Graph Algorithms (Spring 2025)

7. Approximation algorithms

Teacher: Arnaud Casteigts *Assistant: Matteo De Francesco*

What should we do when faced with an NP-hard problem? The main options are: 1) solve it optimally, but in exponential time; 2) restrict the type of instances we can handle, as some restricted classes of instances admit polynomial-time algorithms; 3) abandon the goal of finding an optimal solution, while still trying to approximate it.

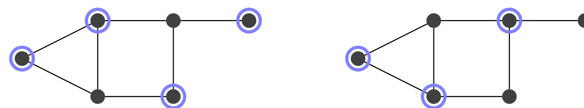
Approximation algorithms fall into the third category: we want a solution that is as good as possible in polynomial time. It turns out that hard problems are not all equal: some are more approximable than others. Suppose we are dealing with a *minimization* problem. Here are three levels of approximability:

- 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:

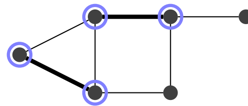


Similarly to graph matchings (which may be maximal or maximum), we can distinguish here *minimal* solutions (no element can be removed, like the one on the left) and *minimum* solutions (no smaller solution exists, like the one on the right). Unfortunately, finding a *minimum* vertex cover is NP-hard.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

$$|\texttt{Maximal Matching}| \leq 2\times |\texttt{Minimum Vertex Cover}|$$

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.

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 MST twice, and the shortcuts we take cannot deteriorate the cost (due to the triangle inequality). We thus have:

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

But how does the cost of an MST relate to that of the TSP? Interestingly, it cannot exceed it:

Lemma 7.3. *cost(MST) $\leq$ cost(TSP)*

Proof. By contradiction, if cost(TSP) $<$ cost(MST), then we can remove an edge from the optimal tour and obtain a spanning tree better than the MST, a contradiction. □

Combining (1) with 7.3, we obtain that $S \leq 2 \cdot \text{cost(TSP)}$.

7.2.4 A 1.5-approximation algorithm for Metric TSP (Christofides, 1976)

The algorithm relies on the same techniques as the 2-approximation above, and uses in addition the concept of *minimum-weight perfect matchings* and *eulerian cycles*. Recall that a perfect matching is a matching that touches all the vertices (see the previous lecture). As for eulerian cycles, they are cycles that traverse all the *edges* exactly once (traversing the vertices an arbitrary number of times).

Here is Christofides' algorithm:

1. Compute a minimum spanning tree T
2. Find the vertices of *odd degree* in T (call them V')
3. Compute a minimum-weight perfect matching M in $G[V']$ (i.e., G restricted to V').
4. Compute the graph $G' = M \cup T$ (possibly as a multigraph, if there are duplicate edges)

5. Find a Eulerian cycle C in G'
6. Perform a depth-first traversal of C, skipping cities that are already visited.

Observe that steps 1 and 6 are essentially the same as in the 2-approximation algorithm. The main difference is that the traversal is made in the graph C that results from intermediate computation.

Feasibility

At first, it is not obvious that all these steps can be performed, but they can! In any graph (and a fortiori, in T), the number of vertices of *odd degree* is necessarily *even* (Handshaking lemma). The original graph G, when restricted to these vertices, is thus a complete graph of even size. Perfect matchings always exist in such graphs (why?), and we can compute one of minimum-weight in polynomial time.

Next, a Eulerian cycle exists in a graph if and only if the graph is connected and all vertices have an *even degree* (to be done in exercises), and if so, it can also be found in polynomial time. If we take the union $M \cup T$, all vertices will have even degree in this multi-graph (either they already had even degree in T and then do not participate in M, or their degree is odd in T and increased by one through M). Since T is connected, $M \cup T$ is connected and thus an Eulerian cycle is guaranteed in $M \cup T$.

Approximation factor of 1.5

Let OPT be the cost of the optimal TSP solution.

The final tour consists of traversing the Eulerian cycle $C \subseteq M \cup T$, skipping already visited vertices. Thanks to these shortcuts, each edge of $M \cup T$ is traversed at most once. The total cost is therefore less than or equal to the total cost of the edges of $M \cup T$. This cost is itself less than the sum of the costs of T and the costs of M. We have already seen that the cost of T is less than OPT. It therefore suffices to show that the cost of M is less than $0.5 \cdot OPT$ to conclude.

In fact, we can show something even stronger, namely that the cost of a minimum perfect matching in the entire original graph is less than half the cost of an optimal tour (this is stronger, once again, thanks to the triangle inequality).

Lemma 7.4. $cost(M) \leq 0.5 \cdot OPT$

Proof. The algorithm computes a minimum-weight perfect matching in the graph $G[V']$, which is a subgraph of G. Let OPT' be the optimal cost of a TSP tour in $G[V']$. We will show two things: (1) $cost(M) \leq 0.5 \cdot OPT'$ and (2) $OPT' \leq OPT$.

For any graph with an even number of nodes, an optimal tour defines two perfect matchings: one taking every "odd" edge, and one taking every "even" edge. Thus, at least one of

these matchings has cost at most half of the total cost of the tour. Thus, a minimum-weight perfect matching (which has at most the same value) cannot exceed half of the total cost of the tour. This proves (1). For (2), we can obtain a tour for $G[V']$ based on a tour for G by skipping the vertices not in V'. Due to the triangle inequality, this tour costs at most that of the optimal tour for G. This implies that an optimal tour OPT' for $G[V']$ also costs less than an optimal tour for G, thus $OPT' \leq OPT$.

As a result, $cost(M) \leq 0.5 \cdot OPT' \leq 0.5 \cdot OPT$. □