Rapport DM 2-approximation du routard

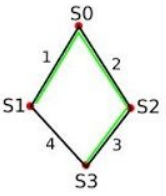
*PB : Nous souhaitons construire un graphe connexe pondéré contenant n sommets tel que, si on note σ∗ une séquence de poids minimum au problème du routard, notre fonction doit construire un graphe pour lequel l'algorithme RoutardApprox pourrait construire une séquence σ telle que w(σ) soit très proche de 2 * w(σ∗).*

Spoiler : j'ai pas réussi, je trouve w(σ) = 1.33 * w(σ∗) (je vous montre que j ai néanmoins chercher)

I) Tâtonnement

1) Avec 4 sommets

J'ai d'abord essayé de trouver un graphe, sur papier, pour lesquels je serai plus rapide que le Routard.

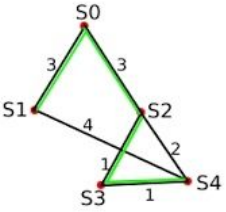


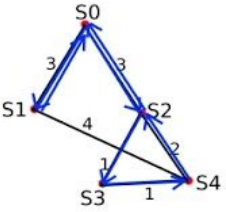
J'ai trouvé le graphe ci-contre (en vert : arbre couvrant de poid minimum à partir de S0) et de la manière dont j'ai codé *RoutardApprox,* en respectant l'algorithme donné dans le sujet, j'ai comme séquence de poids minimum : *σ=['S0', 'S1', 'S0', 'S2', 'S3', 'S2', 'S0']* car en partant de S0 le Routard va au fils le plus proche (S1), puis se dirige vers l'autre fils (S2) en empruntant le chemin le plus court (grâce à l'implémentation de l'algo de Dijkstra), ainsi il repasse par S0 et atteint ensuite S2, puis S3. Pour finir il revient au point de départ (S0).

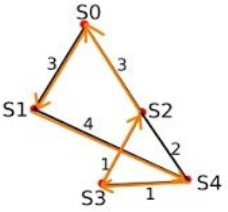
Le poid total par le Routard est de *12* alors que le chemin optimal en fait *10* : *σ*=['S0', 'S1', 'S3, 'S2', 'S0']* on constate un gain de 2 { w(σ) = 1.2 * w(σ∗) } qui s'explique par le poid entre S0 et S2 car le Routard l'empreinte 2 fois.

2) Avec 5 sommets vers une généralisation

Voici un autre graphe avec 5 sommets.
(en vert : arbre couvrant de poid minimum à partir de S0) (en bleu : parcours par le Routard à partir de S0) (en orange : parcours optimal à partir de S0)







Je n'ai pas encore parlé du parcours préfixe, sur tout mes graphes et exemples, celui-ci est volontairement l'ordre alphabétique des nom de sommets (ex pour ce graphe : *['S0', 'S1', 'S2', 'S3', 'S4']*), cela permet de le repérer plus rapidement.

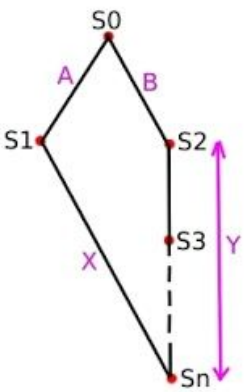
Attardons nous sur le parcours du Routard, *σ=['S0', 'S1', 'S0', 'S2', 'S3', 'S4', 'S2', 'S0']*, il part de S0 et se dirige vers S1, cela est dû au choix arbitraire dans l'implémentation de l'algo de prim : sélectionner le fils qui est le premier selon l'ordre alphabétique donc ici S1 est avant S2.

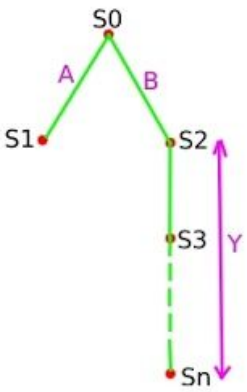
Ensuite il souhaite rejoindre le second fils S2, pour cela il emprunte le chemin le plus court depuis sa position vers S2. Il y 2 chemins de même poid (6) : *['S1', 'S0', 'S2']* ou *['S1', 'S4', 'S3', 'S2']* le Routard choisira le premier chemin car l'implémentation de l'algo de Dijkstra résulte que S2 sera découvert par S0 (lui-même découvert par S1) en premier avec un poid de 6.

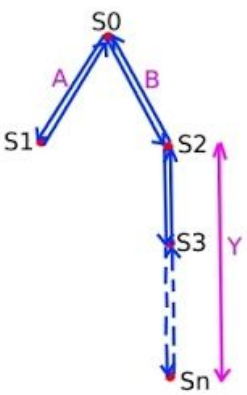
Le poid total par le Routard est de *16* alors que le chemin optimal en fait *12* : *σ*=['S0', 'S1', 'S4', 'S3', 'S2', 'S0']* on constate un gain de 6 { w(σ) = 1.33 * w(σ∗) } qui s'explique par le parcours *['S2', 'S3', 'S4', 'S2']* que le Routard l'empreinte 2 fois.

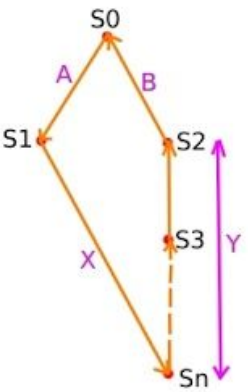
II) Généralisation

1) La théorie









Voici le graphe général, le code couleur reste le même, j'y ajoute en rose le poid des arêtes en tant que variable. La variable Y est la somme de S2 à Sn.

Tout d'abord, le parcours préfixe (en vert) est volontairement dans l'ordre alphabétique des nom de sommets, du au faite que lors de la création de la liste de parcours préfixe je me base sur le premier sommet voisin selon l'ordre alphabétique dans ma fonction *rectListOrdre*.

Pour que le parcours préfixe soit tel que imaginer précédemment, il faut résoudre un système d'inéquation suivante :

$$\begin{cases} A <= B & (1) \\ [Sn->Sn-1] <= X & (2) \\ A+B = X+Y & (3) \end{cases}$$

1) On s'assure que S1 soit le premier sommet extrait dans prim et le plus petit fils de S0.
2) On vérifie que Sn ne peut avoir comme père S1 car le poid entre Sn-1 et Sn est inférieur à Y et donc lors de l'extraction dans prim, on choisira le plus petit entre (Sn-1 et Sn) et X, donc ce sera l'arête (Sn-1 ; Sn).
3) Il faut que le poid du chemin entre S1 et S2 soit aussi long en passant par S0 (A+B) tout comme en passant par Sn jusqu'à S2 (X+Y).

Une fois résout on peut créer le graphe *explication dans le II.2*.

Le Routard part de S0 et se dirige vers S1, cela est dû au choix arbitraire dans l'implémentation de l'algo de prim : sélectionner le fils qui est le premier selon l'ordre alphabétique donc ici S1 est avant S2.

Ensuite il suit le parcours préfixe, donc il veut se rendre de S1 à S2 en prenant le chemin le plus court, Il y 2 choix de parcours de même poid (z) : *['S1', 'S0', 'S2']* ou *['S1', 'Sn', 'Sn-1', … , 'S3', 'S2']*, il choisira le premier parcours car l'implémentation de l'algo de Dijkstra résulte que S2 sera découvert par S0 (lui-même découvert par S1) en premier avec un poid de z.

Puis le Routard continue son exploration selon l'ordre préfixe en passant du sommet Sk à Sk+1 directement car c'est le chemin le plus court. Une fois arrivé à Sn le Routard retour au point de départ en empruntant le chemin le plus court, même explication que pour aller de S1 à S2, détaillé précédemment et finit par dessiner le graphe en bleu.

Le mauvai choix de parcours du Routard l'induit en erreur le poid final du parcours car il emprunt 2 fois la même séquence (aller, retour) *['S2', 'S3', ..., 'Sn-1', 'Sn']* donc *σ=['S0', 'S1', 'S0', 'S2', 'S3', 'S3', ..., 'Sn-1', 'Sn', 'Sn-1', … , 'S3', 'S2', 'S0']* (chemin en bleu) alors que le chemin optimal est : *σ*=['S0', 'S1', 'Sn', 'Sn-1', … , 'S3', 'S2', 'S0']* (chemin en orange).

On constate : le poid de σ est PAS très proche de 2 * w(σ∗) : { w(σ) = 1.33 * w(σ∗) } qui s'explique par le parcours *['S2', 'S3', ..., 'Sn-1', 'Sn']* que le Routard l'empreinte 2 fois.

2) Explication de *ConstruireGrapheDifficile*

Dans un premier temps je vais construire mon arbre couvrant de poid minimum, j'initialise un dictionnaire G avec tout les points de 0 à n-1.

Ensuite je choisi flottant quelconque entre 100 et 1000 avec 9 chiffres après la virgule, puis j'initialise *reste_poid* au flottant moins 1 car je le réserve pour l'arête X.

Je poursuit en ajoutant à S0 ses 2 voisins : S1 et S2. Je choisi pour l'arête entre S0->S1 (A) sera égale à l'arête entre S0->S2 (B) et donc avec un poid : du flottant trouvé précédemment, puis j'ajoute S0 à liste σ∗, ainsi que l'appel la fonction *ajoutVoisin* qui écrit le lien entre S0 et S1 et réciproquement dans le dictionnaire G, jusqu'ici l'équation 1 est vérifiée.

J'ai fini avec le sommet S0 je passe à S1, c'est très rapide : je l'ajoute à liste σ∗.

Pour S2, son voisin sera S3 (S(2+1)) et le poids de cette arête sera un flottant aléatoire entre (1e-9) et la division entière du *reste_poid* avec 9 chiffres après la virgule. Je decrement le *reste_poid* avec le poid de l'arête puis j'appel la fonction *ajoutVoisin* sans oublier d'insérer S2 en 2ème position (2ème position en comptant à partir de 0) dans la liste σ∗.

Je répète l'opération précédente pour tous les autres sommets hormis pour Sn-1 et Sn.

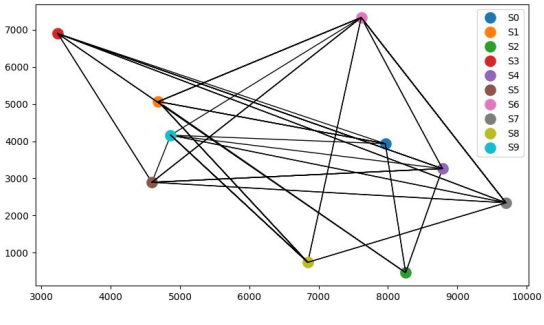
Une fois arrivé à Sn-1, je l'insérer en 2ème position dans la liste σ∗, je défini que son voisin sera Sn avec un poids de : *reste_poid*.

Une fois arrivé à Sn, je défini que son voisin sera S1 avec un poids de : le poid du plus petit court chemin entre Sn et S2 plus le 1 mit de côté plutôt, je l'insérer en 2ème position dans la liste σ∗, jusqu'ici les équations 1,2,3 sont vérifiée.

Mon arbre couvrant de poids minimum est créé, maintenant si j'ai plus de 6 sommets dans G alors j'appel un fonction *ajoutVoisinRandom* qui pour chaque sommets lui ajoute x voisin avec un poids supérieur au poids du plus court chemin entre ces sommets.

III) Exemple

appel : ~$ */usr/bin/python3 RoutardApproxWrapper.py ConstruireGrapheDifficile 10*



σ avec le Routard : *['S0', 'S1', 'S0', 'S2', 'S3', 'S4', 'S5', 'S6', 'S7', 'S8', 'S9', 'S8', 'S7', 'S6', 'S5', 'S4', 'S3', 'S2', 'S0']*

σ∗ optimal : *['S0', 'S1', 'S9', 'S8', 'S7', 'S6', 'S5', 'S4', 'S3', 'S2', 'S0']*

Plus le nombre de sommet est grand plus la le rapport σ/σ∗ se rapproche de 1.333