Indian Institute of Information Technology Design and Manufacturing,
Kancheepuram, Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
COM 209T Design and Analysis of Algorithms -Lecture Notes

Instructor
N.Sadagopan
Scribe
Rashmitha Reddy
Ayesha S.K
K.Avinash

Greedy Algorithms - Part 2

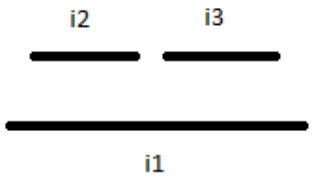
Objective: This module focuses on greedy algorithms for case studies interval scheduling and minimum weight spanning tree.

Case Study: Interval Scheduling

Input:
We have a set of requests $\{1, 2, \ldots, n\}$ on a time axis (an integer time line); the i^{th} request corresponds to an interval of time starting at $s(i)$ and finishing at $f(i)$.
Goal:
To find a largest compatible subset. A subset of requests is compatible if no two of them overlap in time (their line segments do not overlap), and our goal is to accept as large a compatible subset as possible. An optimal set in this context is a compatible set of intervals of maximum size.
We shall explore some strategies (heuristics) for this problem. We present counter examples for strategies that do not work always and present a proof of correctness for a strategy that does work always.

Possible Strategies

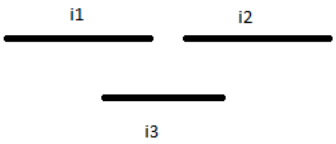
- **Strategy 1:**
  From the given set of intervals, select a request that starts earliest; that is, the one with the least start time (minimum $s(i)$). However, this strategy does not work always as we can see from the following counter example.
  Counter Example:



  The output of strategy 1 : $\{i_1\}$
  Optimum: $\{i_2, i_3\}$
  Reason:
  This method does not yield optimal solutions always. If the earliest request i is a very long interval, then by including request i into the solution, we may have to reject a lot of other requests which are short time intervals and overlapping with i. Clearly, we will end up with a suboptimal solution using this strategy. In the worst case, our strategy picks just one interval and optimum is the remaining set of intervals. The pitfall with this strategy is that finish times of intervals is over looked which is an important parameter to decide optimum.

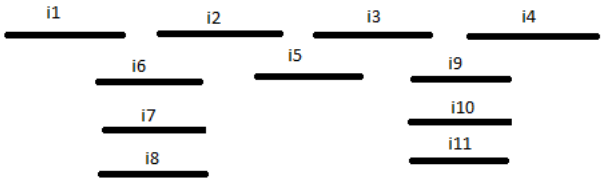
- **Strategy 2:**
  Accept the request that has the smallest interval of time, i.e., the request for which $f(i) - s(i)$ is minimum. It appears to be a better strategy, however, this strategy also can produce a suboptimal

schedule.
Counter Example:



The output of strategy 2= $\{i_3\}$
Optimum: $\{i_1, i_2\}$
Reason:
Accepting a short interval would force us to not pick intervals that overlap with the short interval. However, it may be the case that the one that overlaps with the short interval may be compatible, thus the other two are part of an optimal solution. In this approach, we fail to focus on non-compatible intervals.

- **Strategy 3:**
For each request, we count the number of other requests that are not compatible, and accept the request that has the fewest number of non-compatible requests. In other words, we select the interval with the fewest 'conflicts'.
Counter Example:



The output of strategy 3=$\{i_5, i_1, i_4\}$
Optimum=$\{i_1, i_2, i_3, i_4\}$

Lessons learnt from Strategies 1, 2 and 3:

1. Focus must be given to finish time $f(i)$ and non-compatible requests.
2. The accepted request must finish early so that waiting time for other requests would be minimized, i.e., our strategy must ensure that more requests are being serviced.

Questions:

1. Is there a strategy that yields OPTIMUM ? - Ans: Yes.
2. Is this problem easy (Does it have a polynomial-time algorithm) ? - Ans: Yes.
3. Is it true that every strategy gives suboptimal solution ? - Ans: Brute force strategy gives the optimal solution in $\Omega(2^n n^2)$. 2^n is the time to list all possible subsets of interval requests and it takes $O(n^2)$ time to determine if the generated subset is compatible.

- **Strategy 4:**
Accept first the request that finishes first, that is, the request i for which $f(i)$ is as small as possible. This is also a natural approach and intuitively, we can maximize the number of requests. For the above example, this strategy gives $\{i_1, i_2, i_3, i_4\}$ and the optimum is also $\{i_1, i_2, i_3, i_4\}$. A natural question is; Is Strategy 4 OPTIMAL ?

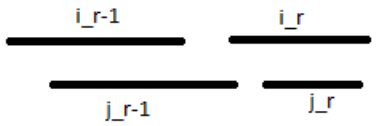


Figure 1: An illustration for the inductive step

Greedy Algorithm: Strategy 4 is Optimal

In this section, we shall present a sequence of structural observations to show that strategy 4 is optimal. i.e., strategy 4 yields an optimum solution, a solution with a maximum number of interval requests.

To prove greedy algorithm yields optimum solution always, we compare the solution output by algorithm and any optimum solution. Let $ALG = \{i_1, i_2, \ldots, i_k\}$ be the set of requests output by our algorithm in order. Let $OPT = \{j_1, j_2, \ldots, j_m\}$. Our goal is to prove that $k = m$. i.e., we need to prove that $|ALG| = |OPT|$. Without loss of generality, assume that the requests in OPT are also ordered in the natural left-to-right order of the corresponding intervals, that is, in the order of the start and finish points. Note that the requests in OPT are compatible, which implies that the start points have the same order as the finish points.

Claim 1:
For all indices $r \leq k$, we have $f(i_r) \leq f(i_k)$.
The intervals in the set ALG returned by the algorithm are compatible. Thus, strategy 4 guarantees that $f(i_r) \leq f(i_k)$.

Lemma 1
For $1 \leq r \leq k, f(i_r) \leq f(j_r)$
Proof: We need to prove that for each $r \geq 1$, the r^{th} accepted request in the algorithm's schedule finishes no later than the r^{th} request in the optimal schedule. We shall prove by mathematical induction on r.
Base Case:
For $r = 1$, the algorithm starts by selecting the request i_1 with minimum finish time. So, for every other request, $f(i_1) \leq f(i_l)$, $2 \leq l \leq k$.
In particular, in OPT, j_1 is either i_1 itself (or) some other request i_l, $2 \leq l \leq k$.
Therefore, $f(i_r) \leq f(j_r)$ when $r = 1$.
Hypothesis:
Assume that the claim is true for $r - 1$, $r \geq 2$. i.e., $f(i_{r-1}) \leq f(j_{r-1})$.
Induction Step:
Let $r \geq 2$. Consider the following illustration. In order for the algorithm's r^{th} interval request not to finish earlier, it would need to 'fall behind' r^{th} interval request of OPT as shown in Figure 1.
We now show that such a configuration is not possible as per our algorithm. Formally, it cannot be the case that $f(i_r) \geq f(j_r)$. Since OPT consists of compatible intervals, it claim that $f(i_{r-1}) \leq s(j_r)$. By the induction hypothesis, we know that $f(i_{r-1}) \leq f(j_{r-1})$, and since OPT consists of compatible requests, we get $f(j_{r-1}) \leq s(j_r)$. Thus, we get $f(i_{r-1}) \leq s(j_r)$.
Thus the interval j_r is in the set R of available intervals at the time when the greedy algorithm selects i_r. The greedy algorithm selects the available interval with smallest finish time; since interval j_r is one of these available intervals, we have $f(i_r) \leq f(j_r)$. This completes the induction step.
Therefore, for each r, the r^{th} interval the ALG selects finishes no later than the r^{th} interval in OPT.

Claim-2:
The greedy algorithm ALG is optimal. i.e., $k = m$.
Proof by Contradiction:
If ALG is not optimal, then any optimal set OPT must have more requests, that is, we must have $m > k$.

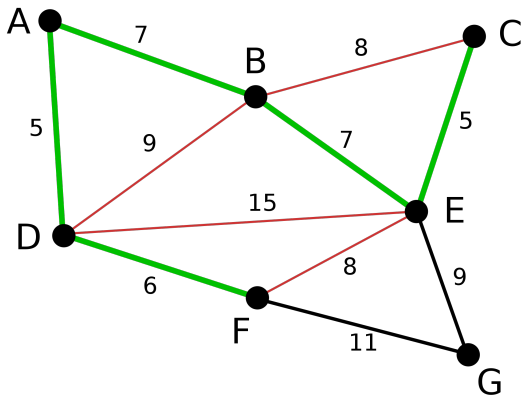


Figure 2: The minimum spanning tree is indicated by the green colored lines

Since $m > k$, there is a request j_{k+1} in OPT. This request starts after request j_k ends, and hence after i_k ends.
i.e., $f(j_k) \leq s(j_{k+1}) \leq f(j_{k+1})$
$f(i_k) \leq s(j_{k+1}) \leq f(j_{k+1})$
So after deleting all requests that are not compatible with requests $i_1, \ldots, i_k$, the set of possible requests R still contains j_{k+1}. j_{k+1} is a candidate request. But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty. However, algorithm stops after i_k. Therefore, j_{k+1} does not exist. Existence of j_{k+1} gives a contradiction to the fact that algorithm terminates at i_k. Therefore, our assumption that $m > k$ is wrong. Hence, the claim follows.

Case Study: Minimum Weight Spanning Tree

Let $G = (V, E)$ be a graph where $V(G)$ is the set of vertices in G and $E(G)$ is the edge set of G. The objective is to find a spanning tree T of G with minimum weight. i.e., we wish to find a subset $E' \subseteq E(G)$ such that it connects all of $V(G)$, the induced graph on E' is acyclic and the total weight of edges in E' is minimum. In this section, we shall present a greedy algorithm along with a proof of correctness, the algorithm is due to Kruskal.

Kruskal's Algorithm

Kruskal's algorithm is quite intuitive. We sort $E(G)$ in non-decreasing order. With respect to this order, we add $(n-1)$ edges in order such that each addition does not create a cycle. Since any acyclic graph with $(n-1)$ edges is a tree, the output of the algorithm is a spanning tree.

We shall now present a formal pseudo code and its sketch: Initially, each vertex itself is a tree and these trees are merged together by repeatedly adding the lowest cost edge that connects two distinct subtrees. Edges are chosen in such a way that it does not create a cycle in the resultant graph. This algorithm maintains the invariant `acyclicity` throughout the algorithm.

1. $A = \emptyset$
2. **for** each vertex $v \in V(G)$
3. MAKE-SET(v)
4. Sort the edges $E(G)$ in non-decreasing order by weight.

5. **for** each edge $\{u, v\} \in E(G)$, taken in non decreasing order by weight

6. if FIND-SET(u) $\neq$ FIND-SET(v)

7. $A = A \cup \{u, v\}$

8. UNION u, v

9. return A

The operation FIND-SET returns a representative element from the set that contains u. Thus we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). To combine trees, we call UNION procedure. Since sorting of $E(G)$ takes $O(e \log e)$ and it is a dominant operation over other operations such as cycle testing, union, etc., the time complexity of this algorithm is $O(e \log e)$ which is $O(e \log n)$.

Proof of correctness

Claim: Any tree constructed by Kruskal's algorithm is optimum, i.e., a spanning tree with minimum weight.
Proof: Let T_{alg} be the tree output by Kruskal's algorithm and T_{opt} be the output of any optimum algorithm. Note that if $w(T_{alg}) = w(T_{opt})$, then T_{alg} is also opt, where $w(T_{alg})$, $w(T_{opt})$ denote the weight of T_{alg} and T_{opt}, respectively.

Suppose, $w(T_{alg}) \neq w(T_{opt})$. Let $E(T_{alg}) = \{e_1, e_2, \ldots, e_t, e_{t+1}, \ldots, e_{n-1}\}$. Since $w(T_{alg}) \neq w(T_{opt}) \Rightarrow E(T_{alg}) \neq E(T_{opt})$. Let e_{t+1} be the first edge in $E(T_{alg})$ and e_{t+1} not in $E(T_{opt})$. We now use cut and paste argument to complete the proof. Let $E(T_{opt}) = \{e_1, \ldots, e_t, e'_{t+1}, \ldots, e'_{n-1}\}$.

Consider the graph H; formed using $T_{opt} + e_{t+1}$, add the edge $e_{(t+1)}$ to T_{opt}. Since H is a graph, but not a tree, there exists a cycle C in H. Clearly, C contains e_{t+1}.

Observe that $\exists e'_j$ in C such that $e'_j \in E(T_{opt})$ and $e'_j \notin E(T_{alg})$. If suppose such an edge e'_j does not exist in C, then every edge in C is part of $E(T_{alg})$. However, as per our algorithm, on adding e_{t+1} to T_{alg}, we will get a cycle and our algorithm would not have added e_{t+1} as part of the construction. The fact that e_{t+1} was added at $(t+1)^{th}$ iteration, implies that the addition of e_{t+1} did not create a cycle. Thus, we observe that e'_j exists in C.

Consider the graph $H' = T_{opt} + e_{t+1} - e'_j$. Clearly H' is a spanning tree. Further,
$w(T_{opt}) \leq w(T_{opt} + e_{t+1} - e'_j)$ as T_{opt} is opt and for every other tree, the weight will be at least the weight of opt tree.
$w(T_{opt}) \leq w(T_{opt}) + w(e_{t+1}) - w(e'_j)$
From the above inequality, we get $w(e_{t+1}) \geq w(e'_j)$.
Note: Kruskal's algorithm selected e_{t+1} over e'_j during $(t+1)^{th}$ iteration, this implies that

$\Rightarrow w(e_{t+1}) \leq w(e'_j)$

$\Rightarrow w(e_{t+1}) = w(e'_j)$, since we know that $w(e_{t+1}) \geq w(e'_j)$.

$\Rightarrow w(T_{opt}) = w(T_{opt} + e_{t+1} - e'_j)$

$\Rightarrow T' = T_{opt} + e_{t+1} - e'_j$ is another MST.

At the end of cut and paste argument, we see that T' and T_{Alg} match at the first $(t+1)$ edges. Continue the above process for $e_{t+2}, \ldots, e_{n-1}$. In each iteration, we modify T_{opt} in such a way that $E(T_{alg}) \cap E(T_{opt})$ increases by one. After, maximum of $(n-1)$ iterations, we get $E(T_{alg}) = E(T_{opt})$. Thus, we obtain $w(T_{alg}) = w(T_{opt})$. Therefore, our claim T_{Alg} is optimum is indeed true.

Some Thoughts

1. In Kruskal's algorithm, in each iteration, the invariant **acyclicity** is maintained. Since the algorithm stops after picking $(n-1)$ edges, we get connectedness property for free.

2. In Prim's algorithm, in each iteration, the invariant **connectedness** is maintained. Since the algorithm stops after picking $(n-1)$ edges, we get acyclic property for free.

3. To get a maximum weight spanning tree, we just negate all the edges in the given graph and use prim's or kruskal's algorithm to find the minimum weight spanning tree T, and negate the edges in the resultant tree T to obtain a maximum weight spanning tree.

4. Is MST unique ? Ans: No, MST is not unique. You get a unique MST only when all the edge costs are distinct.

5. How to find second best MST:

   (a) Let $G = (V, E)$ be an undirected, connected graph whose weight functions is $w : E \to R$, and suppose that $|E| \geq |V|$ and all the edge weights are distinct. We define a second-best minimum spanning tree as follows. Let X be the set of all spanning trees of G, and let T' be a minimum spanning tree of G. Then a **second-best minimum spanning tree** is a spanning tree T such that $w(T)$=MIN $_{T'' \ in \ X-T'}$ $w(T'')$

   (b) Algorithm to find the second best minimum spanning tree:

      i. Pick the first $n-1$ smallest edges from the graph using Kruskal's algorithm and let the tree formed be T

      ii. Let $E' = E(G) \setminus E(T)$ and $E' = \{e_1, \ldots, e_{m-n+1}\}$. $m = |E(G)|$.

      iii. From E', add the smallest edge. On adding the edge, a cycle C is formed, then remove the second maximum edge from C. Let the resultant tree be T^{e_n}.

      iv. Continue the above process for the remaining edges in E'. Let the resultant trees be $\{ T^{e_n}, T^{e_{n+1}}, \ldots, T^{e_m} \}$.

      v. The second best minimum spanning tree is the tree with minimum cost from the above set.

      vi. Time complexity: $m \cdot O(n) + O(e \log e)$.

Acknowledgements: Lecture contents presented in this module and subsequent modules are based on the following text books and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thanks all of them.

References:

1. E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.
2. T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein, Introduction to Algorithms, PHI.
3. Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.
4. Eva Tardos and Kleinberg, Algorithm Design, Pearson.