Approximation Algorithms

Last Updated: April 18th, 2026

1 Introduction

In the previous lecture we saw how there exist many optimization problems that cannot be solved in polynomial time. For if the optimization problem is solvable in polynomial time, then so is its corresponding decision problem, and vice versa. We now return to the study of optimization problems, and ask if there are polynomimal-time algorithms that can at least provide an approximation of the optimal solution.

2 Approximation Ratio

Let L be a minimization problem and suppose $\mathcal{A}$ is an approximation algorithm for L. Let $A(x)$ denote the (numerical) value returned by the algorithm on instance x of L, while $\text{OPT}(x)$ denotes the optimal value for instance x. Then the **approximation ratio** of $\mathcal{A}$ is defined as

$$\max_{x \in L} \frac{\mathcal{A}(x)}{\text{OPT}(x)}.$$

We interpret the above as meaning that the ideal approximation ratio for algorithm $\mathcal{A}$ is 1, but in reality it's only as good as its worst performance with respect to all the problem instances x of L. Here, the worst performance occurs when the ratio $\frac{\mathcal{A}(x)}{\text{OPT}(x)}$ is *maximized*.

Similarly, if L is a maximization problem, then the **approximation ratio** of $\mathcal{A}$ is defined as

$$\min_{x \in L} \frac{\mathcal{A}(x)}{\text{OPT}(x)}.$$

Again, we interpret the above as meaning that the ideal approximation ratio for algorithm $\mathcal{A}$ is 1, but in reality it's only as good as its worst performance with respect to all the problem instances x of L. But in this case the worst performance occurs when the ratio $\frac{\mathcal{A}(x)}{\text{OPT}(x)}$ is *minimized*.

In the next sections we provide algorithms that give approximation-ratio upper bounds on four different intractable problems: `Minimum Vertex Cover`, `Clustering`, `Load Balancing`, and `TSP`.

3 An Approximation Algorithm for Minimum Vertex Cover

Consider the following approximation algorithm for the `Minimum Vertex Cover` optimization problem. Each step the algorithm randomly selects an edge from the current graph, and adds the edge vertices to the cover. Indeed, given current graph $G_i = (V_i, E_i)$, at Step i an edge e is selected from G_i. Then G_{i+1} is formed by removing from G_i both vertices u and v that are incident with e, and all edges that are incident with either u or v. Both u and v are then added to the cover. The algorithm terminates at Step i iff $E_i = \emptyset$. Moreover, the algorithm begins with $G_1 = G$, where G is the input graph.

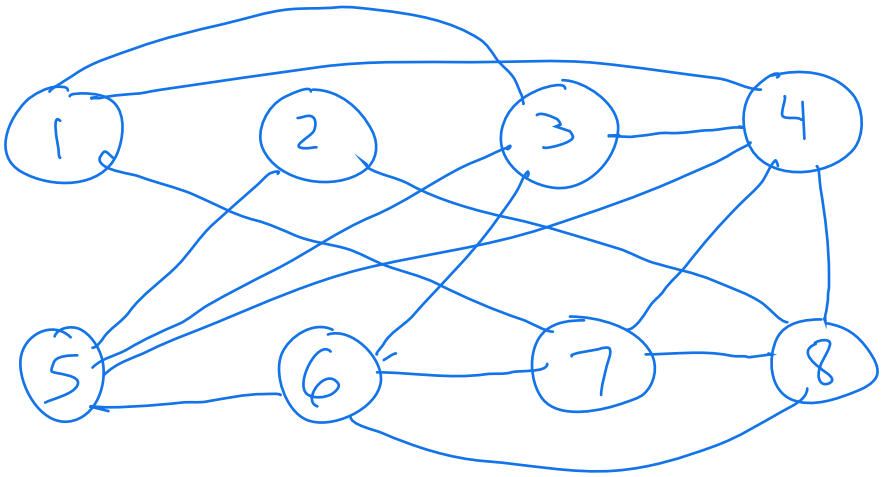
Example 3.1. Demonstrate the above algorithm on the graph whose edges are

$(6,8), (4,8), (2,5), (4,5), (7,8), (5,7), (8,2), (1,4), (4,3), (6,3), (6,7), (4,7), (5,6), (5,3), (1,7), (1,3)$

Compare the size of the minimum cover with that returned by the algorithm.

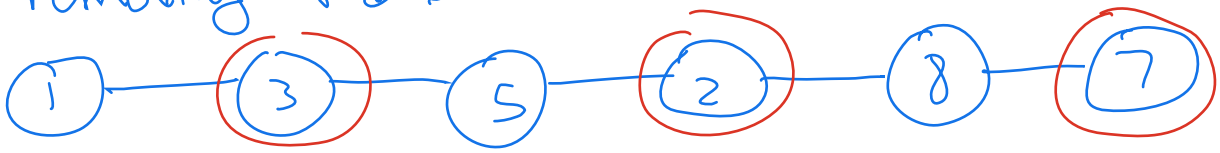
Approximation $C = \{6,8,2,5,1,4\}$ $\frac{A(G)}{OPT(G)} = \frac{6}{5} =$

$\frac{6}{5} = 1.2$



min Cover = $\{2,3,4,6,7\}$

After removing 4 & 6



Theorem 3.2. The above algorithm has an approximation ratio equal to 2.

Proof. At each step of the algorithm at least one of the vertices incident with selected edge e must be in the optimal cover. This is true since *every* edge of G must be incident with at least one vertex of the optimal cover. Therefore, since two vertices are added at each step, the worst case is that, at each step, exactly one of the vertices is not part of the optimal cover, which yields a cover size that is twice the size of the optimal one. Note that the worst case occurs when G has exactly one edge. □

In the Greedy-Algorithm lecture one of the exercises considered the following greedy heuristic for finding a minimum vertex cover. At each step, select the vertex v from G_i having highest degree and add v to the cover. Then G_{i+1} is obtained from G_i by removing v and any edge incident with v.

Theorem 3.3. There is a graph of order $\Theta(n \log n)$ for which the above greedy heuristic produces a cover that is a factor $\Theta(\log n)$ greater in size than the minimum cover.

Proof. Given positive integer n, consider the bipartite graph with vertex set $W = \{w_1, \ldots, w_n\}$ on the left side and vertex sets $U_1, \ldots, U_n$ on the right, where $U_i = \{u_{i1} \ldots, u_{i\lfloor n/i \rfloor}\}$. For the edges of the graph, w_k is adjacent to u_{ij} iff

$$\lceil k/i \rceil = j.$$

We have the following facts about this graph.

Fact 1. The left side has n vertices, while the right side has

$$n + \lfloor n/2 \rfloor + \lfloor n/3 \rfloor + \cdots + \lfloor n/n \rfloor = \Theta(n \log n)$$

vertices.

Fact 2. The values of k that satisfy this equation

$$\lceil k/i \rceil = j.$$

are $k = ij, ij-1, \ldots, ij-i+1$, implying that u_{ij} has degree i.

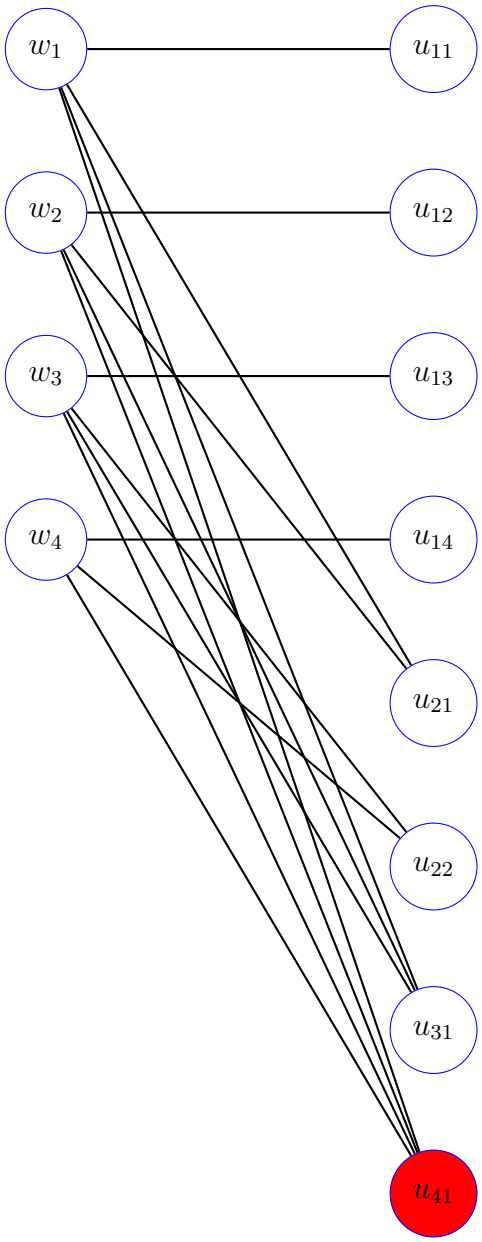
Fact 3. Each w_k is adjacent to at most one vertex in U_i, since the equation $\lceil k/i \rceil = j$ implies a unique j value.

Fact 4. From Fact 3, for all $1 \leq i \leq n$, the subgraph induced by vertices W, $U_1, \ldots, U_i$ is such that each vertex in W has degree at most i and every u_{ij} vertex has exactly degree i.

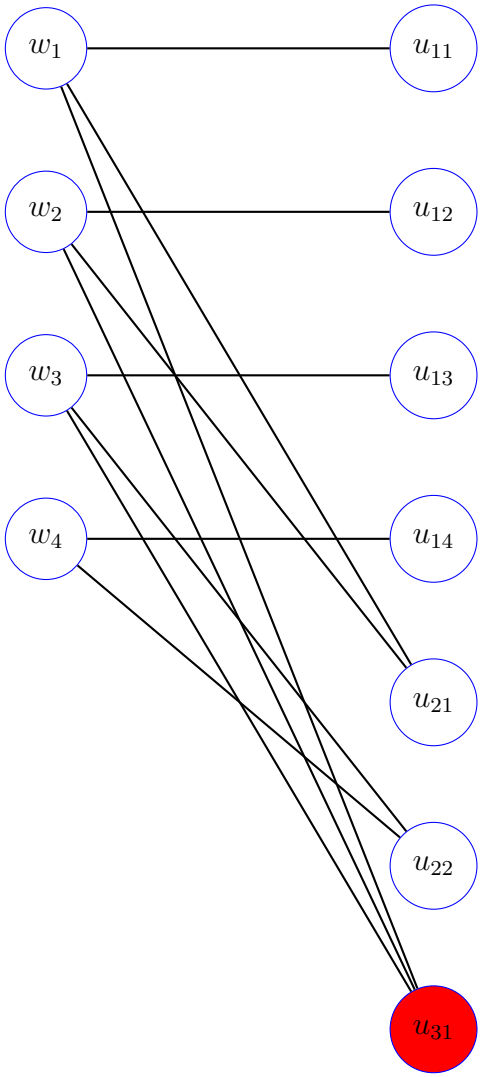
Fact 5. From Fact 4, if the heuristic breaks ties by selecting vertices from the U sets, then the heuristic will select all vertices in U_n, followed by all vertices in U_{n-1}, etc., all the way down to selecting vertices from U_1.

Fact 6. Therefore, the heuristic selects $\Theta(n \log n)$ vertices, yielding a $\Theta(\log n)$ approximation ratio. □

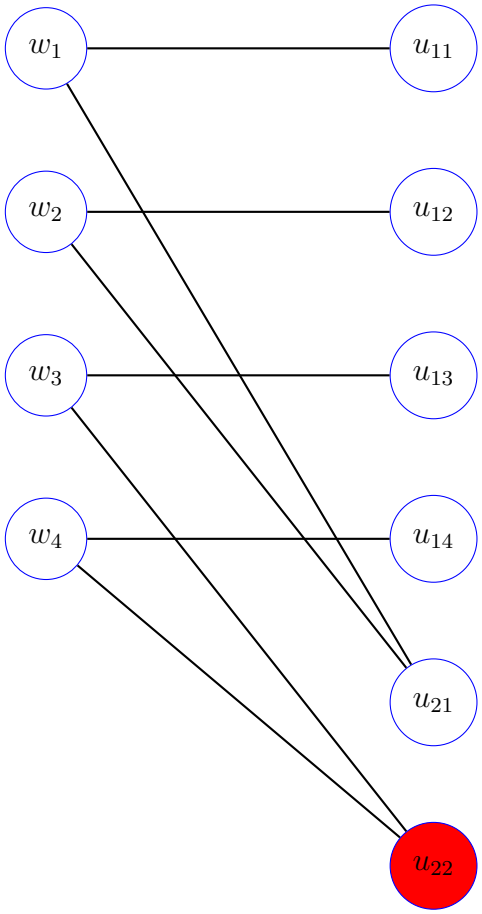
Example 3.4. Verify Theorem 3.3 for $n = 4$.



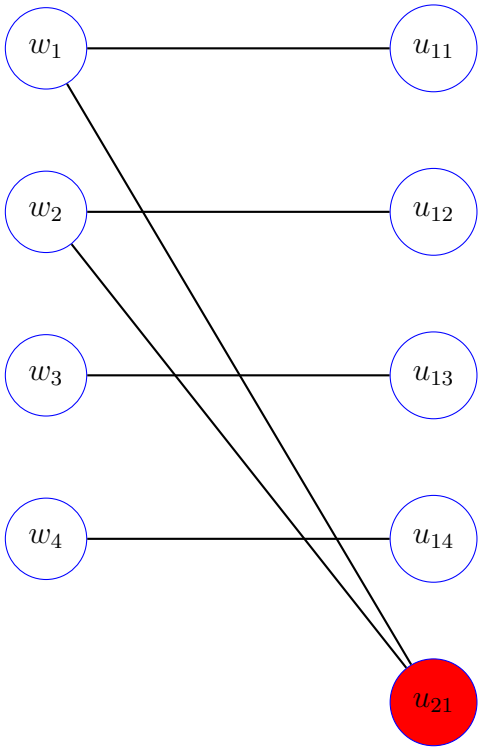
Greedy Heuristic Round 1: Select u_{41}



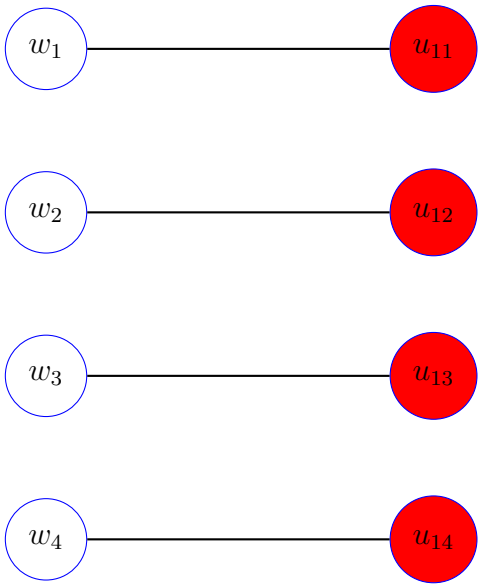
Greedy Heuristic Round 2: Select u_{31}



Greedy Heuristic Round 3: Select u_{22}



Greedy Heuristic Round 4: Select u_{21}



Greedy Heuristic Rounds 5-8: Select u_{14}, u_{13}, u_{12}, u_{11}

4 An approximation algorithm for `Clustering`

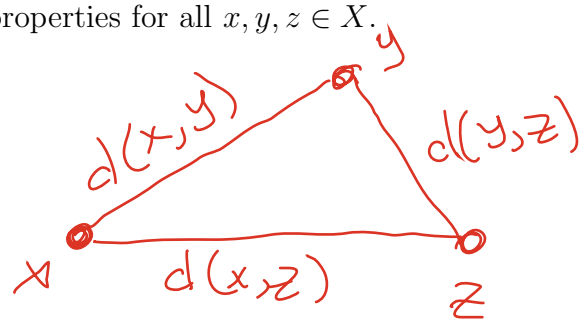
4.1 Preliminary definitions

Definition 4.1. A **metric space** is a pair (X, d), where X is a set of "points" and $d : X \times X \to R$ is a nonnegative function that satisfies the following three properties for all $x, y, z \in X$.

Nonnegative $d(x, y) \geq 0$ and $d(x, y) = 0$ iff $x = y$

Symmetric $d(x, y) = d(y, x)$

Triangle Inequality $d(x, z) \leq d(x, y) + d(y, z)$



For example, (R^n, d) is a metric space, where R^n is the set of all n-dimensional real-valued vectors, and $d(\vec{x}, \vec{y})$ denotes the Euclidean distance between $\vec{x}$ and $\vec{y}$.

Definition 4.2. Given a metric space (X, d) and $A \subseteq X$, the **diameter** of A, denoted diam A is defined as

$$\text{diam } A = \sup\{d(x, y) \mid x, y \in A\},$$

where $\sup B$ denotes the least upper bound for a set of numbers B. Note that, in case A is finite, then

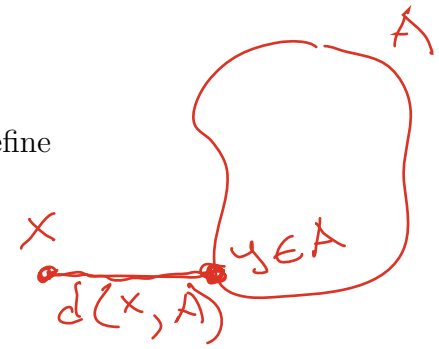
$$\text{diam } A = \max_{a_1, a_2 \in A} d(a_1, a_2).$$

For example, diam $\{1, \ldots, 100\} = 100 - 1 = 99$, while diam $\{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots\} = 1/2$.

Definition 4.3. Given a metric space (X, d), $x \in X$, and $A \subseteq X$, we define

$$d(x, A) = \inf\{d(x, y) \mid y \in A\},$$

where $\inf B$ denotes the greatest lower bound for a set of numbers B



For example, $d(0, \{1, \ldots, 100\}) = |0 - 1| = 1$, while $d(0, \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots\}) = 0$.

Definition 4.4. For $k \geq 1$, A **k-partition** of a set S is a collection $C_1, \ldots, C_k \subseteq S$ of nonempty subsets of S that satisfies the two properties

1. $C_i \cap C_j = \emptyset$ for all $i \neq j$,
2. $C_1 \cup \cdots \cup C_k = S$.

For example, $\{1, \ldots, 25\}$, $\{26, \ldots, 48\}$, and $\{49, \ldots, 100\}$ is a 3-partition of $\{1, \ldots, 100\}$.

Definition 4.5. An instance of the `Clustering` optimization problem is a set of points $S = \{x_1, \ldots, x_n\}$ from a metric space (X, d) and a nonnegative integer $k \geq 1$. The problem is to determine a k-partition of $C_1, \ldots, C_k$ for which

$$\max_j \operatorname{diam} C_j$$

is a minimum amongst all possible k-partitititions of S.

Note that the decision version of `Clustering` is NP-complete.

4.2 The Algorithm

Input: i) set of points $S = \{x_1, \ldots, x_n\}$ from a metric space (X, d), ii) $k \geq 1$.

Ouput: k-partition $C_1, \ldots, C_k$.

Initialize $P = \{x_1\}$. P : set of cluster centers

While $|P| < k$,

P += $\{x\}$, where

$$x = \arg\max_{y \in S} d(y, P).$$

//x is the furthest away from the members of P

Initialize $\mathcal{C} = \emptyset$.

For each $x \in P$

$\mathcal{C}$ += C_x, where $C_x = \{y \in S \mid d(y, x) < d(y, P - \{x\})$

//Cluster C_x contains all points in S that are closer to x than to any other point in P.

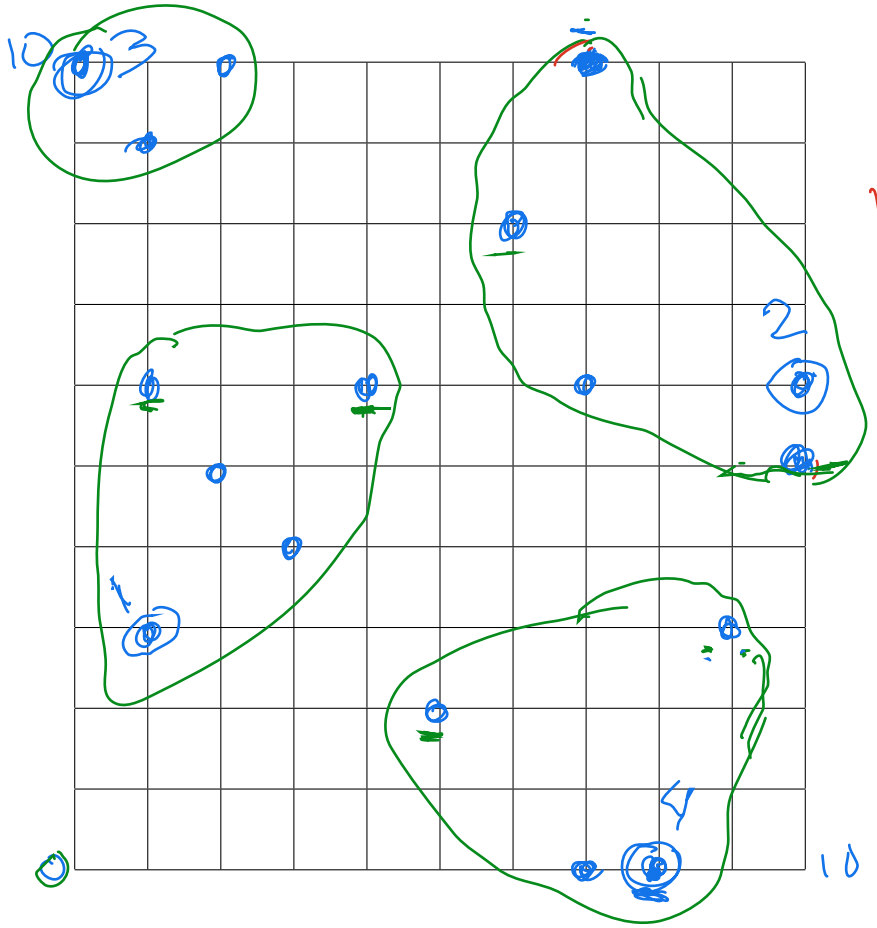
Return $\mathcal{C}$

Example 4.6. Draw the set

$$S = \{(0,10),(1,3),(1,6),(1,9),(2,5),(2,10),(3,4),(4,6),(5,2),(6,8),$$

$$(7,1),(7,6),(7,10),(8,0),(9,3),(10,5),(10,6)\}$$

of points and apply the algorithm to S and $k = 4$. Select $(1,3)$ as the first center.



Theorem 4.7. The `Clustering` algorithm has an approximation ratio that is bounded by 2.

Proof. We may assume that $|S| > k$, since other wise the algorithm returns an optimal set of clusters, each with diameter equal to zero. Let P be the set of centers and consider the point x that satisfies

$$x = \arg\max_{y \in S} d(y, P).$$

In other words, x would be the next center chosen if $k+1$ clusters were sought. Put another way, x is the point that is furthest from its cluster center.

Fact 1. Let $r = d(x, P)$ denote the distance from x to its cluster center. Then every other point must be within r of a center since x is the point that is furthest from its center.

Fact 2. By the triangle inequality, for two points y and z that belong to a cluster with center c, we have

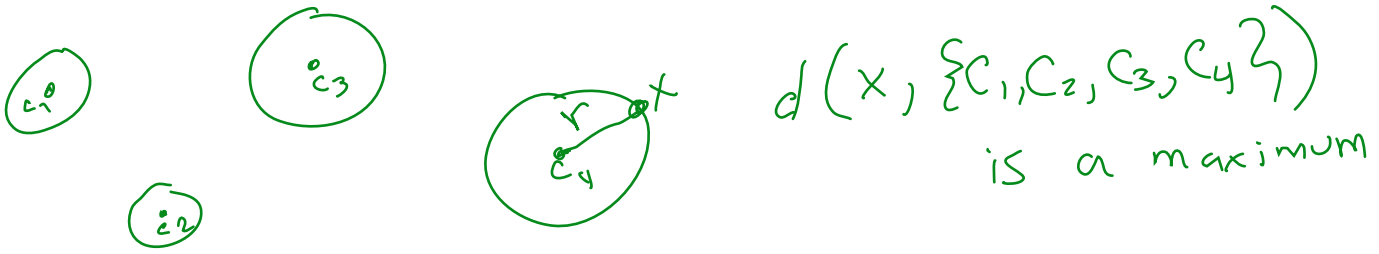
$$d(y, z) \leq d(y, c) + d(c, z) \leq r + r = 2r,$$

and so every cluster has a diameter $\leq 2r$.

$|P \cup \{x\}| = k+1$

Fact 3. For each pair of points $y, z \in P \cup \{x\}$, $d(y, z) \geq r$. This is because $d(x, P) \geq r$ and, each point y added to P in the `while`-loop must have satisfied the predicate $d(y, P) \geq r$ since this predicate is satisfied by x, but x was never chosen, meaning that, if y was chosen instead, y must also satisfiy the predicate since the chosen point is the one which maximizes $d(, P)$.

Fact 4. Finally, any k-partition of S must put at least two points from $P \cup \{x\}$ into the same cluster which thus will have a diameter of at least r. □



5 An approximation algorithm for Load Balancing

An instance of the Load Balancing (LB) optimization problem consists of a positive integer p, representing the number of processors, and a sequence $t_1, \ldots, t_n$ of times, where t_i denotes the amount of time needed for a processor to process the i th task. Each task is assigned to one of the processors and the processors work independently and in parallel in an attempt to complete all the tasks. Then the objective is to minimize the maximum time it takes for any one processor to complete all its assigned tasks.

We now describe an approximation algorithm for LB. To begin, let

$$T_{\text{tot}} = \sum_{i=1}^{n} t_i$$

denote the sum of all the task processing times. Let $\eta \in (0, 1)$ be a constant, and call a task/time t_i *large* iff $t_i \geq \eta T_{\text{tot}}$, and *small* otherwise. Note that there are at most $1/\eta$ large tasks (why?).

The first step of the approximation algorithm is to determine an optimal load balance in the case where only the large tasks are processed. Now, since there are a constant number of different ways to schedule the large tasks, this step can be done in constant time. Note however that this constant has a maximum value of $p^{\frac{1}{\eta}}$ which can become very large when η is small.

The final step is to assign each of the small tasks, where the next small task is assigned to the processor that currently has been assigned the smallest load. Thus, the final step makes use of a greedy heuristic.

Example 5.1. Demonstrate the approximation algorithm for LB using the following tasks with $p = 3$ processors and $\eta = 1/12$.

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Duration	1	7	7	6	7	8	6	2	8	10	4	5	10	9	6

$T_{tot} = 96$

$\frac{1}{12} \cdot T_{tot} = \frac{96}{12} = 8$ Large Tasks: 8, 8, 10, 10, 9

Phase 1: Optimize large tasks

P_1: 10 8

P_2: 10

P_3: 9 8

Longest processing time = 18

Phase 2:

P_1: 10 8 6 6

P_2: 10 7 7 4 6

P_3: 9 8 7 2 5

Longest Processing Time = 35

Theorem 5.2. The above described algorithm provides a $(1 + \eta(p-1))$-approximation for Load Balancing.

Proof.

1. M_{opt} denotes the optimal finish time for the last processor that completes its tasks.
2. M_l denotes the optimal finish time when only large tasks are scheduled.
3. Note: when adding the small tasks, if the latest processor finish time is not increased, then necessarily $M_{\text{opt}} = M_l$ and the algorithm returns an optimal solution.
4. Assume that the latest finish time equals M and that $M > M_l$.
5. T_i denotes the finish time of processor i after the final step.
6. Without loss of generality, assume $T_1 = M \geq T_2 \geq \cdots \geq T_p$. Then at least one small task must have been added to P_1 (why?).
7. t denotes the processing time for the last task that was added to P_1.
8. At the time the last task was assigned to P_1, it must have been true that
$$M \geq T_j \geq M - t$$
for all $j = 2, \ldots, p$.
9. Since $t < \eta T_{\text{tot}}$, we have
$$T_{\text{tot}} = M + \sum_{i=2}^{p} T_i \geq M + (p-1)(M-t) = pM - (p-1)t \geq pM - (p-1)\eta T_{\text{tot}}.$$
10. Solving for M yields the inequality
$$M \leq \frac{T_{\text{tot}}}{p}(1 + \eta(p-1)).$$
11. Also,
$$M_{\text{opt}} \geq \frac{T_{\text{tot}}}{p},$$
since the righthand side represents the best-case scenario in which tasks are divided equally (in terms of processing time) by each of the processors and all processors work the same amount of time, and with no idle time.
12. Therefore,
$$M \leq M_{\text{opt}}(1 + \eta(p-1))$$
which proves the theorem.
13. Note also that $M \geq pM_{\text{opt}}$ (why?), and so the two inequalities can be combined to get
$$p \leq \frac{M}{M_{\text{opt}}} \leq (1 + \eta(p-1)).$$

□

The LB approximation algorithm above is what is called a **polynomial-time approximation scheme (pas)**, since i) LB approximation runs in a polynomial number of steps with respect to the size of the problem instance, and ii) for every $\varepsilon > 0$ we have

$$M \leq (1 + \varepsilon) M_{\text{opt}}$$

so long as we set

$$\eta = \varepsilon/(p - 1).$$

A **fully polynomial-time approximation scheme (fpas)** means that the algorithm runs in time polynomial with respect to both the size of the problem instance, *and* with respect to $1/\varepsilon$. The LB approximation algorithm is *not* an fpas, since the first step of the algorithm requires time $p^{1/\eta}$, and so grows exponentially with respect to $1/\eta$ which is proportional to $1/\varepsilon$.

6 An approximation algorithm for Traveling Salesperson

Recall that an input to the Traveling Salesperson (TSP) optimization problem is a complete weighted graph $G = (V, E, c)$, and the problem is to find a Hamilton cycle whose edge weights sum to a minimum value. We first prove the following.

Theorem 6.1. If there is a polynomial-time ρ-approximation algorithm for TSP, then $\mathtt{P} = \mathtt{NP}$.

Proof. Suppose there is a polynomial-time ρ-approximation algorithm $\mathcal{A}$ for TSP. In other words, there is a constant $\rho \geq 0$ for which

$$\max_{G \in \mathtt{TSP}} \frac{\mathcal{A}(G)}{\mathrm{OPT}(G)} \leq 1 + \rho.$$

Let $G = (V, E)$ be a simple graph. To decide if G has a Hamilton cycle, we define the weighted graph $G' = (V, E \cup \overline{E}, c)$, where

$$c(e) = \begin{cases} 1 & \text{if } e \in E \\ 1 + 1.5\rho n & \text{otherwise} \end{cases}$$

where $n = |V|$. Then G has a Hamilton cycle iff G' has a cycle with minimum weight sum equal to n. Otherwise, the minimum weight sum of any Hamilton cycle is at least

$$n + 1.5\rho n = n(1 + 1.5\rho).$$

Thus, a polynomial-time ρ-approximation algorithm $\mathcal{A}$ would be able to distinguish between the two cases. For example, if G has a Hamilton Cycle, then we must have

$$\mathcal{A}(G') \leq (1 + \rho)\mathrm{OPT}(G) = (1 + \rho)n < (1 + 1.5\rho)n.$$

Inversely, if G does not have a Hamilton Cycle, then

$$\mathcal{A}(G') \geq \mathrm{OPT}(G) \geq (1 + 1.5\rho)n$$

since the optimal solution cannot be lower than this value since it must use at least one edge that is not in G's set of edges E.

Thus G has a Hamilton cycle iff $\mathcal{A}(G')$ returns a value that is less than $(1 + 1.5\rho)n$, which implies $\mathrm{HC} \in \mathtt{P}$ via $\mathcal{A}$. But if an $\mathtt{NP}$-complete problem is in $\mathtt{P}$, then so is every other $\mathtt{NP}$ problem. Therefore, $\mathtt{P} = \mathtt{NP}$. □

It turns out that we can do better if we assume that the vertices of G lie in a **metric space**.

The following proposition provides the key idea behind the 2-approximation algorithm described below. We leave its proof as an exercise.

Proposition 6.2. Let G be a complete weighted graph whose edge weights satisfy the triangle inequality. Then given a path $P = i_1, \ldots, i_n$, $n \geq 2$, we have

$$d(i_1, i_n) \leq \sum_{j=1}^{n-1} d(i_j, i_{j+1}).$$

The following algorithm may be used to obtain a 2-approximation for TSP, assuming the input graph G's edge weights satisfy the triangle inequality. First note that, if T is a minimum spanning tree for G, then the minimum-cost tour is at least equal to the cost of T. This is because any tour forms a single branch (and perhaps non-optimal) spanning tree from the start vertex to the final vertex. Moreover, the tour incurs the additional cost of returning to the start vertex from the final vertex. Now consider a path P that traverses only edges of T, visits every vertex (perhaps more than once), and returns to the start vertex. This can be accomplished by treating the start vertex as the root of T, performing a depth-first traversal along T, and eventually returning to the root vertex. The key observation about this depth-first traversal is that each tree edge is traversed exactly twice. Thus, the cost of P is twice the cost of T. Thus, P provides a 2-approximation of the minimum cost, but unfortunately P is not a Hamilton cycle. However, P can be converted to a Hamilton cycle without increasing the tour cost! To convert P to a Hamilton cycle, suppose $i_1, \ldots, i_k$, $k \geq 3$, is a subpath of P for which

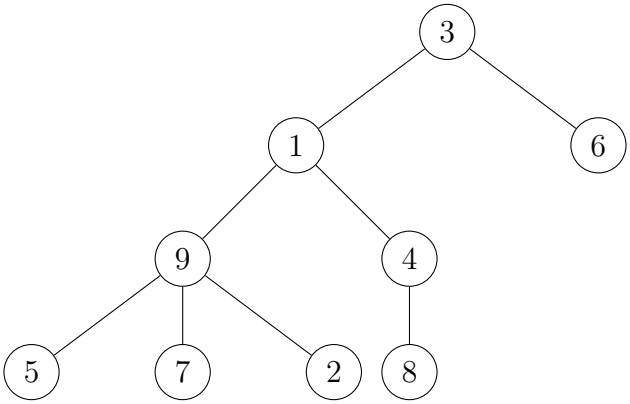
1. vertex i_1 is making its first appearance in P,
2. vertices $i_2, \ldots, i_{k-1}$ have already been visited in P, and
3. either i_k has yet to be visited in P, or i_k is the final vertex of P (and hence equals the start/root vertex).

Then we may replace the edges $(i_1, i_2), \ldots, (i_{k-1}, i_k)$ with the edge (i_1, i_k), and, by Theorem 5, we have

$$d(i_1, i_k) \leq \sum_{j=1}^{k-1} d(i_j, i_{j+1}),$$

and so the revised path is no more costly than the original one. Finally, repeatedly replacing these subpaths eventually results in a Hamilton cycle which is a 2-approximation of the optimal Hamilton cycle.

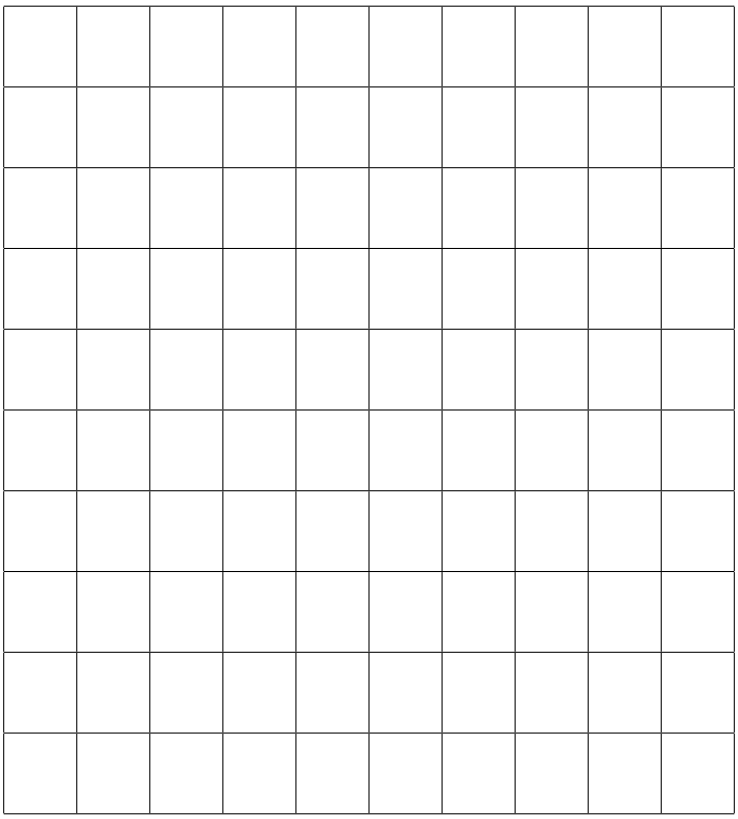
Example 6.3. Suppose G is a weighted complete graph with vertex set $V = \{1, \ldots, 9\}$, and for which a minimum spanning tree is shown below. Assuming the depth-first traversal visits children in the order from left to right, determine the Hamilton cycle produced by the above 2-approximation algorithm for TSP.



Example 6.4. A delivery truck starts at intersection $A = (0, 0)$, and must deliver packages to intersections

$B = (1, 7), C = (3, 6), D = (4, 8), E = (4, 9), F = (5, 7), G = (7, 1), H = (8, 6), I = (8, 10)$ and $J = (9, 7)$.

Provide a Hamilton cycle tour for the driver, using the 2-approximation algorithm. Compute the total cost of the tour. Assume that vertex distances are measured via the "Taxi-Cab" metric: horizontal plus vertical distance along the city grid, and that each grid segment takes unit time to traverse.



An improved TSP approximation algorithm

Given a complete weighted graph $G = (V, E, c)$, Christophides's algorithm also uses a minimum-spanning tree T as the basis for an approximation algorithm for TSP. However, instead of performing a depth-first traversal on T, his algorithm instead adds edges to T so that every vertex has even degree. Moreover, since there are an even number of odd-degree vertices, this can be accomplished in polynomial time by finding a minimum-cost matching M over the odd-degree vertices of T. Let $U = \{u_1, \ldots, u_t\}$ denote the odd-degree vertices of T, and G_U denote the subgraph of G that is induced by U. Then M is a minimum-cost matching for G_U.

Now, once the M-edges have been added, one can perform an Euler tour, and then convert that tour to a Hamilton cycle by skipping already-visited vertices, as was done in the previous algorithm. Futhermore the cost of this cycle is no more than $\text{cost}(T) + \text{cost}(M)$. From the previous algorithm we already know that $\text{cost}(T) \leq C_{\text{opt}}$, where C_{opt} is the minimum cost of any TSP tour. What about $\text{cost}(M)$? Consider a tour of G_U that uses the min-cost TSP tour of G, but skips all vertices that are not in U, then this tour cost does not exceed C_{opt}. Without loss of generality, assume that

$$u_1, u_2 \ldots, u_{t-1}, u_t, u_1$$

is such a tour. Then this tour consists of two different matchings for G_U, namely $M_1 = \{(u_1, u_2), \ldots, (u_{t-1}, u_t)\}$, and $M_2 = \{(u_2, u_3), \ldots, (u_t, u_1)\}$. Thus, since

$$\text{cost}(M_1) + \text{cost}(M_2) \leq C_{\text{opt}},$$

and M is a minimum-cost matching for G_U, it follows that

$$\text{cost}(M) \leq \min(\text{cost}(M_1), \text{cost}(M_2)) \leq C_{\text{opt}}/2,$$

where the last inequality is due to the arithmetic fact that, if two numbers sum to a third number, then one of the two numbers in the sum cannot exceed one-half of the sum. Therefore

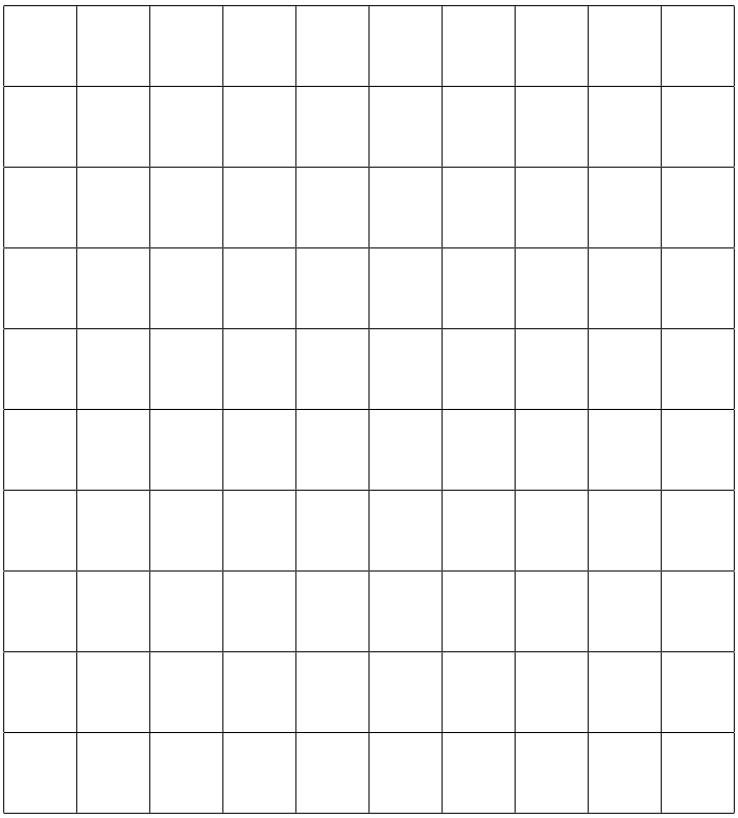
$$\text{cost}(T) + \text{cost}(M) \leq 3/2 C_{\text{opt}},$$

and Christophides's algorithm is a 3/2-approximation algorithm for TSP.

Example 6.5. A delivery truck starts at intersection $(0, 0)$, and must deliver packages to intersections

$B = (1, 7), C = (3, 6), D = (4, 8), E = (4, 9), F = (5, 7), G = (7, 1), H = (8, 6), I = (8, 10)$ and $J = (9, 7)$.

Provide a tour for the driver that uses the Christofides' algorithms. Compute the total cost of the tour. Assume that vertex distances are measured via the "Taxi-Cab" metric: horizontal plus vertical distance along the city grid, and that each grid segment takes unit time to traverse.



Exercises.

1. Prove Theorem 5. Hint: use induction.

Exercise Hints and Answers.

1. The proof is by induction on $n \geq 2$. The statement is trivial for $n = 2$, and is a re-statement of the triangle inequality for $n = 3$. Now assume it is true for any path that traverses $n-1$ vertices, where $n \geq 4$, and consider path $P = i_1, \ldots, i_{n-1}, i_n$. Then, by the inductive assumption, we have

$$d(i_1, i_{n-1}) \leq \sum_{j=1}^{n-2} d(i_j, i_{j+1}).$$

Thus, by the triangle inequality,

$$d(i_1, i_n) \leq d(i_1, i_{n-1}) + d(i_{n-1}, i_n) \leq \sum_{j=1}^{n-2} d(i_j, i_{j+1}) + d(i_{n-1}, i_n) = \sum_{j=1}^{n-1} d(i_j, i_{j+1}),$$

and the theorem is proved.