Rapport de Projet :

Construction d'un Arbre de Classement Single-Peaked à partir de Classements Partiels

Théo-Paul Ladet

Sorbonne Université - Parcours ANDROIDE

Date : 06/05/2025

Table des matières

1. Cahier des charges

Le projet se concentre autour de données électorales de votants à une élection. La requête du client se constitue comme suit :

Le client aimerait obtenir une représentation des données en arbre qui puisse donner une représentation graphique adéquate des proximités entre candidats à une élection à partir des rangements fournis par les votants.

Contraintes

- Les données de rangements sont de nature single-peaked, et ceci doit se refléter sur l'arbre résultant.
- Un individu a des préférences single-peaked sur un ensemble de résultats si ses préférences l'amènent à classer les choix d'une manière hiérarchique nette, représentable de manière linéaire.
- L'individu a un résultat préféré par-dessus tous les autres.
- Plus les autres choix s'éloignent de l'optimum, plus ils sont classés bas dans l'ordre de préférence.
- La nature single peaked permet de faire ressortir l'alignement politique des candidats sur une échelle politique (gauche droite par exemple).
- La recherche de la solution doit se faire par recherche locale ou algorithme génétique.
- Le dataset de référence est l'expérience "voter autrement" 2017-2020 : 11000 rangement, 11 candidats, uniquement les 4 candidats préférés par rangement.
- La solution doit pouvoir permettre au client de charger un dataset en mémoire et de calculer et afficher l'arbre résultant de manière autonome.
- Le calcul de l'arbre doit se faire en un temps raisonnable pour le client.
- Les paramètres d'itérations ou de temps de l'algorithme doivent être paramétrables du côté client.
- La topologie de l'arbre ne doit pas être dégénérée : ne pas être liée à la façon dont on définit la distance.

2. Contexte et spécification du problème

Dans le cadre des systèmes de vote à préférences ordonnées, chaque électeur exprime une préférence single-peaked : il a un candidat favori, puis ses préférences décroissent de part et d'autre de ce pic. Nous disposons cependant de classements partiels, où chaque électeur ne précise que ses k premiers choix parmi n candidats.

L'objectif de ce projet est de déterminer une structure arborescente non orientée sur l'ensemble des candidats, de sorte que, pour chaque classement d'électeur, on puisse orienter temporairement cet arbre à partir du candidat le mieux classé (pic) et mesurer les violations de préférence selon cette orientation.
En d'autres termes, l'arbre lui-même n'a pas de sens de parenté fixe : il reste statiquement non orienté, mais on le « ré-enrôle » différemment pour chaque ranking afin de comparer les ordres d'ancêtre à descendant.

La fonction de coût retenue, dite re-rooted, calcule pour chaque classement r :

- On choisit comme racine l'élément r_1 (première préférence).
- On oriente l'arbre en BFS à partir de cette racine.
- On compte les violations (paires inversées) et les ordres manqués, puis on normalise par le nombre de paires dans r.

L'arbre optimal minimise la moyenne de ces coûts sur tous les classements, ce qui garantit qu'il respecte la nature single-peaked globale.

Ce rapport détaille :

- La formulation formelle du problème avec un arbre non orienté et orientation dynamique (Section 3),
- La conception algorithmique centrée sur la fonction de coût re-rooted (Section 4),
- Les choix d'implémentation (Section 5),
- Les résultats expérimentaux (Section 6) et
- Les conclusions et perspectives (Section 7).

On notera que l'état de l'art en termes d'approche des classements single-peaked par une représentation des données en arbre est très limité et ce projet relève donc d'une démarche expérimentale qui

3. Formulation de la solution

3.1 Cadre mathématique

Ensemble des candidats :
$C = \{c_1, \ldots, c_n\}$.

Classement partiel :
pour chaque électeur e, on dispose d'une suite $r^e = (r_1, \ldots, r_k)$, où r_1 est son candidat préféré (pic), puis ses $k-1$ suivants.

Arbre de structure :
un graphe non orienté $T = (C, E_T)$, connexe et acyclique.

3.2 Orientation dynamique

Pour chaque classement r, on oriente T en prenant la racine temporaire égale à r_1.

Par un parcours en largeur (BFS), on attribue à chaque nœud un niveau de profondeur : plus la profondeur est grande, plus le candidat est perçu comme moins proche du pic.

3.3 Fonction de coût re-rooted

Pour un classement r de longueur k :

- **Violations :**
  nombre de paires (i, j) dans r (ou dans ses paires implicites) pour lesquelles la profondeur de i est strictement supérieure à celle de j, malgré $i \prec_r j$.
- **Ordres manqués :**
  paires (i, j) ordonnées dans r mais où les profondeurs sont égales (i.e. dans la même branche), indiquant une absence de hiérarchie nette.
- **Normalisation :**
  on divise la somme

  $$violations + \gamma \times missed$$

- par le nombre de paires $\frac{k(k-1)}{2}$ de r.

Le coût global est la moyenne de ce coût re-rooted sur tous les m classements :

$$\mathrm{Cost}(T) = \frac{1}{m}\sum_{e=1}^{m} \frac{\mathrm{violations}_e(T) + \gamma \times \mathrm{missed}_e(T)}{\frac{k(k-1)}{2}}$$

Cette formulation garantit que l'arbre, bien que non orienté à l'origine, valorise fortement la structure single-peaked : chaque électeur voit son pic en racine, et on pénalise les profondeurs incohérentes.

En phase exploratoire, les fonctions de coût suivantes ont été évaluées, puis disqualifiées pour ce projet :

- **Violations dans un arbre orienté de facon statique** :
  Cette fonction de coût considère l'arbre orienté avec une racine propre. Les arbre construit avec une telle fonction tentent d'établir un ordre objectif sur tous les ranking. Cette propriétépermet de calculer le coût très rapidement en ouvrant la possibilité à des optimisations recursives, mais ne correspond pas aux premices de notre problèmes, l'abre étént supposé intégrer des ranking ayant des single-peak différents, dans notre cas.

- **Coût binaire de connexité**:
  Cette fonction de coût correspond à la connexité du ranking au sein de l'arbre. Sa nature binaire en fait une fonction pauvre pour quantifier la distance de l'abre aux rankings et peut souffrir d'un déséquilibre fort, venant de sa nature discrète.

- **Coût de connexité maximale**:
  Cette fonction de coût correspond à la mesure du nombre d'element du ranking que l'on peut parcourir avant de perdre sa connexité dans l'abre
  Cette fonction est mieux adaptée que la fonction binaire, mais souffre néanmoins toujours d'une certaine pauvreté, du fait que cette fonction s'arrête dès la premiere discontinuité avec le ranking et n'incorpore pas le reste.
  .

4. Conception de l'Algorithme

4.1 Génération de l'arbre initial

On utilise une séquence de Prüfer pour produire un arbre non orienté T distribué uniformément sur les n candidats.

4.2 Calcul du coût re-rooted

Pour chaque classement r :

- **Orientation** :
  BFS depuis r_1, on calcule la profondeur de chaque candidat.
- **Violation count :**
  pour chaque paire $(i, j) \in r$, si $depth(i) > depth(j)$, on incrémente.
- **Missed count :**
  si $depth(i) = depth(j)$, on incrémente le compteur d'ordres manqués.
- **Coût normalisé :**
  diviser par $\frac{k(k-1)}{2}$

Calcul de la complexité :
L'orientation pour chaque r coûte $O(n)$. Le calcul des violations/missed se fait en $O(k^2)$. Au total, pour m classements, la complexité est $O\big(m(n + k^2)\big)$.

4.3 Recherche locale sur arbre non orienté

4.3.1 Voisinage (coupe–recâblage)

On parcourt chaque arête $\{u, v\}$:

- **Supprimer** $\{u, v\}$, **produire** deux composants.
- **Reconnecter** le plus petit composant en joignant un de ses nœuds à un nœud du composant complémentaire.
- **Former** un nouvel arbre T'.

4.3.2 Critère d'acceptation

On évalue $Cost(T')$ via la fonction re-rooted.
Si le coût diminue, on accepte T'. Sinon, on l'ignore.

4.3.3 Stratégie multi-start

Pour éviter les minima locaux, on répète le processus sur plusieurs arbres initiaux (num_trials) et on retient le meilleur.

4.3.4 Complexité

Chaque itération de coupe–recâblage engendre $O(n)$ voisins, chacun évalué en $O\big(m(n+k^2)\big)$.

En pratique, on limite le nombre d'itérations locales à une constante ou jusqu'à stagnation.

5. Implémentation

5.1 Choix technologiques

Le projet est implémenté en Python 3.11+ pour bénéficier d'une grande expressivité et d'un écosystème riche :

- NetworkX gère la création, l'orientation et le parcours du graphe.
- Matplotlib permet de visualiser l'arbre final.
- Tkinter offre une interface graphique légère permettant à l'utilisateur de sélectionner la fonction de coût, la valeur de gamma, le nombre d'essais et le nombre de candidats.
- Ete3 peut être utilisé pour des rendus phylogénétiques avancés si nécessaire.

5.2 Architecture logicielle

Le code est organisé en fonctions distinctes :

- **read_csv_and_build_rankings.py :**
  lecture du fichier CSV et extraction des classements partiels. Chaque ligne est convertie en liste d'entiers, puis triée par score pour obtenir la liste ordonnée des candidats préférés.
- **random_tree.py :**
  génération de la séquence de Prüfer, conversion en arbre orienté, opérations de sous-arbre et recâblage.
- **local_search.py :**
  algorithme hill-climbing, multi-start et intégration du recuit simulé.
- **TreeGUI.py :**
  interface Tkinter avec champs pour gamma, num_trials, n et menu déroulant pour la fonction de coût; affichage des coûts de chaque essai.
  main.py : script principal lançant l'interface et orchestrant le pipeline.
- **build_position_lookup :**
  - Prépare, pour chaque vote, un dictionnaire « candidat → rang » afin d'interroger rapidement les positions.

- **build_pref_matrices :**

  - Agrège une seule fois tous les votes en matrices de préférences :
    pref_global : combien de fois i ≻ j parmi tous les électeurs

    pref_by_root : idem mais restreint à chaque premier choix

- **root_counts :** population de chaque premier choix

  Ces structures suffisent ensuite à calculer tout coût en $O(n^2)$.

- **get_subtree_nodes / build_forced_pairs / re_root_tree**

  - Opérations de base sur l'arbre :
    extraction d'un sous-arbre, génération de l'ordre partiel imposé,
    changement de racine sans recréer l'arbre.

- **fast_cost_of_tree_re_rooted()**

  - Fonction de coût **principale et optimisée** : exploite les matrices pré-calculées pour évaluer un arbre en temps constant $O(n^2)$;
    remplace l'ancienne version lente mais produit exactement
    les mêmes résultats.

- **violation_plus_missed_cost_of_tree**

  connex_subtree_plus_missed_cost_of_tree

  - Fonctions de coût expérimentales qui réévaluent directement chaque vote ; restent disponibles via le menu déroulant.

- **print_tree / draw_tree**

  - Affichent l'arbre final sous forme indentée et sous forme graphique (layouts **dot** et **planar**).

5.3 Optimisations

- Les matrices M et S sont construites une seule fois par évaluation de coût, puis réutilisées pour chaque classement afin d'éviter les recalculs coûteux.
- L'accès aux positions dans les classements se fait en $O(1)$ grâce à un dictionnaire `position[r][element]` .
- Les fonctions récursives de coût stockent les résultats partiels dans un dictionnaire Python, **réduisant la complexité théorique de $O(k\,!)$ à $O(k^2)$.**
- Effectuer des mises à jour de coûts incrémentielles dans la recherche locale :

  À chaque étape, on coupe un bord $(p \to c)$ et rattache c ailleurs.

  Seuls deux ensembles de paires changent :

  - Paires (x, y) à l'intérieur du sous-arbre déplacé par rapport aux nœuds à l'extérieur de ce sous-arbre.

- La direction de tous les bords sur l'ancien et le nouveau chemin de la racine à

Toutes les autres relations ancêtre/descendant restent identiques, on peut donc mettre à jour les violations et comptages manqués avec seulement :

$$\Delta = co\hat{u}t\_supprim\acute{e} - co\hat{u}t\_ajout\acute{e}$$

Plutôt que de recalculer l'arbre entier.

- Vectorisation de la matrice des paires avec numpy, pour permettre une accélération au niveau du language

6. Expérimentations et Résultats

6.1 Jeux de données et protocole

- **Données synthétiques :**
  permutations aléatoires de $n = 10,50,100$ éléments, classements partiels de taille $k = 5$ suivant une distribution uniforme ou de type Zipf.
- **Données réelles :**
  fichier `borda4.csv` issu d'un sondage sur 11 candidats avec environ 2 000 participants fournissant chacun 4 préférences.

Pour chaque configuration, il a été mesuré :

- **Le coût final :**
  $\mathrm{Cost}(T)$, décomposé en violations et ordres manqués.
- **Le temps d'exécution :**
  total et par phase (initialisation, calcul de coût, recherche locale).
- **La mémoire utilisée.**

Nous obtenons une moyenne de coût de 0.13 avec gamma = 0.3.

6.2 Résultats détaillés

- **Influence du nombre d'essais :**
  le coût moyen diminue de 1 à 20 essais, puis stagne au-delà, indiquant un compromis optimal autour de 20–30 essais.
- **Impact de γ :**
  pour $\gamma = 0.8$, le nombre d'ordres manqués chute drastiquement mais les arbres convergent vers des arbres dégénérés linéaires.
  Une valeur de γ autour de 0.3 produit des arbres maximisant un équilibre.
- **Comparaison des fonctions de coût :**
  la variante "Ré-enraciné" obtient les coûts globaux les plus bas sur les données réelles, tandis que "Violation+Manqué" est plus rapide à converger.

6.3 Analyse temporelle

Pour $n = 11, m = 2000$, sur Intel i5-9300H on observe :

- Temps moyen de génération d'un arbre initial : 0,00037 s.
- Temps moyen de calcul du coût initial : 0,00566 s.
- Temps moyen par itération de recherche locale : 0,406 s.
- Temps total pour 100 itérations : environ 2,863 s.

6.4 Discussion

Les résultats montrent que l'approche multi-start, couplée à une fonction de coût bien calibrée, permet d'obtenir des arbres de haute qualité dans un temps raisonnable pour $n \leq 100$.
Cependant, pour des ensembles plus vastes ($n > 200$), la montée en $O(n^2)$ du calcul de coût devient rapidement contraignante.
Des optimisations en Cython ou la réduction de la recherche locale **(par exemple en explorant un sous-échantillon de voisins)** seraient des pistes naturelles.

7. Difficultés du projet

Sur le plan algorithmique ce projet a soulevé deux types de difficulté :

- **Une difficulté structurelle liée à la complexité** :
  il s'agissait de rester en temps polynomial, dans un contexte où les solutions triviales sont en temps exponentiel
- **Une difficulté classique d'optimisation :**
  Le choix de la fonction de coût est clé dans ce projet, car elle définit l'efficacité de l'heuristique en termes de possibilité de convergence autant qu'en termes de vitesse de convergence, et conditionne également les risques de solution dégénérée. La fonction de coût peut également conditionner la complexité de l'algorithme.
    - Un ensemble de fonctions de coût a été évalué en phase exploratoire, aboutissant à la fonction proposée.

8. Conclusion et Perspectives

En synthèse, ce projet a abouti à une chaîne complète, depuis la formulation formelle du problème jusqu'à l'implémentation logicielle et l'évaluation expérimentale.

L'algorithme proposé, fondé sur des arbres initiaux par séquence de Prüfer et une recherche locale multi-start, atteint un bon équilibre entre précision du classement, topologie de l'arbre en terme de dégénescence, et performance.

Perspectives :

L'extension à des forêts racinées pourrait être envisagée pourmodéliser des préférences multi-groupes.

9. Bibliographie succinte

Recognizing Single-Peaked Preferences on an Arbitrary Graph: Complexity and Algorithms
Bruno Escoffier and Olivier Spanjaard and Magdaléna Tydrichová, 2020
https://arxiv.org/pdf/2004.13602

Preferences Single-Peaked on a Tree: Multiwinner Elections and Structural Results
Dominik Peters, Lan Yu, Hau Chan, Edith Elkind
Journal of Artificial Intelligence Research, 2022

Preferences Single-Peaked on a Tree: Sampling and Tree Recognition
Jakub Sliwinski, Edith Elkind
Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, 2019

Preferences Single-Peaked on Nice Trees Dominik Peters and Edith Elkind Department of Computer Science University of Oxford, UK Dominik Peters, Edith Elkind, Proceedings-of-the-AAAI-Conference-on-Artificial-Intelligence, 2016

10. Annexe 1- Définitions et choix spécifiques

Le codage de Prüfer est une méthode pour décrire de façon compacte un arbre dont les sommets sont numérotés. Ce codage représente un arbre de n sommets numérotés avec une suite P (x1, x2, x3...,x n-2) de n-2 termes. Une suite P donnée correspond à un et un seul arbre numéroté de 1 à n.
Ce codage a été choisi pour ce projet en raison de son efficacité computationnelle dans un contexte d'adressage itératif

Le hill-climbing est une méthode générale d'optimisation permettant de trouver un optimum local parmi un ensemble de configurations, qui prend en entrée trois objets : une configuration, une fonction qui pour chaque configuration donne un ensemble de configurations voisines, et une fonction-objectif qui permet d'évaluer chaque configuration. La méthode consiste à partir de la configuration initiale, à évaluer les solutions voisines, et à choisir la meilleure de celles-ci, et à recommencer l'opération jusqu'à arriver à un optimum local.
Cette méthode a été **choisie** pour ce projet parmi d'autres méthodes d'optimisation **en raison de son approche locale**, cohérente avec la formulation du problème. .

11. Annexe 2- Photos d'écran des entrées-sorties logicielles et instructions d'utilisation

Instructions :

1. Télécharger le fichier python et le dataset .csv

2. Executer avec python >= 3.11

3. Ouvrir le fichier csv avec l'interface
(Le fichier doit être formatté de la manière présente dans borda4.csv)

4.Renseignez le nombre de candidats, selectionner la fonction de coût souhaitée, et ajustez les parametres :

- **Gamma** : Le facteur de pondération entre violations et paires manquées
- **Trials** : Le nombre d'initialisations aléatoires d'arbres pour la recherche locale.
- **Seed** : Graine utilisée pour la génération aléatoire. Entrer « random » pour une graine aléatoire
- **Cost Function :** Fonction de coût pour le calcul de la distance arbre-rankings. « Re-rooted (Fast) » étant la meilleure fonction de coût à la fois en qualité et en vitesse

5 . Lancez la recherche

6. L'arbre résultat est affiché sous deux rendus consécutifs (Spring et Planar), dans des fenêtres permettant la sauvegarde des rendus. La console pyhton affiche également une representation de l'arbre par indentations.

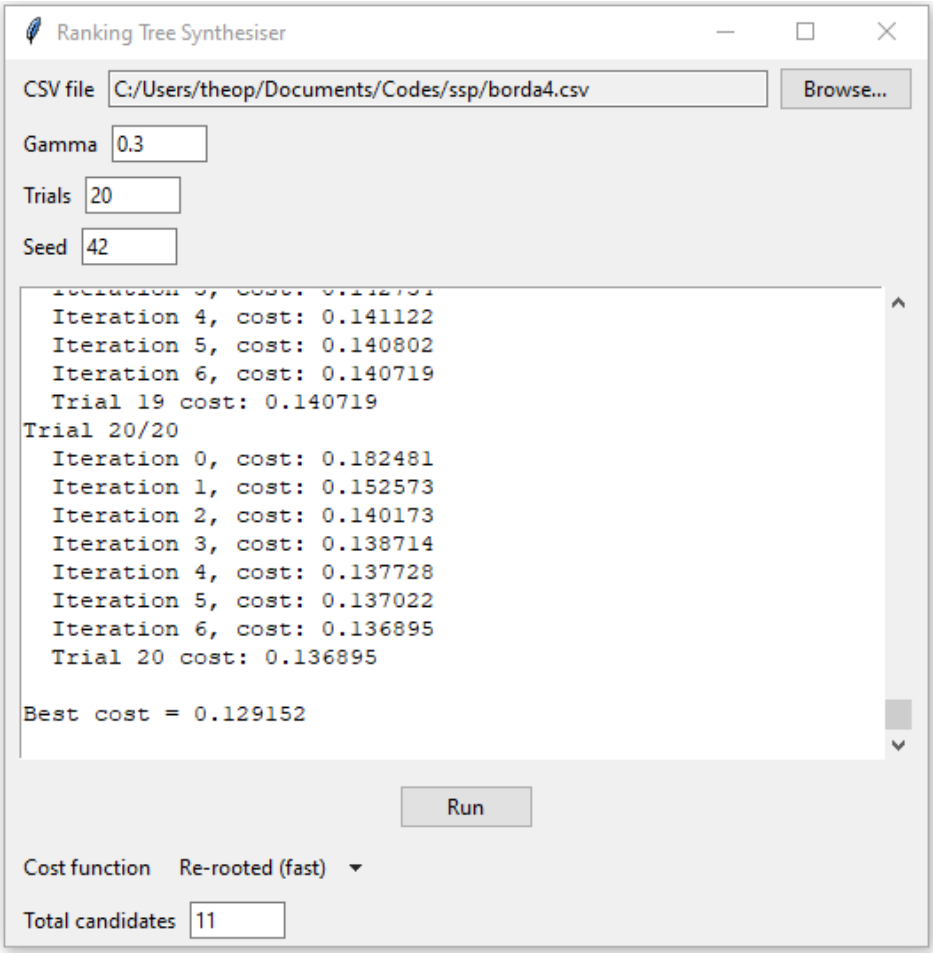
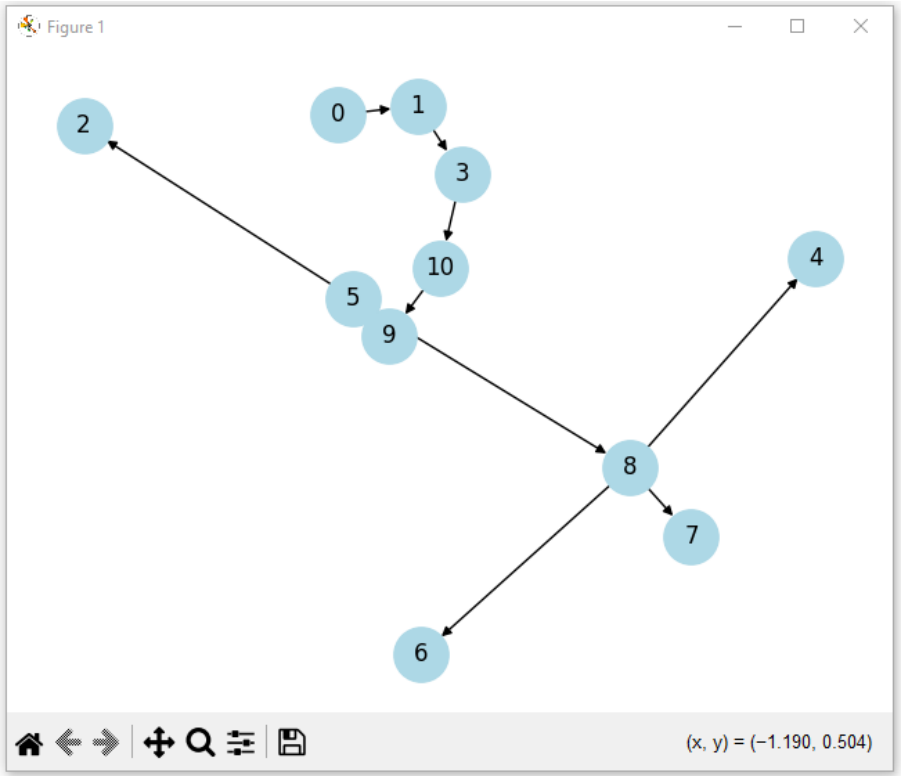
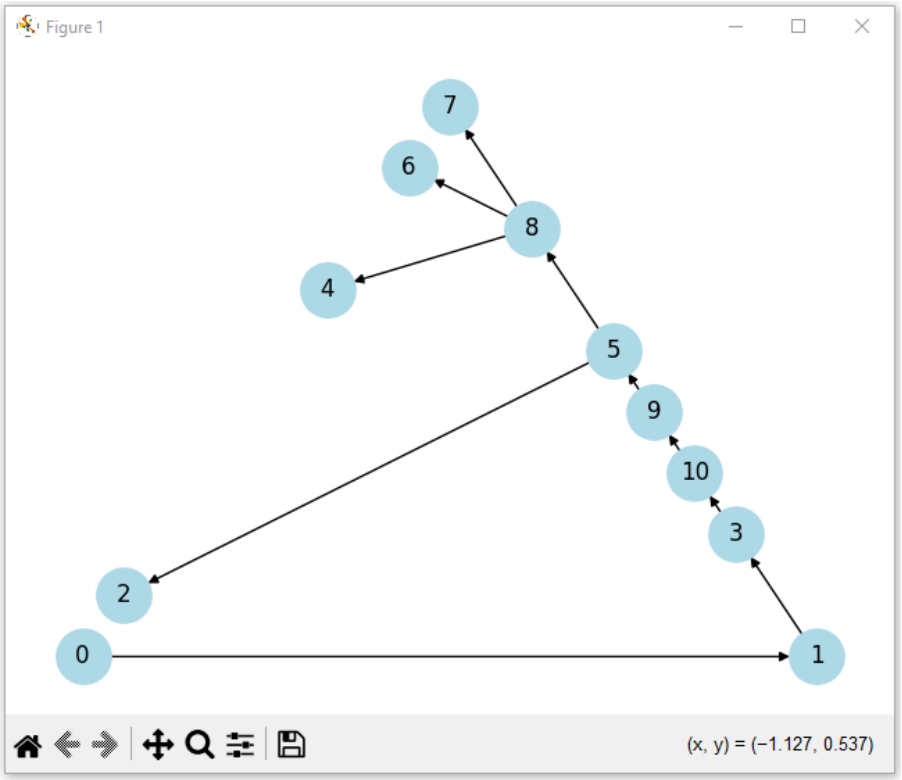


Image de l'interface graphique de l'outil, permettant de paramétrer les variables de l'optimisateur et sélectionner la fonction de coût, avec un retour d'information de la progression du coût par itération.





Rendus de l'arbre final avec networkX avec algorithme spring et planaire respectivement.