1 Vertex Cover

Input: An undirected graph $G(V, E)$.
Output: A set of vertices $S \subseteq V$, such that $\forall e(u, v) \in E$ at least one of u or $v \in S$ and $|S|$ is minimum.

Lemma 1.1. *Let M be any maximal matching. If OPT is the value of the optimal solution, then $|M| \leq$ OPT.*

Proof. For each edge in M, at least one of its end points must belong to any vertex cover. □

The algorithm below returns a vertex cover.

Input: A graph $G(V, E)$
Output: A set of vertices $S \subseteq V$ such that S is a vertex cover of G.
1 $S = \phi$
2 Compute a maximal matching M
3 **foreach** *edge* $e(u, v) \in M$ **do**
4 $S = S \cup \{u, v\}$
5 **end**
6 **return** S

Lemma 1.2. *The above algorithm correctly returns a vertex cover of G.*

Proof. Assume the opposite, that there exists an edge e such that none of its end points belong to S. However, if that were true, then M would not be a maximal matching, since e can be added to M. A contradiction. □

Theorem 1.3. *The above algorithm gives a 2-approximation guarantee.*

Proof. $|S| \leq 2 \cdot |M| \leq 2 \cdot$OPT □

Another algorithm to compute vertex cover of a graph.

Input: A graph $G(V, E)$
Output: A set of vertices $S \subseteq V$ such that S is a vertex cover of G.
1 $S = \phi$
2 **while** *E is not empty* **do**
3 Find a vertex $u \in V$ having the highest degree
4 $S = S \cup \{u\}$
5 $E = E \setminus E'$ where $E' = \{e(u, v) | v \in V\}$
6 **end**
7 **return** S

Theorem 1.4. *The above algorithm gives an approximation guarantee of $O(\log n)$*

Proof. See the proof for Set Cover problem. □

2 k-Center Problem

We are given a complete graph $G(V, E)$ with positive weights on each edge. Let $w(u, v)$ denote the weight on edge $e(u, v)$. The weights on edges obey triangle inequality, that is, $w(u, w) \leq w(u, v) + w(v, w)$. In addition, the weights on edges are symmetric and $w(u, u) = 0, \forall u \in V$.

We define the distance of a vertex u from a set of vertices S to be $d(u, S) = \min_{v \in V} w(u, v)$. Given k, we wish to select a subset S of vertices of size k such that $R = \max_{u \in V} d(u, S)$ is as small as possible. The

algorithm below builds up the set S greedily.

```
1 S = {u}, where u is any arbitrary vertex
2 while |S| ≠ k do
3     Find a vertex v ∈ V \ S, such that d(v, S) is maximum
4     S = S ∪ {v}
5 end
6 return S
```

Theorem 2.1. *The above algorithm gives a 2-approximation guarantee.*

Proof. Suppose the set S contains the vertices $S = \{v_1, v_2, \ldots, v_k\}$ after the completion of the algorithm. Let v_{k+1} be the vertex that would have been added if the algorithm was allowed to run for one more iteration. It follows that the cost of our solution $R = d(v_{k+1}, S)$.

Let $S^* = \{u_1, u_2, \ldots, u_k\}$ be an optimal solution having cost R^*. Consider the set $\{v_1, v_2, \ldots, v_k, v_{k+1}\}$. This set has cardinality $k+1$. By pigeon hole principle, at least two of the vertices, say v_i and v_j, must be covered by the same center, say u_k, in S^*. Since $w(u_k, v_i) \leq R^*$ and $w(u_k, v_j) \leq R^*$, we obtain $w(v_i, v_j) \leq 2 \cdot R^*$.

Also, as the algorithm progresses, the distance of any vertex to the set S does not increase, since each time we add a vertex that has the maximum distance to the set S. It follows that $R \leq 2 \cdot R^*$. □

3 Set Cover

The Set Cover problem is based on a set U of n elements and a collection of m subsets, $S = \{S_1, S_2, \ldots, S_m\}$, of U. The goal is to select a smallest subset $C \subseteq S$, such that the union of all sets in C is equal to the set U.

We develop a greedy algorithm which in each iteration, chooses a set which covers the maximum number of uncovered elements till the current iteration.

```
1  C = φ
2  j = 1
3  U' = U
4  while U' ≠ φ do
5      S_i is a set in S such that |S_i ∩ U'| is maximum
6      Z_j = S_i ∩ U'
7      U' = U' \ Z_j
8      C = C ∪ {S_i}
9      j = j + 1
10 end
11 return C
```

Let n_j be the cardinality of U' at the beginning of the jth iteration. From the algorithm, $|Z_j|$ denotes the number of elements covered in the jth iteration. If k^* is the value of the optimal(C^*) solution, then the following lemma holds.

Lemma 3.1.

$$|Z_j| \geq \frac{n_j}{k^*}$$

Proof. Let $F_j = F \cap U_j$, where $F \in C^*$ and U_j is equal to U' at the beginning of the jth iteration.

$$\begin{aligned} n_j &\leq \sum_{F \in C^*} |F_j| \\ &\leq \sum_{F \in C^*} |Z_j| \\ &= k^* \cdot |Z_j| \end{aligned}$$

□

Theorem 3.2. *The above algorithm gives an approximation guarantee of* $O(\log n)$

Proof. Suppose our algorithm runs for k iterations. Then the cost of our solution is k. (Note that in each iteration, a new set S_i is added to C, ie $|C| = k$). Let $n_1, n_2, \ldots, n_k$ be the number of uncovered elements at the start of each successive iteration.

$$
\begin{aligned}
n_1 &= n \\
n_2 &= n_1 - |Z_1| \leq n_1 - \frac{n_1}{k^*} \leq n(1 - \frac{1}{k^*}) \\
n_3 &= n_2 - |Z_2| \leq n_2 - \frac{n_2}{k^*} \leq n(1 - \frac{1}{k^*})^2 \\
&\vdots \\
n_k &\leq n(1 - \frac{1}{k^*})^{k-1}
\end{aligned}
$$

Since our algorithm performed the kth iteration, we have $1 \leq n_k$.

$$1 \leq n(1 - \frac{1}{k^*})^{k-1}$$

Using the relation, $(1 + x) \leq e^x$,

$$1 \leq n \cdot e^{\frac{-(k-1)}{k^*}}$$

Taking log on both sides of the equation,

$$
\begin{aligned}
0 &\leq \frac{-(k-1)}{k^*} + \log n \\
k &\leq k^* \cdot \log n + 1 \\
k &= O(\log n) \cdot k^*
\end{aligned}
$$

□

We now modify the input to include a weight w_i associated with each set $S_i \in S$. The objective is to select a minimum cost cover, ie a subset C^* of S such that $\sum_{S_i \in C^*} w_i$ is minimized.

By analogy, this time our algorithm selects a set whose "cost per uncovered element" is the least. Using the same notation as above, if U' is the set of all uncovered elements in an iteration of the algorithm, we select a set S_i such that $w_i/|Z_i|$ is *minimum*, where $Z_i = S_i \cap U'$. Thus if $w_1, w_2, \ldots, w_k$ are the weights of the sets selected by our algorithm in each successive iteration, the cost of our solution,

$$k = \sum_{j=1}^{k} |w_j|$$

Lemma 3.3.

$$\frac{w_j}{|Z_j|} \leq \frac{k^*}{n_j}$$

Proof. At every step of our algorithm, we select the set S_i such that $w_i/|Z_i|$ is the least. Thus, for each $F \in C^*$

$$\frac{|F_j|}{w_f} \leq \frac{Z_j}{w_j}$$

$$
\begin{aligned}
n_j &\leq \sum_{F \in C^*} |F_j| \\
&\leq \sum_{F \in C^*} w_f \cdot \frac{|Z_j|}{w_j} \\
&= \frac{|Z_j|}{w_j} \cdot \sum_{F \in C^*} w_f \\
&= \frac{|Z_j|}{w_j} \cdot k^* \\
\frac{n_j}{k^*} &\leq \frac{|Z_j|}{w_j}
\end{aligned}
$$

□

Theorem 3.4. *The above algorithm gives an approximation guarantee of* $O(\log n)$

Proof.

$$
\begin{aligned}
&n_1 = n \\
&n_2 = n_1 - |Z_1| \leq n_1 - \frac{n_1 \cdot w_1}{k^*} \leq n_1(1 - \frac{w_1}{k^*}) \leq n_1 \cdot e^{\frac{-w_1}{k^*}} \leq n \cdot e^{\frac{-w_1}{k^*}} \\
&n_3 = n_2 - |Z_2| \leq n_2 - \frac{n_2 \cdot w_2}{k^*} \leq n_2(1 - \frac{w_2}{k^*}) \leq n_2 \cdot e^{\frac{-w_2}{k^*}} \leq n \cdot e^{\frac{-(w_1+w_2)}{k^*}} \\
&\vdots \\
&n_k \leq n \cdot e^{\frac{-(w_1+w_2+\ldots+w_{k-1})}{k^*}}
\end{aligned}
$$

Since our algorithm performed the kth iteration, we have $1 \leq n_k$.

$$1 \leq n \cdot e^{\frac{-(w_1+w_2+\ldots+w_{k-1})}{k^*}}$$

Taking log on both sides of the equation,

$$
\begin{aligned}
0 &\leq \log n - \frac{w_1 + w_2 + \ldots + w_{k-1}}{k^*} \\
w_1 + w_2 + \ldots + w_{k-1} &\leq \log n \cdot k^*
\end{aligned}
$$

In the kth iteration, $|Z_k| = n_k$. Thus, $w_k \leq k^*$.

$$
\begin{aligned}
w_1 + w_2 + \ldots + w_{k-1} + w_k &\leq \log n \cdot k^* + k^* \\
k &\leq O(\log n) \cdot k^*
\end{aligned}
$$

□

4 Travelling Salesman Problem

Input: A complete graph $G(V, E)$ with weights on edges that satisfy the triangle inequality.
Output: A minimum cost tour that visits each vertex exactly once.

Lemma 4.1. *Let C be the cost of the minimum spanning tree of G. If C^* is the cost of the optimal(OPT) solution, then $C \leq C^*$.*

Proof. Consider the optimal tour. If we remove an edge from this tour, we get a spanning tree of G. The cost of this spanning tree is at least the cost of the minimum spanning tree of G. Thus it follows that $C \leq C^*$. □

If we have a minimum spanning tree T of G, we can convert it to an Eulerian graph by doubling all edges(so that the degree of all vertices become even). The cost of the Eulerian tour is then equal to $2C$. This cost is the upper bound on the cost of our travelling salesman tour. The complete algorithm follows.

```
1 T = minimum spanning tree of G
2 Double all the edges in T
3 Compute an Eulerian tour of T. Let O = {v_0, v_1, ..., v_0} be the sequence of vertices in the Eulerian tour.
4 Some vertices in O may be repeated; remove all but the first occurrence of such vertices.
5 Return O
```

Theorem 4.2. *The above algorithm gives a 2-approximation guarantee.*

The previous algorithm constructs an Eulerian graph by simply doubling all the edges to make the degree of all vertices in T even. In order to get a better approximation guarantee, we should be more careful in making the Eulerian graph.

In any graph there are even number of vertices having odd degree. Therefore, in T we can compute the min-cost perfect matching of all such odd degree vertices and add the edges of the matching to T. The resulting graph is now Eulerian, since the degree of each previously odd degree vertex in T increases by 1. We now compute the Eulerian tour as before and use it to upper bound the cost of the travelling salesman tour.

Theorem 4.3. *The above approach gives a 1.5-approximation guarantee.*

Proof. Let C be the cost of the Eulerian tour. Let $\{v_1, v_2, \ldots, v_{2k}\}$ be the odd degree vertices in T. These vertices lie somewhere on the tour. Consider the two matchings, $M_1 = \{(v_1, v_2), (v_3, v_4), \ldots\}$ and $M_2 = \{(v_2, v_3), (v_4, v_5), \ldots\}$ having costs C_1 and C_2 respectively. We have,

$$C_1 + C_2 \leq C \leq C^*$$

Without loss of generality, let $C_1 \leq C_2$. Then, $C_1 \leq C^*/2$. Since we computed the min-cost matching, cost of our matching is at least $C^*/2$. The cost of the Eulerian graph then is at least $3/2C^*$. □

5 Job Scheduling on a Single Machine

Input: A set of n jobs, with each job i having release time r_i, processing time p_i and deadline d_i.
Output: A schedule that has the least maximum lateness.

Theorem 5.1. *For any instance of the job scheduling problem, it is NP-hard to decide whether $L^*_{\max} \leq 0$.*

Theorem 5.2. *It is not possible to get an α-approximation for the job scheduling problem mentioned above unless $P = NP$.*

Proof. Suppose that an α-approximation for the job scheduling problem existed. If $L^*_{\max} \leq 0$, for that instance, we would report some negative number, and if $L^*_{\max} > 0$, we would report some positive number. Our polynomial time approximation algorithm would then be able to correctly answer whether for a given instance $L^*_{\max} \leq 0$. A contradiction. □

We modify the input to make the problem slightly "less hard". It is now given that $\forall i, d_i < 0$ and $r_i \geq 0$. Thus we know that $L^*_{\max} > 0$. The following lemma gives a lower bound on the value of $L^*_{\max}$.

Lemma 5.3. *If S is a set of jobs,*

$$L^*_{\max} \geq r_{\min}(S) + p(S) - d_{\max}(S)$$

where $r_{\min}(S)$ is the earliest release time of a job in S, $p(S)$ is the sum of the processing time of all jobs in S and $d_{\max}(S)$ is the latest deadline of a job in S.

We use the following algorithm to schedule jobs in S: At any instant when the machine is idle, amongst all the jobs that are released(and not yet scheduled), we choose an *arbitrary* job and schedule it.

Theorem 5.4. *The above algorithm gives a 2-approximation guarantee.*

Proof. Let job j be the one that defines maximum lateness $L_{\max}$ of our solution, ie $L_{\max} = c_j - d_j$ where c_j is its completion time. We look at the state of the machine backwards in time starting at c_j. Let t be the first instant at which we find the machine to be idle during our backward scan. Let S_j be the set of jobs that we encountered during the scan. Clearly none of the jobs in S_j were released before t (otherwise the machine would not have been idle).

$$\begin{aligned} L_{\max} &= c_j - d_j \\ &= t + p(S_j) - d_j \\ &= r_{\min}(S_j) + p(S_j) - d_j \\ &\leq L^*_{\max} - d_j \\ &\leq 2 \cdot L^*_{\max} \end{aligned}$$

□

6 Job Scheduling on Multiple Machines

Input: A set of n jobs with each job i having processing time p_i and a set of m identical machines.
Output: A schedule that has the least makespan.

Lemma 6.1. *If L^* is the value of the optimal solution, then $L^* \geq \frac{\sum_i p_i}{m}$ and $L^* \geq \max p_i$.*

Our first algorithm works as: If a machine is idle, schedule any *arbitrary* job on it.

Theorem 6.2. *The above algorithm gives a 2-approximation guarantee.*

Proof. Let L be the cost of our solution, and let p_l be the processing time of the job that defined it. Suppose that p_l was scheduled at time t. Thus $t \leq \frac{\sum_i p_i}{m}$.

$$\begin{aligned} L &= t + p_l \\ &\leq \frac{\sum_i p_i}{m} + p_l \\ &\leq L^* + p_l \\ &\leq 2 \cdot L^* \end{aligned}$$

□