CS 170
Fall 2017

Efficient Algorithms and Intractable Problems
Prasad Raghavendra and Sanjam Garg

Homework 10

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.
- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.
- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Wednesday, November 29, at 4:59pm

0. Who did you work with?

List all your collaborators on this homework. If you have no collaborators, please list "none".

Solution: N/A

1. (★★★ level) Circular Reductions

Given a set S of non-negative integers $[a_1, a_2, \ldots, a_n]$, consider the following problems:

- **Partition**: Determine whether there is a subset $P \subseteq S$ such that $\sum_{i \in P} a_i = \sum_{j \in S \setminus P} a_j$
- **Subset Sum**: Given some integer k, determine whether there is a subset $P \subseteq S$ such that $\sum_{i \in P} a_i = k$

(a) Find a linear time reduction from Partition to Subset Sum. Then prove your reduction is correct.

(b) Find a linear time reduction from Partition to Knapsack (for Knapsack, refer to pages 164-168 of textbook). Then, prove your reduction is correct.

(c) Find a linear time reduction from Subset Sum to Partition. Then prove your reduction is correct. (Hint: think about adding certain elements to the set S)

Solution:

(a) Let $m = \sum_{i \in S} a_i$, and let $k = \frac{m}{2}$.

Claim: There is a set $P \subset S$ such that $\sum_{i \in P} a_i = \sum_{j \in S \setminus P} a_j \iff$ there is a set $P \subset S$ such that $\sum_{i \in P} a_i = k$.

Proof: Suppose there exists a set P satisfying the LHS. Since $m = \sum_{i \in S} a_i$ and $\sum_{i \in P} a_i = \sum_{j \in S \setminus P} a_j$, we see that $\sum_{i \in P} a_i = \sum_{j \in S \setminus P} a_j = \frac{m}{2} = k$. Therefore, we have a set P such that $\sum_{i \in P} a_i = k$. Suppose there exists a set P satisfying the RHS. Since $\sum_{i \in P} a_i = k = \frac{m}{2}$, $\sum_{j \in S \setminus P} a_j = m - k = \frac{m}{2}$. Therefore, $\sum_{i \in P} a_i = \sum_{j \in S \setminus P} a_j$, and so the LHS is satisfied.

This shows that we can reduce **partition** to **subset-sum** by running **subset-sum**(S, k), if m is even. If m is odd, then the partition problem has no solution. The sum of both halves of the partition have the same parity (i.e. both are even or both are odd), so their sum m must be even. Therefore, WLOG we can assume that m is even. The reduction is linear since computing $k = \frac{m}{2}$ takes linear time.

(b) By part (a), WLOG we can assume that $m = \sum_{i \in S} a_i$ is even. Let $k = \frac{m}{2}$. We can construct a knapsack problem as follows: let S' be the set of n possible items, where the ith item has $v_i = w_i = a_i$, and let k be the total weight that the knapsack can hold.

Claim: There is a set $P \subset S$ such that $\sum_{i \in P} a_i = \sum_{j \in S \setminus P} a_j \iff$ knapsack finds a subset of items with total value k.

Proof: This particular instance of knapsack forces the value and the weight of the knapsack to be the same, for any set of items chosen to be in the knapsack. Since the maximum weight is k, it follows that the maximum value is at most k. Suppose we have a set P satisfying the LHS. From part (a), we know that $\sum_{i \in P} a_i = k$. Let P' be the set containing the items corresponding to the set P. We see that $\sum_{i \in P'} v_i = \sum_{i \in P'} a_i = \sum_{i \in P} a_i = k$, so we have found a knapsack achieving the maximum value of k. Conversely, suppose that the RHS holds. Then knapsack found a set P' such that $\sum_{i \in P'} v_i = k$. Since $k = \sum_{i \in P'} v_i = \sum_{i \in P} a_i$, we see that $\sum_{j \in S \setminus P} a_j = m - \sum_{i \in P} a_i = m - k = k$, so that $\sum_{i \in P} a_i = \sum_{j \in S \setminus P} a_j$.

This shows that we can reduce **partition** to **knapsack**, by running the **knapsack** instance described. The reduction is linear time since constructing the knapsack problem takes $O(n)$ time, where $|S| = n$.

(c) Let $m = \sum_{i \in S} a_i$, and let k be the value chosen for subset sum. Let $W = |m - 2k|$. Let $S' = S \cup \{W\}$.

Claim: There is a set $P \subset S$ such that $\sum_{i \in P} a_i = k \iff$ there is a set $P' \subset S'$ such that $\sum_{i \in P'} a_i =$

$\sum_{j \in S' \setminus P'} a_j$.

Proof: Suppose the RHS holds. We have that $\sum_{i \in P} a_i = k \implies \sum_{j \in S \setminus P} a_j = m - k$. If $W = m - 2k$, then $W + \sum_{i \in P} a_i = m - 2k + k = m - k = \sum_{j \in S \setminus P} a_j$. Similary, if $W = 2k - m$, then $W + \sum_{j \in S \setminus P} a_j = 2k - m + m - k = k = \sum_{i \in P} a_i$. Therefore, if P is a solution to **subset-sum**, then either $P' = P$ or $P' = P \cup \{W\}$ is a solution to **partition**. Conversely, suppose that the LHS holds. Then $\sum_{i \in P'} a_i = \sum_{j \in S' \setminus P'} a_j$. We know that $W = \pm(m - 2k)$. Therefore, WLOG we have that $m - 2k + \sum_{i \in P} a_i = \sum_{j \in S \setminus P} a_j$, as we can swap P' with $S' \setminus P'$ to make this true. We also know that $\sum_{i \in P} a_i + \sum_{j \in S \setminus P} a_j = m$. Solving the two equations, we get that $\sum_{i \in P} a_i = k$.

Thus, we can construct a solution to **subset-sum** from our solution to the **partition** instance, and vice versa. This shows that the reduction is correct. The reduction is linear time since computing W takes linear time.

2. (★★★★ level) Finding Zero(s)

Consider the problem INTEGER-ZEROS.

INPUT: A multivariate polynomial $P(x_1, x_2, x_3, \ldots, x_n)$ with integer coefficients, specified as a sum of monomials.

OUTPUT: Integers $a_1, a_2, \ldots a_n$ such that $P(a_1, a_2, a_3, \ldots, a_n) = 0$.

Show that 3-SAT reduces in polynomial time to INTEGER-ZEROS. (You do not need to show that INTEGER-ZEROS is in **NP**: in fact, it is known *not* to be in **NP**).

Hint 1: Given a 3-SAT formula ϕ in the variables $x_1, x_2, \ldots, x_n$, your reduction f will produce a polynomial P in the same variables such that satisfying assignments correspond to $0, 1$ valued zeros of P.

Hint 2: If your polynomial constructed above has exponential amount of terms, it is not optimal! You need to reduce it to polynomial amount. Think about the equality $a^2 + b^2 = 0$ if and only if $a = b = 0$ when a, b are real to achieve it. [1]

Solution:

Given a 3-SAT formula ϕ in the variables $x_1, x_2, \ldots, x_n$, our reduction f will produce a polynomial P in the same variables such that satisfying assignments correspond to $0, 1$ valued zeros of P. We can use $1 - x_i$ to denote $\neg x_i$ and multiplication to denote $\wedge$. However naïvely translating clauses using this method and multiplying can lead to a polynomial P with 3^m terms, where m is the number of clauses. Thus, we will instead add use the observation that $a^2 + b^2 = 0$ iff $a = b = 0$ when a, b are real to achieve this. We now proceed to describe f. Let the clauses of ϕ be $c_1, c_2, \ldots, c_m$ where $c_i = (y_{i1} \vee y_{i2} \vee y_{i3})$ where y_{ij} are literals. Corresponding to the ith clause we consider the polynomial $p_i(x_1, x_2, \ldots, x_n) := (1 - y_{i1})(1 - y_{i2})(1 - y_{i3})$. For example, corresponding to the clause $x_1 \vee \neg x_2 \vee x_3$, we will get the polynomial $(1 - x_1)x_2(1 - x_3)$. Notice that if x_i are $0, 1$ valued, then p_i is zero if and only if the ith clause is satisfied (interpreting 0 as FALSE and 1 as TRUE). We now need to ensure that x_i are $0, 1$ valued. To do this we need to ensure that the polynomials $q_i(x_1, x_2, \ldots, x_n) := x_i(1 - x_i)$ are all 0. Thus, we need to enforce that all the q_i and p_i polynomials are 0 at a zero of P. This is achieved by setting

$$P(x_1, x_2, \ldots, x_n) := \sum_{i=1}^{m} p_i(x_1, x_2, \ldots, x_n)^2 + \sum_{i=1}^{n} q_i(x_1, x_2, \ldots, x_n)^2.$$

By construction, p can be fully expanded in time polynomial in the size of the clause, and hence the reduction runs in polynomial time.

Now suppose ϕ has a satisfying assignment α. Setting $x_i = 1$ when $\alpha(x_i) = \text{TRUE}$ and $x_i = 0$ otherwise, we see that all the polynomials q_i constructed above are 0. Further, since α is a satisfying assignment, all the polynomials p_i are 0 too. Thus, P has the integral zero as constructed above.

Now suppose P has integral zeros: suppose $P(s_1, s_2, \ldots, s_n) = 0$ for integral s_i. This implies that all the p_i and q_i polynomials are 0. By the latter, we get $s_i \in \{0, 1\}$ for all i. Now setting $x_i = \text{TRUE}$ if $s_i = 1$ and $x_i = \text{FALSE}$ otherwise, we see that all the clauses of ϕ are satisfied since p_i's are 0. Thus, given an integral solution to P, we can construct in polynomial time a satisfying assignment to ϕ.

***Note**: A construction of P which has exponentially many terms should probably lose some points.*

[1] Fun fact: This problem INTEGER-ZEROS is *really* hard: the decision version of the problem is *undecidable*. This means that there is provably no algorithm which, given a multivariate polynomial, will decide correctly whether or not it has integer zeros. However, if we relax the problem to ask if there are *real* numbers at which the given polynomial vanishes, then the problem surprisingly becomes decidable! For Blue and Gold jingoists: the above decidability result was proven by Alfred Tarski, a Berkeley professor, in 1949. The undecidability of INTEGER-ZEROS was proven by Yuri Matiyasevich (at the ripe old age of 23!) in 1970, building upon previous work of another Berkeley professor, Julia Robinson.

3. (★★★ level) Connectivity-Preserving Subgraph

In the CONNECTIVITY-PRESERVING SUBGRAPH problem (CPS), we are given:

1. A directed graph $G = (V, E, w)$ with nonnegative weights w.
2. A set of k *connectivity demands* (s_i, t_i).

The task is to find a subgraph $H \subseteq G$, if one exists, such that for every demand (s_i, t_i), there exists an $s_i \to t_i$ path in H. Furthermore, this subgraph should minimize the total edge weight $\sum_{e \in H} w(e)$.

(a) Suppose CPS is restricted to having only one connectivity demand. What familiar problem is this? *Here, no proof is needed. Just state the problem.*

Solution: SHORTEST s-t PATH.

(b) It turns out that CPS (for general k) is an **NP**-hard optimization problem. Give a k-approximation algorithm for CPS.
Describe the algorithm precisely; justify its correctness and running time. No pseudocode.

Solution: Take the union of the shortest s_i-t_i paths (call them P_i^*, respectively) for all $i \in \{1, \ldots, k\}$, which clearly satisfies the connectivity demands.

Proof of correctness: How costly is this solution? Observe that the optimal solution $H^* \subseteq G$ must contain s_i-t_i paths P_i such that $|P_i| \geq |P_i^*|$, where $|P_i|$ denotes the total edge weight along path P_i.

Therefore, the total weight of H^* must be

$$\sum_{e \in H^*} w(e) = \left| \bigcup_{i=1}^{k} P_i \right| \geq \max_i |P_i| \geq \max_i |P_i^*|$$

Since our solution has cost

$$\sum_i |P_i^*| \leq k \cdot \max_i |P_i^*|$$

it follows that our subgraph $\bigcup_{i=1}^{k} P_i^*$ has cost at most k times that of the optimum, H^*.

Conversely, if there is no solution to the original instance, then there is no way to connect all the s_i-t_i pairs, and thus the approximation algorithm reports that at least one of these pairs does not have a path between them.

Running time: The algorithm takes polynomial time, as we simply need to call Dijkstra's algorithm k times and combine those paths, for at most $O(k \cdot (|V| + |E|) \log |V|)$.

4. (★★★ level) Multiway Cut

In the MULTIWAY CUT problem, the input is an undirected graph $G = (V,E)$ and a set of terminal nodes $s_1, s_2, ..., s_k \in V$. The goal is to find the minimum set of edges in E whose removal leaves all terminals in different components.

(a) Show that this problem can be solved in polynomial time when $k = 2$.

(b) Give an approximation algorithm with ratio at most 2 for the case $k = 3$. (Hint: use part a)

(c) Give an approximation algorithm with ratio at most $k-1$ for any k.

(d) **For Fun (No extra credit)** Give an approximation with ratio strictly less than 2 for the case $k = 3$. Anything less than 2 will get 1 point, but a ratio of at most 4/3 will get 2 points.

(e) **For Fun (No extra credit)** Give an approximation with ratio strictly less than 2 for general k.

Solution:

(a) (i) Reduction: Since $k = 2$, we only have two distinguished vertices s_1 and s_2. We can run max-flow on G from s_1 to s_2, where all edges have capacity 1. This gives us a min-cut (L,R). The set of edges $E' = \{(u,v) | u \in L, v \in R\}$ is a set with the minimum number of edges whose removal will make s_1 and s_2 be in two connected components.

(ii) Proof of Correctness: Suppose after removing E', s_1 and s_2 are still in the same connected component. Then there is a path p from s_1 to s_2 in G that does not use any of the edges of the form $E' = \{(u,v) | u \in L, v \in R\}$. But this is impossible, since (L,R) is a cut, and $s_1 \in L$, $s_2 \in R$, which implies (by definition of a cut) that any path from s_1 to s_2 must use an edge in E'. Thus, s_1 and s_2 are in different connected components. This shows that removing E' disconnects s_1 and s_2.

Suppose there exists a set E_1 such that $|E_1| \leq |E'|$, and removing the edges of E_1 will put s_1 and s_2 into different connected components. Then, any path from s_1 to s_2 in G must use an edge of E_1. Suppose f is a max-flow. Then $val(f) \leq |E_1|$, since the flow goes from s_1 to s_2 using $val(f)$ paths (since all capacities are 1), and every path must use at least one edge of E_1, so there can be at most $|E_1|$ total paths. But, by max-flow min-cut, $|E'| = cap(L,R) = val(f) \leq |E_1|$, and $|E_1| \leq |E'|$, so $|E_1| = |E'|$. Thus, E' has minimal size, so it is an optimal solution.

(iii) Running time: The runtime is polynomial time since max-flow is polynomial time.

(b) (i) High Level Description: Since $k = 3$, we have three distinguished vertices s_1, s_2, s_3. Let $G = (V,E)$ be the graph in the problem. Define $G_1 = (V \cup \{t\}, E \cup \{(s_2,t),(s_3,t)\})$. For all $e \in E$, set $c(e) = 1$, and set $c(s_2,t) = c(s_3,t) = \infty$. We can run the algorithm from part (a) on (G_1, s_1, t) and (G, s_2, s_3) to get two sets $E_1, E_2 \subset E$, such that removing E_1 from G_1 disconnects s_1 and t, and removing E_2 from G disconnects s_2 from s_3. Since we set $c(s_2,t) = c(s_3,t) = \infty$, (s_2,t) and (s_3,t) cannot be in the min-cut in G_1, so $E_1 \subset E$. Removing E_1 from G will therefore disconnect s_1 from s_2 and s_3, and so removing $E_1 \cup E_2$ from G will disconnect s_1, s_2, s_3.

(ii) Proof of Correctness: Since the algorithm from part (a) runs max-flow, and we set $c(s_2,t) = c(s_3,t) = \infty$, the set E_1 returned from part (a) cannot contain (s_2,t) or (s_3,t), as otherwise the max-flow would be infinite, which is clearly impossible since all other edge capacities are 1. Therefore, $E_1 \subset E$. By part (a), removing E_1 from G_1 will disconnect s_1 from t. But s_2 and s_3 will be in the same connected component as t, since the edges (s_2,t) and (s_3,t) were not removed. Therefore, removing E_1 from G_1 disconnects s_1 from s_2 and s_3. Removing t from G_1, we see that removing E_1 from G will make s_1 in a different connected component from s_2 and s_3. By part (a), we also have that removing E_2 from G will make s_2 and s_3 in different connected components. Therefore, removing $E_1 \cup E_2$ will make s_1, s_2 and s_3 all be in different connected components.

Now, we must show that the algorithm has an approximation ratio of 2. Suppose E' is the optimal solution. Then removing the edges from G_1 will disconnect s_1 from s_2 and s_3. However, the solution in part (a) is optimal, so we must have that $|E_1| \leq |E'|$. Similarly, removing E' from G will disconnect s_2 and s_3. Since the algorithm from part (a) gives the optimal solution, we must have that $|E_2| \leq |E'|$. Therefore, $|E_1 \cup E_2| \leq |E_1| + |E_2| \leq 2|E'|$, so the algorithm obtains an approximation ratio of at least 2.

(iii) Running time: The running time is polynomial time since the algorithm from part (a), in particular max-flow, is polynomial time.

(c) Repeat the idea in part (b). Run max-flow/min-cut $k-1$ times, getting sets of edges $E_1, \ldots, E_{k-1}$, where E_i is the min-cut that splits s_i from $s_{i+1}, \ldots, s_k$. Let E' be the optimal solution. $\forall i$, $|E_i| \leq |E'|$, since both E_i and E' split s_i from $s_{i+1}, \ldots, s_k$, and E_i is the minimum sized set which does this. Therefore, $S = \left|\cup_{i=1}^{k-1} E_i\right| \leq \sum_{i=1}^{k-1} |E_i| \leq (k-1)|E'|$. Thus, the algorithm achieves an approximation ratio of $k-1$. The algorithm is polynomial time because each call to max flow takes polynomial time, and there are k (polynomially many) calls.

(d) To do better than a 2-approximation, it is important to keep in mind that in the graph H obtained by deleting the optimal multiway cut C, each edge of C has endpoints in exactly two of the at least k connected components of H.

There are at least two solutions to this problem: one that has approximation ratio 5/3 and one that has approximation ratio 4/3. Given three vertices x, y, and z, the 5/3-approximation finds the best of the three cuts that separate one of $\{x, y, z\}$ from the rest and combines that cut with the min cut of the remaining two vertices in the resulting graph. The first cut cuts at most 2/3 of the optimal multiway cut's edges because one of the three singleton-pair cuts has an endpoint of at most $2/3$ of the multiway cut's edges (see correctness proof for details). The min cut has at most the multiway cut's number of edges. Combining these two cuts yields a solution with $2/3OPT + OPT = 5/3OPT$ edges.

The other solution is as follows. Given three vertices $\{x, y, z\}$, the 4/3-approximation combines the best two singleton-pair cuts. Without loss of generality, suppose that the best two singleton-pair cuts separate x from $\{y, z\}$ and y from $\{x, z\}$.Let C_1 and C_2 be the $x - \{y, z\}$ and $y - \{x, z\}$ minimum cuts and D_1 and D_2 be the sets of edges in the multiway cut with an endpoint in x and y's connected component respectively. By definition of the min cut, $|C_1| \leq |D_1|$ and $|C_2| \leq |D_2|$. $|D_1| + |D_2| = |D_1 \cap D_2| + OPT$, where OPT is the size of the multiway cut. $D_1 \cap D_2$ is the set of edges with one endpoint in x's component and one endpoint in y's component. If we change $x - y$ to $y - z$ or $x - z$, the corresponding sets $D_1 \cap D_2$ are disjoint. Therefore, at least one of the sets (in particular the best choice $x - y$) has $|D_1 \cap D_2| \leq OPT/3$, showing that $|C_1| + |C_2| \leq (4/3)OPT$.

(e) There is a $2 - (2/k)$-approximation for general k. The algorithm directly generalizes the 4/3-approximation from the previous part. Given k points $x_1, x_2, \ldots, x_k$, return the smallest union of $k-1$ cuts of the form $x_j - \{x_1, x_2, \ldots, x_{j-1}, x_{j+1}, \ldots, x_k\}$.

Let D_j be the set of edges in the optimal multiway cut with an endpoint in x_j's component and let C_j be the $x_j - \{x_1, x_2, \ldots, x_{j-1}, x_{j+1}, \ldots, x_k\}$ min cut. By definition of the minimum cut, $|C_j| \leq |D_j|$ for all j. Furthermore, $\sum_{j=1}^{k} |D_j| \leq 2OPT$ because each edge has an endpoint in two connected components of the graph with the multiway cut removed. By leaving out cut i, we make it so that all edges with an endpoint in i's component are counted at most once. Since each edge has an endpoint in 2 components, the average number of endpoints of edges in each component is $(2/k)OPT$. Therefore, there is an i with $|D_i| \geq (2/k)OPT$. In particular, the weight of the solution without that cut is

$$
\begin{aligned}
|\cup_{j=1,j\neq i}^{k} C_j| &\leq \sum_{j=1,j\neq i}^{k} |C_j| \\
&\leq \sum_{j=1,j\neq i}^{k} |D_j| \\
&\leq 2OPT - |D_i| \\
&\leq (2-2/k)OPT
\end{aligned}
$$

as desired.

5. (★★★★ level) TSP via Local Merging

The METRIC TRAVELING SALESMAN problem is the special case of TSP in which the edge lengths form a *metric*, which satisfies the triangle inequality and other properties (review section 9.2.2). In class, we saw a 2-approximation algorithm for this problem, which works by building a minimum spanning tree.

Here is another heuristic for metric TSP:

Summary: maintain a current *subtour* τ on a subset of V, then expand it to include all nodes in G

function TSP-LOCAL-MERGE(complete, undirected graph $G = (V, E, \ell)$ with metric distance)
    $\tau \leftarrow [v_1, v_2]$ where v_1 and v_2 are the closest pair of nodes in G
    while $|\tau| < |V|$ **do**
        $v_j \leftarrow$ the node in $V \setminus \tau$ that is closest to any node v_i in τ
        $v_k \leftarrow$ the node that follows v_i in τ.
        Modify τ by replacing v_i, v_k with v_i, v_j, v_k

Prove that the above "local merging" algorithm also approximates metric TSP to a factor of 2.
Hint: Note the similarity between this algorithm and Prim's.

Solution: Observe that this algorithm picks nodes in precisely the same way as Prim's algorithm.

First, observe that the lightest edge (v_1, v_2) in G is a valid Prim's MST edge. Our subtour starts with two copies of this edge. Thenceforth, each edge (v_i, v_j) is also an edge that would appear in the MST selected by Prim's. When we incorporate v_j into the subtour τ, we remove edge (v_i, v_k) from it, and add in edges (v_i, v_j) and (v_j, v_k). So, on iteration t of the while-loop, we have

$$\text{cost}(\tau_{t+1}) = \text{cost}(\tau_t) - \ell(v_i, v_k) + \ell(v_i, v_j) + \ell(v_j, v_k)$$

By the triangle inequality, $\ell(v_j, v_k) \leq \ell(v_i, v_j) + \ell(v_i, v_k)$, or $\ell(v_j, v_k) - \ell(v_i, v_k) \leq \ell(v_i, v_j)$. Thus

$$\text{cost}(\tau_{t+1}) \leq \text{cost}(\tau_t) + 2 \cdot \ell(v_i, v_j)$$

so for every edge weight that Prim's would incur, we incur at most twice as much. Since any TSP tour is at least as costly as an MST (review section 9.2.3 if this is unclear), we conclude that

$$\text{cost}(\tau_{|V|}) \leq 2 \cdot \text{cost}(\text{MST}) \leq 2 \cdot \text{OPT}$$

Finally, because the algorithm can implemented a la Prim's, its running time is polynomial, making it a valid approximation algorithm. □

6. (★★★★★ level) Steiner Tree

The Steiner tree problem is the following:

Input: An undirected graph $G = (V, E)$ with non-negative edge weights $\text{wt} : E \to \mathbb{N}$, a set $S \subseteq V$ of special nodes.

Output: A Steiner tree for S whose total weight is minimal

A Steiner tree for S is a tree composed of edges from G that spans (connects) all of the special nodes S. In other words, a Steiner tree is a subset $E' \subseteq E$ of edges, such that for every $s, t \in S$, there is a path from s to t using only edges from E'. The total weight of the Steiner tree is the sum of the weights included in the tree, i.e., $\sum_{e \in E'} \text{wt}(e)$.

The Steiner tree problem has many applications in different areas, including creating genealogy trees to represent the evolutionary tree of life, designing efficient networks, to even planning water pipes or heating ducts in buildings. Unfortunately, it is NP-hard.

Here is an approximation algorithm for this problem:

1. Compute the shortest distance between all pairs of special nodes. Use these distances to create a modified and complete graph $G' = (S, E_S)$, which uses the special nodes as vertices, and the weight of the edge between two vertices is the shortest distance between them.
2. Find the minimal spanning tree of G'. Call it M.
3. Reinterpret M to give us a Steiner tree for G: for edge in M, say an edge (r, s), select the edges in G that correspond to the shortest path from r to s. Put together all the selected edges to get a Steiner tree.

Prove that this algorithm achieves an approximation ratio of 2.

(Note that the efficiency of an approximation algorithm does not contradict NP-completeness because it gives an approximate (rather than an exact) solution to the problem.)

Solution: Let W be the weight of the optimal Steiner tree. Let H be the tree outputted by the algorithm. It is clear that $weight(H) \leq weight(M)$, as the quantity $weight(M)$ can count edges in the tree H multiple times. Using the optimal tree, we can construct a tour through all of the special vertices as follows: pick any vertex in the optimal tree, and explore using the DFS ordering. Every time we push a vertex v onto the stack, we add the edge (u, v) to the path, where u is the current vertex. Every time we pop a vertex v off the stack, we add (v, u) to the path, where u is the vertex that is on top of the stack after the pop. This constructs a tour that visits all the vertices in the optimal tree, and uses each edge exactly twice. Since it visits all the vertices in the optimal tree, in particular it visits all the special vertices. The tour has weight $2W$ since it uses each edge exactly twice.

Claim: Let F be any tour that goes through all of the special vertices. Then $weight(M) \leq weight(F)$.

Proof: From any tour F, we can construct a tree $T' \in G' = (S, E_S)$ that visits all the special vertices. First, we can construct a tree T so that the edges of T are in E_S, but the weights are different. We do this by simply following the tour and combining segments of the form $(s_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, s_n)$ into one edge (s_1, s_n) of weight $weight(s_1, v_2) + \cdots + weight(v_{n-1}, s_n)$, where $s_1, s_n \in S$, and all other $v_i \notin S$. If an edge is repeated, we ignore it, and we ignore any edge that would create a cycle. This guarantees that T a tree. Because we count the weight of each edge from F at most once, we see that $weight(T) \leq weight(F)$. Now consider T', a spanning tree in $G' = (S, E_S)$, where the edges of T' are the same as T, just with possibly smaller weights. The weights could be smaller since the edge weights in G' are equal to the lengths of the shortest paths, while the edge weights for T are only the lengths of paths (in particular, not necessarily shortest paths).

This shows that $weight(T') \leq weight(T)$. Since M is an MST in G', $weight(M) \leq weight(T')$. Therefore, $weight(M) \leq weight(T') \leq weight(T) \leq weight(F)$, for any tour F. This proves the claim.

In particular, from the tour we constructed earlier we see that $weight(M) \leq 2W$. Combining with the inequality from earlier, we get that $weight(H) \leq 2W$. But W is the optimal weight, so $\frac{weight(H)}{W} \leq \frac{2W}{W} = 2$. Thus, the algorithm achieves an approximation ratio of 2.