Graph Algorithms (Spring 2026)

7. Approximation algorithms

Teacher: Arnaud Casteigts *Assistant: Himika Das*

What can we do when faced with an NP-hard optimization problem? The main options are: 1) solve it anyway in exponential time; 2) restrict the type of instances we handle (special cases may be polynomial time solvable); 3) abandon the goal of finding an optimal solution, trying to approximate it in polynomial time.

Approximation algorithms fall into the third category: we want a solution that is as good as possible in polynomial time. Hard problems are not all equal: some are more approximable than others. Suppose we are dealing with a *minimization* problem. Here are three levels of approximability:

- k-approximation: Here, the algorithm guarantees that our solution never exceeds k times the optimum OPT, for some constant value k that depends on the problem.
- $(1+\epsilon)$-approximation: Here, we can (asymptotically) get as close as we want to the optimum, with ϵ being a chosen parameter. Of course, the closer we get, the slower the algorithm. For example, achieving a quality of $\epsilon = 1/5$ may cost $O(n^5)$ time, and a quality of $\epsilon = 1/10$ may cost $O(n^{10})$. Still, for any fixed ϵ, time remains polynomial. Such algorithms are also called *polynomial-time approximation scheme* (PTAS).
- Inapproximable: This is the worst scenario, where no k-approximation algorithms exist, whatever the constant k.

The same classification applies to maximization problems, by inverting the factor. For example, a k-approximation guarantees a solution of quality OPT/k.

7.1 Minimum Vertex Cover

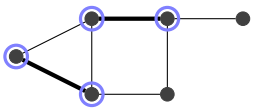
Let $G = (V, E)$ be a graph. A **vertex cover** in G is a set of vertices $V' \subseteq V$ that (collectively) touch all the edges. Intuitively, you can imagine a scenario where you want to monitor all the links of a computer network, by installing a costly software on the computers. Thus, you want to have the smallest possible solution. For example:

Similarly to graph matchings (which may be maximal or maximum), we can distinguish here *minimal* solutions (no element can be removed, like the one on the left) and *minimum* solutions (no smaller solution exists, like the one on the right). Unfortunately, finding a *minimum* vertex cover is NP-hard. But it is 2-approximable!

7.1.1 A 2-approximation algorithm for Minimum Vertex Cover

We will exploit a nice relation betwen vertex covers and matchings. Suppose we have a function that computes a *maximal* matching (not necessarily maximum, here maximal is enough). We can use it to obtain a vertex cover as follows:

```
vertex_cover(G):
    M ← maximal_matching(G)
    S ← ∅
    For each edge uv in M:
        Add u to S
        Add v to S
    Return S
```



In other words, we compute a maximal matching (which is easy with a greedy algorithm), then we select all the vertices that are matched by the matching, which gives us a vertex cover. It turns out that this vertex cover is a 2-approximation. To prove this, we need to show two things: (1) The computed vertex cover is *valid*, and (2) Its size is at most twice the size of a minimum one.

Lemma 7.1 (Validity)**.** *All the edges of G are covered by at least one vertex of S*

Proof. By contradiction, suppose there exists an edge uv that is not covered. By construction, this implies that neither u nor v touch any edge of the matching (otherwise, they would have been added to S). Thus, $M + uv$ would still be a matching, which contradicts the fact that M is maximal. □

Let S^* be an actual minimum vertex cover. We will show that $|S| \leq 2 \cdot |S^*|$.

Lemma 7.2 (Quality)**.** $|S| \leq 2 \cdot |S^*|$

Proof. By definition, a vertex cover must cover all the edges of G. In particular, it must cover all the edges of a maximal matching M. Since the edges of M are disjoint, we need at least one *distinct* vertex for each of them (thus, $|M| \leq |S^*|$). Now, observe that our algorithm selects two vertices per edges of M (thus, $|S| = 2 \cdot |M|$). As a result, $|S| = 2 \cdot |M| \leq 2 \cdot |S^*|$. □

Clearly, the above algorithm runs in polynomial time; essentially, in the same time as the greedy algorithm for maximal matching. It is thus a polynomial-time 2-approximation algorithm for Minimum Vertex Cover.

Note that we have exploited here a so-called **duality** between two problems, namely:

$$|\,\texttt{Minimum Vertex Cover}\,| \;\leq\; 2 \cdot |\,\texttt{Maximal Matching}\,|$$

Many approximation algorithms are based on such relations between the hard problem we want to solve and an easier problem.

7.2 Traveling Salesman Problem

The traveling salesman problem (TSP) is one of the most studied problem in the field of combinatorial optimization. Given a set of n cities and a cost function $c(u, v)$ between some pairs of cities (represented by a weighted graph G), this problem consists of finding a cycle of minimum total cost that visits each city exactly once and returns to the starting point. This problem is known to be NP-hard already with unit cost everywhere (reduction from the Hamiltonian Cycle problem).

In terms of approximation, this problem is quite interesting because it admits several levels of approximability, depending on further restrictions. The general version of TSP is *inapproximable* (unless P=NP). We can gradually restrict the problem into Metric TSP, and then Euclidean TSP, both being still realistic for many applications.

7.2.1 Metric TSP

Metric TSP is a special case of the general TSP, where the costs are guaranteed to satisfy the triangle inequality: for all cities u, v, w, we have $c(u, v) \leq c(u, w) + c(w, v)$. In other words, it is always cheaper to go somewhere directly than to go there via an intermediate city. This assumption also implies that the instance is a *complete* graph. Interestingly, we obtain the same problem if we consider general TSP and allow cities to be visited several times (to be seen in exercises). Metric TSP is still NP-hard, but it admits a polynomial-time 1.5-approximation algorithm (presented below in Section 7.2.3).

7.2.2 Euclidean TSP

Euclidean TSP is, in turn, a special case of the Metric TSP, in which each city is specified by coordinates in the plane and the cost between two cities corresponds to their Euclidean distance. Euclidean TSP is still NP-hard, but this time, it admits a $(1 + \epsilon)$-approximation (i.e. a PTAS), meaning that one can (asymptotically) get as close as desired to the optimal solution, at the cost of a computation time that remains polynomial in n, but depends on the desired error ϵ. One such algorithm is Arora's algorithm (1996), which runs in time $n^{O(1/\epsilon)}$ (this algorithm generalizes to higher dimensions at a slightly higher cost). The general idea is to recursively decompose the space into small sections solved separately, then stitched together using dynamic programming. This algorithm is not part of the course, you can find it in Michel Goemans' course if interested: `https://share.google/rtk6WC5GUqRjB8gyQ`

7.2.3 A 2-approximation algorithm for Metric TSP

We will start by giving a 2-approximation algorithm that is simpler and actually uses a subset of the technique of the 1.5-approximation. The algorithm is quite simple:

1. Compute a minimum spanning tree T of G (using Kruskal or Prim, for example).
2. Perform a depth-first traversal of T, skipping cities that are already visited.

Why does it work?

First, observe that all the cities will be visited exactly once, so the solution we have computed (call it S) in indeed valid. What is the total cost of S in terms of weights? A depth-first traversal crosses each edge of the T twice, and the shortcuts we take cannot deteriorate the cost (due to the triangle inequality). We thus have:

$$\text{cost(S)} \leq 2 \cdot \text{cost(T)} \tag{1}$$

Now, how does the cost of an MST relate to the cost of an optimal TSP tour? Let SS^* be an optimal solution for the TSP. We can show the following:

Lemma 7.3. *cost(T) ≤ cost(S*)*

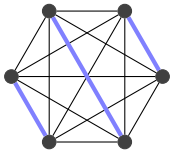
Proof. By contradiction, if cost(T) > cost(S^*), then we can remove an edge from the optimal tour S^* and obtain a better spanning tree than T, contradicting that T is an MST. □

Combining (1) with 7.3, we obtain that $cost(S) \leq 2 \cdot cost(T) \leq 2 \cdot cost(S^*)$.

7.2.4 1.5-Approximation Algorithm for Metric TSP (Christofides, 1976)

The algorithm follows the same strategy as before, but adds steps that use additional concepts from graph theory: *minimum perfect matchings* and *Eulerian cycles*. Let us begin by discussing these two notions.

Minimum perfect matching. A *matching* (already seen in the previous lecture) is a set of edges that do not share endpoints. A matching is *perfect* if it touches all vertices. The existence of perfect matchings is not guaranteed in general, but it is guaranteed in complete graphs with an even number of vertices (see below). Moreover, a minimum-weight perfect matching can be found in polynomial time (not described here).



Eulerian cycle. An Eulerian cycle is a cycle that passes exactly once through each *edge* of the graph. Note that this is not the same as a Hamiltonian cycle (which passes exactly once through each vertex). The Hamiltonian cycle problem is NP-hard, but the Eulerian cycle problem is easy: such a cycle exists if and only if the graph is connected and all vertices have even degree. If these conditions are satisfied, the cycle is also easy to find.

Christofides' algorithm uses these two notions on certain *subgraphs* of the input graph.

Christofides' algorithm

1. Compute a minimum spanning tree T (MST)
2. The number of vertices of *odd* degree in T must be *even*. We can therefore compute a minimum perfect matching M between these vertices in the original graph.
3. Take the union of M and T (possibly duplicating some edges)
4. Find an Eulerian cycle in this union, which now has only even degrees
5. Traverse the obtained cycle while skipping already visited vertices.

Explanation. In any graph (and in particular in T), the number of vertices of *odd* degree is *even* (Handshaking lemma). The subgraph induced by these vertices in the original graph is therefore a complete graph with an even number of vertices, so we can find a minimum perfect matching M in it. If we take the union $T \cup M$ (keeping edge multiplicity), all vertices will then have even degree in this union (either they already had even degree in T, or their degree became even thanks to the matching). Thus, a Eulerian cycle must exist in $T \cup M$.

Approximation factor of 1.5. The final tour S consists of traversing the Eulerian cycle found in $T \cup M$, while skipping already visited vertices. Thanks to the triangle inequality, these shortcuts cannot increase the total cost. Hence:

$$cost(S) \leq cost(T \cup M) \leq cost(T) + cost(M) \tag{1}$$

We already know that $cost(T) \leq cost(S^*)$ (Lemma 7.3). It therefore suffices to show that the cost of M is at most $0.5 \cdot cost(S^*)$. This is done in two steps. First, consider the complete subgraph G' induced by the vertices involved in M and let S'^* be an optimal tour in G'. Clearly, S'^* has smaller cost than S^*, because of the triangle inequality (a tour visiting only a subset of cities must be less expensive than one visiting all cities). Then we show:

Lemma 7.4. $cost(M) \leq 0.5 \cdot S'^*$

Proof. Any optimal tour itself defines a perfect matching (not necessarily minimum) by taking every other edge along the tour. It also defines another one: the remaining edges of the tour. The sum of the costs of these two perfect matchings is exactly equal to S'^*, so one of them is at most half, and therefore a *minimum* perfect matching among these vertices is at most that cost as well. □

In summary, we have:

$$cost(S) \leq cost(T \cup M) \leq cost(T) + cost(M) \leq cost(S^*) + 0.5 \cdot cost(S'^*) \leq 1.5 \cdot cost(S^*) \quad (2)$$

Beautiful, isn't it! :-D