in
  let visited = Array.make n false in
  let parent = Array.make n (-1) in
  dist.(source) <- 0.;
  Pqueue.insert pqueue (0, source)
  while not (Pqueue.is_empty pqueue) do
    let dv, v = Pqueue.extract_min pqueue in
    if not visited.(v) then (
      visited.(v) <- true;
      (* on vient de déterminer la distance de v *)
      List.iter
        (fun (w, dvw) ->
              let d = dv +. dvw in
        )
        g.(v)
    )
  done;
  dist, parent
```

```
let dijkstra_diminue (g: (int * float) list array) (source: int) : float array * int array =
  let n = Array.length g in
  let dist = Array.make n infinity in
  let pqueue = Pqueue.create () in
  let vu = Array.make n false in
  let parent = Array.make n (-1) in
  dist.(0) <- 0.;
  vu.(0) <- true;
  Pqueue.insert pqueue (0, source)
  while not (Pqueue.is_empty pqueue) do
    let dv, v = Pqueue.extract_min pqueue in
    (* on vient de déterminer la distance de v *)
    List.iter
      (fun (w, dvw) ->
        let d = dv +. dvw in
```

```
22
23
24
25
26        )
27        g.(v)
28    done;
29    dist, parent
```

Exercice 1.1.

- Compléter la fonction **disjktra**.
- Dans le 2ème code, on suppose qu'on dispose d'une fonction pour diminuer la priorité d'un sommet. Compléter la fonction **disjktra_diminue** de façon à ce chaque sommet soit ajouté au plus une fois à la file.

Théorème 1.1.

- *L'algorithme de Dijkstra renvoie un tableau contenant pour chaque sommet u la distance $d_G(s,u)$.*
- *Il extrait les sommets de la file de priorité par ordre de $d_G(s,u)$ croissant.*
- *Il a une complexité de $O((|E|+|V|)\log(|V|))$ si la file de priorité est implémenté avec un tas binaire.*

Dans la suite on veut calculer le plus court chemin d'un sommet s à un sommet t.

Exercice 1.2. Proposer dans ce cas une modification de l'algorithme.

2 Algorithme A^*

Définition 2.1. On dispose pour chaque sommet v d'une heuristique $h(v) \geq 0$ qui est sensé approximer la distance de v à t.

Proposition 2.1. *On a que u est un point d'un plus court chemin de s à t si et seulement si $d_G(s,u)+d_G(u,t) = d_G(s,t)$.*

On note $dist[u]$ la plus petite distance de s à u calculée jusqu'à présent.

Définition 2.2. Algorithme A^*

- On stocke $dist$ dans un tableau.
- On a une file de priorité qui contient les sommets à explorer avec comme priorité $dist[u]+h(u)$.
- Quand on explore un sommet, on met à jour la distance de tous ses voisins et on les ajoute à la file.

On s'arrête quand on a atteint la destination.

```
1   let astar_path g src dst h : float * int array =
2    let n = Array.length g in
3    let dist = Array.make n infinity in
4    let pqueue = Pqueue.create () in
5    let parent = Array.make n (-1) in
6    dist.(src) <- 0.;
7    Pqueue.insert pqueue (h src, src);
8    let relax v (w, dvw) =
9      let d = dist.(v) +. dvw in
10     if d < dist.(w) then begin
11         dist.(w) <- d;
12         parent.(w) <- v;
13         Pqueue.insere_ou_diminue pqueue (d +. h w, w)
14         end
15   in
16   let rec loop () =
17     if Pqueue.is_empty pqueue then raise Not_found;
18     let _, v = Pqueue.extract_min pqueue in
```

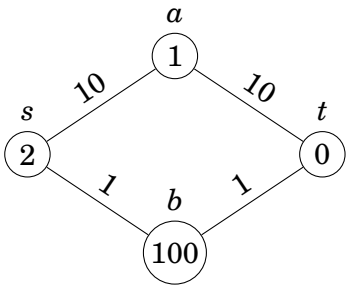
```
19      if v = dst then
20        dist.(dst), parent
21      else (
22        List.iter (relax v) g.(v);
23        loop ()
24      ) in
25    loop ()
```

Exercice 2.1. Éxécutez l'algorithme pour aller de Dijon à Nice.

Exercice 2.2.
- Que se passe t'il si h est constante égale à 0 ?
- Que se passe t'il si $h(v) = d_G(v,t)$ pour tout v ?

3 Heuristique admissible, heuristique monotone

On peut remarquer que l'algorithme A^* n'est pas toujours correct.



Exercice 3.1. Appliquer l'algorithme à :

Définition 3.1. On dit qu'une heuristique est admissible si pour pour tout sommet v, $h(v) \leq d_G(v,t)$.

Théorème 3.1. (*i*) *À tout instant de l'algorithme A^*, pour tout sommet u, $dist[u]$ contient la distance d'un chemin de s à u, et parent le prédécesseur de u sur un tel chemin.*

(*ii*) *Pour tout sommet u, $dist[u]$ est décroissante au cours de l'algorithme.*

(*iii*) *À la fin de l'algorithme, $dist[t] = d_G(s,t)$.*

Remarque 3.1. Même si l'heuristique est admissible, la complexité de A^* peut être exponentielle.

Définition 3.2. Une heuristique est dite monotone si pour tout sommet u et v adjacents $h(u) \leq d(u,v)+h(v)$.

Théorème 3.2. (*i*) *Si h est monotone alors pour tout sommet x, la fonction $v \mapsto h(v)+d_G(x,v)$ est croissante le long de tout plus court chemin de x à t.*

(*ii*) *Si h est monotone et $h(t) = 0$ alors h est admissible.*

Lemme 3.3. *Si h est monotone, alors chaque sommet n'est extrait qu'une seule fois de la file de priorité.*

On en déduit :

Proposition 3.4. *Si h est monotone, la complexité dans le pire cas est linéaire.*

4 Graphes implicites, graphes d'états

Définition 4.1. Un graphe est dit implicite quand la mémoire ne contient pas une liste de tous les sommets et de toutes les arêtes, mais une représentation compacte qu'un programme permet de traduire.

Remarque 4.1. L'algorithme A^* est particulièrement adapté à la recherche d'un plus court chemin dans les graphes implicites. En effet, dans ce cas, l'algorithme peut être sous linéaire ce qui est souvent nécessaire au vu de la taille que peuvent avoir les graphes implicites.

Remarque 4.2. Il faut dans ce cas remplacer les tableaux par des dictionnaires.

Définition 4.2. Un graphe d'état pour un jeu est un graphe où les sommets représentent les états possible et les arêtes les coups possibles.

Remarque 4.3. Ce graphe est en général implicite.