Project: Implementation of a branch-and-bound algorithm for the minimum vertex cover problem

The goal for this project is to implement a branch-and-bound algorithm for the minimum (vertex-weighted) vertex cover problem with the help of an LP-solver. You may use the programming language and optimisation package of your choice. One possibility would be to use Python and an optimisation package like `PuLP` or `cvxpy`. A basic pseudo-code for the algorithm is given below, but you are free to structure your algorithm any way you want. Any improvements and heuristics are encouraged.

1 Algorithm

The basic step of the algorithm is to *branch* on a variable, which means that we try to set the variable to 1 (including the corresponding vertex in the vertex cover), and then try to set it to 0 (excluding the corresponding vertex from the vertex cover). If we do this recursively for each variable, then we would try all possible subsets of vertices and certainly find the smallest cover, but it would take prohibitively long time even for a graph with only, say, 50 vertices.

To improve on this idea, we first solve the LP-relaxation of the problem. Some variables may then be set to 0 (we exclude them) and some to 1 (we include them). Among the remaining fractional variables, we choose one to branch on and recursively call the algorithm on smaller graphs.

We can fix a variable x_i to 0 or 1 by modifying the graph as follows:

- If x_i is to be set to 1, then the corresponding vertex v_i is included in the vertex cover and the graph is modified accordingly to $G - v_i$.
- If x_i is to be set to 0, then v_i is not included in the vertex cover, hence its neighbors $N(v_i)$ must all be included to cover the edges incident on v_i. The graph is then modified to $G - v_i - N(v_i)$.

In the pseudo-code below, the cost of the currently included vertices is passed along in the recursive calls (in the parameter Z) so that the algorithm can know the total cost of a vertex cover in the initial graph whenever one is found in the current (smaller) graph.

The algorithm maintains a global variable UB that contains the value of the currently best found solution. This is used to terminate the search early in some branches when we know that we cannot find a better solution.

2 Pseudo code

Input : A vertex-weighted graph $G = (V, E, c)$, the total additional cost Z of the currently selected vertices

Output : On termination, the global variable UB contains the value of a minimal weighted vertex cover of G

```
1  begin
2      Solve the LP-relaxation to obtain an optimal extreme point x and a lower bound
         LB = LP-Opt(G) + Z
3      if LB ≥ UB then
           /* The best possible solution in this branch is no better than
              the best solution found so far, so we can stop the search in
              this branch                                                  */
4          return
5      end
6      if x is integral then
           /* There is no need to branch further.  Check if the solution is
              better than the best solution found so far                   */
7          if LP-Opt(G) + Z ≤ UB then UB = LP-Opt(G) + Z;
8          return
9      end
10     Let S_p = {v_i : x_i = p} for each p ∈ {0, 1/2, 1}
       /* Include all vertices in S1 and exclude all vertices in S0 (and
          hence include their neighbours)                                  */
11     Let G' = G − S_1 − S_0 − N(S_0)
12     Let Z' = Z + Σ_{v_j ∈ S_1 ∪ N(S_0)} c_j
13     Choose a branch vertex v_i ∈ S_{1/2}
       /* Try including vi in the vertex set                               */
14     Recursively call the algorithm with the graph G' − v_i and additional cost Z' + c_i
       /* Try excluding vi from the vertex set                             */
15     Recursively call the algorithm with the graph G' − v_i − N(v_i) and additional cost
         Z' + Σ_{v_j ∈ N(v_i)} c_j
16 end
```

3 Tips

- Make sure that your LP-solver returns extreme points. This is usually the default behavior but could require setting some additional parameter.
- Make sure that you have a correct algorithm before you try to make any improvements.
- Once you have a working algorithm, you can compare different strategies for choosing branch variable and value. A good choice can greatly improve the performance.
- For instances that take too long to solve, it may be possible to prove a lower bound L on the optimum by fixing UB $= L$ at the start of the algorithm. If the algorithm stops

without finding any solution, then you have effectively proved that the optimum is $> L$.

4 Deliverables

1. The source code of your program and a short user's guide: how do I run your program? which parameters can be modified and how?
2. In your presentation, mention the following:
   - Explain why the Nemhauser-Trotter theorem is needed for the algorithm to be correct.
   - Give a list of instances for which your program has found an optimal solution. For each instance indicate: its optimal cost, the time used to solve it, the number of LP-relaxations solved during the execution, the total number of variables fixed by the LP-relaxations (S_0 and S_1 as opposed to those fixed by branching).
   - A list of instances for which your program did not find an optimal solution. For each instance indicate the cost of the best solution that you found.
   - Explain why finding an optimal solution to the instance `mk11-b2` is so fast.

5 Instances

A set of instances is provided in which a vertex-weighted graph is represented as a text file as follows. The first line lists the number of vertices `n` and the number of edges `m`. Then, follows a line with the vertex weights. Each subsequent line represents an edge with vertices taking values between `1` and `n`. Each edge $\{u, v\}$ is listed once as `u v`, where $u < v$.

Example:

```
4 5
1 1 1 1 1
1 2
1 3
1 4
2 3
3 4
```

The graphs have been taken from `https://networkrepository.com`. All edge weights have been removed and a weight 1 has been added to each vertex.

DIMACS

- `johnson8-2-4`
- `johnson8-4-4`
- `johnson16-2-4`
- `MANN-a9`
- `hamming6-2`

- C125-9
- keller4
- brock200-1

Infrastructure Networks

- inf-USAir97
- inf-euroroad

Ecology Networks

- eco-foodweb-baywet

Miscellaneous Networks

- mk11-b2

Brain Networks

- bn-macaque-rhesus_brain_1

Economic Networks

- econ-mahindas

DIMACS10

- delaunay_n10