Approximating the best electrical network configuration

Marc-Antoine Weisser

9 octobre 2024

Résumé

The objective of this lab is to implement the approach presented in this course. First, we will focus on modeling the problem based on a set of specifications. Then, we will address questions of complexity and approximability. We will implement at least two resolution methods (approximation, branch and bound, simulated annealing, etc.). Finally, we will examine the advantages and disadvantages, along with a performance evaluation.

This lab is staged, so be sure to carefully analyze what the interlocutors say. The dialogues contain important information.

1 Expected Work

The lab session is spread over 3 classes and can be completed at home. It can be done in pairs, but be aware that the expected workload is slightly higher in that case (see report outline).

The lab session will result in a report written in LaTeX. It must be structured according to the plan presented below. We accept reports in either French or English. The source code of your programs must be commented and provided along with the report. All your work must be grouped in a zip or tar.gz archive (we do not accept rar, bzip, or other 7zip formats). This archive should be organized as follows. A parent folder titled `report_algo_NAME` (or `report_algo_NAME1_NAME2` if you are in a group) will contain the PDF of the report, as well as a "source" subdirectory containing the code of your programs and the data.

Here is the report outline. You will complete it with the information gathered during the lab sessions. If you are working alone, you may choose not to implement the exact method.

- Modeling
  - Problem presentation
  - Decision problem
  - Optimization problem
- Theoretical Results
  - Complexity (proof of NP-completeness)
  - Approximability
    - Main known results and reference to a compendium
    - Presentation of an approximability algorithm
    - Proof of the approximation ratio of the presented algorithm
- Metaheuristic (Simulated Annealing)
  - Description of the method
  - Neighborhood
- Exact method (Branch & Bound) [optional if alone, mandatory in group]
  - Description of the method
  - Presentation of the bound
- Performance evaluation

During the lab sessions, it is strongly recommended to interact with other student groups : don't isolate yourself.

Your supervisors will contact you to schedule a video meeting between each session. This will allow for discussions, progress updates, and assistance if needed.

2 Modeling

2.1 Specifications

You are Camille and you have recently started working in a computer science research lab. You are contacted by Alix, who works at a fiber optic installation company : Aerian Fiber. Her company is facing an optimization problem and cannot find a satisfactory solution. Here is a transcript of your first meeting.

ALIX: A few years ago, the government launched a public financing plan for the installation of fiber optics. These subsidies are primarily aimed at the deployment of optical networks in "non-conventional" areas. These are areas where no operator has yet deployed an optical network, mainly because the investment/benefit ratio is not attractive enough. Mostly in rural areas. Our company responds to this type of demand. Our specialty is using existing telecom infrastructure and attaching fibers to it.

CAMILLE: Attaching ? In the aerial network ?

ALIX: Yes, exactly. For underground network deployment, it's necessary to pass the fibers through already buried conduits or even dig to install conduits. It's very expensive, and the local governments that finance us rarely have the means. So we prefer to use the existing poles along the roads. There are already almost everywhere, carrying electric or telephone installations. We can add an optical fiber to them. Sometimes the pole needs to be replaced, but it's much cheaper than burying the fiber.

CAMILLE: Ok, I get the idea, but how can our lab help ?

ALIX: To respond to project tenders, we need to be competitive. This means, in particular, proposing the network requiring the least installation possible.

CAMILLE: What do you mean by the least installation possible ?

ALIX: Well, we have access to the map of electric and telecom installations. These are essentially poles following roads and intersections. The cost of installing fiber along a road is known, and generally, it's proportional to its length.

CAMILLE: Ok, this is what we call a graph. Your intersections are the nodes, and the roads are the edges. We can associate a weight to each edge to represent the installation cost.

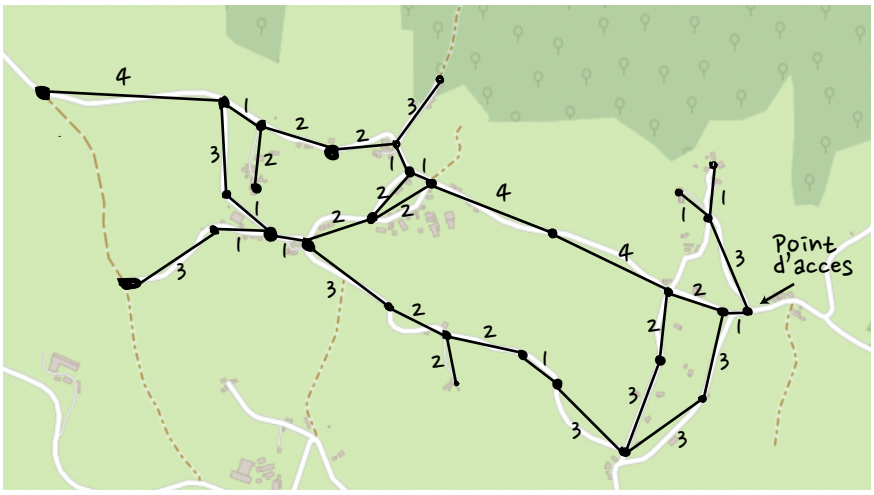
ALIX: Yes, that sounds about right.

CAMILLE: And this seems relatively simple to me. You are looking to create a structure that connects all the intersections in your graph. For this structure to have minimum weight, it must be a tree. You're looking for what we call the minimum spanning tree.

ALIX: Minimum weight ?

CAMILLE: Yes, the weight of a tree is the sum of the weights of the edges that compose it. So we are looking for the tree with the smallest total weight. The one that will require the least investment for fiber installation.

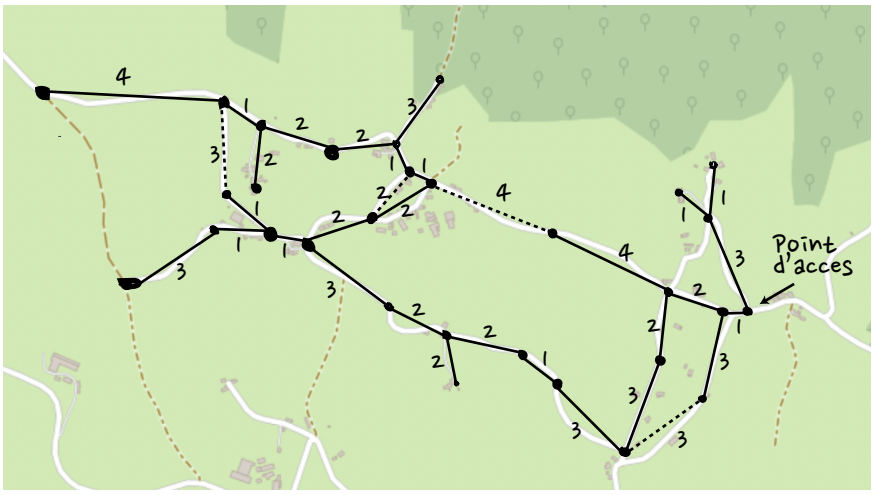
ALIX: Wait, let's do an example, it will be clearer. I must have an example of topology. A network with an estimated installation cost for fiber along the roads.



CAMILLE: You have marked "access point", is that the interconnection point with the national network ?

ALIX: Exactly. The access point must therefore serve all the neighborhoods. But fiber links are bidirectional, so the role of the exchange point is not special. We call them terminals in our jargon. A terminal can be a neighborhood or the exchange point. What's important to us is that there is a path between every pair of terminals.

CAMILLE: Ok, perfect, so I can calculate and represent the minimum spanning tree with solid lines. The dashed lines are links we don't need. This tree is easily found using Kruskal's or Prim's algorithm. It covers all the nodes and has the minimum weight. It's an algorithm with polynomial complexity, so it would take really large topologies before it becomes a problem.



ALIX: I'm not sure, what do you mean by "cover all the nodes" ?

CAMILLE: Well, all the nodes of the graph belong to the tree, they are covered.

ALIX: Okay, but why cover these intersections ? They are useless, there are no neighborhoods here !

CAMILLE: You're not wrong... We would need something that only covers the terminals.

ALIX: But maybe that's not a problem ?

CAMILLE: Well, I don't know. I mean, it's not an issue, it's the problem you're trying to solve. But I need to look into it further. What I suggest is to model the problem and come back to you so we can validate this approach.

ALIX: Perfect. Let's do that.

2.2 Optimization Problem

We are now ready to dive into this story. After the first meeting with Aerian Fiber, you get to work. The first step is to formalize the optimization problem. Write it in a standardized form :

— What are the input data ?

— What is a feasible solution ?

— What is the weight of a solution ?

You recall that there are lists of classical optimization problems and that it might be a good idea to start by checking if there is a corresponding problem. It's probably a network problem and, in particular, a tree problem. Such problems can be found in Viggo's compendium [1].

Question 1. *Search the compendium for the problem that you believe corresponds to the one you are tackling.*

Hint. *To know if you've found the right problem, copy the web address of the page describing the problem. It should be in the form :* `https://www.csc.kth.se/~viggo/wwwcompendium/nodeXX.html` *where XX is a number. Calculate the hash of this address using this site* `https://www.sha1.fr/`*. If you've found the correct problem, the hash of the address should be*

$$0342e07ad89553a6052357b732221f69e58bd2f9$$

Question 2. *Include the problem you found in the compendium in your report, keeping the formalism.*

2.3 NP-Completeness

Congratulations, you've found a problem that seems to correspond to the one described by Alix, your contact at Aerian Fiber. You decide to discuss your idea with Charlie, who works with you and is well-versed in these topics.

CAMILLE: Here's what I managed to identify using the compendium.

CHARLIE: Not bad. I think you're right, but I'm not sure it's good news.

CAMILLE: How so ?

CHARLIE: You know that all the problems on this list are NP-complete, right ? There is no known polynomial-time algorithm to solve any of them.

CAMILLE: Indeed...

CHARLIE: True, indeed... But, no offense, I don't think you're the one who's going to prove that P = NP.

CAMILLE: Okay, thanks. Remind me never to ask you for help again !

CHARLIE: Well, who knows, maybe I'm wrong about you !

CAMILLE: Exactly.

CHARLIE: So, I have a little challenge for you. I think I know how to prove that this problem is NP-Complete. It shouldn't be too hard for you, right ?

CAMILLE: Challenge accepted ! So, first, we're talking about the associated decision problem. It doesn't make sense to talk about NP-completeness for an optimization problem.

CHARLIE: True, that's an abuse of language. But you're right, we're talking about the decision problem. So instead of looking for a minimum-weight tree, we're looking for a tree whose weight is less than a parameter K.

CAMILLE: That works for me. And clearly, the problem is in NP.

1. `https://www.csc.kth.se/tcs/compendium/`

CHARLIE: Yes, for sure. If we have a positive instance and a solution tree, we can verify in polynomial time that this tree meets the constraints and the weight limit K.
CAMILLE: Good. I suppose you also want a polynomial reduction ?
CHARLIE: Of course... But I'm happy to give you a hint ! It's a reduction from Exact Set Cover to our problem.
CAMILLE: Well... give me 5 minutes ?
CHARLIE: Listen, I have a meeting starting soon. Send me your proposal by email. I should have time to look at it—my afternoon doesn't seem too

You return to your office with the firm intention of proving to Charlie that you have what it takes. You open a browser and start by finding a definition of Exact Set Cover [2]. You then begin searching for a suitable reduction.

Question 3. *Find a polynomial reduction that could be suitable. You can discover it yourself or find it on the Internet, but in the latter case, you should make it your own. In any case, integrate the proof of NP-completeness (membership in NP and reduction) into your report.*

2.4 Theoretical Results

Once your polynomial reduction is completed, you send it to Charlie. You don't wait for his response to reread the theoretical results presented in the compendium and to make a quick call to Alix. You quickly discuss your model to ensure everything fits well. Later in the afternoon, you meet Charlie again.

CAMILLE: So, this reduction ? What do you think ?
CHARLIE: Well, nothing at all, I didn't read it !
CAMILLE: ...
CHARLIE: However, I still worked for you. I quickly reviewed the known results for this problem and even dug up an approximate algorithm I found in an old course.
CAMILLE: Oh, great ! So ?
CHARLIE: Well, first, there's good news. An approximation algorithm exists.
CAMILLE: An approximation algorithm ?
CHARLIE: An algorithm that runs in polynomial time and returns a good solution.
CAMILLE: An exact solution ?
CHARLIE: Oh no, the problem is NP-complete, so unless P=NP, there is no polynomial algorithm to solve the problem exactly.
CAMILLE: So what's a good solution ?
CHARLIE: It's a solution that is within a certain distance from the optimal. In our case, the best known constant-ratio approximation algorithm provides solutions that are at most 1.55 times larger than the optimum. To be precise, it's $1 + \frac{\ln(3)}{2}$, but I'm nitpicking.
CAMILLE: Okay, so the distance to the optimal solution for an approximate algorithm is a multiplicative constant ?
CHARLIE: Yes, for minimization problems, it's an upper bound between the solution returned by the approximation algorithm and the exact solution. It's a value that's always greater than 1.
CAMILLE: And for maximization problems ?
CHARLIE: In that case, the ratio is always between 0 and 1.
CAMILLE: And is the approximation ratio always a multiplicative constant ?

2. `https://www.csc.kth.se/~viggo/wwwcompendium/node147.html`

CHARLIE: It's not always a constant. It can be a function depending on the input size of the problem. But I have bad news. From what I've read, the problem is APX-complete. That means there's no approximation whose ratio can get arbitrarily close to 1. This is what we call approximation schemes. An algorithm with an approximation ratio of $1+\epsilon$ whose complexity is in $\mathcal{O}(n^{\frac{1}{\epsilon}})$ or even sometimes polynomial [3] in n and $\frac{1}{\epsilon}$. That's not our case, because for us, no such algorithm exists. Let's leave that for another time.

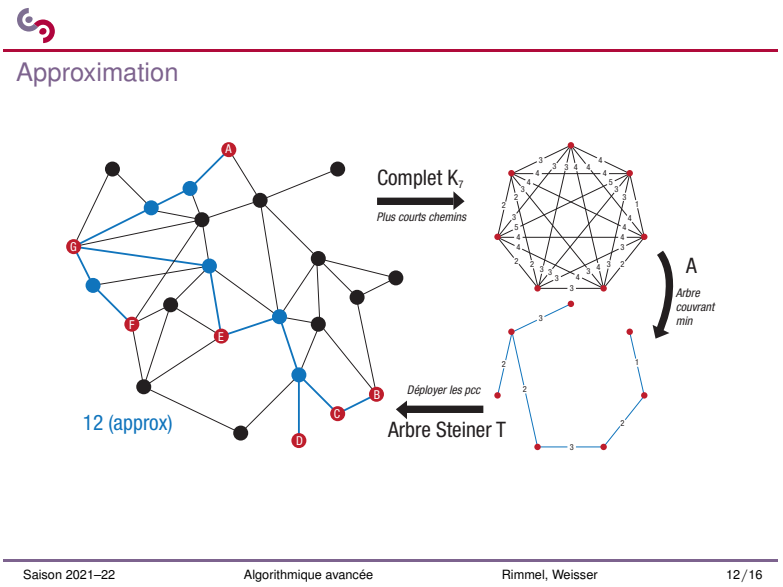
CAMILLE: That works for me. So, you mentioned an old course you found. What was it about ?

CHARLIE: Well, it was precisely about an approximation algorithm for your problem. It doesn't have the best ratio ; it's only 2. But it's simple to code, and you should be able to implement it quickly. That'll give you a small demo for the next time you go to Aerian Fiber.

CAMILLE: Great ! Send me your slides by email, I'll take a look tomorrow.

CHARLIE: Yep. One last thing, you'll see in my slides that the algorithm is presented for an unweighted graph, but it also works if the edges are weighted. And when you're done, we can talk about performance evaluation because there's something to explore there as well...

Question 4. *Write the 2-approximation algorithm and its proof. Integrate everything into your report.*



3. $\mathcal{O}(n^c \left(\frac{1}{\epsilon}\right)^d)$ with c and d two constants

Preuve de l'approximation

Preuve

- $T^* \leq T \leq A$
- On veut montrer que $A \leq 2T^*$
- Soit C un cycle eulérien obtenu en duplicant les arêtes de T^*, $C = 2T^*$
- On parcours les terminaux de C par ordre de première apparition et on construit C' en utilisant des pcc $C' \leq C$
- C' correspond à un cycle dans K, notons le C''
- On supprime une arête à C'', on obtient un arbre A''
- $A'' \leq C'' = C' \leq C = 2T^*$
- Tout arbre couvrant minimum de K a un poids $A^* \leq A''$
- $T^* \leq A^* \leq A'' \leq C'' = C' \leq C = 2T^*$

2.5 2-approximation

Your first day of work was dedicated to theoretical results, and you are eager to begin the second day to move on to implementation. Upon arriving at the lab, we stop by to see Charlie to gather his valuable advice before you start.

CHARLIE: So, this algorithm ? Did you understand it well ?
CAMILLE: Yes, I think so. It's not very complicated. Now, I'm tackling the development.
CHARLIE: What language and library are you going to use ?
CAMILLE: I'm not sure yet. I'm thinking that if I want it to be fast, it's better to code it in C.
CHARLIE: Oh yeah ? But what's more important ? That you develop quickly, or that the program runs quickly ? Because for a demo, it's better if you code quickly. Nothing in your algorithm is too costly, so you don't need to optimize the code at all costs. If I were you, I would code it in Python with the NetworkX library. It includes all the classic graph algorithms, all the ones you need. But do as you wish.
CAMILLE: Better the machine waste time than me...
CHARLIE: Exactly... Let me quickly show you. We'll start with basic manipulations. For example, here's how to import the networkx module and create a function that builds the graph for the 2-approximation. You see, we can create an empty graph and add nodes one by one using the `add_node(i)` method, which takes the node label as a parameter ; or several at once with `add_nodes_from(seq)`, which takes a sequence of labels. It's the same for edges. Here, they don't have weights, but you can certainly add weights. You can find the documentation for these functions online [4].

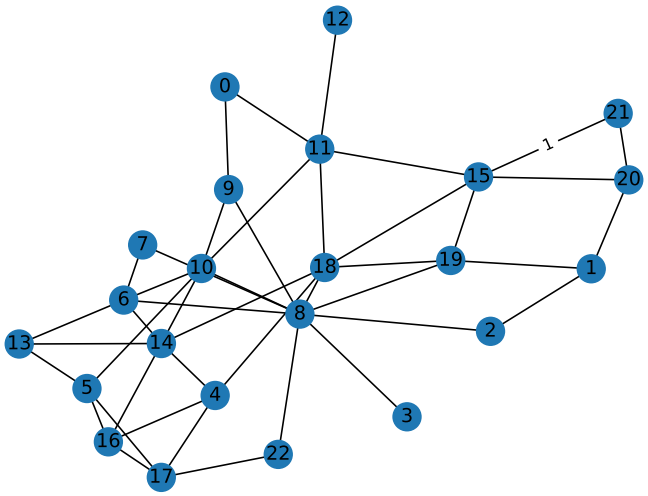
```
import networkx as nx

def create_test_graph():
    G = nx.Graph()
```

4. https://networkx.org/documentation/networkx-1.7/reference/classes.graph.html

```
    G.add_node(0)
    G.add_nodes_from(range(1,23))
    G.add_edge(0, 9)
    G.add_edges_from([(0, 11), (1, 2), (1, 19), (1, 20), (1, 2),
    (2, 8), (3, 8), (4, 14), (4, 16), (4, 17), (4, 18), (5, 10),
    (5, 13), (5, 16), (5, 17), (6, 7), (6, 8), (6, 10), (6, 14),
    (6, 13), (7, 8), (8, 9), (8, 10), (9, 10), (10, 11), (10, 14),
    (11, 12), (11, 15), (11, 18), (13, 14), (14, 16), (14, 18),
    (15, 18), (15, 19), (15, 20), (15, 21), (16, 17), (17, 22),
    (18, 19), (18, 8), (19, 8), (20, 21), (22, 8)])
    return G
```

CHARLIE: We can easily retrieve the list of nodes and edges in the graph. We can set or modify the weight of an edge. And most importantly, we can display the graph using the `matplotlib` module and save it as a PDF. It's quite handy for testing.

```
def test():
    G = create_test_graph()
    print(G.nodes)
    print(G.edges)
    G[21][15]["weight"] = 1

    pos = nx.spring_layout(G)
    nx.draw(G, with_labels=True, pos=pos)
    labels = nx.get_edge_attributes(G, "weight")
    print(labels)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    plt.savefig('test_graph.pdf')
    plt.show()
```



CHARLIE: Well, there are still some limitations. For instance, we don't really control the placement of the nodes. The library tries to spread them out as best as possible, but it's not always readable. Anyway, here's our test graph.

CAMILLE: It might be worth checking that we didn't make any mistakes when entering it, but it doesn't matter. It will do.

CHARLIE: Yes, indeed, it's not crucial. You probably won't use this graph much, except for a few tests. Eventually, you will likely need to generate a large number of random graphs. That's also relatively easy. The library allows us to generate them according to different models. For example, here's how to generate a binomial random graph.

CAMILLE: Like the binomial distribution ?

CHARLIE: Yes. We choose the number of nodes we want and a probability of an edge appearing between two nodes.

CAMILLE: There's no guarantee that the graph will be connected ?

CHARLIE: No. But we can simply test if a graph is connected and regenerate one if needed.

```
def random_binomial_graph(n, p=0.5):
    G = nx.generators.binomial_graph(n, p)
    while nx.is_connected(G) is False:
        G = nx.generators.binomial_graph(n, p)
    return G
```

CAMILLE: What does the value 0.5 associated with p in the function definition correspond to ?

CHARLIE: It's an optional parameter. If you only provide one parameter when calling the function, the second will automatically be 0.5.

CAMILLE: Convenient.

CHARLIE: Of course, all the `networkx` generators are documented [5]. And there are not only random generators. For example, there is a method to create a complete graph : `nx.complete_graph`, which takes as a parameter either the number of nodes or a sequence containing the node labels.

CAMILLE: The label sequence ?

CHARLIE: Yes, by default the n nodes are numbered from 0 to $n-1$, but you can associate them with other integers or even letters. So, for instance, you can pass the list `["A", "B", "C"]` as a parameter. The complete graph will then have 3 nodes labeled "A", "B", and "C".

CAMILLE: Coming back to random generation, I see that you used a built-in function to test if a graph is connected. That's pretty handy. Are there others ? I'm thinking particularly of the 2-approximation. To implement it, I need shortest paths and a minimum spanning tree.

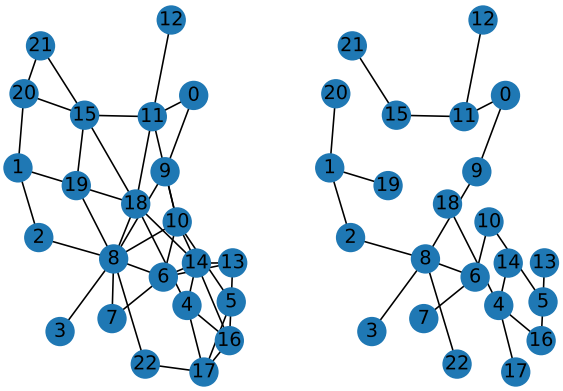
CHARLIE: Yes, exactly. That's why `networkx` is so convenient. For instance, we can add random weights to all the edges of the graph and compute a minimum spanning tree. There's a dedicated function. It's easy to integrate, and we can combine it with the "subplots" from matplotlib.

5. `https://networkx.org/documentation/stable/reference/generators.html`

```
def test_spanning_tree():
    G = create_test_graph()
    for u, v in G.edges:
        G[u][v]["weight"] = random.randint(1, 10)
    T = nx.algorithms.maximum_spanning_tree(G, weight="weight")
    print(T)

    plt.subplot(121)
    pos = nx.spring_layout(G)
    nx.draw(G, with_labels=True, pos=pos)

    plt.subplot(122)
    nx.draw(T, with_labels=True, pos=pos)
    plt.savefig('test_minimum_tree.pdf')
    plt.show()
```

CAMILLE: Indeed, the computation of the minimum spanning tree is almost free. But I don't quite understand the `subplot` part. What do 121 and 122 correspond to ?

CHARLIE: We can tell `matplotlib` to draw multiple times on the same graph. The command `plt.subplot(121)` tells `matplotlib` to split its space into one row and two columns. This is the meaning of the hundreds and tens in 121. Each section of the space is numbered starting from the top left. This is what the unit refers to. Thus, 121 corresponds to the left cell, and 122 to the right cell. When in doubt, it's best to look at the `matplotlib` documentation. It is quite comprehensive and there are several ways to achieve the same result [6].



CAMILLE: Perfect, we indeed have a spanning tree, 21 edges for 22 vertices. Well, obviously we need to check the weights of the edges.

6. `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.subplots.html`

CHARLIE: Yes, let's trust Python on this one. Come on, while we're at it, let's look for the shortest paths. It works well too. We use the function `nx.algorithms.all_pairs_dijkstra_path`. It takes a graph as a parameter and returns the shortest paths. For the example, we will display the path between vertex 0 and vertex 22.

```
def test_shortest_paths():
    G = create_test_graph()
    for u, v in G.edges:
        G[u][v]["weight"] = random.randint(1, 10)

    distances = dict(nx.algorithms.all_pairs_dijkstra_path(G, weight="weight"))

    pos = nx.spring_layout(G)
    nx.draw(G, with_labels=True, pos=pos)
    labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    plt.savefig('test_shortest_paths.pdf')
    plt.show()

    print(distances[0])
    print(distances[0][22])
```

CAMILLE: Ok, that seems quite clear to me. Except maybe the value returned by the function `all_pairs_dijkstra_path`. Why is it necessary to convert it into a dictionary ?

CHARLIE: What the function returns is an iterator, an object that you cannot manipulate directly with the bracket notation. You need to convert it into a dictionary.

CAMILLE: And the function only returns the shortest paths, not their weights ?

CHARLIE: No, you need to calculate that afterwards if you need it. Otherwise, you can use the function `all_pairs_dijkstra_path_length` but it only returns the distances, not the paths...

CAMILLE: That's a bit of a hassle, isn't it ?

CHARLIE: Yes, I admit. But for what you have to do, the work is already quite well laid out. You mainly need to put the pieces back in the right order.

CAMILLE: You're right, I have everything I need to start coding the approximation, and we can discuss performance evaluation later. You mentioned that last time ; I'm interested.

CHARLIE: No problem, we'll talk about it again. I'll send you the Python program with the various tests we just saw right away.

Question 5. *Implement the 2-approximation, test it on simple examples and then on randomly generated graphs.*

Your two approximations should return a list of edges (pairs of vertices).

Take inspiration from Charlie's functions. They are in the file ***`test_networkx.py`****. The file also contains a function to verify that a solution is valid.*

2.6 Series of Random Graphs

Now that your 2-approximation is coded, you schedule a meeting with Alix to present your initial results. The date is set for next week. You prepare some slides to illustrate the problem modeling

and the 2-approximation, but you're not entirely satisfied with what you're writing. In particular, the factor of 2 seems quite large. You're afraid Alix won't think much of it. It's true that a solution worth twice the optimal isn't such great news... Nevertheless, Charlie seemed quite confident, so you decide to ask him how to present things.

CAMILLE: Mission accomplished, the 2-approximation is developed.

CHARLIE: Bravo ! Does it run quickly ?

CAMILLE: Yes, in the end, there was no need to optimize the code at all costs. At least not for a proof of concept. So, I'm going to ask for your advice once again. I'm worried that the 2-approximation won't be satisfactory for Aerian Fiber. I feel like twice the optimal solution is still quite a lot.

CHARLIE: Yes, it is a lot, but remember, we're talking about the worst-case scenario. What this 2-approximation guarantees is that the approximate solution will never be more than twice the optimal. In practice, the solutions will be relatively close to the optimal. It really takes some twisted instances to get a ratio of 2. Actually, it's a good exercise.

CAMILLE: To find a bad instance ?

CHARLIE: Yes. Because in the end, the proof of the 2-approximation only tells us that the solutions are at most 2 away. Nothing tells us there isn't another proof that guarantees the solutions are within 1.5 or even less. However, if we find a bad instance—an instance where our approximation algorithm performs poorly, say at a distance of $2 - \epsilon$, for example—that would mean our bound is tight [7]. It would mean the best approximation ratio we can prove for this algorithm is 2.

CAMILLE: I'll start looking for such an instance. Was there nothing about this in your courses ?

CHARLIE: No, the professor didn't talk about it. We discussed it for other problems, but not for this one.

CAMILLE: In any case, this bad instance is important to me because it will help me know if the proof of the ratio is good. But it won't help me convince Alix that my algorithm performs well in practice.

CHARLIE: True. However, what could help is generating a large number of random instances, running your algorithm, and looking at its average performance. For example, you could write a function that takes a series of graphs as input and returns the average value of the 2-approximation. Then, you generate sets of similar graphs. For example, series of n graphs, of size s, following the same random generation model.

CAMILLE: Okay, so the idea is to vary the parameters one by one to see their impact on the algorithm's quality ?

CHARLIE: That's the idea. For instance, you use the binomial random graph model and generate 100 graphs of sizes 10, 20, 50, 100, and 200. Each time, you calculate the average weight of the approximate solution. This allows you to see how it evolves with the size of the graph. You can repeat the operation on graphs of the same size but vary the number of edges, for example, by increasing or decreasing the probability of an edge appearing between two vertices.

CAMILLE: Well, don't reduce it too much, or generating random graphs will take too long since the graph needs to be connected. If the probability is too low, it will be hard to get connected graphs.

CHARLIE: True. And what we've just discussed for binomial random graphs can also be done with other types of graphs to see how they perform. You can make some nice graphs

7. in English, tight bound

using `matplotlib` to illustrate it all. It'll look a bit more serious, which should reassure Alix.

CAMILLE: Yes, but how do I convince them the results are good ? I mean, I'll make a graph with the average weight of the solutions, but there's no comparison.

CHARLIE: You're touching on something very important. Ideally, when doing this kind of exercise, you either need to be able to compute the optimal solutions or have a lower bound on the optimal solution. For each series, you could then calculate a reference, the average weight of the solution or the lower bound.

CAMILLE: But the problem is NP-complete, we can't compute the optimal solution's weight.

CHARLIE: Yes, we can. It just takes some time. For many problems, there are databases with test instances. For these instances, we know the optimal solutions. This allows us to test new algorithms. I'll check if I can find some for this problem. In the meantime, you can use the second option : find a lower bound.

CAMILLE: Not simple. We know that for k terminals, we need at least $k-1$ edges to form a tree. We could make a list with the cheapest edge for each of the k terminals. If we remove the most expensive edge from this list, we should have a lower bound. But I doubt it's relevant.

CHARLIE: That does seem like a very weak bound. It's the configuration where all the terminals are directly connected to each other, and each time with the cheapest edge. In fact, it doesn't take into account what can be costly, namely the intermediate vertices. When connecting two terminals, we sometimes have to go through vertices that aren't terminals.

CAMILLE: For that, we could look at the two terminals that are furthest apart. I mean, in the optimal solution, there exists a path between every pair of terminals. And this path is always at least as long as the shortest path. So, if I find the distance between the two furthest terminals, I have a lower bound.

CHARLIE: Exactly. And that doesn't prevent you from complementing this bound with the edges we mentioned earlier. This path between the two furthest terminals can be seen as the vertical spine of the tree we're looking for. All other terminals have to connect to it. And we know that for that, each terminal will have to pay at least the price of its smallest edge.

CAMILLE: Yes and no, because a terminal could be an intermediate vertex on the shortest path, so that doesn't quite work.

CHARLIE: Fair enough...

CAMILLE: ...

CHARLIE: If we have k terminals, the optimal solution O_k is greater than any solution for two terminals chosen arbitrarily among the k. And we know the optimal solution for two terminals is the shortest path. So, the optimal solution O_k is greater than the largest shortest path between two terminals.

CAMILLE: Yes, but that's already been said. Slightly differently, but it's already been mentioned.

CHARLIE: True, but in this version, we can generalize. If we know the exact solution for 3 terminals, we can build a lower bound for k terminals.

CAMILLE: Okay, and I suppose you know how to build a bound for three terminals ?

CHARLIE: Yes, do you think you can figure it out ?

CAMILLE: Your riddles are getting annoying... I'll get to it...

You return to your office with new ideas : find a bad instance, generate series of graphs, calculate approximate solutions, and compute average lower bounds. Enough to keep you busy until the next meeting with Alix.

Question 6. *Find the "best" of the worst possible instances. Try to construct an instance where the ratio between the solution provided by the 2-approximation and the optimal solution is the worst possible. Integrate this instance into your report, following the proof of the approximation ratio.*

Question 7. *Write a function that generates a series of n graphs of size s according to a given random model.*

Question 8. *Write a function that takes a series of graphs as input and returns the average solution.*

Question 9. *Write a function that takes several series of graphs varying according to a chosen parameter. Your function should plot a curve showing the average solution value as a function of the chosen parameter.*

Question 10. *Write a function that computes the lower bound of the optimal solution. Write a function that computes the average lower bound for a series of graphs. Integrate the lower bound value into the function that generates the curve.*

Question 11. *Write a resolution algorithm for the 3-terminal problem. Use this algorithm to write a lower bound for k terminals.*

3 The Second Meeting

Two weeks after your first exchange with Alix, you've made significant progress on the problem and the program that allows you to solve instances. You are fine-tuning the presentation you'll give at Aerian Fiber's offices when you receive an email from Charlie.

I found something that might interest you. There is a library of instances for your problem. It is available at this address[8]*. For some classes of instances, the library also provides optimal solutions. This should allow you to compare your algorithm!*
Using it is quite simple. You just need to install a Python module with the command `pip install steinlib`*. Then, with the following lines of code, you can load an instance. For example, here I load the* `b04` *instance, which is located in the file* `./data/B/b04.stp`*.*

```
from steinlib.instance import SteinlibInstance
from steinlib.parser import SteinlibParser

class MySteinlibInstance(SteinlibInstance):
    def __init__(self):
        self.my_graph = nx.Graph()
        self.terms = []

    def terminals__t(self, line, converted_token):
        self.terms.append(converted_token[0])

    def graph__e(self, line, converted_token):
        e_start = converted_token[0]
        e_end = converted_token[1]
        weight = converted_token[2]
```

8. `https://steinlib.zib.de`

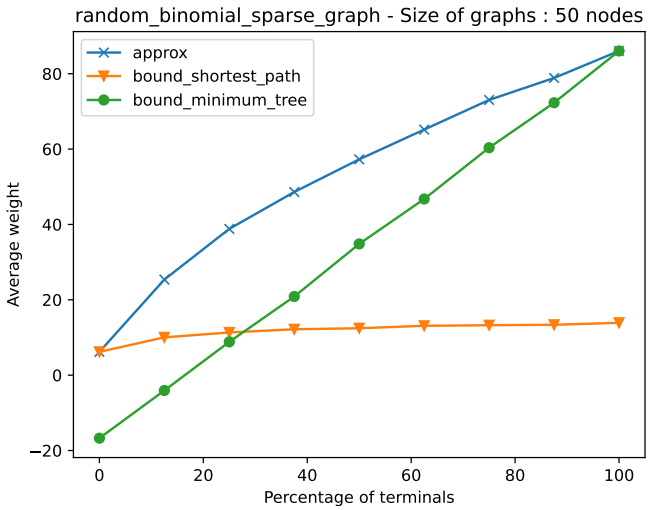
```
        self.my_graph.add_edge(e_start, e_end, weight=weight)


def load_B04():
    stein_file = "data/B/b04.stp"
    my_class = MySteinlibInstance()
    with open(stein_file) as my_file:
        my_parser = SteinlibParser(my_file, my_class)
        my_parser.parse()
        terms = my_class.terms
        graph = my_class.my_graph
    return graph, terms
```

You quickly visit the library's website, download the B-class instance, and write a few lines of code to test your algorithm on these instances. You integrate these results into your presentation before heading out to meet Alix, who is eagerly waiting for you. You begin to explain your findings...

CAMILLE: To sum up, we have managed to show that the problem you're facing is NP-complete. Simply put, this means we cannot find a fast and exact solution algorithm. However, we have identified a solution algorithm that is fast but provides an approximate solution. It's not the optimal solution, but it's close to the optimum.

ALIX: How close is it ?

CAMILLE: Well, in the worst case, it's at most twice the optimal. I know that's quite a lot, but we're talking about the worst case. I've run simulations, and in practice, I think it performs quite well. For example, on this chart, we have the performance of our approximation.



CAMILLE: In blue, with cross markers, we have the results of our approximation. It's an average. For each point, we generated 50 random graphs with 50 vertices and averaged the results. We see that the weight of the returned solution increases slowly as the percentage of terminals increases.

ALIX: Ok, so initially, the tree returned by the algorithm when we have very few terminals is worth about 5 on average. On the opposite end, when all the nodes in the graph are terminals, the solution's value averages around 85.

CAMILLE: Yes, that's right.

ALIX: So on the x-axis, 100% means that all nodes in the graph are terminals, but what does 0% terminals mean ?

CAMILLE: Actually, we have at least 2 terminals, and the percentage refers to the number of terminals among the remaining vertices.

ALIX: Ok, got it. And the other two curves ?

CAMILLE: These are lower bounds. We know that the optimal solution value is at least as high as these bounds. The first bound, in yellow with triangles, is based on the shortest paths. We know that the value of the tree we're looking for is at least the shortest path between two terminals. It's a good bound when we have few terminals. The second bound, in green with circles, is based on a minimum spanning tree. When we have many terminals, the tree we're looking for is close to a minimum spanning tree.

ALIX: Ok, so the optimal curve should lie somewhere between the upper curve, the approximation, and the two lower curves... It does seem that when the number of terminals is high, the gap between the bounds and the approximation is small. It's less clear when there are few terminals.

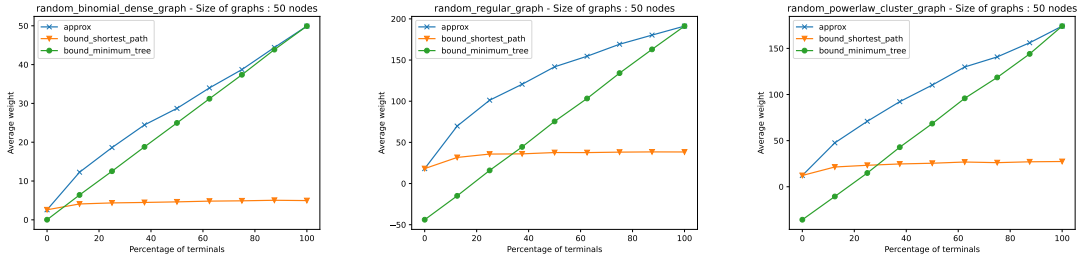
CAMILLE: That's true. But in this case, our bounds are probably not very good. In fact, we even see that the bound based on the spanning tree becomes negative. That doesn't really make sense...

ALIX: Why do the bound curves cross the approximation curve at 0 and 100% ?

CAMILLE: For two terminals, we know that the optimal solution is the shortest path connecting them. The first bound is thus equal to the optimal solution. On the other hand, when all the nodes in the graph are terminals, we know that the optimal solution is exactly a minimum spanning tree. That's the value given by the second bound. Things get more complicated in between.

ALIX: Here, we're in a fairly specific case, with 50-node randomly generated graphs. Is that close to the cases I'll be dealing with ?

CAMILLE: It's still to be validated, but it seems that the results we're getting are fairly representative. We've tested larger graphs, and we get similar results. Here, we generated random graphs following the binomial model, but we've also tested with other models, and we get similar results. Look.

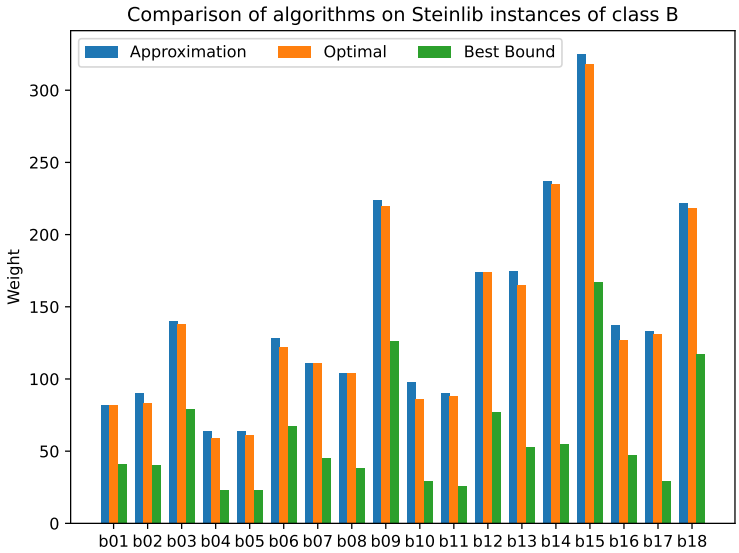


ALIX: Indeed, that's quite similar. Except maybe for the first one, where the approximation results seem particularly good.

CAMILLE: Yes, it's a dense graph, meaning it has many edges. In this case, the shortest paths are always very short. The spanning trees are small. It's a bit of a special case. The other two types of graphs are regular graphs, where all vertices have the same number of neighbors, and power-law graphs. These are graphs with a few vertices having many neighbors and most other vertices having few neighbors. Graphs with concentration points.

ALIX: I find these initial results really encouraging, but I'm still not entirely convinced that this approach is the best. The gap between the bounds and the approximation is still quite large. Can't we be more precise ?

CAMILLE: Yes ! At least for certain graphs. We recently found a library of instances for which we have the optimal solutions. We ran our approximation on each of them. Here's what we found.



CAMILLE: So far, we've looked at 18 instances from the library. These are the ones with 50 vertices, just like the random graphs we generated. On this chart, I show the value of the optimal solution, the approximate solution, and the best of the two lower bounds.

ALIX: Not bad at all ! The approximate solutions are very close to the optimal ones. But it's clear that the lower bound isn't great.

CAMILLE: No, not at all ! But what's positive is that the approximation seems, at least for these instances, to be empirically good. However, the bound we obtain is far from the optimum, making it difficult to draw conclusions about the quality of our approximation. Except, of course, in the case of dense graphs, but that's a very specific case...

ALIX: Yes, and it's not very relevant for us. The graphs we deal with are anything but dense... We're more likely to work with graphs containing few edges.

CAMILLE: Exactly. We can also add that the vertex degrees are quite low, and the graphs are probably planar or close to it.

ALIX: Planar ?

CAMILLE: A graph that can be drawn on a plane without any edges crossing.

ALIX: Oh yes, there are rarely crossings. Usually, when two lines cross, it's at a pole. So there's a vertex in the graph, and the edges don't cross. Anyway, I find these initial results very encouraging. What's next ?

CAMILLE: Well, I'm going to look at two aspects. The first is finding an exact solution method.
I want to use a Branch

Question 12. *Use the functions you have coded previously to generate curves describing the behavior of your algorithm. Integrate these curves into your report in the performance evaluation section, detailing what they represent and how they were created.*

Question 13. *Optionally, test your algorithm on instances from classes B and C. Observe the behavior and integrate it into your report, still in the performance evaluation section.*

4 Meta-heuristic and Exact Method

We have now reached the end of the scripted part of this lab. We hope this will provide you with the logic and mindset needed to best embody Camille in the remaining tasks, the most perilous ones...

Question 14. *If you are alone in your group, implement a Simulated Annealing algorithm. Include it in your report in a separate section to describe its functioning. Incorporate the results you obtain using it in the performance evaluation section. Compare these with the approximation algorithm.*

Question 15. *If you are two in your group, you may also implement a branch and bound algorithm to solve instances and refine your analysis. If this method seems difficult to implement, you can also choose to carry out another heuristic method of your choice.*