# 1 Premier rendez-vous

## 1.1 Question 1

L'algorithme nécessite bien entendu autant d'étapes qu'il y a de balises.

## 1.2 Question 2

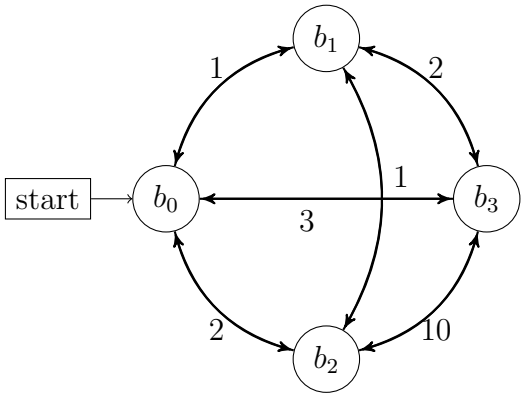


FIGURE 1 – Parcours non optimal : chemin optimal = $b_0 \to b_2 \to b_1 \to b_3$ de poids 5 mais chemin de l'algorithme = $b_0 \to b_1 \to b_2 \to b_3$ de poids 12.

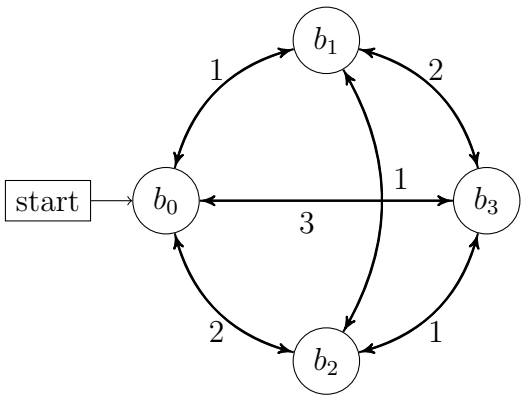


FIGURE 2 – Parcours optimal : chemin optimal = chemin de l'algorithme = $b_0 \to b_1 \to b_2 \to b_3$ de poids 3.

## 1.3 Question 3

Le problème peut être modélisé sous forme d'un graphe $< S, \delta >$ où $S = \{b_1, b_2 \dots b_n\}$ désigne les balises et $\delta : S^2 \rightarrow \mathbb{R}^+$ associe à tout couple de balises le temps qu'il faut pour joindre l'une à l'autre.

Cette vision est équivalente à celle qui consiste à définir $\delta$ comme étant une matrice symmétrique positive telle que $\delta_{i,j}$ représente le temps pour aller de la balise $b_i$ à la balise $b_j$.

## 1.4 Question 4

### 1.4.1 4.a

Je présente ici une version récursive en caml où le premier formalisme a été utilisé. La balise 0 est assimilée au point de départ du bateau.

```
let rec poids_chemin =
  function
    |[0]->0.
    |balise_b::balise_a::chemin->
       (delta (balise_a,balise_b))+.(poids_chemin (balise_a::chemin))
;;

let poids_optimal = ref infinity;;
let chemin_optimal = ref [0];;

let rec recherche_chemin_optimal chemin=
  if (List.length chemin)=n
  then
    let poids = poids_chemin chemin in
      if poids<(!poids_optimal)
      then
begin
  poids_optimal:=poids;
  chemin_optimal:=chemin;
end;
  else
    for i=1 to n do
      if not(List.exists (function x->x=i) (!chemin_optimal))
```

```
      then recherche_chemin_optimal (i::chemin);
    done;
;;

recherche_chemin_optimal [0];;
!chemin_optimal;;
```

### 1.4.2 4.b

Dans cet algorithme, il faut bien entendu $n!$ étapes pour trouver le chemin optimal, avec $n$ le nombre de balises.

### 1.4.3 4.c

A chaque itération, l'algorithme fait de l'ordre de $n$ opérations élémentaires (pour calculer la longueur d'un chemin). Puisqu'il fait $n!$ fois cette étape, on en conclut donc qu'il fait au total de l'ordre de $n(n!)$ opérations élémentaires.

En prenant 271 balises, cela représente donc $271 * (271!)$ opérations, soit $3.10^{-9}271(271!)$ secondes. La formule de Stirling donne donc l'approximation suivante : $3.10^{-9}271\sqrt{2\pi 271}(\frac{271}{e})^{271}$s soit de l'ordre de $10^{537}$s.

Pour info, cela représente environs $10^{522}$ millions de millards d'années :)

# 2 Second rendez vous

Au second rendez-vous, on attend des élèves qu'ils aient abordés l'idée de ce qu'est une approche *breadth-first search* ou *depth-first search*. Ils doivent également connaître les approches par *backtracking* et *branch and bound*.

Je résume rapidement en dessous les différents algorithmes.

## 2.1 Parcours en largeur et en profondeur

Ce sont les parcours les plus usuels. Ils ont de toute facon utilisés l'une ou l'autre de ces méthodes lors de la première scéance pour formuler leur algorithme. L'exemple donné pour le premier rendez-vous est de type *depth first search*.

Est-ce vraiment exact ? La notion de parcours de l'arbre des possibilités restreint déjà le champs des possibilités si le graphe de départ n'est pas complet. En effet, contrairement à ce qui est attendu lors du premier rendez

vous, on remarque ici qu'on ne parcours que des possibilités réelles qui sont constructibles par le graphe.

## 2.2 Backtracking

L'algorithme de backtracking est un algorithme de type *depth first search*. Le principe est très simple : si on se rend compte prématurément qu'un chemin qu'on est en train de construire est déjà trop long, alors on ne cherche pas à aller plus loin et on passe au suivant.

Voici l'algorithme présenté pour le premier rendez-vous en version backtracking :

```
let rec poids_chemin =
  function
    |[0]->0.
    |balise_b::balise_a::chemin->
       (delta (balise_a,balise_b))+.(poids_chemin (balise_a::chemin))
;;

let poids_optimal = ref infinity;;
let chemin_optimal = ref [0];;

let rec recherche_chemin_optimal chemin=
  if (List.length chemin)=n
  then
    let poids = poids_chemin chemin in
      if poids<(!poids_optimal)
      then
begin
  poids_optimal:=poids;
  chemin_optimal:=chemin;
end;
  else
      if not((poids_chemin chemin)>poids_optimal)
      then
        for i=1 to n do
  if not(List.exists (function x->x=i) (!chemin_optimal))
  then recherche_chemin_optimal (i::chemin);
```

```
done;
;;

recherche_chemin_optimal [0];;
!chemin_optimal;;
```

## 2.3 Branch and bound

L'algorithme *branch and bound* est beaucoup plus subtil. L'idée est que cette fois ci l'arbre de recherche est parcouru en largeur. A chaque fois que l'on fait un pas de plus dans la profondeur de l'arbre, on trouve une borne minimale et maximale pour la longueur des chemins des sous arbres qui partent du noeud courrant. Cette borne est estimée grace aux valeurs extrêmes de la fonction de transition du graphe.

Sur un graphe tel que celui qu'on s'attend à considérer, l'apport de cette méthode est néanmoins probablement mauvais, du fait que certaines balises sont beaucoup plus proches que d'autres.