CS4234: Optimisation Algorithms Lecture 1

The Limits of Tractability: MIN-VERTEX-COVER

V1.0: Seth Gilbert, V1.1: Steven Halim *August 15, 2017*

Abstract

Today, we are talking about the MIN-VERTEX-COVER problem. MIN-VERTEX-COVER is a classic NP-hard optimization problem, and to solve it, we need to compromise. We look at three approaches. First, we consider restricting our attention to a special case: a (binary) tree. Then, we look at an exponential algorithm parameterized by the size of the vertex cover. Finally, we look at approximation algorithms (both a deterministic approach and a simple randomized approach) and analyze their approximation ratios (both are 2-approximation).

1 Overview

Today, we will study the problem of MIN-VERTEX-COVER, a classical NP-hard optimization problem. MIN-VERTEX-COVER is a great model problem to think about at the beginning of the semester because, on the one way, it has a relatively simple structure and the algorithms we will see today are quite simple; on the other hand, it illustrates several of the basic techniques we will use this semester, and many problems that show up in the real world are quite similar to MIN-VERTEX-COVER.

We begin by defining the problem.

Definition 1 *A **vertex cover** for a graph $G = (V, E)$ is a set $S \subseteq V$ such that for every edge $e = (u, v) \in E$, either $u \in S$ or $v \in S$.*

That is, every edge is covered by one of the vertices in the set S.

Definition 2 *The* **MIN-VERTEX-COVER** *problem is defined as follows: Given a graph $G = (V, E)$, find a minimum-sized set S that is a vertex cover for G.*

Imagine, for example, that you are given a map of Singapore and you want to choose where to open Starbucks coffee shops to ensure that, no matter where you are in Singapore, there is a Starbucks on a nearby corner. (Assume that you have decided to open your coffee shops only at an intersection.) If you find a vertex cover, you can be sure that nobody in Singapore is too far away from a Starbucks! But is your vertex cover of minimum possible size? See Figure 1 for an example.

As we will see shortly in Section 2, MIN-VERTEX-COVER is a hard problem—specifically it is NP-hard. We are not likely to come up with an efficient algorithm (unless $P = NP$).

When you have a hard computational problem, there are three things that you want a solution to be:

1. Fast (i.e., polynomial time)
2. Optimal (i.e., yielding the best solution possible)
3. Universal (i.e., good for all instances/inputs)

But when you have an NP-hard problem like MIN-VERTEX-COVER, you can only have two of these three[1]! You can sacrifice speed, and have an algorithm that runs in exponential time. You can sacrifice optimality, and explore heuristics

[1] These options appear in challenging programming contest problems, see [3].

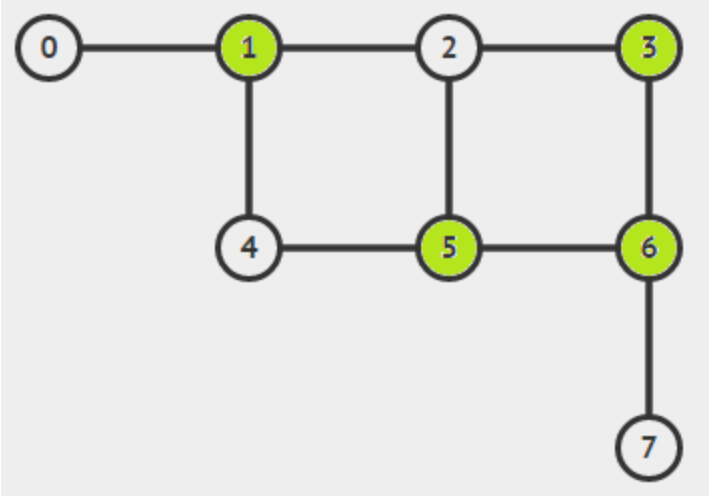


Figure 1: Example of a graph with a vertex cover of size 4. The green highlighted vertices are in the vertex cover. Is 4 the size of the MIN-VERTEX-COVER? If yes, is the solution unique? If no, can you find MIN-VERTEX-COVER of size 3 or less?

and approximation algorithms—try to find a solution that is *good enough*[2]. Or you can sacrifice universality, and focus on solving special cases. Today, we will talk about these three different approaches for coping with an otherwise intractable problem.

1. First, we consider the case where the input to your problem is somehow special (and easier than the general case); specifically, we will look at finding a MIN-VERTEX-COVER on a (binary) tree. (Here we sacrifice universality.)
2. Second, we will see a *parameterized* solution that can guarantee good performance when the MIN-VERTEX-COVER being sought is small (e.g., we cannot afford to open more than 10 Starbucks, regardless). In this case, we given an algorithm that achieves an exponential-time solution that is exponential only in the parameter k, not the size of the problem n. (Here we sacrifice speed: It is exponential-time, but hopefully reasonable.)
3. Third, we will consider approximation algorithms that do not find an *optimal* solution, but find a solution that is not too far from optimal. We will discuss a deterministic 2-approximation algorithm and also a randomized 2-approximation algorithm. (Here we sacrifice optimality, finding a solution that is "good enough.")

2 NP-completeness

Before we get to algorithms, let us quickly review[3] the proof that VERTEX-COVER is NP-complete (for details, see Chapter 34 of [1])[4].

It is relatively easy to show that VERTEX-COVER (as a decision problem) is NP-complete. First, it is easy to observe that it is in NP: Given a set S of size k, we can easily verify whether or not it is a vertex cover simply by checking whether, for every edge, one of the two endpoints can be found in S.

[2] "Le mieux est l'ennemi du bien," according to Voltaire. ("The best is the enemy of the good.")

[3] Recall CS3230!

[4] To be a bit pedantic, it is actually a mistake to refer to MIN-VERTEX-COVER as NP-complete. The complexity class NP refers only to *decision* problems, not to *optimization* problems. The version of MIN-VERTEX-COVER that *is* NP-complete is the VERTEX-COVER problem described as follows: Given a graph $G = (V, E)$ and an integer k, does graph G has a vertex cover of size k (vertices)? Clearly, if we could efficiently find a MIN-VERTEX-COVER, then we could also efficiently answer the decision version of the question. Hence solving the optimization version of MIN-VERTEX-COVER is at least as hard as the decision version, and hence we term it NP-hard.

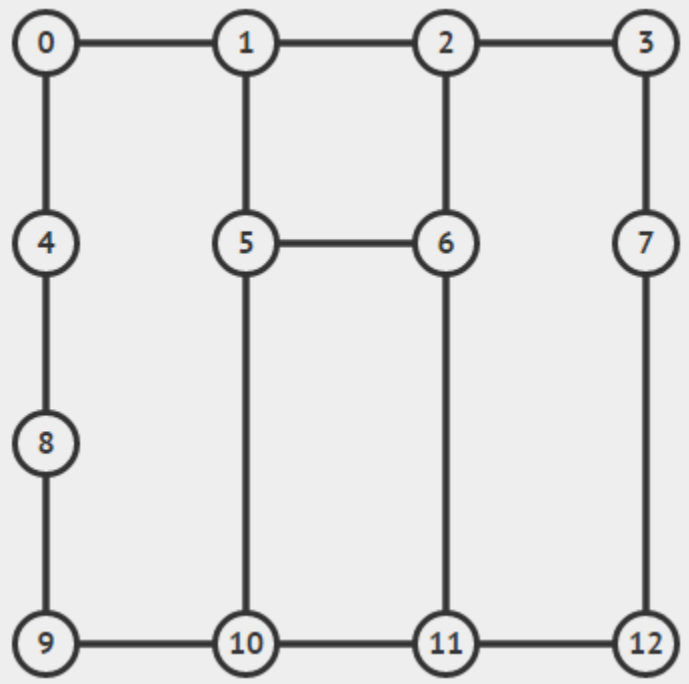


Figure 2: Mini challenge: What is the minimum-sized vertex cover of this graph?

Next, we have to show that it is NP-hard. We can show this by reducing some other NP-hard problem to VERTEX-COVER, thus showing that solving VERTEX-COVER is at least as hard as solving this other problem. The original NP-complete problem is 3-SAT (see Section 34.4 of [1] or Chapter 3 of [2]), and we could[5] reduce 3-SAT to VERTEX-COVER. Instead, we focus on a different problem: CLIQUE, that of finding the a clique[6] of size k in a graph.

Definition 3 *The (decision version of)* **CLIQUE** *problem is defined as follows: Given a graph $G = (V, E)$ and an integer k, is there a subset $C \subseteq V$ of size k (vertices) such that C is a clique in G?*

The goal of the CLIQUE problem is to determine whether a given graph G contains a clique of size k. CLIQUE is known to be NP-complete[7]. We can show that VERTEX-COVER is NP-hard by giving an algorithm for solving CLIQUE by using VERTEX-COVER, that is, we reduce CLIQUE $\leq_P$ VERTEX-COVER. First, we need to define the ***complement*** of a graph G, i.e., the graph that contains only the edges that are *not* in G:

Definition 4 *Given a graph $G = (V, E)$ we define the* ***complement***[8] *of G to be the graph $G' = (V, E')$ where edge $e \in E'$ if and only if $e \notin E$.*

We can now give an algorithm for solving the CLIQUE problem. Assume we have an algorithm VC-ALGO which solves the problem of VERTEX-COVER. On an input of graph $G = (V, E)$ with n vertices, and an integer k, execute the VERTEX-COVER algorithm on G', the complement of G; return **true** if and only if G' has a VERTEX-COVER of size $\leq n - k$.

[5] But we wouldn't, as it is too complicated.

[6] A clique is a set of vertices $K \subseteq V$ such that every pair of vertices in K is connected by an edge.

[7] We can also say upfront that VERTEX-COVER is known to be NP-complete (e.g. from CS3230, a.k.a. 'proof from previous module') without going through all these proofs in this Section 2. However, for the sake of CS3230 review, we assume that we 'don't know' that VERTEX-COVER is NP-complete but we somehow recall that CLIQUE is NP-complete.

[8] We can compute the complement of a graph G in $O(V^2)$ time.

Algorithm: Clique($G = (V, E)$, k)
Procedure:
`/* computable in polynomial time` $O(V^2)$ `*/`
Let G' be the complement of G.
Execute VC-ALGO on G' with parameter $n - k$.
if $\exists$ *vertex cover of* G' *of size* $\leq n - k$ **then**
    return true.
else return false.

The correctness of this algorithm for CLIQUE immediately shows that VERTEX-COVER is NP-complete and depends on the following claim:

Lemma 5 *Graph $G = (V, E)$ with n vertices has a clique of size at least k if and only if its complement G' has a vertex cover of size at most $n - k$.*

The complete proof can be reviewed in Section 34.5.2 of [1].

3 MIN-VERTEX-COVER on a (Binary) Tree

While finding a minimum sized vertex cover is NP-hard in general, there are many special cases that are tractable. Consider an example where the graph $G = (V, E)$ is a rooted binary tree. In this case, we can find a MIN-VERTEX-COVER in linear time using Dynamic Programming (DP). Here we outline the solution for determining the *size* of the MIN-VERTEX-COVER. It is relatively easy to turn this idea into an algorithm for finding the MIN-VERTEX-COVER.

For each vertex v in the tree, we define two variables:

- $in(v)$ is the size of the minimum vertex cover for the sub-tree of G rooted at v, assuming that v is contained in the vertex cover.
- $out(v)$ is the size of the minimum vertex cover for the sub-tree of G rooted at v, assuming that v is not contained in the vertex cover.

Notice that if r is the root of the tree, the size of the minimum vertex cover is simply $\min[in(r), out(r)]$.

We can now traverse the tree starting from the leaves up to the root calculating for every vertex v both $in(v)$ and $out(v)$. For a leaf, we can trivially define $in(v) = 1$ and $out(v) = 0$. Now consider a vertex v in the tree with children w and z.

- To calculate $out(v)$: In this case, both children w and z *must* be included in the vertex cover. Thus $out(v) = in(w) + in(z)$.
- To calculate $in(v)$: In this case, we can either include or not include w or z in the tree, depending on which solution is better. In either case, we add 1 to the count, since we are including v. Thus, $in(v) = 1 + \min[in(w), out(w)] + \min[in(z), out(z)]$.

It is easy to see that this calculation can be performed in linear time, and that it results in the MIN-VERTEX-COVER. (The proof would proceed by induction over the height of the tree.)

Note (in addition) that the same idea can be used for more general trees, and also for weighted variants of MIN-VERTEX-COVER (where each vertex has a cost for including it in the vertex cover) — discussed in the next Lecture

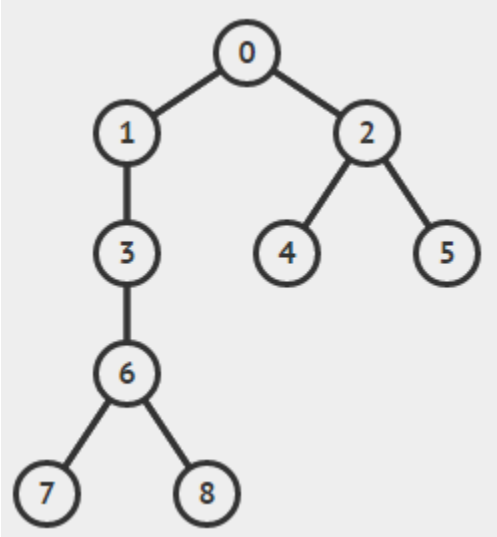


Figure 3: Example of a Binary Tree. Apply the Dynamic Programming algorithm discussed in this section to obtain the MIN-VERTEX-COVER of this Binary Tree.

02. A sample implementation of MIN-VERTEX-COVER (unweighted version) on not-necessarily-binary trees in C++ is given in Section 4.7.1 of [4].

4 Parameterized Complexity

Another special case is when you know, in advance, that the vertex cover is small. For example, if you are told that the vertex cover is up to $k = 2$ vertices, then there is an easy polynomial time algorithm: Simply test every pair of vertices (and every individual vertex[9]) and check whether that vertex is a valid vertex cover. Assuming that you can check the vertex cover of a graph $G = (V, E)$ with n vertices and m edges in time $O(m)$, then this algorithm runs in time $O(n^2 m)$.

More generally, if you know that the vertex cover is of size at most k, you can find it in time $O(n^k m)$. Unfortunately, as n gets large, the term n^k becomes prohibitively large and the algorithm rapidly becomes unusable, even for small values of k. For example, if $k = 10$, $n = 100$, $m = 1\,000$ and you use a CPU that can perform 10 billion instructions per second (e.g., an Intel Core i7), then it would take more than 300 thousand years to find the vertex cover.

Instead, we would like an algorithm that runs in time $O(f(k)g(n, m)$, for some functions f and g where g is a polynomial in n and m. That is, the algorithm may be exponential in k, but the exponential function should not have any dependence on n and m. If k is small, this might yield a usable algorithm.

Here's the basic idea. For every edge $e = (u, v)$, we know that either u or v is in the vertex cover. So let's consider both cases. For every vertex $z \in V$, define G_{-z} to be the graph G where z and all edges adjacent to z have been deleted. If we can find a vertex cover of size $k - 1$ in either G_{-u} or G_{-v}, when we can construct a vertex cover for G. The key lemma here is as follows:

Lemma 6 *Let $G = (V, E)$ be a graph and let $e = (u, v)$ be an edge in E. The graph G has a vertex cover of size k if and only if either G_{-u} or G_{-v} has a vertex cover of size $k - 1$.*

(We leave the proof as an exercise.) This yields the following algorithm:

[9] There is a possibility that the answer is just 1 vertex, so we also need to check for this possibility.

```
1  Algorithm: ParameterizedVertexCover(G = (V, E), k)
2  Procedure:
       /* Base case of recursion:  ⊥ = failure, ∅ = empty set          */
3      if k = 0 and E ≠ ∅ then return ⊥
4      if E = ∅ then return ∅
       /* Recurse on arbitrary edge e:                                 */
5      Let e = (u, v) be any edge in G.
6      S_u = ParameterizedVertexCover(G_{-u}, k − 1)
7      S_v = ParameterizedVertexCover(G_{-v}, k − 1)
8      if S_u ≠ ⊥ and |S_u| < |S_v| then
9          return S_u ∪ {u}
10     else if S_v ≠ ⊥ and |S_v| ≤ |S_u| then
11         return S_v ∪ {v}
12     else return ⊥
```

The algorithm picks an arbitrary edge $e = (u, v)$, and recursively searches for a vertex cover of size $k - 1$ in G_{-u} and G_{-v}. When it has explored both options, it then return the smaller of the two vertex covers (adding either u or v as is appropriate).

Notice that this recursive algorithm has the following structure: At every step we remove one vertex from the graph and add it to the vertex cover, and we recurse twice. As a base case, if we have removed k vertices from the graph, and yet we have not found a vertex cover (i.e., some edges still remain in the graph), then we abort and return failure. Another base case is when we have no more edge to be covered (and $k \geq 0$), in this case we return an empty set.

Assuming that the initial graph G has a vertex cover of size at most k, the running time of this algorithm is captured by the following recurrence:

$$T(k, m) \leq 2 \times T(k - 1, m) + O(m)$$

(here we assume that it takes approximately $O(m)$ time to delete a vertex and its adjacent edges from a graph, and we ignore the fact that the number of edges is reduced in the recursive calls).

Solving this recurrence, we find that $T(n) = O(2^k m)$. For the setting above ($k = 10$, $n = 1\,000$), this improved algorithm would run in just about 0.1ms :O...

5 Approximation Algorithms

Alas, sometimes a problem is still too hard to solve. The graph does not satisfy any of the (known) special cases. The parameterized complexity is *still* too large (i.e., the vertex cover is not sufficiently small). In this case, we cannot guarantee that you will find an optimal solution.

Instead, we relax our goals: We will not find an *optimal* solution, but we will find an *almost* optimal solution. This is the goal of an approximation algorithm. Consider, as an example, the problem of MIN-VERTEX-COVER. For a given graph G, define $OPT(G)$ to be a minimal-sized vertex cover[10].

Definition 7 *An algorithm A is an* ***α-approximation algorithm*** *for* MIN-VERTEX-COVER *if, for every graph G:*

$$|A(G)| \leq \alpha \times |OPT(G)| \,.$$

[10]Notice that we often seem to implicitly assume there is only *one* possible optimal solution. This is not true! There may be many possible optimal vertex covers. Here we just choose one.

That is, an algorithm A is a good approximation[11] algorithm for MIN-VERTEX-COVER if it guarantees that it will find a vertex cover that is not too much larger than the best possible vertex cover, e.g. see Figure 4.

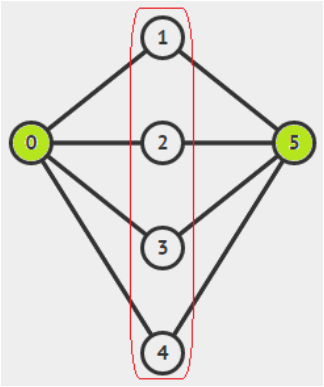


Figure 4: Example of a graph with an optimal vertex cover of size 2 (the two green-highlighted vertices: Vertex $\{0, 5\}$). The red-highlighted vertices (Vertex $\{1, 2, 3, 4\}$) in the middle also form a vertex cover of size 4. This vertex cover of size 4 is not optimal. To be precise, it is two times larger than the optimal vertex cover.

In the context of approximation algorithms, MIN-VERTEX-COVER is a particularly interesting problem to study. It is very easy[12] to design a 2-approximation algorithm (and we will see two such algorithms today).

5.1 Deterministic Approximation Algorithm

First, we consider a deterministic algorithm for approximating MIN-VERTEX-COVER. There are three natural candidate algorithms (see below). Each of these algorithms greedily adds vertices to the cover, with progressing attempts at cleverness. The first algorithm simply adds arbitrary vertices until every edge is covered. The second algorithm considers the edges in arbitrary order, but adds *both* endpoints. The third algorithm is the 'cleverest', greedily adding the vertex that covers the most edges that remain uncovered. Stop for a minute and think about which algorithm you think is best.

```
/* This algorithm adds vertices greedily, one at a time, until everything
   is covered.  The edges are considered in an arbitrary order, and for
   each edge, an arbitrary endpoint is added.                              */
Algorithm: ApproxVertexCover-1(G = (V, E))
Procedure:
    C ← ∅
    /* Repeat until every edge is covered:                                 */
    while E ≠ ∅ do
        Let e = (u, v) be any edge in G.
        C ← C ∪ {u}
        G ← G_{−u} // Remove u and all adjacent edges from G.
    return C
```

[11] $\alpha = 1$ means an optimal algorithm. α is > 1 for all approximation algorithms for a minimisation problem and Computer scientists aim to make α as close as possible to 1.

[12] Whether $\alpha = 2$ is a good bound or not depends on the requirements. There are other possible algorithms that we will learn later in this course that can reach better performance than the 2-approximation algorithms that we study today even though they have no such bound.

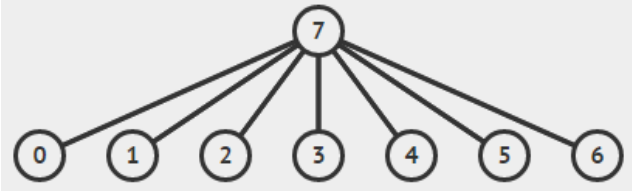


Figure 5: Counterexample for the simple vertex cover algorithm 1 where an *arbitrary* vertex is added in each step. When each edge is considered, we can be very unlucky whereby the leaf vertex is always chosen, yielding a vertex cover of size $n - 1$. The minimum vertex cover contains only vertex 7.

```
/* This algorithm adds vertices greedily, two at a time, until everything
   is covered.  The edges are considered in an arbitrary order, and for
   each edge, both endpoints are added.                                    */
Algorithm: ApproxVertexCover-2(G = (V, E))
Procedure:
    C ← ∅
    /* Repeat until every edge is covered:                                */
    while E ≠ ∅ do
        Let e = (u, v) be any edge in G.
        C ← C ∪ {u, v}
        G ← G_{−{u,v}} // Remove u and v and all adjacent edges from G.
    return C
```

```
/* This algorithm adds vertices greedily, one at a time, until everything
   is covered.  At each step, the algorithm chooses the next vertex that
   will cover the most uncovered edges.                                    */
Algorithm: ApproxVertexCover-3(G = (V, E))
Procedure:
    C ← ∅
    /* Repeat until every edge is covered:                                */
    while E ≠ ∅ do
        Let d(x) = number of uncovered edges adjacent to x.
        Let u = argmax_{x∈V} d(x)
        C ← C ∪ {u}
        G ← G_{−{u}} // Remove u and all adjacent edges from G.
    return C
```

It turns out that neither Algorithm 1 nor Algorithm 3 achieve a good approximation ratio. See Figure 5 for a graph where Algorithm 1 can perform poorly and Figure 6 for a graph where Algorithm 3 can perform poorly. We now proceed to analyze `ApproxVertexCover-2` and show that it is a 2-approximation algorithm. This may seem somewhat surprising: We are wastefully adding *both* endpoints of an edge e, when only one would suffice. On the other hand, thinking back to the analysis of the randomized algorithm, we know that each time at least *one* of the two vertices being added is a "correct" vertex, i.e., one that is added by $OPT(G)$. This guarantees that we get a vertex cover that is at most twice as big as optimal.

We now try to formalize this intuition. For pedagogic purposes, we are going to do this a little more carefully and less directly. In general, the technique for analyzing an approximation algorithm is to find some way to bound the size of OPT. If we can show that OPT cannot be too small, then it is easier to show that our algorithm is close to OPT. To this end, we are going to define another combinatorial concept:

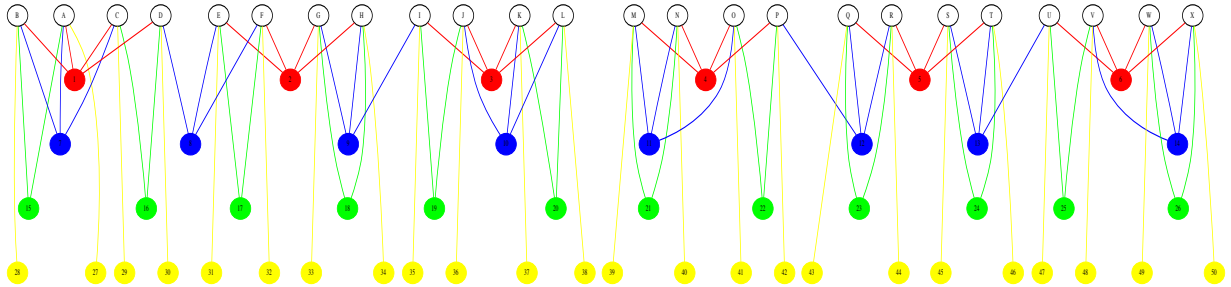


Figure 6: Counterexample for the greedy vertex cover algorithm 3. In this example, $n = 4$. The top row consists of $n! = 24$ white vertices, each of degree $n = 4$. The next row consists of $n!/n = 6$ red vertices, each of degree $n = 4$. The next row consists of $n!/(n-1) = 8$ blue vertices of degree 3. The next row consists of $n!/(n-2) = 12$ green vertices of degree 2. The last row consists of $n!/(n-3) = 24$ yellow vertices of degree 1. Notice that there is a vertex cover of size $n!$, i.e., all the white vertices. However, the greedy algorithm might first add the red vertices (reducing the degree of every white vertex by one), then add the blue vertices (reducing the degree of every white vertex by one), then add the green vertices, and then the yellow vertices. The total number of vertices added to the vertex cover is $n! \sum_{i=1}^{n}(1/i) = n! \log n$, which is a factor of $\log n$ from optimal.

Definition 8 *Given a graph $G = (V, E)$, we say that a set of edge $M \subseteq E$ is a **matching** if no two edges in M share an endpoint, i.e., $\forall e_1, e_2 \in M : \; e_1 \cap e_2 = \emptyset$.*

A matching is a set of edges that pairs up vertices: Each vertex in the matching gets exactly one partner. Note that a matching can be of any size, consisting of one edge or up to $\lfloor n/2 \rfloor$ edges. If every vertex in the graph is adjacent to an edge in the matching, then it is a ***perfect matching***. We will later revisit this problem in the middle of the semester.

The key observation here is that for every edge in a matching, one of the two endpoints *must* be in the vertex cover. This yields the following lemma:

Lemma 9 *Let M be any matching of graph $G = (V, E)$. Then $|OPT(G)| \geq |M|$.*

Proof Observe that for every edge $e = (u, v)$ in the matching M, either u or v has to be in the vertex cover $OPT(G)$. Moreover, since M is a matching, neither u nor v cover any other edges in M. Thus every edge in M must have a unique vertex in $OPT(G)$, and hence $|OPT(G)| \geq |M|$. □

This lemma gives us a nice way of showing that our vertex cover approximation algorithm is good: All we need to do is show that there exists some matching that is not too much smaller than the vertex cover it produces. Luckily, the algorithm constructs a matching as it executes:

Lemma 10 *Let E' be the set of edges considered by* `ApproxVertexCover-2` *during its execution. Then E' is a matching.*

Proof After each edge $e = (u, v)$ in E' is considered, both the vertices u and v are removed from the graph G. Thus no edge considered later can contain either u or v. (Formally, the proof proceeds by induction, maintaining the invariant that for every edge e in E', neither endpoint of e remains in G and hence the new edge being considered can be safely added to E'.) □

Combining these two lemmas yields our final claim:

Theorem 11 `ApproxVertexCover-2` *is a 2-approximation algorithm for vertex cover.*

Proof For a given graph $G = (V, E)$, let C be the set output by the algorithm and let E' be the edges considered. Notice that $|C| = 2 \times |E'|$, since for every edge in E' we added two vertices to C. Since E' is a matching, we know that $|OPT(G)| \geq |E'|$, and hence we conclude:

$$\begin{aligned} |C| &= 2 \times |E'| \\ &\leq 2 \times |OPT(G)| \end{aligned}$$

□

5.2 Randomized Approximation Algorithm

Next, we consider a very simple randomized approach: For every edge in the graph, simply flip a coin and randomly choose one of the two endpoints of the edge to add to the vertex cover. Repeat this process until every edge is covered.

Algorithm: RandomizedVertexCover($G = (V, E)$)
Procedure:
  $C \leftarrow \emptyset$
  `/* Repeat until every edge is covered: */`
  while $E \neq \emptyset$ **do**
    Let $e = (u, v)$ be any edge in G.
    $b \leftarrow$ **Random**$(0, 1)$ `// Returns` $\{0, 1\}$ `each with probability` $1/2$.
    if $b = 0$ **then** $z = u$
    else if $b = 1$ **then** $z = v$
    $C \leftarrow C \cup \{z\}$
    $G \leftarrow G_{-z}$ `// Remove` z `and all adjacent edges from` G.
  return C

It is easy to see that, when the algorithm completes, the resulting set C is a vertex cover: Each edge is removed from G only when it is covered by a vertex that has been added to C.

We now argue that the resulting vertex cover is, in expectation, at most twice the size of $OPT(G)$. The intuition here is that each time we add a vertex to C, we have (at least) $1/2$ probability of being right: For every edge $e = (u, v)$, either $u \in OPT(G)$ or $v \in OPT(G)$ (or both of them are), and so with probability (at least) $1/2$ we choose the correct one. Thus we would expect C to be only about twice as large as OPT.

To make this intuition a little more formal, we define two sets: $A = C \cap OPT(G)$ and $B = C \setminus OPT(G)$. The set A contains all the "correct" elements that we have added to C, i.e., all the elements that are in both C and $OPT(G)$. The set B contains all the "incorrect" elements that we have added to C, i.e,. all the elements that are in C but not in $OPT(G)$. For every edge e considered by the algorithm, define the indicator random variable:

$$X_e = \begin{cases} 0 & \textbf{if} \quad \text{the vertex added to } C \text{ after considering } e \text{ is } \textbf{not} \text{ in } OPT \\ 1 & \textbf{if} \quad \text{the vertex added to } C \text{ after considering } e \text{ is in } OPT \end{cases}$$

Notice that $\mathrm{E}[X_e] \geq 1/2$, since the probability of choosing the correct vertex is at least $1/2$. Let E' be the set of edges considered by the algorithm. Since $|A| = \sum_{E'} X_e$, we conclude that:

$$\begin{aligned} \mathrm{E}[|A|] &= \mathrm{E}\left[\sum_{E'} X_e\right] \\ &= \sum_{E'} \mathrm{E}[X_e] \qquad \textit{(linearity of expectation)} \\ &\geq |E'|/2 \,. \end{aligned}$$

Similarly, we see that $|B| = \sum_{E'}(1 - X_e)$, and hence $\mathrm{E}[|B|] \leq |E'|/2$. Putting these two together, we observe that:

$$\mathrm{E}[|B|] \leq \mathrm{E}[|A|] \ .$$

The final observation we need is that $|A| \leq |OPT(G)|$ (since A only contains vertices that are also in $OPT(G)$), and hence $\mathrm{E}[|A|] \leq |OPT(G)|$. Putting together the preceding equations, we see:

$$\mathrm{E}[|B|] \leq \mathrm{E}[|A|] \leq |OPT(G)| \,.$$

Finally, we conclude that calculation:

$$\begin{aligned} \mathrm{E}[|C|] &= \mathrm{E}[|A|] + \mathrm{E}[|B|] \qquad \textit{(linearity of expectation)} \\ &\leq |OPT(G)| + |OPT(G)| \\ &\leq 2 \times |OPT(G)| \end{aligned}$$

This yields the following theorem:

Theorem 12 `RandomizedVertexCover` *yields a 2-approximation for* MIN-VERTEX-COVER *(in expectation).*

Note: Since April 2017, Steven and his FYP student: Muhammad Rais has created `https://visualgo.net/en/mvc` visualization tool that covers many parts of this lecture [5].

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 3rd edition, 2009.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[3] Jonathan Irvin Gunawan. Understanding Unsolvable Problem. *Olympiads In Informatics*, 10:87–98, 2016.

[4] Steven Halim and Felix Halim. *Competitive Programming: The New Lower Bound of Programming Contests*. Lulu, 3rd edition, 2013.

[5] Steven Halim and Muhammad Rais. VisuAlgo - Min Vertex Cover Visualization, howpublished= https://visualgo.net/en/mvc.

CS4234: Optimisation Algorithms Lecture 2

Introduction to LP

V1.0: Seth Gilbert, V1.1: Steven Halim *August 10, 2016*

Abstract

Today we introduce Linear Program(ming) (LP), a powerful tool for solving optimization problems. We discuss some terminology and notation, and we give some examples of how to use LPs to solve basic optimization problems. Then we talk about Integer Linear Programs (ILPs), and discuss how to relax ILPs to solve Combinatorial Optimization Problems that require integer answers. We use MIN-WEIGHT-VERTEX-COVER as an example, giving a 2-approximation algorithm.

1 Linear Programming

Linear Program (or Linear Programming, both usually abbreviated as LP) is a general and very powerful technique[1] for solving optimization problems where the objective (i.e., the thing being optimized) and the constraints are *linear*. Out in the real world, this is the standard approach for solving the Combinatorial Optimization Problems (COPs) that arise all the time. This technique is so common that an LP solver is now included in most common spreadsheets, e.g., Excel. (Note that the term "programming" refers not to a computer program, more to a program in the sense of an "event program," i.e., a plan for something.)

A typical Linear Program consists of three components:

- A list of (real-valued) variables $x_1, x_2, \ldots, x_n$. The goal of your optimization problem is to find good values for these variables.
- An objective function $f(x_1, x_2, \ldots, x_n)$ that you are trying to maximize or minimize. The goal is to find the best values for the variables so as optimize this function.
- A set of constraints that limits the feasible solution space. Each of these constraints is specified as an inequality.

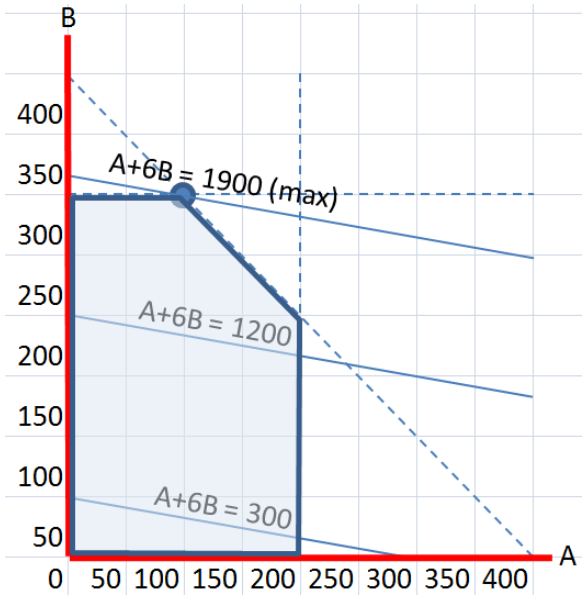
In a Linear Programming problem, both the objective function and the constraints are *linear* functions of the variables.

Example 1. You are employed by Acme Corporation, a company that makes two products: widgets and bobbles. Widgets sell for 1 SGD/widget, and bobbles sell for 6 SGD/bobble. Bobbles clearly make more money for you, and so ideally you would like to sell as many bobbles as you can. However, after doing some market research, you have discovered that there is only demand for at most 300 bobbles and at most 200 widgets. It also turns out that your factory can only produce at most 400 units, whether they are widgets or bobbles. How many widgets and bobbles should you make, in order to maximize your total revenue?

We will represent this as a Linear Programming problem. This problem has two variables: A and B. The variable A represents the number of widgets and the variable B represents the number of bobbles. Your revenue is equal to $A + 6B$, i.e., 1/widget and 6/bobble. Your goal is to maximize this revenue. Your constraints are that $A \leq 200$ and $B \leq 300$: that represents the demand constraints from the market. Your other constraint is a supply constraint: $A + B \leq 400$, since you can only make 400 units total. Finally, we will include two more constraints that are obvious: $A \geq 0$ and $B \geq 0$. These were not included in the problem, but are important to exclude negative solutions. Put together, this yields the following Linear Program:

[1] It is taught in Business School too!

$$\begin{array}{rcll} \max\,(A+6B) & & \text{where:} \\ A & \leq & 200 \\ B & \leq & 300 \\ A+B & \leq & 400 \\ A & \geq & 0 \\ B & \geq & 0 \end{array}$$



On the left, is the LP represented mathematically, specified in terms of an objective function and a set of constraints. On the right is a picture representing the LP geometrically, where the variable A is drawn as the x-axis and the variable B is drawn as the y-axis.

The dashed lines here represent the constraints: $A \leq 200$ (i.e., a vertical line), $B \leq 300$ (i.e., a horizontal line), and $A + B \leq 400$ (i.e., the diagonal line). Each constraint defines a ***halfspace***, i.e., it divides the universe of possible solutions in half. In two-dimensions, each constraint is a line. In higher dimensions, a constraint is defined by a hyperplane[2].

Everything that is beneath the three lines represents the ***feasible region***[3], which is defined as the values of A and B that satisfy all the constraints. In general, the feasible region is the intersection of the halfspaces defined by the hyperplanes, and from this we conclude that the feasible region is a convex[4] polygon.

Definition 1 *The **feasible region** for a Linear Program with variables $x_1, x_2, \ldots, x_n$ is the set of points $(x_1, x_2, \ldots, x_n)$ that satisfy all the constraints.*

Notice that the feasible region for a Linear Program may be: (i) empty, (ii) a single point, or (iii) infinite.

For every point in the feasible region, we can calculate the value of the objective function: $A + 6B$. The goal is to find a point in the feasible region that maximizes this objective. For each value of c, we can draw the line for $A + 6B = c$. Our goal is to find the maximum value of c for which this line intersects the feasible region. You can see, in the picture above, we have drawn in this line for three values of c: $c = 300$, $c = 1\,200$, and $c = 1\,900$. The last line, where $A + 6B = 1\,900$ intersects the feasible region at exactly one point: $(100, 300)$. This point, then, is the maximum value that can be achieved.

One obvious difficulty in solving LPs is that the feasible space may be infinite, and in fact, there may be an infinite number of optimal solutions. (Remember, we are considering real numbers here, so any line segment has a infinite number of points on it.) Let's think a little bit about whether we can reduce this space of possible solutions.

Imagine that I give you possible solution $(100, 300)$ and ask you to decide if it maximizes the objective function. How might you decide?

[2] It will be (much) harder to visualize an LP on more than 3 dimensions. Most textbook examples are on 2D (2 variables).
[3] In [1], this feasible region is called a 'simplex', hence the name of Simplex Method in Section 2.1.
[4] Mathematically, for each pair of points in the set, the line joining the points is wholly contained in the set.

1. You draw the geometric picture and look at it. Recall that $f(100, 300) = 1\,900$. The line represented by the objective function $A + 6B = 1\,900$ cannot move up any farther. Thus, clearly $1\,900$ is the maximum that can be achieved and hence $(100, 300)$ is a maximum.

2. Maybe we can prove algebraically that $(100, 300)$ is maximal. Recall, one of the constraints shows that $A+B \leq 400$. We also know that $B \leq 300$, and so $5B \leq 1\,500$. Putting these facts together, we conclude that:

$$\begin{array}{rcl} A + B & \leq & 400 \\ 5B & \leq & 1\,500 \\ \hline A + 6B & \leq & 1\,900 \end{array}$$

   Since the objective function is equal to $A + 6B$, this shows that we cannot possibly find a solution better than $1\,900$. Since we have found such a solution, we know that $(100, 300)$ is optimal.

   This may seem like a special case, but in fact it turns out that you can always generate such a set of equations to prove that you have solved your LP optimally!

3. One important fact about Linear Programs is that this maximum is always achieved at a vertex of the polygon defined by the constraints (if the feasible region is not empty). Notice that there may be other points (e.g., on an edge or a face) that also maximize the objective, but there is always a vertex that is at least as good. (We will not prove this fact today, but if you think about the geometric picture, it should make sense.) Therefore, one way to prove that your solution is optimal is to examine all the vertices of the polygon.

   How many vertices can there be? In two dimensions, a vertex may occur wherever two (independent) constraints intersect. In general, if there are n dimensions (i.e., there are n variables), a vertex may occur wherever n (linearly independent) hyperplanes (i.e., constraints) intersect. Recall that if you have n linearly independent equations and n variables, there is a single solution—that solution defines a vertex. Of course, if the equations are not linearly independent, you may get many solutions—in that case, there is no vertex. (For example, if the constraints define hyperplanes that are parallel and do not intersect, there is no vertex.) Or, alternatively, if the intersection point is outside the feasible region, this too is not a vertex.

   So in a system with m constraints and n variables, there are $\binom{m}{n} = O(m^n)$ vertices.

   We have thus discovered an exponential time $O(m^n)$ time algorithm for solving a Linear Program: Enumerate each of the $O(m^n)$ vertices of the polytope, calculate the value of the objective function for each point, and take the maximum.

2 Solving Linear Programs

In this class, for the most part, we will ignore the problem of solving Linear Programs. There exist fast and efficient LP solvers, and we will rely on these as a *black-box*. Please study Chapter 29 of [1] if you are keen to explore more details. However, here we give a few hints as to how these LP solvers work. (It is a fascinating and well-studied problem, and if you can build a faster LP solver, you can make a lot of money!)

2.1 Simplex Method

One of the earliest techniques for solving an LP—and still one of the fastest today—is the Simplex method. It was invented by Dantzig in 1947, and remains in common use today. There are many variants, but all take exponential time in the worst-case. However, in practice, for almost every LP that anyone has ever generated, it is remarkably fast.

The basic idea behind the Simplex method is remarkably simple. Recall that if an LP is feasible, its optimum is found at a vertex. (Again, remember that the optimum may be achieved at other points as well, but this does not matter to us.) Hence, the basic algorithm can be described as follows, where the function f represents the objective function.

1. Find any (feasible) vertex v.
2. Examine all the neighboring vertices of v: $v_1, v_2, \ldots, v_k$.
3. Calculate $f(v), f(v_1), f(v_2), \ldots, f(v_k)$. If $f(v)$ is the maximum (among its neighbors), then stop and return v.
4. Otherwise, choose *one of* the neighboring vertices v_j where $f(v_j) > f(v)$. Let $v = v_j$.
5. Go to step (2).

There are several things to notice about this algorithm. First, if the algorithm stops in step (3), then it really has found an optimum point. To prove this fact, we need to examine the geometry of the polytope; because the feasible region is convex and the objective function linear, we can conclude that if $f(v)$ is maximum among its neighbors, then it is maximum in the entire feasible region. This fact ensures that the Simplex Method always returns the right answer.

Second, observe that it will eventually terminate. At some point, it will have explored all the vertices of the polytope. Recall that at one of the vertices we will find an optimum, and hence we will eventually terminate. This also bounds the worst-case running time as $O(m^n)$ for an LP with n variables and m constraints.

Third, notice that Step (4) is somewhat underspecified: Which neighboring vertex should we choose? Depending on how we choose the neighboring vertex, we can get very different performance. (In the case where $n = 2$, there are not many choices. But as the dimension grows, there become a much larger number of neighboring vertices to choose among.) The rule for choosing the next vertex is known as the *pivot rule*, and a large part of designing an efficient simplex implementation is choosing the pivot rule. Even so, all known pivot rules take worst-case exponential time.

Fourth, implementing this algorithm efficiently requires some care. How should one find neighboring vertices and determine where to go next (without re-calculating all the vertices in every step)? In fact, using some basic Linear Algebra and matrix manipulation[5], this can be implemented quite efficiently (e.g., using techniques like Gaussian elimination).

2.2 An Example

As an example, consider running the Simplex Method on Example 1 above (i.e., the Acme corporation optimization problem). In this case, the Simplex method might start with the feasible vertex $(0, 0)$.

First iteration. In the first iteration, it would calculate $f(0, 0) = 0$. It would also look at the two neighboring vertices, calculating that $f(0, 300) = 1\,800$ and $f(200, 0) = 200$. Having discovered that $(0, 0)$ is not optimal, it would choose one of the two neighbors. *Assume*, in this case, that the algorithm chooses next to visit neighbor $(200, 0)$[6].

Second iteration. In the second iteration, it would calculate[7] $f(200, 0) = 200$. It would also look at the two neighboring vertices, calculating that $f(0, 0) = 0$ and $f(200, 200) = 1\,400$. In this case, there is only one neighboring vertex that is better, and it would move to $(200, 200)$.

[5] Hence we need MA1101R as pre-requisite for this module.
[6] Notice that it does not greedily choose the best local move at all times.
[7] Notice that we have computed this value before. Some form of memoization can be used to avoid re-computation.

Third iteration. In the third iteration, it would calculate $f(200, 200) = 1\,400$. It would also look at the two neighboring vertices, calculating that $f(200, 0) = 200$ and $f(100, 300) = 1\,900$. In this case, there is only one neighboring vertex that is better, and it would move to $(100, 300)$.

Fourth iteration. In the fourth iteration, it would calculate $f(100, 300) = 1\,900$. It would also look at the two neighboring vertices, calculating that $f(200, 200) = 1\,400$ and $f(0, 300) = 1\,800$. After discovering that $(100, 300)$ is better than any of its neighbors, the algorithm would stop and return $(100, 300)$ as the optimal point.

Notice that along the way, the algorithm might calculate some points that were not vertices. For example, in the second iteration, it might find the point $(400, 0)$—which is not feasible. Clearly, a critical part of any good implementation is quickly calculating the feasible neighboring vertices.

For the remainder of today, we will ignore the problem of *how to solve* Linear Programs, and instead focus on *how to use* Linear Programming to solve combinatorial graph problems. For now, here's what you need to know about solving Linear Programs:

- If you can represent your Linear Program in terms of a polynomial number of variables and a polynomial number of constraints, then there exist polynomial time algorithms for solving them.
- You can find an LP solver in Excel to experiment with.
- The existing LP-solvers are very efficient for almost all the LPs that you would want to solve. You can try Stanford's ACM ICPC Simplex code library[8].

3 MIN-WEIGHT-VERTEX-COVER and Integer Linear Program

We now introduce the *weighted* version of the MIN-VERTEX-COVER problem, and show how to represent it as an *Integer* Linear Program (ILP). We can then *relax* the ILP, leading to a Linear Program that we can solve. Finally, we use the LP to construct a 2-approximation algorithm.

3.1 MIN-WEIGHT-VERTEX-COVER

To this point, we have looked at *unweighted* version of MIN-VERTEX-COVER: Every vertex has the same (unit) weight 1. It is natural to consider a weighted version of the problem where different vertices have different weights.

Definition 2 *A* **MIN-WEIGHT-VERTEX-COVER** *for a graph $G = (V, E)$ where each vertex $v \in V$ has* ***weight*** *$w(v)$, is a set $S \subseteq V$ such that for every edge $e = (u, v) \in E$, either $u \in S$ or $v \in S$. The cost of vertex cover S is the sum of the weights, i.e., $\sum_{v \in S} w(v)$.*

For MIN-WEIGHT-VERTEX-COVER problem, the 2-approximation algorithm (both the randomized and the deterministic variants) presented in the previous lecture is no longer sufficient. See the examples in Figure 1. In both examples, the simple deterministic 2-approximate algorithm examines exactly one edge and adds both endpoints, incurring a cost of 101. And yet in both cases, there are much better solutions, of cost 9 in the first case and of cost 1 in the second case. Therefore, the deterministic 2-approximate algorithm for MIN-VERTEX-COVER is *no longer* a 2-approximation algorithm for the weighted variant.

There are several methods for solving the MIN-WEIGHT-VERTEX-COVER problem, and our goal over the rest of this lecture is to develop one such solution. In order to do so, we will use Linear Programming that we have been exposed with earlier.

[8] See https://github.com/jaehyunp/stanfordacm/blob/master/code/Simplex.cc

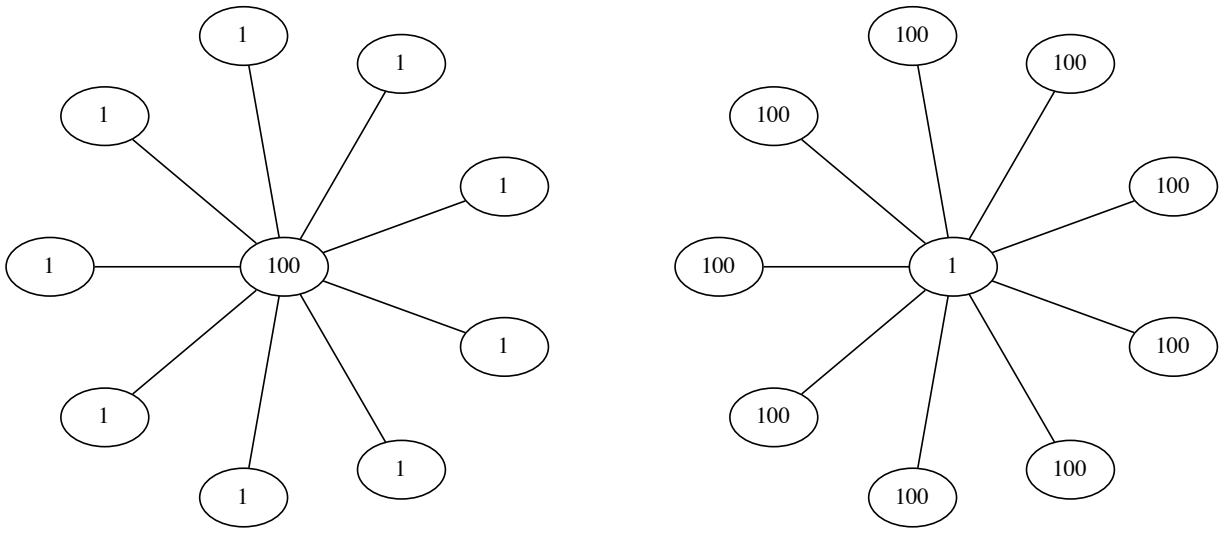


Figure 1: These are two examples of the MIN-WEIGHT-VERTEX-COVER problem, where the value in each vertex represents the weight. Notice in the first case, the optimal vertex cover includes all the leaves; in the second case, the optimal vertex cover includes only the center vertex. In both cases, the simple (deterministic) 2-approximate vertex cover algorithm fails badly, always including a vertex of cost 100.

3.2 MIN-WEIGHT-VERTEX-COVER as an Integer Linear Program

Assume we are given a graph $G = (V, E)$ with weight function $w : V \to \mathbb{R}$, and we want to find a minimum weight vertex cover.

The first step is to define a set of variables. In this case, it is natural to define one variable for each vertex in the graph. Assuming there are n vertices in the graph[9], we define variables $x_1, x_2, \ldots, x_n$. These variables should be interpreted as follows: $x_j = 1$ implies that vertex v_j is included in the vertex cover; $x_j = 0$ implies that vertex v_j is *not* included in the vertex cover.

Given these variables, we can now define the objective function. We want to minimize the sum of the weights of the vertices included in the vertex cover. That is, we want to minimize: $\sum_{j=1}^{n} w(v_j) \cdot x_j$.

Finally, we define the constraints. First, we need to ensure that every edge is covered: For every edge $e = (v_i, v_j)$, we need to ensure that either v_i or v_j is in the vertex cover. We need to represent this constraint as a linear function. Here is one way to do that: $x_i + x_j \geq 1$. This ensures that either x_i or x_j is included in the vertex cover.

We need one more constraint: for each variable x_j, we need to ensure that either $x_j = 1$ or $x_j = 0$. What would it mean if the Linear Programming solver returned that $x_j = 0.4$? This would not be useful. Unfortunately, there is no good way to represent this constraint as a linear function.

And it should not be surprising to you that we cannot represent the MIN-VERTEX-COVER problem as a Linear Program! We know, already, that MIN-VERTEX-COVER is NP-hard. We also know that Linear Programs can be solved in polynomial time. Thus, if we found a Linear Programming formulation for MIN-VERTEX-COVER, that would imply that $P = NP$.

[9] As n can go beyond 3 (variables/dimensions), it is rather impossible to properly visualize the n-dimensional Linear Program of this MIN-WEIGHT-VERTEX-COVER problem.

Instead, we formulate an *Integer* Linear Program:

Definition 3 *An **Integer Linear Program** (ILP) is a Linear Program in which all the variables are constrained to be integer values.*

Thus, we can now represent MIN-WEIGHT-VERTEX-COVER as an Integer Linear Program as follows:

$$\min\left(\sum_{j=1}^{n} w(v_j)\cdot x_j\right) \qquad \text{where:}$$

$$\begin{aligned} x_i + x_j &\geq 1 \quad \text{for all } (i,j)\in E \\ x_j &\geq 0 \quad \text{for all } j \in V \\ x_j &\leq 1 \quad \text{for all } j \in V \\ x_j &\in \mathbb{Z} \quad \text{for all } j \in V \end{aligned}$$

Notice that the objective function is a linear function of the variables x_j, where the weights are simply constants. (There is no term in the expression that looks like $x_i x_j$, i.e., multiplying variables together nor term that looks like x^y for $y \neq 1$.) Similarly, each of the constraints is a linear function of the variables. The only constraint that cannot be expressed as a linear function is the last one, where we assert that each of the variables must be integral. (We will often abbreviate the last three lines by simply stating that $x_j \in \{0,1\}$.)

3.3 Relaxation

Unfortunately, there is no polynomial time algorithm for solving ILPs. We have already effectively shown that solving ILPs is NP-hard. If there were a polynomial time algorithm, we would have proved that $P = NP$. Instead, we will *relax* the ILP to a (regular) LP. That is, we will consider the same optimization problem, but dropping the constraint that the variables be integers. Here, for example, is the MIN-WEIGHT-VERTEX-COVER **relaxation**:

$$\min\left(\sum_{j=1}^{n} w(v_j)\cdot x_j\right) \qquad \text{where:}$$

$$\begin{aligned} x_i + x_j &\geq 1 \quad \text{for all } (i,j)\in E \\ x_j &\geq 0 \quad \text{for all } j \in V \\ x_j &\leq 1 \quad \text{for all } j \in V \end{aligned}$$

Notice that the solution to this LP is no longer guaranteed to be a solution to MIN-WEIGHT-VERTEX-COVER! In fact, there is no obvious way to interpret the solution to this LP. What does it mean if we decide that $x_j = 0.4$?

Solving the relaxed ILP does tell us something: The solution to the Linear Program is *at least as good* as the optimal solution for the original ILP. In the case of MIN-WEIGHT-VERTEX-COVER, imagine that we solve the relaxed ILP and the LP solver returns a set of variables $x_1, x_2, \ldots, x_n$ such that $\left(\sum_{j=1}^{n} w(v_j)\cdot x_j\right) = c$, for some value c. Then we know that $OPT(G) \geq c$, where $OPT(G)$ is the optimal (integral) solution for MIN-WEIGHT-VERTEX-COVER.

Why? Imagine there were a better solution $x'_1, x'_2, \ldots, x'_n$ return by $OPT(G)$. In that case, this solution would also be a feasible solution for the relaxed Linear Program: Each of these variables x'_j is either 0 or 1, and hence would be a valid choice for the relaxed case where $0 \leq x_j \leq 1$. Hence the LP solver would have found this better solution.

The general rule is that when you expand the space being optimized over, your solution can only improve. By relaxing an ILP, we are expanding the range of possible solutions, and hence we can only find a better (or at least equal) solution.

Lemma 4 *Let I be an Integer Linear Program, and let $L = relax(I)$ be the relaxed Integer Linear Program. Then $OPT(I) \geq OPT(L)$.*

3.4 Solving MIN-WEIGHT-VERTEX-COVER

Returning to MIN-WEIGHT-VERTEX-COVER, we have defined an ILP I for solving MIN-WEIGHT-VERTEX-COVER. We have relaxed this ILP and generated an LP L. The first thing we must argue is that there is a feasible solution the Linear Program:

Lemma 5 *The relaxed ILP for the vertex cover problem has a feasible solution.*

Proof Consider the solution where each $x_j = 1$. This solution satisfies all the constraints. □

Assume we have now solved the LP L using an LP solver and discovered a solution $x_1, x_2, \ldots, x_n$ to the Linear Program L. Our goal is to use this (non-integral) solution to find an (integral) solution to the MIN-WEIGHT-VERTEX-COVER problem.

Here is a simple observation: If (u, v) is an edge in the graph, then either $x_u \geq 1/2$ or $x_v \geq 1/2$. Why? Well, the Linear Program guarantees that $x_u + x_v \geq 1$. The Linear Program may well choose non-integral values for x_u and x_v, but it will always ensure that all the (linear) constraints are met.

Consider, then, the following procedure for ***rounding*** our solution to the Linear Program:

- For every vertex $u \in V$: if $x_u \geq 1/2$, then add u to the vertex cover C.

We claim that the resulting set C is a vertex cover:

Lemma 6 *The set C constructed by rounding the variables $x_1, x_2, \ldots, x_n$ is a vertex cover.*

Proof Assume, for the sake of contradiction, that there is some edge (u, v) that is not covered by the set C. Since neither u nor v was added to the vertex cover, this implies that $x_u < 1/2$ and $x_v < 1/2$. In that case, $x_u + x_v < 1$, which violates the constraint for the LP, which is a contradiction. □

It remains to show that the rounded solution is a good approximation, i.e., that we have not increased the cost too much[10].

Lemma 7 *Let C be the set constructed by rounding the variables $x_1, x_2, \ldots, x_n$. Then $cost(C) \leq 2 \times cost(OPT)$.*

Proof The proof relies on two inequalities. First, we relate the cost of OPT to the cost of the Linear Program solution:

[10] Unless, of course, being up to two times worse than optimal, a.k.a. 200%-off from optimal, is not an acceptable criteria.

$$cost(OPT) \geq \sum_{j=1}^{n} w(v_j) \cdot x_j \tag{1}$$

This follows because the x_j were calculated as the optimal solution to a relaxation of the original MIN-WEIGHT-VERTEX-COVER problem. Recall, by relaxing the ILP to an LP, we can only improve the solution (i.e., in this case, we can only get a solution that is $\leq$ the integral solution).

Second, we related the cost of the LP solution to the cost of the rounded solution. To represent the rounded solution, let $y_j = 1$ if $x_j \geq 1/2$, and let $y_j = 0$ otherwise. Now, the cost of the final solution, i.e., $cost(C)$, is equal to $\sum_{j=1}^{n} w(v_j) \cdot y_j$. Notice, however, that $y_j \leq 2x_j$, for all j. Therefore:

$$\begin{aligned}
\sum_{j=1}^{n} w(v_j) \cdot y_j &\leq \sum_{j=1}^{n} w(v_j) \cdot (2x_j) \\
&\leq 2\left(\sum_{j=1}^{n} w(v_j) \cdot x_j\right) \\
&\leq 2 \times OPT(G)
\end{aligned}$$

Thus, the rounded solution has cost at most $2 \times OPT(G)$. □

3.5 General Approach

We have thus discovered a polynomial time 2-approximation algorithm for the MIN-WEIGHT-VERTEX-COVER:

- Define the MIN-WEIGHT-VERTEX-COVER problem as an Integer Linear Program (ILP).
- Relax the ILP to a standard Linear Program (LP).
- Solve the LP using an existing LP solver of your choice.
- Round the solution, adding vertex v to the cover if and only if $x_j \geq 1/2$.

The general approach we have defined here for MIN-WEIGHT-VERTEX-COVER can also be applicable to a *wide variety* of Combinatorial Optimization Problems (COPs). The basic idea is to find an ILP formulation, relax it to an LP, solve the LP, and then round the solution until it is integral. MIN-WEIGHT-VERTEX-COVER yields a solution that is particularly easy to round. For other problems, however, rounding the solution may be more difficult.

One question, perhaps, is when this approach works and when this approach fails. Sometimes, the non-integral solution (returned by the LP solver) will be much better than the optimal integral solution. In such a case, it will be very difficult to round the solution to find a good approximation. This ratio is defined as the ***integrality gap***. Assume I is some Integer Linear Program:

$$\textbf{integrality gap} = \frac{OPT(relax(I))}{OPT(I)} \tag{2}$$

For a specific problem, if the integrality gap is at least c, then the best approximation algorithm you can achieve by rounding the solution the relaxed Integer Linear Program is a c-approximation. This is a fundamental limit on this technique for developing approximation algorithms.

4 Linear Programming Terminologies

In this section, we will review some basic Linear Programming terminologies.

4.1 Definitions

In general, a Linear Program consists of:

- A set of variables: $x_1, x_2, \ldots, x_n$.
- A linear objective to maximize (or minimize): $c_1x_1 + c_2x_2 + \cdots + c_nx_n$. Thinking of c and x as vectors, this is often written more tersely as: c^Tx (where c^T represents the transpose of c, and multiplication here represents the dot product.)
- A set of linear constraints of the form: $a_{j,1}x_1 + a_{j,2}x_2 + \cdots + a_{j,n}x_n \leq b_j$. (This version represents the jth constraint.) This is often abbreviated by the matrix equation: $Ax \leq b$.

Putting these pieces together, a Linear Program is often presented in the following form:

$$\begin{array}{rcl} \max cx & & \text{where} \\ Ax & \leq & b \\ x & \geq & 0 \end{array}$$

4.2 Terminology

Standard terminologies for Linear Programs:

- A point x is ***feasible*** if it satisfies all the constraints.
- An LP is bounded if there is some value V such that $c^Tx \leq V$ for all points x.
- Given a point x and a constraint $a^Tx \leq b$, we say that the constraint is ***tight*** if $a^Tx = b$; we say that the constraint is ***slack*** if $a^tx < b$.
- A ***halfspace*** is the set of points x that satisfy one constraint. For example, the constraint $a^tx \leq b$ defines a halfspace containing all the points x for which this inequality is true. A halfspace is a convex set.
- The ***polytope*** of the LP is the set of points that satisfy all the constraints, i.e., the intersection of all the constraints. The polytope of an LP is convex, since it is the intersection of halfspaces (which are convex).
- A point x is a vertex for an n-dimensional LP if there are n linearly independent constraints for which it is tight.

For every Linear Program, we know that one of the following three cases holds:

- The LP is infeasible. There is no value of x that satisfies the constraints.
- The LP has an optimal solution.
- The LP is unbounded.

(Mathematically, this follows from the fact that if the LP is feasible and bounded, then it is a closed and bounded subset of $\mathbb{R}^n$ and hence has a maximum point.)

4.3 Standard Form

In general, we will think of Linear Programs as having a standard (or canonical) form (see Section 29.1 of [1], omitting the slack form for now). Many ready made LP solvers out there assume this standard form.

For a maximization problem, the standard form is:

$$\begin{array}{rcl} \max cx & & \text{where} \\ Ax & \leq & b \\ x & \geq & 0 \end{array}$$

For a minimization problem, the standard form is:

$$\begin{array}{rcl} \min cx & & \text{where} \\ Ax & \geq & b \\ x & \geq & 0 \end{array}$$

Notice that the standard forms obey the following rules:

- Inequalities: Every constraint is an inequality. For a maximization problem, the inequalities are all $\leq$ (except for the $x \geq 0$ constraints). For a minimization problem, the inequalities are all $\geq$.
- Non-negative: Every variable is constrained to be ≤ 0.

(Beware in other classes or other textbooks, there may be different preferred "standard forms".)

It is easy to translate Linear Programs in other forms into standard form. For example, we can translate a minimization problem into a maximization problem (and vice versa) by negating the objective function. We can reverse the direction of an inequality by negating the constraint. We can translate an equality into an inequality by introducing two constraints. Etc.

Imagine, for example, that you are given the following Linear Program and your LP solver is designed to solve a maximization problem:

$$\begin{array}{rcl} \min x_1 + 2x_2 - x_3 & & \text{where // this is a minimization problem} \\ x_1 + x_2 & = & 7 \text{ // this is an equality} \\ x_2 - 2x_3 & \geq & 4 \text{ // this is a } \geq \text{ inequality} \\ x_1 & \leq & 2 \end{array}$$

First, we can translate this into a maximization problem by negating the objective function:

$$\begin{array}{rcl} \max -(x_1 + 2x_2 - x_3) & & \text{where} \\ x_1 + x_2 & = & 7 \text{ // this is an equality} \\ x_2 - 2x_3 & \geq & 4 \text{ // this is a } \geq \text{ inequality} \\ x_1 & \leq & 2 \end{array}$$

Then, we can replace the equality with two inequalities:

$$\begin{aligned} \max -(x_1 + 2x_2 - x_3) & & \text{where} \\ x_1 + x_2 & \leq & 7 \\ x_1 + x_2 & \geq & 7 \text{ // this is a } \geq \text{ inequality} \\ x_2 - 2x_3 & \geq & 4 \text{ // this is a } \geq \text{ inequality} \\ x_1 & \leq & 2 \end{aligned}$$

Next, we can flip the direction on the inequalities that are in the incorrect direction:

$$\begin{aligned} \max -(x_1 + 2x_2 - x_3) & & \text{where} \\ x_1 + x_2 & \leq & 7 \\ -(x_1 + x_2) & \leq & -7 \\ -(x_2 - 2x_3) & \leq & -4 \\ x_1 & \leq & 2 \end{aligned}$$

The last step in putting the LP in standard form is to constrain all the variables to be positive. To accomplish this, we create new variables to replace the variables here. Specifically, we define variables $x_1^+, x_1^-, x_2^+, x_2^-, x_3^+, x_3^-$. We define them as follows:

$$\begin{aligned} x_1 &= (x_1^+ - x_1^-) \\ x_2 &= (x_2^+ - x_2^-) \\ x_3 &= (x_3^+ - x_3^-) \end{aligned}$$

We can then constrain the new variables to all be positive. We end up with the following LP in standard form:

$$\begin{aligned} \max -((x_1^+ - x_1^-) + 2(x_2^+ - x_2^-) - (x_3^+ - x_3^-)) & & \text{where} \\ (x_1^+ - x_1^-) + (x_2^+ - x_2^-) & \leq & 7 \\ -((x_1^+ - x_1^-) + (x_2^+ - x_2^-)) & \leq & -7 \\ -((x_2^+ - x_2^-) - 2(x_3^+ - x_3^-)) & \leq & -4 \\ (x_1^+ - x_1^-) & \leq & 2 \\ x_1^+ & \geq & 0 \\ x_1^- & \geq & 0 \\ x_2^+ & \geq & 0 \\ x_2^- & \geq & 0 \\ x_3^+ & \geq & 0 \\ x_3^- & \geq & 0 \end{aligned}$$

Solving this final LP (and translating back to the initial variables) gives us the correct optimal solution for the original LP. (Notice that it may increase the number of variables and constraints by a constant factor).

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 3rd edition, 2009.

Index

CS4234: Optimisation Algorithms Lecture 3

MIN-SET-COVER

V1.0: Seth Gilbert, V1.1: Steven Halim *August 23, 2016*

Abstract

Here we consider the problem of MIN-SET-COVER. We begin with an example, and then discuss the classical greedy solution. Finally, we analyze the Greedy Algorithm and show that it is a $O(\log n)$-approximation algorithm. In the analysis, notice how we begin by giving a lower bound on OPT, showing that OPT can only do so well. We then relate this to the performance of the Greedy Algorithm to get the approximation ratio.

1 MIN-SET-COVER

The **MIN-SET-COVER** is a Combinatorial Optimization Problem (COP) that frequently shows up in real-world scenarios, typically when you have collections of needs (e.g., tasks, responsibilities, or capabilities) and collections of resources (e.g., employees or machines) and you need to find a minimal set of resources to satisfy your needs. In fact, MIN-SET-COVER generalizes MIN-VERTEX-COVER that we have seen in the earlier lectures. In vertex cover, each vertex (i.e., set) covers the adjacent edges (i.e., elements); in set cover, each set can cover an arbitrary set of elements.

Unfortunately, MIN-SET-COVER is NP-hard (i.e., NP-complete as a decision problem). See **Example 2.** below to see that MIN-VERTEX-COVER can be reduced to MIN-SET-COVER.

In fact, it is known that MIN-SET-COVER is hard to approximate [1]. There are no polynomial-time α-approximation algorithms *for any constant*, assuming $P \neq NP$.

1.1 Problem Definition

We first define the problem, and then give some examples that show how MIN-SET-COVER problem might appear in the real world.

Definition 1 *Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of n elements. Let $S_1, S_2, \ldots, S_m$ be subsets of X, i.e., each $S_j \subseteq X$. Assume that every item in X appears in some set, i.e., $\bigcup_j S_j = X$. A **set cover** of X with S is a set $I \subseteq \{1, 2, \ldots, m\}$ such that $\bigcup_{j \in I} S_j = X$. The solution for* **MIN-SET-COVER** *problem is a set cover I of minimum size.*

That is, a minimum set cover is the smallest set of sets $\{S_{i_1}, S_{i_2}, \ldots, S_{i_k}\}$ that covers X.

Example 1. Assume you have a set of software developers: Alice, Bob, Collin, and Dave. Each programmer knows at least one programming language. Alice knows C and C++. Bob knows C++ and Java. Collin knows C++, Ruby, and Python. Dave knows C and Java. Your job is to hire a team of programmers. You are given two requirements: (i) there has to be at least one person on the team who knows each language (i.e., C, C++, Java, Python, and Ruby), and (ii) your team should be as small as possible (maybe your team is running on a tight budget).

This is precisely a MIN-SET-COVER problem. The base elements X are the 5 different programming languages. Each programmer represents a set. Your job is to find the minimum number of programmers (i.e., the minimum number of sets) such that every language is covered.

See Figure 1 for an example of this problem, depicted here as a (directed) bipartite graph. You will notice that any set cover problem can be represented as a (directed) bipartite graph, with the sets represented on one side, the base elements represented on the other side, and edges only go from sets to base elements.

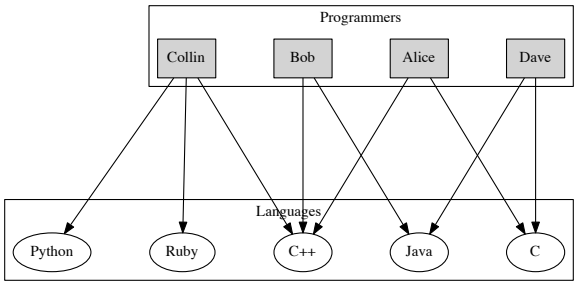


Figure 1: The problem of hiring software developers, represented as a bipartite graph. The goal is to choose a minimum sized set of programmers to ensure that every language is known by at least one of your programmers.

In this case, Alice, Bob, and Collin form a set cover of size 3. However, Collin and Dave form a set cover of size 2, which is optimal, i.e. the solution for this MIN-SET-COVER problem instance.

Example 2. Any vertex cover problem can be represented as a set cover problem, i.e. we can reduce MIN-VERTEX-COVER $\leq_p$ MIN-SET-COVER. Assume you are given a graph $G = (V, E)$ and you want to find a vertex cover. In this case, you can define $X = E$, i.e., the elements to be covered are the edges. Each vertex represents a set. We define the set $S_u = \{(u, v) : (u, v) \in E\}$, i.e., S_u is the set of edges adjacent to u. The problem of vertex cover is now to find a set of sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ such that every edge is covered. For example, see Figure 2.

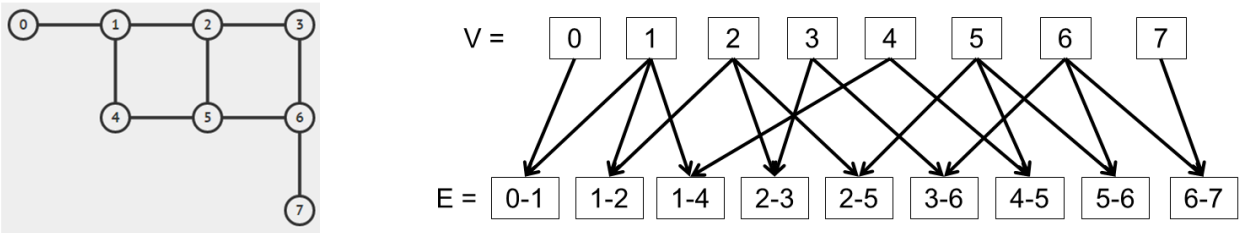


Figure 2: We can reduce an MIN-VERTEX-COVER instance into an MIN-SET-COVER instance in polynomial time.

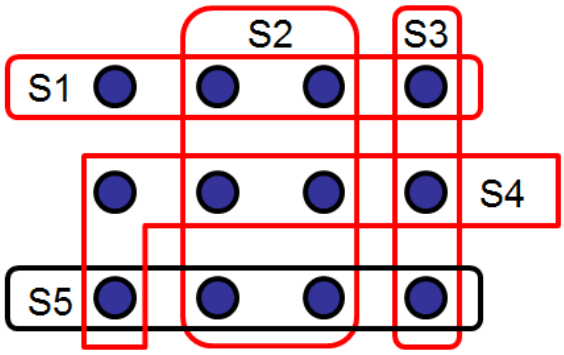
1.2 Greedy Algorithm

There is a simple Greedy Algorithm for tackling MIN-SET-COVER, albeit sub-optimally as we will analyze later:

```
/* This algorithm adds sets greedily, one at a time, until everything is
   covered.  At each step, the algorithm chooses the next set that will
   cover the most uncovered elements.                                    */
Algorithm: GreedySetCover(X, S_1, S_2, ..., S_m)
Procedure:
    I ← ∅
    /* Repeat until every element in X is covered:                      */
    while X ≠ ∅ do
        Let d(j) = |S_j ∩ X| // This is the number of uncovered elements in S_j
        Let j = argmax_{i∈{1,2,...,m}} d(i) // Break ties by taking lower i
        I ← I ∪ {j} // Include set S_j into the set cover
        X ← X \ S_j // Remove elements in S_j from X.
    return I
```

The algorithm proceeds greedily, adding one set at a time to the set cover until every element in X is covered by at least one set. In the first step, we add the set the covers the most elements. At each ensuing step, the algorithm chooses the set that covers the most elements that remain uncovered.

Let's look at Figure 3. Here we have 12 elements (represented by the dots) and 5 sets: $S1, S2, S3, S4, S5$. In the first iteration, we notice that set $S2$ covers the most elements, i.e., 6 elements, and hence it is added to the set cover. In the second iteration, set $S3$ and $S4$ both cover 3 new elements, and so we add set $S3$ to the set cover. In the third iteration, set $S4$ covers 2 new elements, and so we add it to the set cover. Finally, in the fourth step, set $S1$ covers one new element and so we add it to the set cover. Thus, we end up with a set cover consisting of $S1, S2, S3, S4$. Notice, though, that the optimal set cover consists of only three elements: $S1, S4, S5$.



Set	d(j) - 1	d(j) - 2	d(j) - 3	d(j) - 4
S1	4	2	1	1
S2	6			
S3	3	3		
S4	5	3	2	
S5	4	2	1	0

Figure 3: On the left is an example of set cover consisting of twelve elements and five sets. On the right is a depiction of what happens when you execute the GreedySetCover algorithm on this example. Each column represents the number of new elements covered by each set at the beginning of the step.

1.3 Analysis

Our goal is to show that the Greedy Algorithm for MIN-SET-COVER shown above is a $O(\log n)$-approximation of optimal. As is typical, in order to show that it is a good approximation of optimal, we need some way to bound the optimal solution. Throughout this section, we will let OPT refer to the MIN-SET-COVER.

Intuition. To get some intuition, let's consider Figure 3 and see what we can say about the optimal solution. Notice that there are 12 elements that need to be covered, and none of the sets cover more than 6 elements. Clearly, then, any solution for this instance of MIN-SET-COVER problem requires at least $12/6 = 2$ sets. Now consider the situation after the first iteration, i.e., after adding set $S3$ to the set cover. At this point, there are 6 elements that remain to be covered, and none of the sets cover more than 3 elements. Any solution to the (smaller) MIN-SET-COVER problem requires at least $6/3 = 2$ sets.

In general, if at some point during the Greedy Algorithm, there are only k elements that remain uncovered and none of the sets covers more than t elements, then we can conclude that $OPT \geq k/t$. We will apply this intuition to show that the Greedy Algorithm is a good approximation of OPT.

Definitions. Assume we run the Greedy Algorithm for MIN-SET-COVER on elements X and sets $S_1, S_2, \ldots, S_m$. When we run the algorithm, let us label the elements in the order that they are covered. For example, let's use Figure 3 again:

$$\underbrace{x_1, x_2, x_3, x_4, x_5, x_6}_{S_2}, \underbrace{x_7, x_8, x_9}_{S_3}, \underbrace{x_{10}, x_{11}}_{S_4}, \underbrace{x_{12}}_{S_1}$$

That is, x_1 is the first element covered, x_2 is the second element covered, etc. Under each element, we indicate *the first set that covered it*. In this example, notice that the first set chosen (S_2) covers 6 new elements, the second set chosen (S_3) covers 3 new elements, the third set chosen (S_4) covers 2 new elements (with element x_3, x_4, x_8 already covered by two other sets previously), etc. Each successive set covers *at most* the same number of elements as the previous one, *because the algorithm is greedy*: For example, if S_3 here had covered more new elements than S_2, then it would have been selected before S_2.

For each element x_j, let c_j be the number of elements covered at the same time. In the example, this would yield:

$$c_1 = 6, c_2 = 6, c_3 = 6, c_4 = 6, c_5 = 6, c_6 = 6, c_7 = 3, c_8 = 3, c_9 = 3, c_{10} = 2, c_{11} = 2, c_{12} = 1$$

Notice that $c_1 \geq c_2 \geq c_3 \geq \cdots$, because the algorithm is greedy.

We define $cost(x_j) = 1/c_j$. In this way, the cost of covering all the new elements for some set is exactly 1. In this example, the cost of covering $x_1, x_2, x_3, x_4, x_5, x_6$ is 1, the cost of covering x_7, x_8, x_9 is 1, etc. In general, if I is the set cover constructed by the Greedy Algorithm, then:

$$|I| = \sum_{j=1}^{n} cost(x_j) \; .$$

Key step. Let's consider the situation after elements $x_1, x_2, \ldots, x_{j-1}$ have already been covered, and the elements $x_j, x_{j+1}, \ldots, x_n$ remain to be covered. Let OPT be the optimal solution for covering all n elements.

What is the best that OPT can do to cover the element $x_j, x_{j+1}, \ldots, x_n$? How many sets does OPT need to cover these remaining elements?

Notice that there remain $n - j + 1$ uncovered elements. However, no set covers more than $c(j)$ of the remaining elements. In particular, all the sets already selected by the Greedy Algorithm cover *zero* of the remaining elements. Of the sets not yet chosen by the Greedy Algorithm, the one that covers the most remaining elements covers $c(j)$ of those elements: Otherwise, the Greedy Algorithm would have chosen a different set.

Therefore, OPT needs at least $(n - j + 1)/c(j)$ sets to cover the remaining $(n - j + 1)$ elements. We thus conclude that:

$$OPT \geq \frac{n-j+1}{c(j)} \geq (n-j+1)cost(x_j)$$

Or to put it differently:

$$cost(x_j) \leq \frac{OPT}{(n-j+1)}$$

We can now show that the Greedy Algorithm provides a good approximation:

$$\begin{aligned} |I| &= \sum_{j=1}^{n} cost(x_j) \\ &\leq \sum_{j=1}^{n} \frac{OPT}{(n-j+1)} \\ &\leq OPT \sum_{i=1}^{n} \frac{1}{i} \\ &\leq OPT(\ln n + O(1)) \end{aligned}$$

(Notice that the third inequality is simply a change of variable where $i = (n-j+1)$, and the fourth inequality is because the Harmonic series $1/1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n$ can be bounded by $\ln n + O(1)$.)

We have therefore shown than the set cover constructed is at most $O(\log n)$ times optimal, i.e., the Greedy Algorithm is an $O(\log n)$-approximation algorithm:

Theorem 2 *The algorithm* GREEDYSETCOVER *is a* $O(\log n)$*-approximation*[1] *algorithm for* MIN-SET-COVER *problem.*

There are two main points to note about this proof. First, the key idea was to (repeatedly) bound OPT, so that we could relate the performance of the greedy algorithm to the performance of OPT. Second, the proof crucially depends on the fact that the algorithm is *greedy*. (Always try to understand how the structure of the algorithm, in this case the greedy nature of the algorithm, is used in the proof.) The fact that the algorithm is greedy leads directly to the bound on OPT by limiting the maximum number of new elements that any set could cover.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm.* MIT Press, 3rd edition, 2009.

[1] Approximation ratio of $O(\log n)$ means that the approximation ratio (slowly) grows as the instance size n increases. This may not be an ideal situation but we can use this Greedy Set Cover as a starting point for other advanced algorithms.

Index

CS4234: Optimisation Algorithms Lecture 3

STEINER-TREE (3 variants)

V1.0: Seth Gilbert, V1.1: Steven Halim *August 23, 2016*

Abstract

Today we consider a new network construction problem where we are given a set of vertices in a graph to connect. A Steiner Tree is a subgraph that connects a set of required terminals, and we want to find the Steiner Tree with the minimum cost, i.e. the solution to STEINER-TREE problem. Today, we discuss three variants of this problem: the EUCLIDEAN-STEINER-TREE, the METRIC-STEINER-TREE, and the GENERAL-STEINER-TREE Problem, and show how to approximate each of these.

1 STEINER-TREE

In this section, we will define the STEINER-TREE[1] problem. We begin with a well-known problem, i.e., that of a MIN-SPANNING-TREE, and then generalize from there.

1.1 MIN-SPANNING-TREE

Imagine you were given a map containing a set of cities, and were asked to develop a plan for connecting these cities with roads. Building a road costs 1 000 000 SGD per kilometer, and you want to minimize the length of the highways. Perhaps the map looks like this (drawn on a 2D plane, units: kilometers):

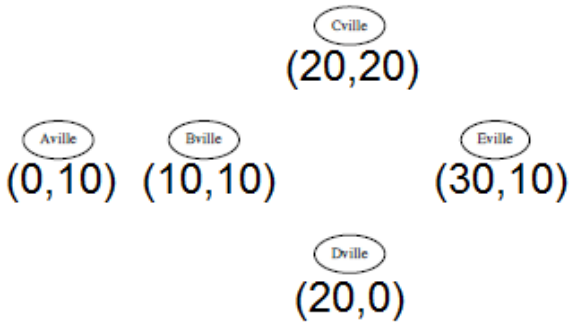


Figure 1: How do you connect the cities with a road network as cheaply as possible?

This is a standard network design question, and the solution is typically to find the minimum cost spanning tree. Recall that the problem of finding a MIN-SPANNING-TREE[2] is defined as follows:

Definition 1 *Given a graph $G = (V, E)$ and edge weights $w : E \to \mathbb{R}$, find a subset of the edges $E' \subseteq E$ such that: (i) the subgraph (V, E') is a spanning tree, and (ii) the sum of edge weights $\sum_{e \in E'} w(e)$ is minimized.*

We can then solve the road network problem described above using the following general approach:

[1] This problem is named after a Swiss mathematician named Jakob Steiner (1796-1863).
[2] Recall CS1231/CS2010/CS2020!

- For every pair of location (u, v) calculate the distance $d(u, v)$. This part is $O(V^2)$.
- Build a graph $G = (V, E)$ where V is the set of locations, and for every pair of vertices $u, v \in V$, define edge $e = (u, v)$ with weight $d(u, v)$. This results in a complete graph K_V.
- Find a minimum spanning tree of G using either Kruskal's Algorithm[3] or Prim's Algorithm[4]. Please review `http://visualgo.net/mst` for more details about these two classic MST algorithms that run in $O(E \log V)$, or $O(V^2 \log V)$ in this case as we are working with a complete graph K_V.
- Return the minimum spanning tree.

In this case, you will get a road network that looks something like on the left side of Figure 2.

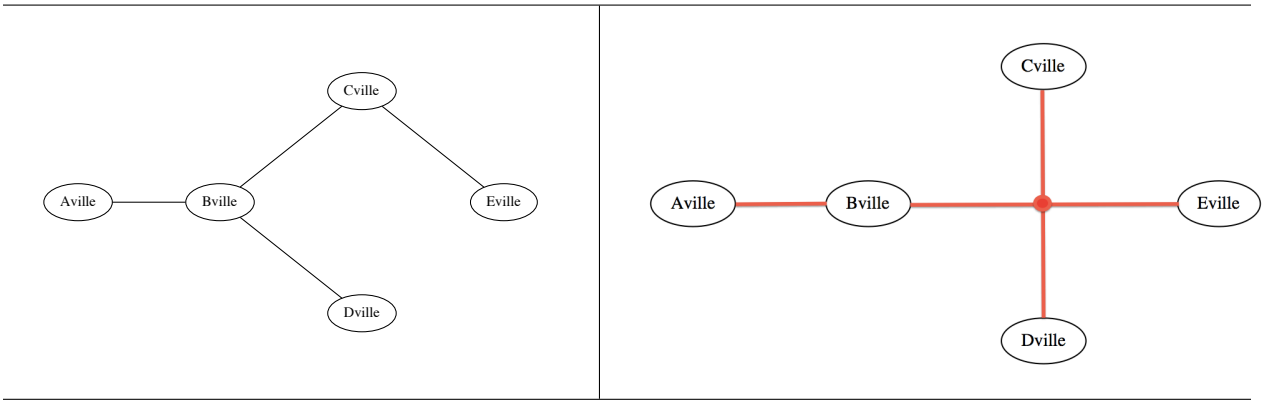


Figure 2: On the left is the graph that results from finding a minimum spanning tree with cost ≈ 52.42 M SGD. On the right is an alternate method of connecting the cities with a road network, by creating a new intersection in the middle with cost 50 M SGD.

1.2 EUCLIDEAN-STEINER-TREE

Is the minimum spanning tree solution, however, really the best solution to the problem? What about the road network on the right side of Figure 2? Notice that this road network is not actually connected in the original graph! The two edges $(B(ville), E(ville))$ and $(C(ville), D(ville))$ do not share an endpoint. Instead, they *intersect* at some new point in the middle of nowhere. Thus the minimum spanning tree approach *will never find* this solution[5].

The goal, then, of the EUCLIDEAN-STEINER-TREE problem is to find the best way to connect the cities when you are allowed to add new vertices to the graph in Euclidean space (e.g., the new intersection above). Formally, we define it as follows:

Definition 2 *Assume that you are given a set R of n distinct points in the Euclidean (2-dimensional) plane. Find a set of points S and a spanning tree $T = (R \cup S, E)$ such that that the weight of the tree is minimized. The **weight** of the tree is defined as:*

$$\sum_{(u,v)\in E} |u - v| ,$$

*where $|u - v|$ refers to the Euclidean distance from u to v. The resulting tree is called a **Euclidean Steiner Tree** and the points in S are called **Steiner** points.*

[3]Recall, Kruskal's Algorithm iterates through the edges from lightest to heaviest, adding an edge if it does not create a cycle.

[4]Recall, Prim's Algorithm grows a spanning tree from a single source vertex, repeatedly takes lightest neighboring edge that does not create a cycle until the tree spans the entire graph.

[5]That is, if we stop at CS2010/CS2020 level, we will not know about the existence of these better solution(s).

As you can see above, adding new points to the graph can results in a spanning tree of lower cost than just the standard minimum spanning tree result! The goal of the EUCLIDEAN-STEINER-TREE problem is to determine how much we can reduce the cost.

Unlike the MIN-SPANNING-TREE problem, finding the minimum solution for EUCLIDEAN-STEINER-TREE problem is NP-hard [1]. Unfortunately we do not have an easy-to-describe-yet-reasonably-good algorithm for this variant, so we will not discuss this variant in depth. We do, however, know some facts about the structure of any optimal Euclidean Steiner Tree:

- Each Steiner point in an optimal solution has degree 3.
- The three lines entering a Steiner point form 120 degree angles, in an optimal solution.
- An optimal solution has at most $n - 2$ Steiner points.

Using these known facts, we can revisit Figure 1 and Figure 2 above to come up with an even better solution, shown in Figure 3 below.

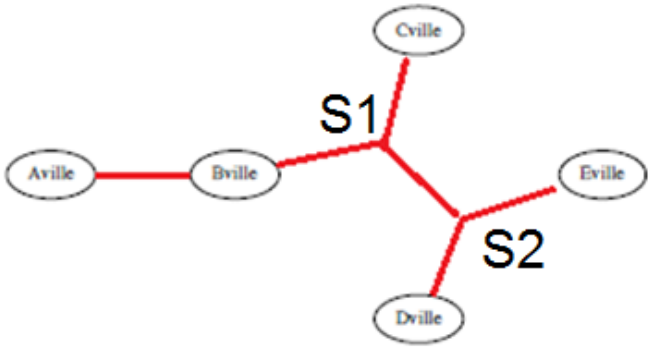


Figure 3: An even better solution with cost ≈ 48.61 M SGD

1.3 METRIC-STEINER-TREE

It is natural, at this point, to generalize beyond the Euclidean plane. Assume you are given n points, as before. In addition you are given a distance function $d : V \times V \to \mathbb{R}$ which gives the pairwise distance between any two vertices (u, v). If the points are in the Euclidean plane, we can simply define $d(u, v)$ to be the Euclidean distance between u and v. However, the distance function d can be *any* metric function, e.g. Manhattan distance[6]. Recall the definition of a metric:

Definition 3 *We say that function $d : V \times V \to \mathbb{R}$ is a **metric** if it satisfies the following properties:*

- Non-negativity*: For all $u, v \in V$, $d(u, v) \geq 0$.*
- Identity*: For all $u \in V$, $d(u, u) = 0$.*
- Symmetric*: For all $u, v \in V$, $d(u, v) = d(v, u)$.*
- Triangle inequality*: For all $u, v, w \in V$, $d(u, v) + d(v, w) \geq d(u, w)$.*

[6]Or taxicab distance. Manhattan distance of two points (a, b) and (c, d) is formally defined as $| a - c | + | b - d |$.

(Technically, this is often referred to as a pseudometric, since we allow distances $d(u, v)$ for $u \neq v$ to equal 0.) The key aspect of the distance function d is that it must satisfy the triangle inequality.

If we want to think of the input as a graph, we can define $G = (V, E)$ where V is the set of points, and E is the set of all $\binom{n}{2}$ pairs of edges, where the weight of edge (u, v) is equal to $d(u, v)$.

As in the Euclidean case, we can readily find a minimum spanning tree of G. However, the STEINER-TREE problem is to find if there is any better network, if we are allowed to add additional points. Unlike in the Euclidean case, however, it is not immediately clear which points can be added. Therefore, we are also given a set S of possible Steiner points to add. The goal is to choose some subset of S to minimize the cost of the spanning tree. The Metric Steiner Tree problem is defined more precisely as follows:

Definition 4 *Assume we are given:*

- *A set of **required** vertices R,*
- *A set of **Steiner** vertices S,*
- *A distance function $d : (R \cup S) \times (R \cup S) \to \mathbb{R}$ that is a distance metric on the points in R and S.*

The **METRIC-STEINER-TREE** *problem is to find a subset $S' \subset S$ of the Steiner vertices and a spanning tree $T = (R \cup S', E)$ of minimum weight. The **weight** of the tree $T = (R \cup S', E)$ is defined to be:*

$$\sum_{(u,v)\in E} d(u, v) \ .$$

1.4 GENERAL-STEINER-TREE

At this point, we can generalize even further to the case where d is not a distance metric. Instead, assume that we are simply given an arbitrary graph with edge weights, where some of the vertices are required vertices and some of the vertices are Steiner vertices.

Definition 5 *Assume we are given:*

- *a graph $G = (V, E)$,*
- *edge weights $w : E \to \mathbb{R}$,*
- *a set of **required** vertices $R \subseteq V$,*
- *a set of **Steiner** vertices $S \subseteq V$.*

Assume that $V = R \cup S$. The **GENERAL-STEINER-TREE** *problem is to find a subset $S' \subset S$ of the Steiner vertices and a spanning tree $T = (R \cup S', E)$ of minimum weight. The **weight** of the tree $T = (R \cup S', E)$ is defined to be:*

$$\sum_{(u,v)\in E} d(u, v) \ .$$

1.5 Summarizing the Three Different Variants

So far, we have defined three variants of the STEINER-TREE problem:

- *Euclidean:* The first variant assumes that we are considering points in the Euclidean plane.
- *Metric:* The second variant assumes that we have a distance metric.
- *General:* The third variant allows for an arbitrary graph.

Notice that the GENERAL-STEINER-TREE problem is clearly a generalization of the METRIC-STEINER-TREE problem. On the other hand, if the set of Steiner points/vertices is restricted to be finite (or countable)[7], then the METRIC-STEINER-TREE problem is not simply a generalization of the EUCLIDEAN-STEINER-TREE problem as the EUCLIDEAN-STEINER-TREE problem allows *any* points in the plane to be a Steiner point!

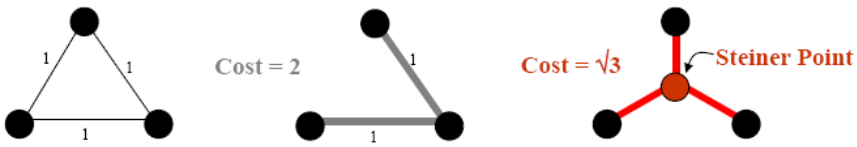
All the variants of the problem are relatively important in practice. In almost any network design problem, we are really interested in Steiner Trees, not simply Minimum Spanning Trees[8]. (One common example is VLSI layout, where we need to route wires between components on the chip.)

All three variants of the problem are NP-hard [1].

2 Steiner Tree Approximations: OK and Bad Examples

There is a simple and natural heuristic for solving the Steiner Tree problem: Just ignore all the Steiner vertices and simply find a minimum spanning tree for the required vertices R. Does this find a good approximation? We will see that for the Euclidean Steiner Tree problem and the Metric Steiner Tree problem, an MST is a good approximation of the optimal Steiner Tree. However, for the General Steiner Tree problem, an MST is *not* a good approximation.

First, consider this example of the Euclidean Steiner Tree problem:

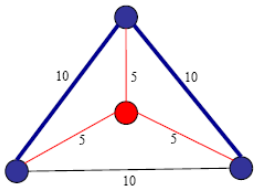


On the left picture, we are given an equilateral triangle with side-length 1. The minimum spanning tree for this triangle includes any two of the edges, and hence has length 2 (as in the middle). However, the minimum Steiner tree for the triangle adds one Steiner point: The point in the middle of the triangle (as on the right picture). With this extra Steiner point, now the total cost is $\sqrt{(3)}$. (You can calculate this based on the fact that the angle in the middle is 120 degrees.) Thus, a minimum spanning tree is, at best, a $2/\sqrt{(3)}$-approximation (or 1.15-approximation) of the optimal Euclidean Steiner tree. Is it always at least a $2/\sqrt{(3)}$-approximation of optimal? That remains an open conjecture. As of today, no one knows of any example that is worse than the triangle.

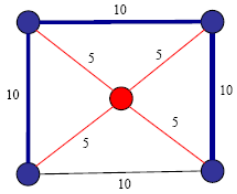
Now consider the Metric Steiner Tree problem. Obviously, a minimum spanning tree still cannot be better than a $2/\sqrt{(3)}$-approximation, since again we could consider the triangle with a single Steiner point in the middle. Now, however, we can construct a better example.

[7] Which actually makes the STEINER-TREE problem 'easier'...

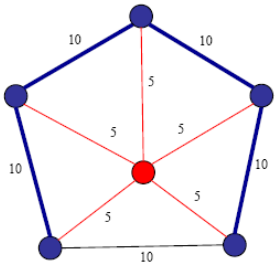
[8] So technically, stopping at CS2010/CS2020 level is not really enough to deal with these kinds of problems :O...



MST: 20
Steiner Tree: 15
Ratio: 1.33



MST: 30
Steiner Tree: 20
Ratio: 1.5



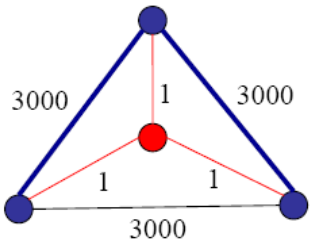
MST: 40
Steiner Tree: 25
Ratio: 1.8

Assume in this example that the distance between every pair of required vertices is 10. (In the picture, only outer edges between adjacent vertices are drawn, but the distance metric must define the distance between every pair of vertices.)

Here, we have three cases: a triangle, a square, and a pentagon (and we can generalize this further as these cases are Wheel Graphs). Each n-gon has n required vertices, where each pair of vertices is connected by edges of length 10. Each has a single Steiner vertex in the middle. There is an edge from each required vertex to the Steiner vertex in the middle, and each of these edges has length 5. Notice that this is not Euclidean: For example, the diagonal of a square would have length $10\sqrt{(2)}$, but here has only length 10. However, all the distance satisfy the triangle inequality, so the distances are metric. (Verify that this is true!)

For an n-gon where all distances are 10, a minimum spanning tree has exactly $n-1$ edges, and hence has cost $10(n-1)$. On the other hand, the minimum Steiner tree uses the Steiner vertex in the middle and has n edges, one connecting the Steiner vertex to each required vertex. The total cost of the Steiner tree is $5n$. Thus, the minimum spanning tree is no better than a $2(n-1)/n$-approximation of the optimal spanning tree. As n gets large, this approaches a 2-approximation. We will later show that a minimum spanning tree is (at least) a 2-approximation of optimal.

Finally, it should now be clear that the minimum spanning tree is not a good approximation for General Steiner Tree problem. Consider this example:



In this case, just considering the three outer vertices as required, the minimum spanning tree has cost 6 000. However, the minimum Steiner tree, including the Steiner vertex in the middle, has cost 3. Moreover, this ratio can be made as large as desired. Notice that this depends critically on the fact that the triangle inequality does not hold. That is, the distances here are not a metric.

3 METRIC-STEINER-TREE Approximation Algorithm

We are now going to show that, in a metric space, a minimum spanning tree is a good approximation of a Steiner tree. This yields a simple algorithm for finding an approximately optimal Steiner tree, in the metric case: Simply ignore the Steiner points and return the minimum spanning tree!

Theorem 6 *For a set of required vertices R, a set of Steiner vertices S, and a metric distance function d, the minimum spanning tree of (R, d) is a 2-approximation of the optimal Steiner Tree for (R, S, d).*

Proof Throughout the proof, we will use as an example the graph in Figure 4. Here we have a graph with six required vertices (i.e., the blue ones) and two Steiner vertices (i.e., the red ones). All the edges drawn have distance 1; all the other edges (not drawn) have distance 2.

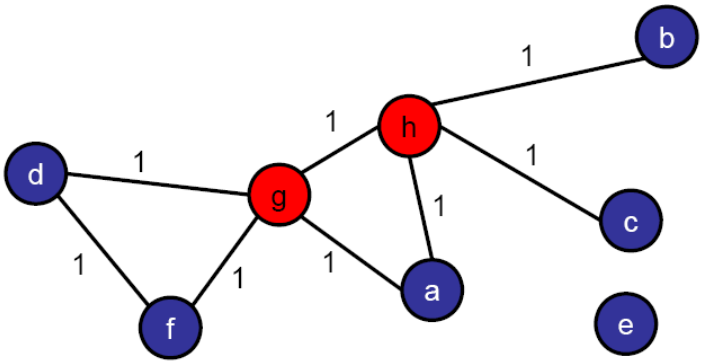


Figure 4: Example for showing that a minimum spanning tree is a 2-approximation of the optimal Steiner tree, if the distances are a metric. Assume that all edges *not* drawn have distance 2. Verify that these distances satisfy the triangle inquality.

Let $T = (V, E)$ be the optimal Steiner tree, for some $V \subseteq R \cup S$. For our example, we have drawn this optimal Steiner tree in Figure 5.

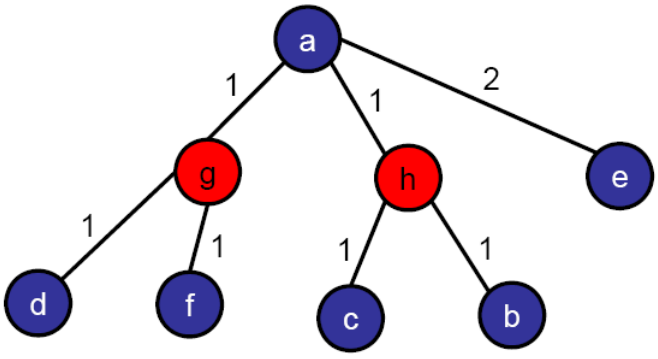


Figure 5: Here we have drawn the optimal Steiner tree T of graph shown in Figure 4.

The first step of the proof is to transform the tree T into a cycle C of at most twice the cost. We accomplish this by performing a DFS of the tree T, adding each edge to the cycle as it is traversed (both down and up). Notice that the cycle begins and ends at the root of the tree, and each edge appears in the cycle exactly twice: Once traversing down from a parent to a child, and once traversing up from a child to a parent. (Also notice that this cycle visits some vertices multiple times: A vertex with x children in the tree will appear $2(x+1)$ times ($2x$ for root vertex), twice for each of the $x+1$ (x for root vertex) adjacent edges.)

In our example, the cycle C is as follows:

$$(a,g) \to (g,d) \to (d,g) \to (g,f) \to (f,g) \to (g,a) \to (a,h) \to$$
$$(h,c) \to (c,h) \to (h,b) \to (b,h) \to (h,a) \to (a,e) \to (e,a)$$

Notice that this cycle has 14 edges, whereas the original tree has 7 edges and 8 vertices.

We have already defined the $cost(T) = \sum_{e \in E} d(e)$. Similarly, the cost of cycle C is defined as $cost(C) = \sum_{e \in C} d(e)$. Since every edge in the tree T appears exactly twice in the cycle C, we know that $cost(C) = 2 \times cost(T)$. In our example, we see that the cost of the original tree T is 8 and the cost of the cycle C is 16.

The cycle C contains both required vertices in R and Steiner vertices in S. We now want to remove all the Steiner vertices from C, without increasing the cost of the cycle C. Find any two consecutive edges in the cycle (u,v) and (v,w) where the intermediate vertex v is a Steiner vertex. (At this point, it does not matter whether u and w are required or Steiner). Replace the two edges (u,v) and (v,w) with a single edge (u,w), thus deleting the Steiner vertex v. We refer to this procedure as "short-cutting v." Notice that this replacement does not increase the cost of the cycle C, since $d(u,w) \leq d(u,v) + d(v,w)$—by the triangle inequality. *(Hint: Always pay attention to where we use the assumption; here is where the proof depends on the triangle inequality.)* Continue short-cutting Steiner vertices until all the Steiner vertices have been deleted from C.

In our example, the Steiner vertices to be removed are g and h. Thus, we update the cycle C as follows:

$$(a,d) \to (d,f) \to (f,a) \to (a,c) \to (c,b) \to (b,a) \to (a,e) \to (e,a)$$

This revised cycle now has cost[9]: $2+1+2+2+2+2+2+2 = 15$, which is no greater than the original cost of the cycle C (which was 16).

We now have a cycle C where $cost(C) \leq 2 \times cost(T)$. (Notice that the cost may have decreased during the short-cutting process, but it could not have increased.) Moreover, this cycle visits every required vertex in R at least twice (and there are no Steiner vertices in the cycle). The next step is to remove duplicates. Beginning at the root, traverse the cycle labeling each vertex: The first time a vertex is visited, mark it *new*; mark every other instances of that vertex in the cycle as *old*. (But mark the root vertex visited at the beginning and end as *new*.) As with Steiner vertices, we now short-cut all the *old* vertices: Find two edges (u,v) and (v,w) where v is an old vertex and replace those two edges in the cycle with a single edge (u,w). As before, this does not increase the cost of the cycle.

In our example, the cycle C visits the vertices in the following order:

$$\mathbf{a} \to \mathbf{d} \to \mathbf{f} \to \underline{a} \to \mathbf{c} \to \mathbf{b} \to \underline{a} \to \mathbf{e} \to \mathbf{a}$$

The new vertices are colored with blue color, including vertex a at the beginning and the end. The old vertices are colored with red color and underlined. This leaves two instances of vertex a to be shortcut. Once we shortcut past the old vertices, we are left with the following cycle C:

$$(a,d) \to (d,f) \to (f,c) \to (c,b) \to (b,e) \to (e,a)$$

[9] When you compute the new cost, refer to the cost in original Figure 4. For example, there is an edge with weight 1 that connects vertex d and f in Figure 4.

This cycle has cost[10]: $2+1+2+2+2+2=11$, which is no greater than the original cost of the cycle (which was 16). This revised cycle C is depicted in Figure 6.

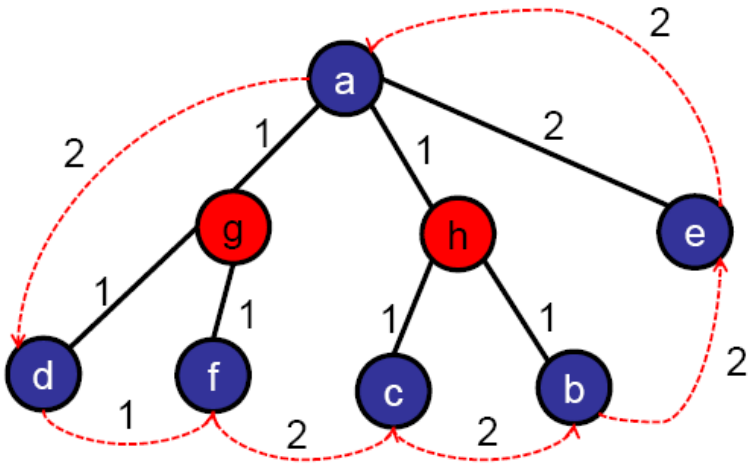


Figure 6: Here we have drawn the cycle C after the Steiner vertices have been short-cut and the repeated vertices have been deleted.

We now have a cycle C where $cost(C) \leq 2 \times cost(T)$, and each required vertex in R appears exactly once. Finally, remove any one arbitrary edge from C. (Again, this cannot increase the cost of C.) At this point, C is a path which traverses each vertex in the graph exactly once. That is, C is a spanning tree with cost at most $2 \times cost(T)$. In our example, the spanning tree C (where one arbitrary edge (e.g. the last edge) of the cycle has been deleted is):

$$(a,d) \rightarrow (d,f) \rightarrow (f,c) \rightarrow (c,b) \rightarrow (b,e)$$

This is a spanning tree with cost 9.

Let M be the minimum spanning tree of the required vertices R. Since M is the spanning tree of minimum cost, clearly $cost(M) \leq cost(C) \leq 2 \times cost(T)$. From this we conclude that M is a 2-approximation for the minimum cost Steiner tree of $R \cup S$. □

To summarize, the proof goes through the following steps:

1. Begin with the optimal Steiner Tree T.
2. Use a DFS traversal to generate a cycle of twice the cost.
3. Eliminate the Steiner vertices and repeated required vertices, without increasing the cost. (Use the assumption that d satisfies the triangle inequality.)
4. Remove on edge from the cycle, yielding a spanning tree of cost at most twice the cost of T.
5. Observe that the minimum spanning tree can have cost no greater than the constructed spanning tree, and hence no greater than twice the cost of T.

This implies that the minimum spanning tree has cost at most twice the cost of T, the optimal Steiner Tree.

[10] Again, when you compute the new cost, refer to the cost in original Figure 4. For example, there are edges with weight 2 that connects vertex f and c, vertex b and e, and vertex e and a in Figure 4 (all not drawn).

4 GENERAL-STEINER-TREE Approximation Algorithm

In general, a minimum spanning tree is *not* a good approximation for the GENERAL-STEINER-TREE problem. Here we want to show how to find a good approximation in this case. Instead of developing an algorithm from scratch, we are going to use a reduction. Part of the goal is to demonstrate how to use reductions when we are talking about approximation algorithms.

The typical process, with a reduction, is something like as follows:

- Begin with an instance of the GENERAL-STEINER-TREE problem.
- Via the reduction, construct a new instance of the METRIC-STEINER-TREE problem.
- Solve the METRIC-STEINER-TREE problem using our existing algorithm.
- Convert the solution to the METRIC-STEINER-TREE instance back to a solution for the GENERAL-STEINER-TREE problem.

The key to the analysis would typically be a lemma that says something like, "If we have an algorithm for finding an optimal solution to the METRIC-STEINER-TREE problem, then our construction/conversion process yields an optimal solution to the GENERAL-STEINER-TREE problem."

With approximation algorithms, however, you have to be a little more careful. Normally, it is sufficient to show that an optimal solution to the translated problem yields an optimal solution to the original problem. However, we are looking here at approximation algorithms. Hence we need to show that, even though we are only finding an approximate solution to the METRIC-STEINER-TREE problem, that still yields an approximate solution to the GENERAL-STEINER-TREE problem. A reduction that preserves approximation ratios is known as a ***gap-preserving*** reduction.

Assume we are given a graph $G = (V, E)$ and non-negative edge weights $w : E \to \mathbb{R}^{\geq 0}$. The vertices V are divided into required vertices R and Steiner vertices S. Our goal is to find a minimum cost Steiner Tree. There are no restrictions on the weights w (i.e., they do not necessarily satisfy the triangle inequality).

Defining a metric. In order to perform the reduction, we need to construct an instance of the METRIC-STEINER-TREE problem. In particular, we need to define a metric. The specific distance metric we are going to define is known as the ***metric completion*** of G.

Definition 7 *Given a graph $G = (V, E)$ and non-negative edge weights w, we define the **metric completion** of G to the be the distance function $d : V \times V \to \mathbb{R}$ constructed as follows: For every $u, v \in V$, define $d(u, v)$ to be the distance of the* shortest path *from u to v in G with respect to the weight function w.*

Notice that the metric completion d provides distances between every pair of vertices, not just the edges in E. Also notice that, computationally, d is relatively easy to calculate, e.g., via an All-Pairs-Shortest-Paths algorithm such as Floyd-Warshall which runs in $O(V^3)$ time[11]. Critically, d is a metric:

Lemma 8 *Given a graph $G = (V, E)$, the metric completion d is a metric with respect to V.*

Proof Since all the edge weights are non-negative, clearly $d(u, v) \geq 0$ for all $u, v \in V$. Similarly, $d(u, u) = 0$, by definition. Since the original graph is undirected, the shortest path from u to v is also the shortest path from v to u; hence $d(u, v) = d(v, u)$.

[11] Compared to the NP-hardness of the original STEINER-TREE problem, running an $O(V^3)$ algorithm for its approximation algorithm is generally viewed as OK.

The most interesting property is the triangle inequality. We need to show that for all vertices $u, v, w \in V$, the distances $d(u, v) + d(v, w) \geq d(u, w)$. Assume, for the sake of contradiction, that this inequality does not hold, i.e., $d(u, v)+d(v, w) < d(u, w)$. Let $P_{u,v}$ be the shortest path from u to v in G (with respect to the weight function w), and let $P_{v,w}$ be the shortest path from v to w (with respect to the weight function w). Consider the path $P_{u,v} + P_{v,w}$, which is a path from u to w of length $d(u, v) + d(v, w)$. This path is of length less than $d(u, w)$. But that is a contradiction, since $d(u, w)$ was defined to be the length of the shortest path from u to w.

From this, we conclude that d satisfies the triangle inequality, and hence is a metric. □

In the following, we will sometimes be calculating the cost with respect to the metric completion d, and sometimes with respect to the edge weights w. To be clear, we will use $cost_d$ to refer to the former and $cost_w$ to refer to the latter. Similarly, we will refer to graph G^g when talking about the GENERAL-STEINER-TREE problem, and G^m when talking about the METRIC-STEINER-TREE problem.

Converting from General to Metric. We can now reduce the original instance of the GENERAL-STEINER-TREE problem to an instance of the METRIC-STEINER-TREE problem. Assume we have an algorithm A that finds an α-approximate minimum cost METRIC-STEINER-TREE.

- Given a graph $G^g = (V, E^g)$ with requires vertices R, Steiner vertices S, and a non-negative edge weight function w:
- Let d be the metric completion of G^g.
- Consider the METRIC-STEINER-TREE problem: (R, S, d).
- Let $T^m = A(R, S, d)$ be the α-approximate minimum cost Metric Steiner tree.

At this point, we have converted our GENERAL-STEINER-TREE problem into a METRIC-STEINER-TREE problem, and solved it using our existing approximation algorithm.

Converting from Metric back to General. We are not yet done, however, since T^m is defined in terms of edges that may not exist in G^g, and in terms of a different set of costs. We need to convert the tree T^m back into a tree in G^g.

For every edge $e = (u, v)$ in the tree T^m, let p_e be the shortest path in G^g from u to v. Let $P = \bigcup_{e \in T} p_e$, i.e., the set of all paths that make up the tree T^m. Notice that some of these paths may overlap.

Define the cost of a path in G^g (with respect to w) to be the sum of the costs of the edge weights, i.e., $cost_w(p) = \sum_{e \in p} w(e)$. Notice that the cost of a path in G^g is with respect to edge weights, while the cost of the tree T^m is with respect to the metric completion d. (This difference is because we are converting back from the metric to the general problem.) If $e = (u, v)$ is an edge in the tree T^m, then $cost_w(p_e) = d(u, v)$.

Consider all the paths $p_e \in P$, i.e., for every edge e in the tree T^m: add every edge that appears in any path $p_e \in P$ to a set E'. Notice that the graph $G' = (V, E')$ is connected, since the original tree T^m was connected and if there was an edge (u, v) in T^m then there is a path connecting u to v in E'. Also, notice that the cost of all the edges in E' (with respect to w) is no greater than the cost of all the edges in T^m (with respect to d), since each path p_e costs the same amount as the edge e in T^m.

Finally, we need to remove any cycles from the graph (V, E') so that we have a spanning tree. Let T^g be the minimum spanning of (V, E'). The tree T^g in the graph $G^g = (V, E^g)$ with edge weights w has cost no greater than the tree T^m with edge weights d.

Analysis. To analyze this, we need to prove two key lemmas (proof omitted). Notice that you need to prove two things: You need to relate the solution in the metric version to OPT in the general version, and you also need to relate the final solution in the general version to the solution found in the metric version.

Lemma 9 *Let OPT^g be the optimal minimum cost Steiner tree for G^g. Then $cost_d(T^m) \leq \alpha \cdot cost_w(OPT^g)$.*

Lemma 10 *Let T^g be the Steiner tree calculated by converting T^m back to graph G^g. Then $cost_w(T^g) \leq cost_d(T^m)$*

Putting these lemmas together, we get our final result:

Theorem 11 *Given an α-approximation algorithm for* METRIC-STEINER-TREE *problem, we can find an α-approximation for a* GENERAL-STEINER-TREE *problem.*

Proof Assume we have a graph $G^g = (V, E^g)$ with required vertices R, Steiner vertices S, and a non-negative edge weight function w. Let T^m be an α-approximate Steiner tree for (R, S, d), where d is the metric completion of G^g. Let T^g be the spanning tree constructed above by converting the edges in T^m into paths in $G^g = (V, E^g)$ and removing cycles. We will argue that T^g is an α-approximation of the minimum cost spanning tree for G.

First, we have shown that $cost_d(T^m) \leq \alpha \cdot cost_w(OPT^g)$. Second, we have shown that $cost_w(T^g) \leq cost_d(T^m)$. Putting the two pieces together, we conclude that $cost_w(T^g) \leq \alpha \cdot cost_w(OPT^g)$, and hence T^g is an α-approximation for the minimum cost Steiner tree for G with respect to w. □

References

[1] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

Index

CS4234: Optimisation Algorithms Lecture 4

TRAVELLING-SALESMAN-PROBLEM (4 variants)

V1.0: Seth Gilbert, V1.1: Steven Halim *August 30, 2016*

Abstract

The goal of the TRAVELLING-SALESMAN-PROBLEM is to find a tour that connects all the vertices in a graph at a minimal cost. There are several variants, depending on whether repeated visits are allowed, and depending on whether the distances satisfy a metric. We discuss the relationship between these variants, and give a simple 2-approximation algorithm. We then develop a more involved 1.5-approximation algorithm that relates the TSP to Eulerian tours.

1 The TRAVELLING-SALESMAN-PROBLEM

Today we consider the TRAVELLING-SALESMAN-PROBLEM[1], often abbreviated TSP. The "TSP"[2] is perhaps one of the most famous[3] (and most studied) Combinatorial Optimization Problem (COP).

1.1 Problem Definition

Given a set of cities (i.e., points, or vertices), the goal of the TSP is to find a minimum cost circuit (or cycle, or tour) that visits all the points. More formally, the problem is stated as follows:

Definition 1 *Given a set V of n points and a distance function $d : V \times V \to \mathbb{R}$, find a cycle C of minimum cost that contains all the points in V. The cost of a cycle $C = (e_1, e_2, \ldots, e_n)$ is defined to be $\sum_{e \in C} d(e)$, and we assume that the distance function is non-negative (i.e., $d(x, y) \geq 0$).*

Notice that, unlike the STEINER-TREE problem, there are no Steiner vertices: You have to visit *every* city. The cities/points may be in some geometric space (e.g., the Euclidean plane), or they may not. If the points are in the Euclidean plane, then it is natural to define d to be the Euclidean distance. Otherwise, d can be arbitrary, i.e. non-metric. See Figure 1 for an example of one instance of the TSP.

1.2 Variants

As with the STEINER-TREE problem, there are several variants:

- *Metric vs. General:* In the *metric* version of the TSP, the distance function d is a metric, i.e., it satisfies *the triangle inequality*[4]. In the general version, d can assign any arbitrary weight to an edge.
- *Repeated-Visits vs. No-Repeats:* The goal of the TSP is to find a cycle that visits every vertex. Can the cycle contain repeated vertices (or does it have to be a simple cycle)? In the version with No-Repeats, the TSP cycle must visit each vertex *exactly* once. In the version with repeats, it is acceptable to visit each vertex more than once (if that results in a shorter route).

[1] The word 'travelling' (2 'l's) is in British and the word 'traveling' (1 'l') is in American.
[2] As the abbreviation "TSP" already includes the word 'Problem', we do not say "TSP problem" anymore.
[3] We even have a movie for this, see `http://www.travellingsalesmanmovie.com/`.
[4] In the previous lecture about Steiner Tree, we have highlighted that metric version is somewhat easier than the general version.

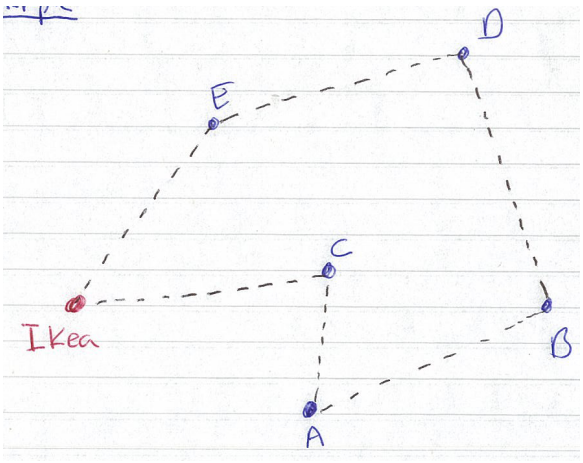


Figure 1: Example of the TSP. Here, there are six locations. The problem is to find the shortest delivery circuit starting from Ikea and visiting each of the five houses, and then returning to Ikea.

We will summarize these four variants as follows:

	Repeats	**No-Repeats**
Metric	M-R-TSP	M-NR-TSP
General	G-R-TSP	G-NR-TSP

1.3 NP-hard

All the variants of the TSP are NP-hard [2].

For the No-Repeats variants of the problem, this is easily seen by reduction from the HAMILTONIAN-CYCLE problem: a Hamiltonian cycle is a cycle that contains every vertex exactly once; the goal of the HAMILTONIAN-CYCLE problem is to determine whether a given graph has a Hamiltonian cycle. Deciding whether a graph has a Hamiltonian cycle is NP-hard (and this can be shown by reduction from 3-SAT). Clearly, if we can solve the TSP, then we can solve the HAMILTONIAN-CYCLE problem (by setting the distances properly). There are similar reductions for the other variants.

The general No-Repeats version of TSP (the G-NR-TSP) is NP-hard even to approximate! So we will temporarily ignore this variant in this Lecture.

Even so, we are going to see how to (easily!) approximate the other three variants. In almost all common real-world cases (e.g., where the distance function satisfies the triangle inequality, or where repeats are acceptable), there are good approximation algorithms.

2 Equivalence

The first thing we are going to see is that the three other variants are equivalent. That is, M-R-TSP, M-NR-TSP, and G-R-TSP are all equivalent: If we have a c-approximation algorithm for any one of these variants, then we also can

construct a c-approximation algorithm for the other two.

We first focus on the issue of repeats: As long as the distance function is a metric, it does not matter whether or not we allow repeats. In many ways, this should be unsurprising (given our discussion of STEINER-TREE last week): If we have repeated vertices on the cycle, we can always "short-cut" past them producing a cycle with no repeats at the same cost.

Lemma 2 *There exists a c-approximation algorithm for* M-NR-TSP *if and only if there exists a c-approximation algorithm for* M-R-TSP.

Proof The proof has three claims. First, we show that both the version with repeats and no-repeats have the same optimal cost. We then show how to transform the solution from the NR variant into the R variant (which is trivial) and from the R variant into the NR variant (which involves skipping repeats).

Claim 1. Let (V, d) be an input for Metric TSP (for both NR or R variants). Let $OPT(R)$ be the minimum cost TSP cycle overall (i.e., with repeats), and let $OPT(NR)$ be the minimum cost TSP-cycle with no repeats. Then $OPT(R) = OPT(NR)$.

To show this, first we observe that any cycle with no repetitions is also a legal TSP-cycle overall (i.e., for the version that allows repeats). And so it follows immediately[5] that $OPT(R) \leq OPT(NR)$ — **Part A**.

Next, let C be a cycle with repeats, for example:

$$C = 0, 1, 2, 3, 2, 4(, 0)$$

We can now create cycle C' by skipping the repeated vertices. For example:

$$C' = 0, 1, 2, 3, 4(, 0)$$

By the triangle inequality, we know that $d(C') \leq d(C)$. For example, in constructing C' from C, we replace $(3, 2, 4)$ with $(3, 4)$; we know that $d(3, 4) \leq d(3, 2) + d(2, 5)$, by the triangle inequality. We can repeat this inductively for every skipped vertex.

If we assume that cycle $C = OPT(R)$, then we conclude that $OPT(NR) = d(C') \leq d(C) = OPT(R)$ — **Part B**.

Putting together the two inequalities in **Part A** and **Part B**, we have shown that $OPT(NR) \leq OPT(R)$ and $OPT(R) \leq OPT(NR)$, which yields our desired conclusion that $OPT(R) = OPT(NR)$.

Claim 2. If A is a c-approximation algorithm for M-NR-TSP, then A is a c-approximation algorithm for M-R-TSP.

Assume algorithm A outputs cycle C. By assumption, C has no repeats, and $d(C) \leq c \cdot OPT(NR)$. As $OPT(R) = OPT(NR)$ from **Claim 1**, we have = $d(C) \leq c \cdot OPT(R)$. Thus we see that C is also a valid solution for M-R-TSP and provides a c-approximation.

Claim 3. If A is a c-approximation algorithm for M-R-TSP, then we can construct an algorithm A' which is a c-approximation for M-NR-TSP.

Specifically, construct algorithm A' as follows: Run algorithm A to get cycle C (which may have repeats); then construct C' from C by skipping any repeated vertex. There are two things we have to show.

First, the cost of cycle C is a good approximation of the optimal for M-NR-OPT, i.e. $d(C) \leq c \cdot OPT(R)$. As $OPT(R) = OPT(NR)$ from **Claim 1**, we have = $d(C) \leq c \cdot OPT(NR)$.

[5]Remember that in Lecture 2 about relaxed ILP, when we minimize over a larger space of possible solutions, we always get a solution that is at least as good as the best solution of the smaller subspace.

Second, the cycle we have constructed in A' has cost bounded by the cost of C, i.e. $d(C') \leq d(C)$ due to the triangle inequality, similar as in **Claim 1**.

Putting these two facts together, we see that the algorithm A' that produces C' is a c-approximation algorithm for M-NR-TSP as $d(C') \leq c \cdot OPT(NR)$. □

We can show exactly the same type of equivalence for the general and metric versions, as long as we allow repeats. In this case, we cannot create "shortcuts" because the triangle inequality does not hold; however, we do not need to create shortcuts because repeats are allowed.

Lemma 3 *There exists a c-approximation algorithm for* M-R-TSP *if and only if there exists a c-approximation algorithm for* G-R-TSP.

Proof In this case, the proof contains two claims: One transforming G into M (which is trivial), and the other M into G (need some work).

Claim 1. If A is a c-approximation algorithm for G-R-TSP, then A is a c-approximation algorithm for M-R-TSP.

This claim is immediately and obviously true. The two problems are identical, except for the restriction that the input to M-R-TSP must be a metric. However, if a given input *is* a metric, then it clearly satisfies the requirements of G-R-TSP and we can just execute algorithm A on that instance. Moreover, the optimal solution will be the same for G-R-TSP and M-R-TSP (since they are optimizing over the exact same set of possible cycles).

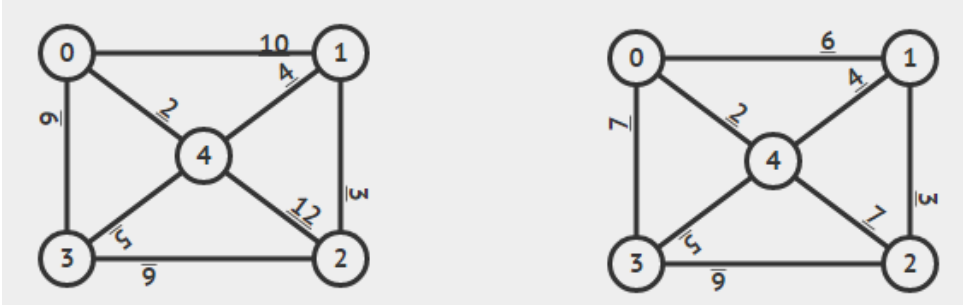


Figure 2: Example of constructing A', reducing G-R-TSP to M-R-TSP, e.g. $C = \{0, 4, 1, 2, 3, 0\}$, then $C' = \{0, 4, 1, 2, \underline{3, 4, 0}\}$.

Claim 2. If A is a c-approximation algorithm for M-R-TSP, then we can construct algorithm A' is a c-approximation algorithm for G-R-TSP.

Let (V, d_g) be the input to algorithm A', i.e., an input for G-R-TSP. We construct algorithm A' as follows.

First, we need to construct a distance metric d_m. For every pair of vertices (u, v), define $d_m(u, v)$ to be the shortest path from u to v according to the distances d_g. (Recall that d_g is not a metric, i.e., does not necessarily satisfy the triangle inequality.) We can find d_m by construct the complete graph on V, assign the weight of edge (u, v) to be $d_g(u, v)$, and executing an All-Pairs-Shortest-Paths algorithm, e.g. $O(V^3)$ Floyd Warshall's algorithm. You will notice that d_m is now a distance metric, i.e., it satisfies the triangle inequality. Recall that d_m is called the metric completion of the graph[6].

[6]We have done similar thing in Lecture 3b: Steiner Tree, General versus Metric.

Second, execute algorithm A on (V, d_m), producing cycle C. We then need to construct a new cycle C' out of C. For each edge (u, v), we add the shortest path from u to v according to d_g to our output cycle C'. Notice this produces a cycle C' that may contain may repeated vertices[7].

We now argue that the result is a good approximation of the $OPT(V, d_g)$. First, observe that $d(C') = d(C)$. Each edge in C got replaced by a path in C' with the exact same cost, and hence we end up with the same cost cycle.

Second, we know by assumption that $d(C) \leq c \cdot OPT(V, d_m)$, by the assumption that A is a c-approximation algorithm.

Finally, we argue that $OPT(V, d_m) \leq OPT(V, d_g)$. In particular, if R is the optimal cycle for (V, d_g), then R is also a cycle in (V, d_m). Moreover, for every pair (u, v), we know that $d_m(u, v) \leq d_g(u, v)$ (because d_m is defined as the shortest path). Thus $d_m(R) \leq d_g(R)$. Since the optimal cycle with metric d_m has to be at least as good as $d_m(R)$, we conclude that $OPT(V, d_m) = d_m(R) \leq d_g(R) = OPT(V, d_g)$.

Putting the pieces together, we conclude that $d(C') = d(C) \leq c \cdot OPT(V, d_m) \leq c \cdot OPT(V, d_g)$, and hence algorithm A' is a c-approximation algorithm for G-R-TSP. □

3 2-Approximation Algorithm

We now present a 2-approximation algorithm for G-R-TSP. We already know that this will also yield a 2-approximation algorithm for the other two variants. Assume that the input is (V, d) where V is a set of points and d is a distance function (but not necessarily a metric). Consider the following algorithm:

1. Construct the complete graph $G = (V, E)$ with weights w where E contains every pair $(u, v) \in V \times V$ and $w(u, v) = d(u, v)$.
2. Let T be the Minimum Spanning Tree of G.
3. Let C be the cycle constructed by performing a Depth-First Search of T.

To analyze this algorithm, as usual[8], we start with the optimal solution and work backwards. Let C^* be the optimal (minimum cost) TSP cycle for input (V, d) and let E^* be the edges in C^*. Notice that the graph $G^* = (V, E^*)$ is connected, because C^* is a cycle that includes every point. Let T^* be the Minimum Spanning Tree of $G = (V, E^*)$. At this point, we know that:

$$d(T^*) \leq d(C^*) = OPT$$

Since the tree T is a minimum spanning tree of $G = (V, E)$ and $E^* \subseteq E$, we know that:

$$d(T) \leq d(T^*)$$

Finally, since C is construct by a Depth-First Search traversal of T, we know that C includes each edge in T exactly twice, and so:

$$d(C) = 2 \times d(T)$$

Putting these inequalities together, we conclude that:

$$d(C) = 2 \times d(T) \leq 2 \times d(T^*) \leq 2 \times d(C^*) \leq 2 \times OPT$$

That is, the cost of C is at most twice the cost of OPT, and hence the algorithm is a 2-approximation algorithm.

[7] Again, this is the same as with Lecture 3b: Steiner Tree, General versus Metric.
[8] We have done similar thing in Lecture 3b: Steiner Tree, General versus Metric.

4 Eulerian Cycles

We now want to develop a better approximation algorithm, one that will have a better approximation ratio than 2. In order to do that, we are going to need a few additional tools. Let's begin with a famous problem, known as the *Bridges of Konigsberg*.

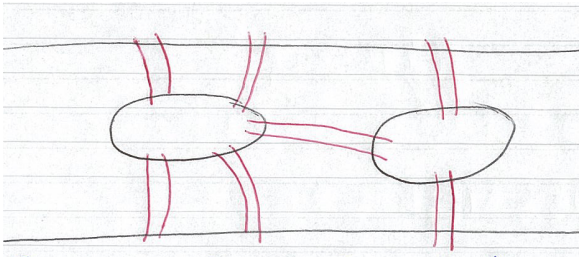


Figure 3: Illustration of the Bridges of Konigsberg.

In Konigsberg, there is a river with two islands. Historically, these two islands were connected by seven bridges. The bridges were beautiful and famous, and so the question that everyone wanted to answer was whether it was possible to cross each bridge exactly once—and end up back where you started. With a little bit of thought, it becomes obvious that you cannot. But how would you prove it? Euler solved this problem, and his solution is often seen as marking the beginning of graph theory as a mathematical field. Notably, he realized that the problem could be represented as a graph:

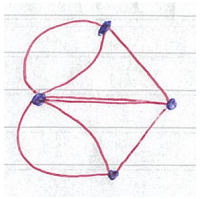


Figure 4: Illustration of the Bridges of Konigsberg as a graph.

Once the problem was represented abstractly as a graph, we can apply more powerful techniques to solve the problem.

For the purpose of this section, we will be talking about *multigraphs* rather than simple graphs. A *multigraph* is a graph $G = (V, E)$ where E is a multiset, rather than a set, of edges. This means that each edge can appear in the graph more than once. For example, a given edge (u, v) might appear in graph G four times.

Definition 4 *A **Eulerian Cycle** in a multigraph G is a cycle that crosses each edge exactly once.*

Notice that, unlike a TSP-cycle, an Eulerian Cycle focuses on edges, rather than vertices. The goal is to cross each edge exactly once—though that may mean visiting a given vertex many times. Clearly the problem of the Bridges of Konigsberg is simply asking whether there is an Eulerian Cycle in the associated graph.

The key claim is to detect when a graph has an Eulerian Cycle. Amazingly, there is a very simple characterization:

Lemma 5 *A multigraph $G = (V, E)$ has an Eulerian Cycle if and only if:*

- *It is connected (except for vertices of degree 0).*

- *Every vertex has even degree.*

Proof Notice that one direction of this claim is easy. Assume graph G has an Eulerian Cycle. In that case, since the cycle cross every edge, clearly the cycle visits every vertex (except those vertices with degree zero) and hence graph is connected—except for vertices with degree zero. In addition, the cycle enters and exits each vertex the same number of times. For example, if the cycle enters a vertex v some k times, then it also exits vertex v the same k times. From this we conclude that every vertex has even degree. (In our example, vertex v has degree $2k$, which is even.)

The surprising result is the opposite direction. Assume graph G is connected (except for vertices of degree 0) and that every vertex has even degree. We need to construct a Eulerian Cycle. We will proceed by induction on the number of edges, proving a stronger invariant as we go: If every vertex in G has even degree, then each connected component with > 1 vertices has an Eulerian Cycle.

We will begin (in the base case) with an empty graph containing only the vertices, and add one edge at a time back to the graph. At every step, we will construct an Eulerian Cycle for each connected component.

Base case: In this case, $m = 0$, i.e., there are no edges, and hence trivially the claim holds.

Inductive step: Assume we have m edges. Let $V_1, V_2, \ldots, V_k$ be the connected components of G that contain > 1 vertices. We consider two cases:

Case 1: $k \geq 2$: There are at least two connected components. Each connected component has $< m$ edges. Hence, by induction, each component has an Eulerian Cycle.

Case 2: $k = 1$: V_1 is the only connected component with > 1 vertices. Recall that all the vertices in V_1 have even degree, and every vertex note in V_1 has degree zero.

First, we claim that there must be a cycle in V_1. Let $n_1 = |V_1|$. If there are no cycles in V_1, then V_1 is a tree and contains exactly $n_1 - 1$ edges. However, the degree of every vertex in V_1 is at least 2 (since the degree is even and > 0). Thus there are at least $2V_1$ endpoints of edges in V_1, and hence at least V_1 edges in V_1. (Notice in general that if $d(v)$ is the degree of vertex v, then a graph has exactly $\sum_{u \in V} d(u)/2$ edges.) This contradicts the claim that V_1 is a tree.

Let C be any cycle in V_1. Let G' be the the graph G where the edges in C are removed. Let $V'_1, V'_2, \ldots, V'_{k'}$ be the new connected components of G'. (By removing the cycle C, we may have split V_1 into several distinct connected components; or V_1 may remain connected.) Each of these components now has $< m$ edges. Hence, by induction, each of these components has an Eulerian Cycle. Let $C_1, C_2, \ldots, C_k$ be the Eulerian Cycles for connected components $V'_1, V'_2, \ldots, V'_{k'}$.

It remains to stitch these Eulerian Cycles back together. Notice that these connected components are all connected by the initial cycle C. That is, if we add back the cycle C, we will have a single connected component. Here's how we do that:

Let $C = (v_1, v_2, \ldots, v_\ell)$. Begin at vertex v_1. We know that vertex v_1 is part of one of the Eulerian Cycles V_j. Follow the Eulerian Cycle V_j until it returns to vertex v_1. Then proceed to vertex v_2. If v_2 is part of a new Eulerian Cycle (i.e., not V_j), then follow the new Eulerian Cycle until it returns to v_2. Otherwise move on to v_3. And so on. At each step, we move to the next vertex v_i in the cycle C. If that vertex v_i is part of a new Eulerian Cycle V_i that has not yet been traversed, then we traverse the cycle until it returns to v_i. Then we continue to v_{i+1}. This continues until we return to vertex v_1.

Notice that the new cycle we have constructed traverses every edge in the graph G, as it traverses all the edges in the connected components $V'_1, \ldots, V'_{k'}$ as well as all the edges in C. Hence we have constructed an Eulerian Cycle for G. □

The other interesting fact is that this Eulerian Cycle can be easily constructed in polynomial time, exactly by following the steps of the algorithm:

1. Find a cycle in G (e.g., via DFS). If there is none, then skip the next step.
2. Remove the cycle.
3. Find the connected components in the residual graph.
4. Recurse: Find an Eulerian Cycle in each connected component.
5. If a cycle was found in Step 1, paste the cycles back together.

This can readily be implemented in $O(m^2)$ time. Can you optimize it further? (see [3]).

5 A Better TSP Approximation

We consider the M-R-TSP variant again, and as before, if we can have a c-approximation algorithm for this variant, then we can also approximate the other two variants: M-NR-TSP and G-R-TSP. Assume (V, d) is the input to our problem.

5.1 Basic Idea

Imagine that we built a multigraph $G = (V, E)$ that contained an Eulerian Cycle. Then that cycle would give us a feasible solution to the TSP problem. Conversely, if had a solution to the TSP problem, we could build a multigraph consisting of only the edges in the TSP-cycle, which would result in a graph with a Eulerian Cycle. Hence solving TSP is equivalent to finding a minimum cost set of edges E such that $G = (V, E)$ has an Eulerian Cycle.

For example, consider the following proposed algorithm:

1. Find the MST T of the complete graph on V with weights defined by d.
2. Add every edge in T *twice* to the set E. This ensures that each vertex in V has even degree.
3. Since every vertex in multigraph $G = (V, E)$ has even degree, we can find a Eulerian Cycle C for G.
4. Return C (skipping repeated vertices).

This is a correct algorithm for solving the TSP problem. Unfortunately, it will not yield any improvement over the earlier 2-approximation algorithm because we are still adding each edge in the MST twice, leading to a cost that is at most twice optimal.

Notice that it is quite inefficient to add each edge twice! Our only goal is to ensure that each vertex has even degree—if a vertex already has even degree, why are we doubling all of its outgoing edges, and thus increasing our cost? Consider the example in Figure 5.

Thus the question we must answer is: How do we add the minimal number of edges to a multigraph to ensure that each vertex has even degree? Thus, in summary, the resulting algorithm should look like:

1. Find the MST T of the complete graph on V with weights defined by d.
2. Add every edge in T to the set E.

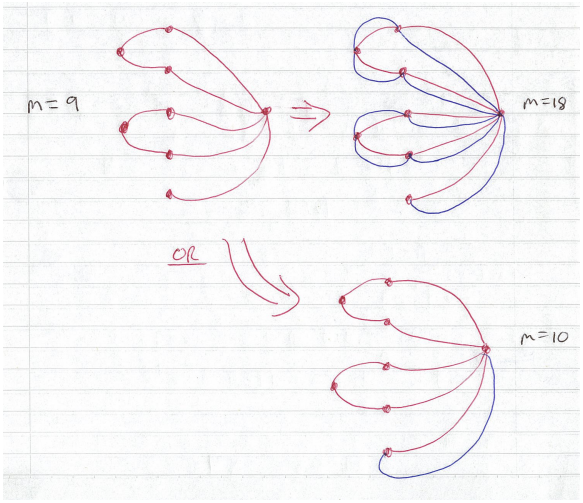


Figure 5: Example of finding a TSP by creating vertices with even degree. Initially, we have a graph with nine edges. If we double every edges, we have a graph with 18 edges. However, if we want to ensure that eachvertex has even degree, it suffices to add just one edge to the graph, yielding a graph with only 10 edges.

3. Add the minimum cost set of edges E' to E such that in $E \cup E'$, every vertex has even degree.
4. Since every vertex in multigraph $G = (V, E \cup E')$ has even degree, we can find a Eulerian Cycle C for G.
5. Return C (skipping repeated vertices).

We already know that $cost(T) \leq cost(OPT)$, since the optimal TSP-cycle is also a spanning tree (once you have removed a single edge). The key is to prove that $cost(E') \leq OPT/2$.

5.2 Matchings

The last tool we need is a perfect matching:

Definition 6 *We say that (V, M) is a **matching** if no two edges in M share an endpoint, i.e., all vertices in V have degree ≤ 1. We say that (V, M) is a perfect matching if every vertex in V has exactly degree 1, i.e., every vertex is matched.*

Notice that a perfect matching only exists if $|V|$ is even: if there are an odd number of vertices, it is clearly impossible to match them all.

If each edge has a weight, then we may want to find the minimum cost perfect matching. One of the most beautiful results in combinatorial optimization yields exactly that:

Theorem 7 *There is a polynomial time algorithm for finding the minimum cost perfect matching.*

This finds the optimal solution, and it does it efficiently! Unfortunately, this algorithm is beyond the scope of today's class. (See Edmonds's algorithm [1]. The solution involves his famous *Blossom Algorithm* for finding matchings, along with the primal-dual method and linear programming.) For now, we are just going to assume that we can find a minimum cost perfect matching.

5.3 Evens and Odds

We want to use the idea of perfect matchings to pair up the vertices in the multigraph that have odd degree. If we can assign each of them a partner, then they will have even degree. But there is one potential problem: what if there are an odd number of vertices that need a partern? Luckily, that is impossible due to a nice counting argument:

Lemma 8 *In every graph $G = (V, E)$, there are an even number of vertices with odd degree.*

Proof Let O be the set of vertices with odd degree, and $V - O$ the set of vertices with even degree. Let $deg(u)$ be the degree of u.

Recall that we can count the number of edges in the graph by summing up the total degree and diving by 2, i.e.:

$$\sum_{u \in V} deg(u) = 2|E|$$

That is, the sum of the degrees of all the vertices is even.

Now, consider just the vertices in $V - O$: since each vertex has even degree, the sum of their degrees must also be even:

$$\sum_{u \in V-O} deg(u) = 2k$$

Summing everything, we conclude that:

$$\begin{aligned} \sum_{u \in O} deg(u) + \sum_{u \in V-O} deg(u) &= \sum_{u \in V} deg(u) \\ \sum_{u \in O} deg(u) + 2k &= 2|E| \end{aligned}$$

Since the right-hand-side of the expression is even, the left-hand side of the expression must also be even. That is:

$$\sum_{u \in O} deg(u) = 2\ell$$

But each vertex in O has odd degree. Hence the only way that the sum of their degrees can be even is if there are an even number of odd vertices, i.e., $|O|$ is even. □

5.4 Christofides Algorithm

Putting the pieces together, we now have the following algorithm:

1. Find the MST T of the complete graph on V with weights defined by d.
2. Add every edge in T to the set E.
3. Let O be the vertices in T with odd degree. Notice that $|O|$ is even.
4. Let M be the minimum cost perfect matching for O.
5. Construct the multigraph $G = (V, E \cup M)$.
6. Since every vertex in multigraph $G = (V, E \cup M)$ has even degree, we can find a Eulerian Cycle E for G.

7. Return E (skipping repeated vertices).

To analyze this, we notice that the cost consists of the following:

$$cost(E) = \sum_{e \in E} d(e) + \sum_{e \in M} d(e)$$

The first part of the expression comes from the MST, and hence we already know that:

$$\sum_{e \in E} d(e) \leq OPT$$

Since OPT forms a cycle, we can remove any edge from the cycle to get a spanning tree; the MST T must have cost no greater than this cycle.

We now need to argue that $2 \times \sum_{e \in M} d(e) \leq OPT$. Let C be the cycle for OPT. Let C' be the same cycle as C where we skip all the vertices in $V - O$, and we skip repeats. That is, C' is a cycle on the odd vertices with no repeats. Notice that $cost(C') \leq cost(C)$, since we have only skipped vertices (and the triangle inequality holds).

Also, notice that cycle C' has an even number of vertices (because there are an even number of odd vertices) and an even number of edges. Assume that $C' = (v_1, v_2, \ldots, v_{2k})$.

We can now construct two different perfect matchings M_1 and M_2. We define them as follows:

$$\begin{aligned} M_1 &= (v_1, v_2), (v_3, v_4), (v_5, v_6), \ldots, (v_{2k-1}, v_{2k}) \\ M_2 &= (v_2, v_3), (v_4, v_5), (v_6, v_7), \ldots, (v_{2k}, v_1) \end{aligned}$$

Notice that each of these perfect matchings has $k = |O|/2$ edges, and both are valid perfect matchings for the set O. Since M is the minimum cost perfect matching, we conclude that:

$$\begin{aligned} cost(M) &\leq cost(M_1) \\ cost(M) &\leq cost(M_2) \end{aligned}$$

Notice, though, that $cost(M_1) + cost(M_2) = cost(C') \leq cost(C)$. So, if we sum the matchings, we get:

$$2 \times cost(M) \leq cost(M_1) + cost(M_2) \leq cost(C) = OPT$$

Thus we have shown that $cost(M) \leq OPT/2$.

Putting the pieces together, we see that the cost of the cycle output by the algorithm is:

$$\begin{aligned} cost(E) = \sum_{e \in E} d(e) + \sum_{e \in M} d(e) &\leq cost(T) + cost(M) \\ &\leq OPT + OPT/2 \\ &\leq 1.5 \cdot OPT \end{aligned}$$

Thus, we have discovered a 1.5-approximation algorithm for the M-R-TSP.

References

[1] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal on Maths*, 17:449–467, 1965.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[3] Steven Halim and Felix Halim. *Competitive Programming: The New Lower Bound of Programming Contests*. Lulu, 3rd edition, 2013.

Index