Chapter 6

Approximation algorithms

By Sariel Har-Peled, December 10, 2013[1] Version: 0.31

6.1 Greedy algorithms and approximation algorithms

A natural tendency in solving algorithmic problems is to locally do whats seems to be the right thing. This is usually referred to as ***greedy algorithms***. The problem is that usually these kind of algorithms do not really work. For example, consider the following optimization version of Vertex Cover:

VertexCoverMin

Instance: A graph G, and integer k.
Question: Return the smallest subset $S \subseteq V(\mathsf{G})$, s.t. S touches all the edges of G.

For this problem, the greedy algorithm will always take the vertex with the highest degree (i.e., the one covering the largest number of edges), add it to the cover set, remove it from the graph, and repeat. We will refer to this algorithm as GreedyVertexCover.

It is not too hard to see that this algorithm does not output the optimal vertex-cover. Indeed, consider the graph depicted on the right. Clearly, the optimal solution is the black vertices, but the greedy algorithm would pick the four white vertices.

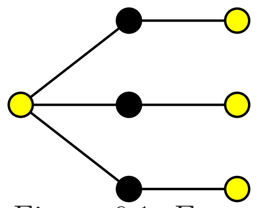


Figure 6.1: Example.

This of course still leaves open the possibility that, while we do not get the optimal vertex cover, what we get is a vertex cover which is "relatively good" (or "good enough").

Definition 6.1.1. A ***minimization problem*** is an optimization problem, where we look for a valid solution that minimizes a certain target function.

Example 6.1.2. In the VertexCoverMin problem the (minimization) target function is the size of the cover. Formally $\mathrm{Opt}(\mathsf{G}) = \min_{S \subseteq V(\mathsf{G}), S \text{ cover of } G} |S|$.

The VertexCover(G) is just the set S realizing this minimum.

[1]This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/3.0/` or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

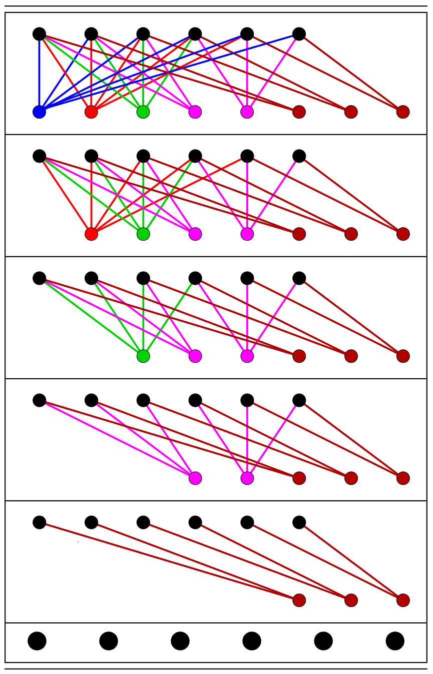
Definition 6.1.3. Let Opt(G) denote the value of the target function for the optimal solution.

Intuitively, a vertex-cover of size "close" to the optimal solution would be considered to be good.

Definition 6.1.4. Algorithm **Alg** for a minimization problem Min achieves an approximation factor $\alpha \geq 1$ if for all inputs G, we have:

$$\frac{\mathbf{Alg}(\mathsf{G})}{\mathrm{Opt}(\mathsf{G})} \leq \alpha.$$

We will refer to **Alg** as an ***α-approximation algorithm*** for Min.



As a concrete example, an algorithm is a 2-approximation for VertexCoverMin, if it outputs a vertex-cover which is at most twice the size of the optimal solution for vertex cover.

So, how good (or bad) is the **GreedyVertexCover** algorithm described above? Well, the graph in Figure 6.1 shows that the approximation factor of **GreedyVertexCover** is *at least* 4/3.

It turns out that **GreedyVertexCover** performance is considerably worse. To this end, consider the following bipartite graph: $G_n = (L \cup R, E)$, where L is a set of n vertices. Next, for $i = 2, \ldots, n$, we add a set R_i of $\lfloor n/i \rfloor$ vertices, to R, each one of them of degree i, such that all of them (i.e., all vertices of degree i at L) are connected to distinct vertices in R. The execution of **GreedyVertexCover** on such a graph is shown on the right.

Clearly, in G_n all the vertices in L have degree at most $n-1$, since they are connected to (at most) one vertex of R_i, for $i = 2, \ldots, n$. On the other hand, there is a vertex of degree n at R (i.e., the single vertex of R_n). Thus, **GreedyVertexCover** will first remove this vertex. We claim, that **GreedyVertexCover** will remove all the vertices of $R_2, \ldots, R_n$ and put them into the vertex-cover. To see that, observe that if $R_2, \ldots, R_i$ are still active, then all the nodes of R_i have degree i, all the vertices of L have degree at most $i-1$, and all the vertices of $R_2, \ldots, R_{i-1}$ have degree strictly smaller than i. As such, the greedy algorithms will use the vertices of R_i. Easy induction now implies that all the vertices of R are going to be picked by **GreedyVertexCover**. This implies the following lemma.

Lemma 6.1.5. *The algorithm* **GreedyVertexCover** *is* $\Omega(\log n)$ *approximation to the optimal solution to VertexCoverMin.*

Proof: Consider the graph G_n above. The optimal solution is to pick all the vertices of L to the vertex cover, which results in a cover of size n. On the other hand, the greedy algorithm picks the set R. We have that

$$|R| = \sum_{i=2}^{n} |R_i| = \sum_{i=2}^{n} \left\lfloor \frac{n}{i} \right\rfloor \geq \sum_{i=2}^{n} \left(\frac{n}{i} - 1 \right) \geq n \sum_{i=1}^{n} \frac{1}{i} - 2n = n(H_n - 2)\,.$$

Here, $H_n = \sum_{i=1}^{n} 1/i = \lg n + \Theta(1)$ is the nth harmonic number. As such, the approximation ratio for **GreedyVertexCover** is $\geq \dfrac{|R|}{|L|} = \dfrac{n(H_n-2)}{n} = \Omega(\log n)$. ∎

Theorem 6.1.6. *The greedy algorithm for VertexCover achieves $\Theta(\log n)$ approximation, where n is the number of vertices in the graph. Its running time is $O(mn^2)$.*

Proof: The lower bound follows from Lemma 6.1.5. The upper bound follows from the analysis of the greedy of Set Cover, which will be done shortly.

As for the running time, each iteration of the algorithm takes $O(mn)$ time, and there are at most n iterations. ■

6.1.1 Alternative algorithm – two for the price of one

One can still do much better than the greedy algorithm in this case. In particular, let **ApproxVertexCover** be the algorithm that chooses an edge from G, add both endpoints to the vertex cover, and removes the two vertices (and all the edges adjacent to these two vertices) from G. This process is repeated till G has no edges. Clearly, the resulting set of vertices is a vertex-cover, since the algorithm removes an edge only if it is being covered by the generated cover.

Theorem 6.1.7. **ApproxVertexCover** *is a 2-approximation algorithm for VertexCoverMin that runs in $O(n^2)$ time.*

Proof: Every edge picked by the algorithm contains at least one vertex of the optimal solution. As such, the cover generated is at most twice larger than the optimal. ■

6.2 Fixed parameter tractability, approximation, and fast exponential time algorithms (to say nothing of the dog)

6.2.1 A silly brute force algorithm for vertex cover

So given a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with n vertices, we can approximate VertexCoverMin up to a factor of two in polynomial time. Let K be this approximation – we know that any vertex cover in G must be of size at least $K/2$, and we have a cover of size K. Imagine the case that K is truly small – can we compute the optimal vertex-cover in this case quickly? Well, of course, we could just try all possible subsets of vertices size at most K, and check for each one whether it is a cover or not. Checking if a specific set of vertices is a cover takes $O(m) = O(n^2)$ time, where $m = |\mathsf{E}|$. So, the running time of this algorithm is

$$\sum_{i=1}^{K} \binom{n}{i} O\left(n^2\right) \leq \sum_{i=1}^{K} O\left(n^i \cdot n^2\right) = O\left(n^{K+2}\right),$$

where $\binom{n}{i}$ is the number of subsets of the vertices of G of size exactly i. Observe that we do not require to know K – the algorithm can just try all sizes of subsets, till it finds a solution. We thus get the following (not very interesting result).

Lemma 6.2.1. *Given a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with n vertices, one can solve VertexCoverMin in $O(n^{\alpha+2})$ time, where α is the size the minimum vertex cover.*

```
fpVertexCoverInner $(X, \beta)$
        // Computes minimum vertex cover for the induced graph $\mathsf{G}_X$
        // $\beta$: size of VC computed so far.
    if $X = \emptyset$ or $\mathsf{G}_X$ has no edges then return $\beta$
    $\mathsf{e} \leftarrow$ any edge $uv$ of $\mathsf{G}_X$.
    $\beta_1 =$ fpVertexCoverInner $\left(X \setminus \{u, v\}, \beta + 2\right)$
    // Only take $u$ to the cover, but then we must also take
    //  all the vertices that are neighbors of $v$,
    //  to cover their edges with $v$
    $\beta_2 =$ fpVertexCoverInner $\left(X \setminus \left(\{u\} \cup N_{\mathsf{G}_X}(v)\right), \beta + |N_{\mathsf{G}_X}(v)|\right)$
    // Only take $v$ to the cover...
    $\beta_3 =$ fpVertexCoverInner $\left(X \setminus \left(\{v\} \cup N_{\mathsf{G}_X}(u)\right), \beta + |N_{\mathsf{G}_X}(u)|\right)$
    return $\min(\beta_1, \beta_2, \beta_3)$.

algFPVertexCover $(\mathsf{G} = (\mathsf{V}, \mathsf{E}))$
    return fpVertexCoverInner $(\mathsf{V}, 0)$
```

Figure 6.2: Fixed parameter tractable algorithm for VertexCoverMin.

6.2.2 A fixed parameter tractable algorithm

As before, our input is a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, for which we want to compute a vertex-cover of minimum size. We need the following definition:

Definition 6.2.2. Let $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ be a graph. For a subset $S \subseteq \mathsf{V}$, let G_S be the ***induced subgraph*** over S. Namely, it is a graph with the set of vertices being S. For any pair of vertices $x, y \in \mathsf{V}$, we have that the edge $xy \in \mathsf{E}(\mathsf{G}_S)$ if and only if $xy \in \mathsf{E}(\mathsf{G})$, and $x, y \in S$.

Also, in the following, for a vertex v, let $N_{\mathsf{G}}(v)$ denote the set of vertices of G that are adjacent to v.

Consider an edge $\mathsf{e} = uv$ in G. We know that either u or v (or both) must be in any vertex cover of G, so consider the brute force algorithm for VertexCoverMin that tries all these possibilities. The resulting algorithm **algFPVertexCover** is depicted in Figure 6.2.

Lemma 6.2.3. *The algorithm* **algFPVertexCover** *(depicted in Figure 6.2) returns the optimal solution to the given instance of VertexCoverMin.*

Proof: It is easy to verify, that if the algorithm returns β then it found a vertex cover of size β. Since the depth of the recursion is at most n, it follows that this algorithm always terminates.

Consider the optimal solution $Y \subseteq \mathsf{V}$, and run the algorithm, where every stage of the recursion always pick the option that complies with the optimal solution. Clearly, since in every level of the recursion at least one vertex of Y is being found, then after $O(|Y|)$ recursive calls, the remaining graph would have no edges, and it would return $|Y|$ as one of the candidate solution. Furthermore, since the algorithm always returns the minimum solution encountered, it follows that it would return the optimal solution. ■

Lemma 6.2.4. *The depth of the recursion of* **algFPVertexCover**(G) *is at most α, where α is the minimum size vertex cover in* G.

Proof: The idea is to consider all the vertices that can be added to the vertex cover being computed without covering any new edge. In particular, in the case the algorithm takes both u and v to the cover, then one of these vertices must be in the optimal solution, and this can happen at most α times.

The more interesting case, is when the algorithm picks $N_{\mathsf{G}_X}(v)$ (i.e., β_2) to the vertex cover. We can add v to the vertex cover in this case without getting any new edges being covered (again, we are doing this only conceptually – the vertex cover computed by the algorithm would not contain v [only its neighbors]). We do the same thing for the case of β_3.

Now, observe that in any of these cases, the hypothetical set cover being constructed (which has more vertices than what the algorithm computes, but covers exactly the same set of edges in the original graph) contains one vertex of the optimal solution picked into itself in each level of the recursion. Clearly, the algorithm is done once we pick all the vertices of the optimal solution into the hypothetical vertex cover. It follows that the depth the recursion is $\leq \alpha$. ■

Theorem 6.2.5. *Let* G *be a graph with* n *vertices, and with the minimal vertex cover being of size* α*. Then, the algorithm* **algFPVertexCover** *(depicted in Figure 6.2) returns the optimal vertex cover for* G *and the running time of this algorithm is* $O(3^\alpha n^2)$*.*

Proof: By Lemma 6.2.4, the recursion tree has depth α. As such, it contains at most $2 \cdot 3^\alpha$ nodes. Each node in the recursion requires $O(n^2)$ work (ignoring the recursive calls), if implemented naively. Thus, the bound on the running time follows. ■

Algorithms where the running time is of the form $O(n^c f(\alpha))$, where α is some parameter that depends on the problem are ***fixed parameter tractable*** algorithms for the given problem.

6.2.2.1 Remarks

Currently, the fastest algorithm known for this problem has running time $O(1.2738^\alpha + \alpha n)$ [CKX10]. This algorithm uses similar ideas, but is considerably more complicated.

It is known that no better approximation than 1.3606 is possible for VertexCoverMin, unless P = NP. The currently best approximation known is $2 - \Theta\left(1/\sqrt{\log n}\right)$. If the ***Unique Games Conjecture*** is true, then no better constant approximation is possible in polynomial time.

6.3 Traveling Salesman Person

We remind the reader that the optimization variant of the TSP problem is the following.

TSP-Min

Instance: $\mathsf{G} = (V, E)$ a complete graph, and $\omega(e)$ a cost function on edges of G.
Question: The cheapest tour that visits all the vertices of G exactly once.

Theorem 6.3.1. *TSP-Min can not be approximated within* ***any*** *factor unless* NP = P*.*

Proof: Consider the reduction from Hamiltonian Cycle into TSP. Given a graph G, which is the input for the Hamiltonian cycle, we transform it into an instance of TSP-Min. Specifically, we set the weight of every edge to 1 if it was present in the instance of the Hamiltonian cycle, and 2 otherwise. In the resulting complete graph, if there is a tour price n then there is a Hamiltonian cycle in the original graph. If on the other hand, there was no cycle in G then the cheapest TSP is of price $n+1$.

Instead of 2, let us assign the missing edges in H a weight of cn, for c an arbitrary number. Let H denote the resulting graph. Clearly, if G does not contain any Hamiltonian cycle in the original graph, then the price of the TSP-Min in H is at least $cn+1$.

Note, that the prices of tours of H are either (i) equal to n if there is a Hamiltonian cycle in G, or (ii) larger than $cn+1$ if there is no Hamiltonian cycle in G. As such, if one can do a c-approximation, in polynomial time, to TSP-Min, then using it on H would yield a tour of price $\leq cn$ if a tour of price n exists. But a tour of price $\leq cn$ exists if and only if G has a Hamiltonian cycle.

Namely, such an approximation algorithm would solve a NP-Complete problem (i.e., Hamiltonian Cycle) in polynomial time. ■

Note, that Theorem 6.3.1 implies that TSP-Min can not be approximated to within any factor. However, once we add some assumptions to the problem, it becomes much more manageable (at least as far as approximation).

What the above reduction did, was to take a problem and reduce it into an instance where this is a huge gap, between the optimal solution, and the second cheapest solution. Next, we argued that if had an approximation algorithm that has ratio better than the ratio between the two endpoints of this empty interval, then the approximation algorithm, would in polynomial time would be able to decide if there is an optimal solution.

6.3.1 TSP with the triangle inequality

6.3.1.1 A 2-approximation

Consider the following special case of TSP:

TSP$_{\triangle\neq}$-Min

Instance: $\mathsf{G} = (V, E)$ is a complete graph. There is also a cost function $\omega(\cdot)$ defined over the edges of G, that complies with the triangle inequality.
Question: The cheapest tour that visits all the vertices of G exactly once.

We remind the reader that the ***triangle inequality*** holds for $\omega(\cdot)$ if

$$\forall u, v, w \in \mathsf{V}(\mathsf{G}), \qquad \omega(u, v) \leq \omega(u, w) + \omega(w, v).$$

The triangle inequality implies that if we have a path σ in G, that starts at s and ends at t, then $\omega(st) \leq \omega(\sigma)$. Namely, ***shortcutting***, that is going directly from s to t, is always beneficial if the triangle inequality holds (assuming that we do not have any reason to visit the other vertices of σ).

Definition 6.3.2. A cycle in a graph G is ***Eulerian*** if it visits every edge of G exactly once.

Unlike Hamiltonian cycle, which has to visit every vertex exactly once, an Eulerian cycle might visit a vertex an arbitrary number of times. We need the following classical result:

Lemma 6.3.3. *A graph G has a cycle that visits every edge of G exactly once (i.e., an Eulerian cycle) if and only if G is connected, and all the vertices have even degree. Such a cycle can be computed in $O(n+m)$ time, where n and m are the number of vertices and edges of G, respectively.*

Our purpose is to come up with a 2-approximation algorithm for $\mathsf{TSP}_{\triangle\neq}$-Min. To this end, let C_{opt} denote the optimal TSP tour in G. Observe that C_{opt} is a spanning graph of G, and as such we have that

$$\omega(C_{\mathrm{opt}}) \geq \mathrm{weight}\Big(\text{cheapest spanning graph of } \mathsf{G}\Big)\,.$$

But the cheapest spanning graph of G, is the minimum spanning tree (MST) of G, and as such $\omega(C_{\mathrm{opt}}) \geq \omega(\mathrm{MST}(\mathsf{G}))$. The MST can be computed in $O(n \log n + m) = O(n^2)$ time, where n is the number of vertices of G, and $m = \binom{n}{2}$ is the number of edges (since G is the complete graph). Let T denote the MST of G, and covert T into a tour by duplicating every edge twice. Let H denote the new graph. We have that H is a connected graph, every vertex of H has even degree, and as such H has an Eulerian tour (i.e., a tour that visits every edge of H exactly once).

As such, let C denote the Eulerian cycle in H. Observe that

$$\omega(\mathsf{C}) = \omega(H) = 2\omega(T) = 2\omega(MST(\mathsf{G})) \leq 2\omega(C_{\mathrm{opt}})\,.$$

Next, we traverse C starting from any vertex $v \in V(\mathsf{C})$. As we traverse C, we skip vertices that we already visited, and in particular, the new tour we extract from C will visit the vertices of $V(\mathsf{G})$ in the order they first appear in C. Let π denote the new tour of G. Clearly, since we are performing shortcutting, and the triangle inequality holds, we have that $\omega(\pi) \leq \omega(\mathsf{C})$. The resulting algorithm is depicted in Figure 6.3.

It is easy to verify, that all the steps of our algorithm can be done in polynomial time. As such, we have the following result.

Theorem 6.3.4. *Given an instance of TSP with the triangle inequality ($\mathsf{TSP}_{\triangle\neq}$-Min) (namely, a graph G with n vertices and $\binom{n}{2}$ edges, and a cost function $\omega(\cdot)$ on the edges that comply with the triangle inequality), one can compute a tour of G of length $\leq 2\omega(C_{\mathrm{opt}})$, where C_{opt} is the minimum cost TSP tour of G. The running time of the algorithm is $O(n^2)$.*

6.3.1.2 A 3/2-approximation to TSP$_{\triangle\neq}$-Min

Let us revisit the concept of matchings.

Definition 6.3.5. Given a graph $\mathsf{G} = (V, E)$, a subset $M \subseteq E$ is a ***matching*** if no pair of edges of M share endpoints. A ***perfect matching*** is a matching that covers all the vertices of G. Given a weight function w on the edges, a ***min-weight perfect matching***, is the minimum weight matching among all perfect matching, where

$$\omega(M) = \sum_{e \in M} \omega(e)\,.$$

The following is a known result, and we will see a somewhat weaker version of it in class.

Theorem 6.3.6. *Given a graph G and weights on the edges, one can compute the min-weight perfect matching of G in polynomial time.*

Lemma 6.3.7. *Let $\mathsf{G} = (V, E)$ be a complete graph, S a subset of the vertices of V of even size, and $\omega(\cdot)$ a weight function over the edges. Then, the weight of the min-weight perfect matching in G_S is $\leq \omega(\mathrm{TSP}(\mathsf{G}))/2$.*

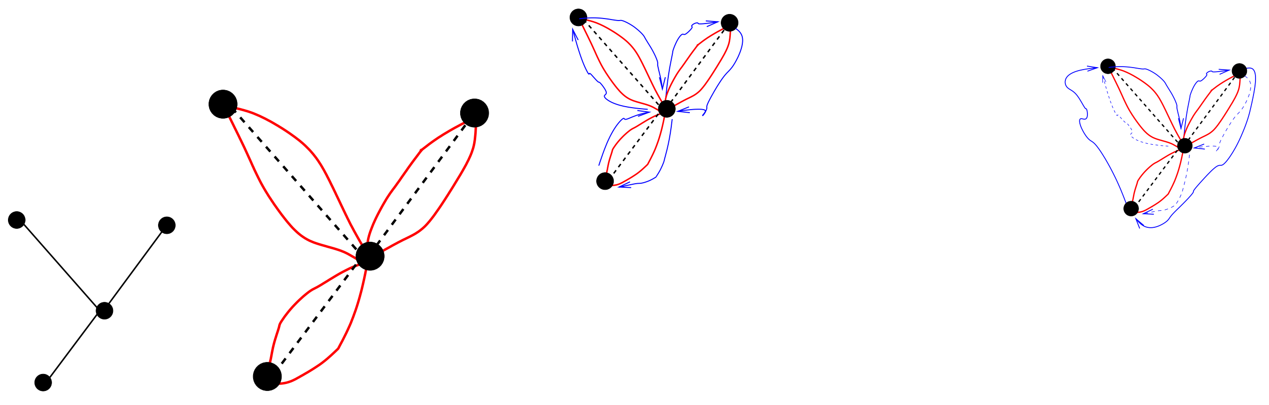
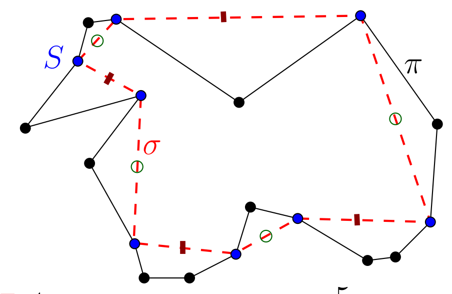


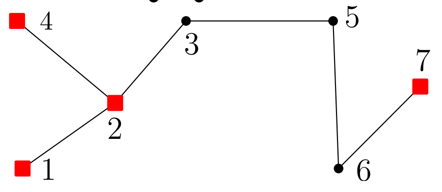
Figure 6.3: The TSP approximation algorithm: (a) the input, (b) the duplicated graph, (c) the extracted Eulerian tour, and (d) the resulting shortcut path.



Proof: Let π be the cycle realizing the TSP in G. Let σ be the cycle resulting from shortcutting π so that it uses only the vertices of S. Clearly, $\omega(\sigma) \leq \omega(\pi)$. Now, let M_e and M_o be the sets of even and odd edges of σ respectively. Clearly, both M_o and M_e are perfect matching in G_S, and

$$\omega(M_o) + \omega(M_e) = \omega(\sigma)\,.$$

We conclude, that $min(w(M_o)\,, w(M_e)) \leq \omega(\mathrm{TSP}(\mathsf{G}))/2$. ■



We now have a creature that has the weight of half of the TSP, and we can compute it in polynomial time. How to use it to approximate the TSP? The idea is that we can make the MST of G into an Eulerian graph by being more careful. To this end, consider the tree on the right. Clearly, it is almost Eulerian, except for these pesky odd degree vertices. Indeed, if all the vertices of the spanning tree had even degree, then the graph would be Eulerian (see Lemma 6.3.3).

In particular, in the depicted tree, the "problematic" vertices are $1, 4, 2, 7$, since they are all the odd degree vertices in the MST T.

Lemma 6.3.8. *The number of odd degree vertices in any graph G' is even.*

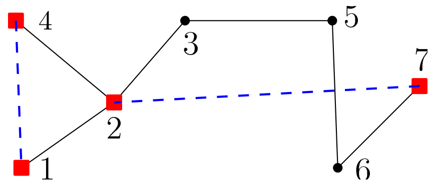
Proof: Observe that $\mu = \sum_{v \in V(G')} d(v) = 2|E(G')|$, where $d(v)$ denotes the degree of v. Let $U = \sum_{v \in V(G'), d(v) \text{ is even}} d(v)$, and observe that U is even as it is the sum of even numbers.

Thus, ignoring vertices of even degree, we have

$$\alpha = \sum_{v \in V, d(v) \text{ is odd}} d(v) = \mu - U = \text{even number},$$

since μ and U are both even. Thus, the number of elements in the above sum of all odd numbers must be even, since the total sum is even. ■

So, we have an even number of problematic vertices in T. The idea now is to compute a minimum-weight perfect matching M on the problematic vertices, and add the edges of the matching to the tree. The resulting graph, for our running example, is depicted on the right. Let $H = (V, E(M) \cup E(T))$ denote this graph, which is the result of adding M to T.



We observe that H is Eulerian, as all the vertices now have even degree, and the graph is connected. We also have

$$\omega(H) \;\; = \;\; \omega(\mathrm{MST}(\mathsf{G})) + \omega(M) \leq \omega(\mathrm{TSP}(\mathsf{G})) + \omega(\mathrm{TSP}(\mathsf{G}))/2 = (3/2)\omega(TSP(\mathsf{G})),$$

by Lemma 6.3.7. Now, H is Eulerian, and one can compute the Euler cycle for H, shortcut it, and get a tour of the vertices of G of weight $\leq (3/2)\omega(\mathrm{TSP}(\mathsf{G}))$.

Theorem 6.3.9. *Given an instance of TSP with the triangle inequality, one can compute in polynomial time, a* $(3/2)$*-approximation to the optimal TSP.*

6.4 Biographical Notes

The 3/2-approximation for TSP with the triangle inequality is due to Christofides [Chr76].

Bibliography

[Chr76] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[CKX10] J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theor. Comput. Sci.*, 411(40-42):3736–3756, 2010.