1 Greedy Algorithms for $\mathcal{NP}$-complete Problems

In general, when faced with an $\mathcal{NP}$-complete problem, we do not expect to find an algorithm that is both polynomial time and correct. This leaves us two options. First, we could have a super-polynomial time, accepting the limitations that come with that. Second, we could sacrifice correctness for speed.

1.1 Metric Traveling Salesperson

Suppose I want to solve Traveling Salesperson but I know the following information:

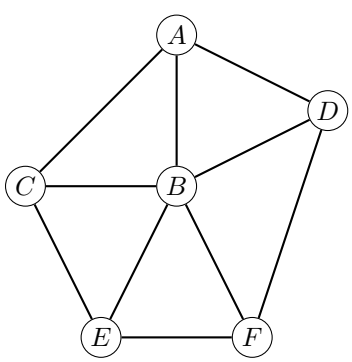
- The graph is *complete* : for all pairs of vertices, there is an (undirected) edge between them.
- Distances on the graph obey the triangle inequality; $c_{uv} + c_{vw} \geq c_{uw}$.

If we are going to have an incorrect algorithm, we want to know how inaccurate our output is. One way to measure this is *approximation ratio*. The following are constraints on an efficient approximation algorithm:

- The path it outputs is *valid*: it will visit every vertex exactly once.
- The cost of the path I output might not be optimal. Ideally, though, I will be able to know "how bad" it will be.
- The algorithm must be polynomial time.

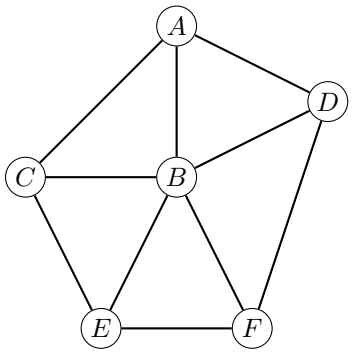
1.2 Tightness of Approximation Bound

Consider the following graph; for simplicity, not every edge is shown. If an edge is drawn on the paper, it has a cost of 1. If an edge is not drawn, it has a cost of 2.



Question 1. What is the optimal TSP journey for that graph?

Here is the graph again. If an edge is drawn on the paper, it has a cost of 1. If an edge is not drawn, it has a cost of 2.



Question 2. What is the Minimum Spanning Tree for that graph?

Question 3. What is the worst TSP journey our algorithm can find on this graph? Can you generalize to a larger number of vertices?

Question 4. How could you evaluate how likely that worst-case outcome is? How do we evaluate an approximation algorithm experimentally?

1.3 Hardness of Approximation

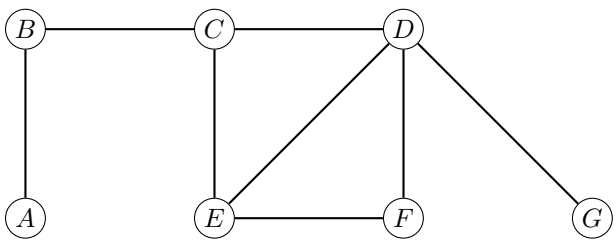
Do we expect to be able to approximate TSP without the triangle inequality requirement? Suppose you had a ρ-approximate solution. What could you do with it?

1.4 Vertex Cover

Recall the VERTEX COVER problem. The decision version of the problem was: given a graph G and integer k, determine if there is some subset of the vertices V', $|V'| \leq k$ such that each edge is incident to some vertex in V'.

The *optimization* version is to find a vertex cover V' of minimal total size. In other words, given a graph G, find the minimum possible value of k for which the decision version will return true with this graph.

Here is a sample graph:



Question 5. What is the optimal Vertex Cover for this graph?

Question 6. How does the following algorithm perform on this graph? In general?

$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$ **do**
  Select any $e = (u, v) \in E'$
  Add u, v to C
  Remove all edges incident to u or v
return C

Here are some additional study questions related to Vertex Cover approximation:

1. Draw graphs such that our algorithm never finds the optimal solution. Try to do this with one where the optimal cover is of even size.
2. Recall the similarity between Vertex Cover and Independent Set. Suppose we calculate C with our algorithm but want to solve Independent Set. Is "keep $V - C$" a 2-approximation for Independent Set? Why or why not?
3. Here's another greedy algorithm for Vertex Cover:

   - Select vertex of highest degree.
   - Add it to C
   - Remove it and all incident edges

   Prove that this **does not** have ratio 2.

1.5 Load Balancing Problem

Suppose we have m machines and a set of n jobs. Each job j takes time t_j to complete. We want to assign each job to a machine so that the total time for the last machine to finish is minimized.

One of the key ideas in this problem, compared to the previous one, is to *reason about bounds on the optimum*. We are going to see a 2-approximation and then an improvement that produces a better result.

First, an example. What is the optimal solution to the following input?
Example: six jobs, $t = \{2, 3, 4, 6, 2, 2\}$, $m = 3$

How does the following algorithm perform on the same input?

`// Define` $T_i = \sum_{j \in A_i} t_j$
  for each job j **do**
    Let M_i be the machine with minimal T_i
    Assign job j to T_i

Question 7. Can you find two ways to express a *lower bound* on the optimal makespan?

Question 8. What can we say about the last job j placed on the highest-makespan machine during the run of the above algorithm?

Question 9. Is this the best analysis for that algorithm? Is there an input that can be "the worst" (or nearly the worst, within the guarantee)?

Question 10. How can we improve the algorithm to avoid this case? Does it improve the situation in general?