CSE 331: Introduction to Algorithm Analysis and Design

Greedy Algorithms

1 Interval Scheduling: Maximize Intervals Scheduled

1.1 Problem

Input: A set of n intervals $I = \{i_0, i_1, \ldots, i_{n-1}\}$ such that each i_j is defined by its start time $s(i_j)$ and finish time $f(i_j)$ as the interval $[s(i_j), f(i_j))$. Note that the interval does not contain $f(i_j)$ so an interval can be scheduled with a start time equal to the previous tasks finish time.

Output: A schedule A of maximum size such that $A \subseteq I$ and no two intervals in A conflict.

1.2 Algorithm

1. Sort the intervals in I by increasing finish time
2. Initialize $A = \emptyset$
3. Iterate through the intervals in I
   (a) If the current interval does not conflict with any interval in A, add it to A
4. return A as the maximum set of scheduled intervals

1.3 Correctness

Greedy stays ahead: This is the first of two proofs techniques we will see for greedy algorithms. The idea is to prove that any point during execution, the algorithm has found the best possible solution on the sub-problem containing only the section of the input that has already been iterated over.

Proof. For this proof, we will assume we have an optimal solution $\mathcal{O} = \{j_0, j_1, \ldots, j_{m-1}\}$ and a solution from the algorithm $A = \{i_0, i_1, \ldots, i_{k-1}\}$. The optimal solution could be any arbitrary solution as long as it maximizes the number of intervals scheduled. We will prove that $|A| = |\mathcal{O}|$ which is equivalent to $k = m$.

We will assume that both A and $\mathcal{O}$ are sorted by the time of the intervals. Since there are no conflicts, sorting by start time or finish time will result in the same order. Now we want to prove that $f(i_r) \leq f(j_r)$ for all r by induction. This is the staying ahead part of the proof. We are proving that the greedy algorithm is always ahead (behind in this case) in terms of finish time. This can also be thought of as the greedy solution having more time remaining to schedule other tasks after the earliest $r + 1$ tasks are scheduled.

Base case: At $r = 0$, each algorithm has only a single task scheduled. Since the greedy algorithm chooses the task with the earliest finish time, we must have $f(i_0) \leq f(j_0)$. The interval i_0 must be

the interval with the earliest finish time so there is no possible interval that can be in $\mathscr{O}$ with an earlier finish time.

Induction step: We will assume $f(i_{r-1}) \leq f(j_{r-1})$ and prove $f(i_r) \leq f(j_r)$. We will assume $f(i_r) > f(j_r)$ and show that this leads to a contradiction. Since j_r is the next interval in the sorted schedule $\mathscr{O}$, we know that $s(j_r) \geq f(j_{r-1})$ and doesn't conflict any intervals in $\mathscr{O}$. Since $f(i_{r-1}) \leq f(j_{r-1})$, we know that j_r also doesn't conflict with any intervals already in the schedule A. Since j_r doesn't conflict with A and it has an earlier finish time than i_r, the algorithm would have chosen j_r over i_r. This means the algorithm didn't choose the earliest finish time interval with no conflicts which is a contradiction. Therefor, our assumption is false and $f(i_r) \leq f(j_r)$ as desired.

We have proven by induction that $f(i_r) \leq f(j_r)$ for all r, but this is not enough to prove the algorithms correctness. It is important to use this fact as a lemma to finish the proof. We have $|A| = k$ and $|\mathscr{O}| = m$ and we are proving that $k = m$. Since we assume that $\mathscr{O}$ is optimal, we can't have $k > m$, so we just need to show that $m \not> k$ which we will do by contradiction.

Assume that $m > k$. We know that at $r = k - 1$ (k intervals scheduled) that $f(i_{k-1}) \leq f(j_{k-1})$ and that $\mathscr{O}$ has at least one more interval in its schedule j_k. By the same reasoning as the induction step earlier in this proof, the greedy algorithm would add j_k to its schedule since it doesn't conflict with any previously scheduled algorithm. This implies that $|A| \neq k$ which is a contradiction. Our assumption must have been false and $k = m$. Since the greedy solution has the same number of intervals scheduled as an optimal solution, the greedy solution must be optimal.

□

1.4 Runtime

1. Sort the intervals in I by increasing finish time: Sorting n elements can be performed in $O(n \cdot log(n))$ time.
2. Initialize $A = \emptyset$: $O(1)$
3. Iterate through the intervals in I: This loop will iterate through all n intervals, visiting each exactly once for $O(n)$ total iterations.
   (a) If the current interval does not conflict with any interval in A, add it to A: By tracking the finish time of the most recently scheduled interval, this check can be done with one comparison of the previous finish time and the current intervals start time. This comparison takes $O(1)$ time.
4. return A as the maximum set of scheduled intervals: $O(1)$

This gives a total runtime of $O(n \cdot log(n)) + O(1) + O(n) \cdot O(1) + O(1)$ which is equivalent to $O(n \cdot log(n))$.

2 Interval Scheduling: Minimize Maximum Lateness

2.1 Problem

Input: A set of n intervals $I = \{i_0, i_1, \ldots, i_{n-1}\}$ such that each i_j is defined by its deadline $d(i_j)$ and duration $t(i_j)$. The input also includes a global start time s which we will assume is 0. If $s \neq 0$ we can shift s and all the deadlines to make $s = 0$ so we will use this assumption to make things simpler.

Output: A schedule of all n tasks with no conflicts that minimizes the maximum lateness. For the schedule, each task is given a start time $s(i_j)$ and we define a finish time $f(i_j) = s(i_j) + d(i_j)$. The lateness of a task is defined as $l(i_j) = max(f(i_j) - d(i_j), 0)$ and the maximum lateness of a schedule is the lateness of the latest task $L = \max_{0 \leq j \leq n-1} l(i_j)$. The algorithm should output a schedule that minimizes L.

2.2 Algorithm

1. Sort the tasks in I by increasing deadline
2. Initialize the current finish time to the global start time $f = s = 0$
3. Iterate through the tasks in I. Call the current task i_j
   (a) Set the start time of i_j to the current finish time $s(i_j) = f$
   (b) Compute the finish time of i_j to $f(i_j) = s(i_j) + t(i_j)$
   (c) Update the finish time to include the current task $f = f + t(i_j)$
4. return the resulting schedule

2.3 Correctness

Exchange argument: We will prove the correctness of the algorithm by proving the following three properties. We define an inversion as two tasks i_j and $i_{j'}$ in a schedule such that i_j is scheduled before $i_{j'}$ and $d(i_{j'}) < d(i_j)$:

- Any two schedules with 0 idle time and 0 inversions have the same max lateness.
- Greedy schedule has 0 idle time and 0 inversions.
- There is an optimal schedule with 0 idle time and 0 inversions.

If all three of these properties are true, we know there is an optimal solution with 0 idle time and 0 inversions and that the greedy solution has 0 idle time and 0 inversions. Since any two solution that both have 0 idle time and 0 inversions have the same maximum lateness, the greedy solution must be optimal.

Lemma 1. *Any two schedules with 0 idle time and 0 inversions have the same max lateness.*

Proof. Let S_0 and S_1 be two arbitrary schedules which both have 0 idle time and 0 inversions. From this, we know that both schedules performs tasks consecutively and that for any two tasks such that $d(i_j) < d(i_{j'})$, that task i_j must be scheduled before $i_{j'}$ in both S_0 and S_1.

This implies that the schedules must be identical except for tasks such that $d(i_k) = d(i_{k'})$. If the deadlines are identical, the tasks are scheduled consecutively but in arbitrary order. We will show that this order does not affect the maximum lateness of the schedule. For completeness, our argument can be applied to an arbitrary number of tasks all with the same deadline, but for clarity we present the argument with just two tasks.

For two tasks i_k and $i_{k'}$ such that $d(i_k) = d(i_{k'}) = d'$ switching the tasks doesn't change the maximum lateness of the two tasks. No matter which task is scheduled first, the end time of the later task will be $t_f = f' + t(i_k) + t(i_{k'})$ where f' is the finish time of the previously scheduled task. Given this the maximum lateness of these tasks is $max(t_f - d', 0)$ which is the same regardless of the order of i_k and $i_{k'}$. To expand this proof to an arbitrary number of tasks with the same deadline, the argument is the same except $t_f = f' + \sum t(i_j)$ for all j in the set of tasks with the same deadline.

□

Lemma 2. *The greedy schedule has 0 idle time and 0 inversions.*

Proof. This follows directly from the definition of the algorithm. Since the algorithm schedules each task to start at the end of the previously scheduled task, the resulting schedule will have 0 idle time. The algorithm schedules the tasks in order of increasing deadline, so there are no inversions. □

Lemma 3. *There is an optimal schedule with 0 idle time and 0 inversions.*

Proof. Here we will apply the core of the exchange argument. Having 0 idle time and 0 inversions is the fundamental property that a schedule from the greedy algorithm will have. For this proof we will assume there is an optimal solution that does not have 0 idle time and 0 inversions and prove that it's maximum lateness L can't increase by adding this property. We will show this in two part, one for removing idle time and one for removing inversions.

Removing idle time from the optimal schedule can not increase the maximum lateness of the schedule. Removing idle time before a task will decrease the finish time for that task. By the equation for the lateness of a task $l(i_j) = max(f(i_j) - d(i_j), 0)$, we can see that decreasing $f(i_j)$ can't increase $l(i_j)$.

Removing an inversion from the optimal solution can not increase the maximum lateness of the schedule. If a schedule has any inversions, there must exist two consecutive tasks that are inverted that we'll call i_p and i_{p+1}. We will switch the order of these tasks and show that it can't increase the maximum lateness of the schedule. We know that $d(i_{p+1}) < d(i_p)$ and that switching the tasks does not change the finish time of the later scheduled task which we'll call f. Before the switch, the lateness of the tasks are $l(i_p) = max(f - t(i_{p+1}) - d(i_p), 0)$ and $l(i_{p+1}) = max(f - d(i_{p+1}), 0)$ making the maximum lateness between the two task $l = max(f - t(i_{p+1}) - d(i_p), f - d(i_{p+1}), 0)$. After switching the tasks, the maximum lateness is $l' = max(f - d(i_p), f - t(i_p) - d(i_{p+1}), 0)$ and we need to show that $l' \leq l$. We know that $f - t(i_p) - d(i_{p+1}) < f - d(i_{p+1})$ so we only need to show that $f - d(i_p) \leq l$ which we will do by showing that $f - d(i_p) \leq f - d(i_{p+1})$. We can rewrite this inequality as follows:

$$f - d(i_p) \leq f - d(i_{p+1})$$
$$-d(i_p) \leq -d(i_{p+1})$$

$$d(i_p) > d(i_{p+1})$$

Which is true from the definition of an inversion.

By fixing all such inversions, we will arrive at a schedule with 0 inversions that could only be more optimal. Since the solution was already assumed to be optimal, it must still be optimal now that it has 0 idle time and 0 inversions which completes the proof.

□

2.4 Runtime

1. Sort the tasks in I by increasing deadline: Sorting n elements in $O(n \cdot log(n))$ time.
2. Initialize the current finish time to the global start time $f = s = 0$: $O(1)$
3. Iterate through the tasks in I. Call the current task i_j: Iterate through all n tasks for $O(n)$ total iterations.
   (a) Set the start time of i_j to the current finish time $s(i_j) = f$: $O(1)$
   (b) Compute the finish time of i_j to $f(i_j) = s(i_j) + t(i_j)$: $O(1)$
   (c) Update the finish time to include the current task $f = f + t(i_j)$: $O(1)$
4. return the resulting schedule: $O(1)$

The total runtime is $O(n \cdot log(n)) + O(1) + O(n) \cdot (O(1) + O(1) + O(1)) + O(1)$ which is $O(n \cdot log(n))$.

3 Shortest Path With Weighted Edges

3.1 Problem

Input: A weighted graph $G = (V, E)$ (directed or undirected) and a starting node $s \in V$. Each edge $e \in E$ has a weight $l_e > 0$.

Output: The length of the shortest path from s to t for all $t \in V$.

3.2 Dijkstra's Algorithm

1. $d'(s) = 0$
2. $R = \{s\}$
3. While $\exists_{(u,v) \in E}$ where $u \in R \wedge v \notin R$
   (a) Choose v with the smallest $d'(v)$
   (b) $R = R \bigcup \{v\}$
   (c) $d(v) = d'(v)$
   (d) $d'(w) = \min(d'(w), d(v) + l_e)$ for each $e = (v, w)$
4. return d for all nodes

3.3 Correctness

Let P_v be the length of the shortest path from s to v.

We will prove the correctness of Dijkstra's algorithm by induction on the size of R throughout the run of the algorithm. Specifically, at any point in time during a run of the algorithm, for any node $v \in R$, $d(v)$ is the length of the shortest path from s to v.

Proof. **Base case:** For $|R| = 1$, the only node in R is s and $d'(s)$ is initialized to 0 meaning $d(0)$ ends up being 0 after the first execution of the loop. With no negative edges, we can't find a path shorter than 0 length.

Induction step: Assume that at any point during the run of the algorithm, $d(v)$ is the length of the shortest path from s to v for all $v \in R$. Prove that the next node w to be added to R that $d(w)$ will be the length of the shortest path from s to w.

Since w is the next node to be added to R, there must exist an edge $e = (u, w)$ such that $u \in R$. We want to prove that $d(u) + l_e$ is the length of the shortest path from s to w which we will do by contradiction. To this end, assume that there exists a path P from s to w with length less than $d(u) + l_e$. Since $s \in R$ and $w \notin R$, there must be an edge $e' = (x, y)$ in P such that $x \in R$ and $y \notin R$. Since the total length of P is less than $d(u) + l_e$, then we know that $d(x) + l_{e'} < d(u) + l_e$ which implies that $d'(y) < d'(w)$. This is a contradiction since the algorithm always chooses to add the node with the smallest d' value to R, but the algorithm chose to add w, not y.

□

3.4 Runtime

1. $d'(s) = 0$ $O(1)$
2. $R = \{s\}$ $O(1)$
3. While $\exists_{(u,v)\in E}$ where $u \in R \wedge v \notin R$ Each iteration of the loop adds a node to R making the total number of iterations $O(n')$ where n' is the size of the connected component of s. Since s can't be connected to more nodes than the number of edges in the graph, this loop also runs $O(m)$ times which we will use for the analysis.
   (a) Choose v with the smallest $d'(v)$ This can be preformed in $O(1)$ time with a priority queue implementation.
   (b) $R = R \bigcup \{v\}$ $O(1)$
   (c) $d(v) = d'(v)$ $O(1)$
   (d) $d'(w) = \min(d'(w), d(v) + l_e)$ for each $e = (v, w)$ With a priority queue implementation, this step can be performed in $\log(n)$ time to fix up the heap. Though the loop executes $O(n)$ times and this step can execute $O(m)$ times if v is connected to every edge in the graph, the total number of times d' values are updated is $O(m)$ since each edge is checked at most twice (see the total number of neighbors proof or the runtime analysis for BFS). This makes the total runtime of this step throughout the entire run of the algorithm $O(m \log n)$.
4. return d for all nodes $O(1)$

By utilizing a priority queue, the total runtime is $O(1) + O(1) + O(m) \cdot (O(1) + O(1) + O(1)) + O(m \log n) + O(1)$ which is $O(m \log n)$.

4 Minimum Spanning Tree

4.1 Problem

Input: A weighted undirected connected graph $G = (V, E)$ where each edge $e \in E$ has weight $l_e > 0$.

Output: A minimum spanning tree (MST) of G. Specifically, the MST is defined by (V, T) where $T \subseteq E$, all nodes are connected by the edges in T, and the sum of the edge weights in T is minimum ($\sum_{e \in T} l_e$ in minimized).

4.2 Prim's Algorithm

1. Pick a starting node s
2. Initialize $S = \{s\}$
3. Iteratively add the node with the least cost crossing edge $e = (u, v)$ to S such that $u \in S$ and $v \notin S$ with minimum l_e.

4.3 Kruskals's Algorithm

1. Sort the edges in E by increasing weight.
2. Initialize $T = \emptyset$
3. Iteratively add the the edges in E to T unless adding it would create a cycle.

4.4 Cut Property Lemma

Lemma 4. *Given a connected undirected graph $G = (V, E)$ with distinct edge weights and a cut $S \subset V$ such that $S \neq \emptyset$ and $V \setminus S \neq \emptyset$, then the crossing edge $e = (u, v)$ with $u \in S$ and $v \in V \setminus S$ with the smallest weight is in every MST of G.*

Proof. We will prove the cut property lemma by contradiction. For the contraction, we will assume that we have an MST of G defined by the edges T and for some cut S such that $S \neq \emptyset$ and $V \setminus S \neq \emptyset$, that the minimum weight crossing edge $e = (u, v)$ with $u \in S$ and $v \in V \setminus S$ is not in T ($e \notin T$). Since the MST is connected by T, we know that there must be at least one crossing edge between S and $V \setminus S$ in Tthat connects these disjoint sets of nodes[1].

We will then add the minimum crossing edge e to the set T. Since T defined a tree, we know that it is connected, has $n-1$ edges, and no cycles. By adding e to this set, we must have introduced a cycle that contains e. If we follow this cycle starting with e moving from u to v, it must end back at u. This implies that at some edge e', the cycle crosses S and $V \setminus S$ again. We then remove e'

[1]This proof will differ slightly from the one in class in that we will not label this crossing edge so we can avoid using a double prime later. Suffice to know that such an edge exists.

from T to remove the cycle. Since we removed one edge in a cycle, the graph will still be connected since any path connecting two nodes containing e' can instead go around the remaining edges in the cycle instead.

We'll call the resulting set of edges after this swap T' and argue that it has less weight than T. We can see that $w(T') = w(T) + w(e) - w(e')$ where we overload $w()$ to mean either the weight of an edge or the sum of weights for a set of edges. Since we know that $w_e < w_{e'}$, we have $w(e) - w(e') < 0$ so it follows that $w(T') < w(T)$. Since have found a tree defined by T' that has less total weight than T, we have a contradiction of the fact that T defined a minimum spanning tree. Therefor, our assumption that there exists an MST that does not contain e is false since we can always decrease it's weight by swapping e for some e' with larger weight. Thus, the minimum crossing edge must be in all MST's.

□

To utilize this lemma in the following proofs of correctness, we will use a trick to address property of distinct edge weights which is required in the cut property lemma. Specifically, if there are edges in E with identical weights there can be multiple MST's for Gthat an algorithm can output[2]. This is a problem since the proofs depend on the argument that there are edges that must be in all MST's. If there is an arbitrary choice between two edges, either can be in an MST, but neither must be in all MST's.

The trick we use to address this is to adjust the edge weights in E to artificially make the weights distinct, but not by enough to change the order of edge weights in the graph. This effectively removes the ambiguous decisions that an algorithm would have to make to the point that there is only one MST for G and we will prove that the algorithm returns this exact MST. This is an artificial construct that we use to address the unique edge weights constrain in the correctness proofs and is not used in the algorithms themselves.

4.5 Correctness of Prim's Algorithm

Theorem 1. *Prim's algorithm run on a connected undirected graph $G = (V, E)$ will always output a Minimum Spanning Tree (MST) of G.*

Proof. To prove the correctness of Prim's algorithm, we will utilize the cut property lemma. To do this, we must make our argument fit into the conditions of the lemma. We first use the trick described above to address the condition of unique edge weights. We now define the set S from the lemma to be the conveniently labeled S from the algorithm. Since we initialize $S = \{s\}$ and never remove any element from S, we have $S \neq \emptyset$. The algorithm runs until $S = V$, which implies $V \setminus S \neq \emptyset$ whenever an edge is being added to T. This satisfies all the condition of the lemma.

By the definition of the algorithm, it is always adding the smallest weight crossing edge between S and $V \setminus S$. By the cut property lemma, we know that this edge must be in all MST's. Since the algorithm is only adding edges that must be in every MST, it can't make a mistake by adding an edge that doesn't lead to minimum weight. Since the algorithm explores all edges starting at s, it will explore the connected component of s. The graph is connected, so it will explore the entire graph while only adding edges that must be in every MST. The result must be an MST of G. □

[2]If there exists a cycle in G with two or more edges having identical weight, the choice of which of these edges to add to the MST is arbitrary

4.6 Correctness of Kruskal's Algorithm

Theorem 2. *Kruskal's algorithm run on a connected undirected graph $G = (V, E)$ will always output a Minimum Spanning Tree (MST) of G.*

Proof. This proof is similar to the proof of Prim's algorithm in the sense that we will set up the proof to utilize the cut property lemma for the crucial part of the proof. Specifically, when an edge $e = (u, v)$ is added to T, we will show that it fits the conditions for the cut property and therefor must be in any MST. We will define S to be the current connected component of u in T at this point of the algorithm. Since $u \in S$, we have $S \neq \emptyset$. We know that the algorithm is adding e to T, so it doesn't create a cycle. This implies that $v \in V \setminus S$ since if $v \in S$ it would create a cycle because S is connected. Thus we have $V \setminus S \neq \emptyset$.

By contradiction, e must be the minimum crossing edge. If e were not the minimum crossing edge, then the minimum crossing edge would have already been considered by the algorithm. Since S and $V \setminus S$ are not connected, this crossing edge would not create a cycle and it would have been added to T which is a contradiction of the sets being disconnected. Therefor, e is the minimum crossing edge and must be in every MST. The algorithm only adds edges that must be in all MST's and considers all edges in the graph adding the as long as they don't create a cycle. The result must be a minimum spanning tree.

□

4.7 Runtime

Both Prim's and Kruskal's algorithms can be implemented in $O(m \cdot \log n)$ time by using efficient data structures. Prim's can be implemented using a priority queue (not the that comes with java) and Kruskal's can be reach this bound by using a union-find data structure. The sorting step in Kruskal's takes $O(m \cdot \log m)$, but since m is $O(n^2)$ we have $\log m$ is $O(\log n^2) = O(2 \log n)$ which is $O(\log n)$