M2 Intelligence Artificielle, Systèmes, Données (IASD)
Monte Carlo search and games

Euclidean Steiner tree problem : A Monte Carlo tree search approach

Srir Mohamed Ali
Courty Théo

I Introduction

1 The problem

The Euclidean Steiner Tree Problem (ESTP) is a classical combinatorial optimization problem in computational geometry. Given a finite set of points (called terminals) in the Euclidean plane, the goal is to construct a network of minimal total length that connects all terminals. The network is allowed to include additional points, called Steiner points, which are not part of the original set but can reduce the overall connection cost. In an optimal Euclidean Steiner tree, each Steiner point has exactly three edges meeting at angles of 120 degrees, and terminal points have a degree of maximum 2. The ESTP is known to be NP-hard, making exact solutions computationally challenging for large instances. As a result, many approximation algorithms and heuristics, such as MST-based methods and greedy Steiner point insertion techniques, have been developed to tackle the problem efficiently.

2 Our formulation

In our project, we implemented an interactive Euclidean Steiner Tree game where players start from the Minimum Spanning Tree (MST, tree of minimal total length without Steiner points) connecting a given set of terminals. Inspired by classical MST-based approximation techniques like greedy local Steiner optimization methods, we designed a system where players can merge points (simulating the insertion of Steiner points) and swap connections between nodes. These operations allow players to explore the space of possible Steiner topologies while maintaining the core conditions for constructing optimal Steiner trees [3]. Through successive local transformations, players are encouraged to discover how introducing Steiner points and adjusting connections can significantly shorten the tree. In our github repository, you can find an interactive version of the game in addition to the Monte Carlo Tree Search method to look for the optimal solution. It differs from the implementation in [2], where they choose to implement a modified version of Prim algorithm that allow addition of steiner points. A gameplay example is given in figure 1.

II Methods

1 Greedy

The greedy search algorithm serves as a baseline since it is an easy-to-implement and simple approach to compute. The idea is to start from the minimum spanning tree (MST), select the child of the node that yields the best improvement at each step. It stops either when the best move is to stop playing, or if the maximal depth specified by the user is attained. This approximation is often used as a fast but suboptimal solution for the ESTP. However, one should note that it often leads to good solutions.

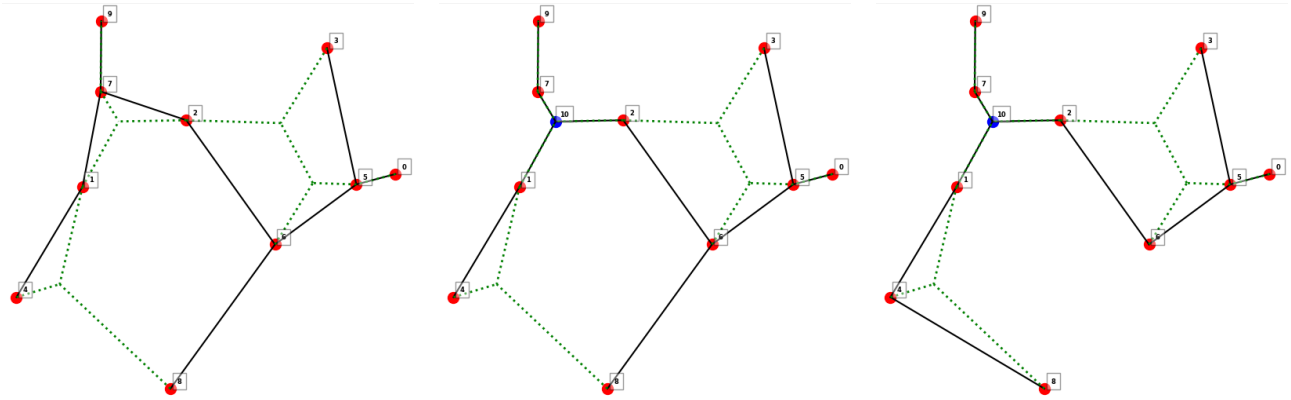
2 Flat Monte carlo

Our first real "Monte Carlo search" method is the flat Monte Carlo which consists in performing a fixed number of playouts for each child, and selecting the one with the highest score. One major downside of this approach is the computational time which can be large as we want to explore uniformly every child. A natural extension is the UCT method presented below which addresses this issue by exploring wisely the moves to play.

3 UCT

We applied the UCT algorithm to our Steiner tree problem, with a minor modifications with respect to the regular version for traditional games : we play the best found move for exploration instead of the average best move. Since we are not playing against an opponent in our game, we can also take a "greedier" approach by selecting the move to play based on the best valued instead of the most visited one. We perform exploration n_{sim} times. The UCB value rewrites as follow :

$$\text{value} = s_i + C\sqrt{\frac{\log n}{n_i}} \tag{1}$$



(a) Initialization with MST (b) Merging 1,2,7 (c) Swapping (6,8) and (4,8)

FIGURE 1 – Gameplay example showcasing different chosen moves. Terminals are the red dots, while the blue ones are the added Steiner points. The optimal tree is represented with a dotted green line.

Where s_i is the score, n (n_i) the number of visits of the parent (child) node. The value of the constant C determines how much exploration the algorithm will do; we discuss its impact in the next section (see Table 1). The score is computed similarly as in [2] :

$$s_i = 1 - \rho_i \tag{2}$$

Where ρ_i is the Steiner ratio (total length out of MST length) of the tree with child node i. Writing s_i this way guarantees that it is growing when the total length of the Steiner tree decreases [1]. The UCT algorithm settles if the maximal depth of the tree [2] specified by the user is reached, or if the best move is to stop playing. Overall, we attain good performance, and we sometimes approach the best theoretical score.

4 Main results

We tested our method on the OR-Lib dataset. UCT results in comparison with flat Monte Carlo and greedy are reported in Table 2. Overall, all methods yield improvements over the MST. Greedy shows systematically better scores than flat, except for one dataset. As discussed earlier, the greedy method is a really fast method to compute, while on the other hand, the flat Monte Carlo takes a significant amount of time. Since we start the game from the MST, the starting state of the game is already a good position, which is a good framework for the greedy approach that will only perform direct improvements at each step. In our implementation, contrary to [2], having a "short-sighted" method is not entirely bad and can perform decently.

We experimented with UCT for $C = \{0.4, \sqrt{2}\}$ in order to explore the impact of the exploring constant on the results. We tested UCT on the first three datasets with 10 and 20 points with $n_{\text{sim}} = 1000$ and $n_{\text{sim}} = 10000$ respectively. With $C = \sqrt{2}$ we empirically observe that the moves are more uniformly explored, however it seems to perform systematically worse than $C = 0.4$ on almost all the datasets for our value of n_{sim}. Higher values of C may spend the exploration budget inefficiently. With our value of n_{sim} and $C = 0.4$ we still witness that every move is still at least explored a dozen times, and it seems to be a fine choice of hyperparameter value. The results are reported in Table 1.

We tried fixing $C = 0.4$ and increasing the number of simulations to 10^5, and reported the results in Table 2. We can witness that the increase in exploration magnitude is beneficial and leads to significantly better performance for UCT than its $n_{\text{sim}} = 10^4$ counterpart. It sometimes even outperforms the greedy algorithm.

1. the graph, not to be confused with a Monte carlo tree
2. The Monte Carlo tree this time

# points	dataset	C	n_{sim}	score ($\times 10^3$)	theoretical best score ($\times 10^3$)
10	1	$\sqrt{2}$	1000	4.48	4.77
10	1	0.4	1000	4.48	4.77
10	2	$\sqrt{2}$	1000	35.82	43.78
10	2	0.4	1000	43.78	43.78
10	3	$\sqrt{2}$	1000	6.52	11.50
10	3	0.4	1000	11.50	11.50
20	1	$\sqrt{2}$	10000	5.85	23.00
20	1	0.4	10000	5.85	23.00
20	2	$\sqrt{2}$	10000	21.53	37.25
20	2	0.4	10000	21.53	37.25
20	3	$\sqrt{2}$	10000	6.08	27.94
20	3	0.4	10000	11.65	27.94

TABLE 1 – Results for our UCT implementation for different datasets with two values of C. Maximal depth is fixed to 7.

# points	dataset	UCT	Greedy	Flat Monte Carlo	Ground Truth
10	0	**43.0**	43.0	31.28	43.0
10	1	4.77	4.77	4.77	4.77
10	2	43.78	43.78	43.78	43.78
10	3	11.5	11.5	11.5	11.5
10	4	**24.6**	24.6	22.6	24.6
10	5	**46.08**	26.83	26.12	46.08
10	6	**44.26**	44.26	40.51	44.26
10	7	14.07	14.07	**15.22**	15.86
10	8	**19.06**	19.06	15.37	24.95
10	9	**19.79**	19.79	19.14	19.79
20	0	43.77	**43.81**	24.5	43.94
20	3	21.34	**22.06**	11.69	27.94
20	4	37.66	**38.91**	15.07	38.91
20	6	**12.66**	12.66	7.4	21.0
20	7	**47.37**	47.37	28.65	47.58
20	8	**15.51**	15.51	6.9	15.51

TABLE 2 – Comparison of the three proposed methods with $C = 0.4$ for UCT with 100k simulations for 10 points and 200k for 20 points and 1M simulations for flat Monte Carlo. (Scores are in ($\times 10^3$).

III Conclusion

In this work we explore a way to solve the ESTP with Monte Carlo Tree search. Experimental results using benchmark problems from OR-Lib demonstrated that the proposed algorithms can achieve near-optimal solutions. Future work will focus on adapting the Monte Carlo Tree Search approach to handle problems with higher dimensionality. While our work focused on implementing UCT similarly as [2], but with a different state and action space, other MCTS methods has been proposed to study the Steiner tree problem such as [1] exploring the problem in the non-Euclidean cases.

Bibliographie

[1] Reyan Ahmed, Mithun Ghosh, Kwang-Sung Jun, and Stephen Kobourov. Nearly optimal steiner trees using graph neural network assisted monte carlo tree search. *arXiV*, 04 2023.

[2] Michał Bereta. Monte carlo tree search algorithm for the euclidean steiner tree problem. *Journal of Telecommunications and Information Technology*, 4 :71–81, 01 2018.

[3] Warren D. Smith. How to find steiner minimal trees in euclideand-space. *Algorithmica*, 7(1–6) :137–177, Jun 1992.

GitHub structure

In order to use the code, one must use the shell script *download_dataset.sh* before launching the simulations to get access to the datasets. The general structure of the game is computed using the script *steiner.py* in the *src* folder. The *interactive.py* is used for illustrating the gameplay with an interactive interface. The MCTS methods are stored in the files *naive_methods.py* and *uct.py*. The **_xp.sh* scripts are used to obtain the results reported in the tables. The *main* branch is up-to-date, others may be deprecated and should not be considered.