COMP3121: Algorithms & Programming Techniques
Summary notes – Week 4

Gerald Huang

Updated: August 12, 2020

Contents

1 Lecture 6A – The Greedy method

Date: August 12, 2020

1.1 Activity selection problem

(*Activity selection problem*)

- **Key points**:
  - You have a list of activities a_i for $1 \leq i \leq n$.
  - Each activity a_i contains a starting time s_i and finishing time f_i.
  - No two activities may take place simultaneously.
- **Task**: Find a *maximum size* subset of compatible activities.

- **Solution**: Among the activities that do not conflict with the previous chosen times, always choose the one with the earliest end time!
- **Optimality**: *Optimal solution transforms into greedy solution.*

  - Find the first place where the chosen activity violates the greedy choice.
  - Show that replacing that activity with the greedy choice produces a **non conflicting selection** with the same number of activities.
  - Continue in this manner until you "morph" your optimal solution into the greedy solution, thus proving the greedy solution is also optimal.

- **Time complexity**: $\mathcal{O}(n \log n)$.
  We represent activities by ordered pairs of their starting and finishing times, sort them in increasing order of their finishing times. We go through such a sorted list in order, looking for the first activity whose starting time is after the finishing time of the last taken activity. Every activity is handled exactly once, so it takes $\mathcal{O}(n)$ to traverse through the list. The sorting algorithm takes $\mathcal{O}(n \log n)$ which dominates the linear search.

1.2 Minimising job lateness

(*Minimising job lateness*)

- **Key points**:
  - You have a start time T_0 and a list of jobs a_i for $1 \leq i \leq n$.
  - Each job a_i contains a duration time t_i and deadline d_i.
  - Each job must be completed and no two jobs may take place simultaneously.
  - If a job is completed at a time $f_i > d_i$, then we say that it has incurred lateness $\ell_i = f_i - d_i$.
- **Task**: Schedule all the jobs so that the lateness of the job with the largest lateness is minimised.

- **Solution**: Schedule jobs in increasing order of deadlines.
- **Optimality**: *Minimise the number of inversions.*
  Consider any optimal solution. We say that jobs a_i and a_j form an inversion if job a_i is scheduled before a_j but $d_j < d_i$.
  - We observe that an optimal solution occurs when there is no inversion since an inversion will guarantee that there is some lateness incurred.
  - Observe that if we eliminate inversions among adjacent jobs, then we will have eliminated all inversions (as per the bubble sort).
  - Finally, swapping two adjacent inverted jobs does not increase the lateness incurred, only decrease it or stay the same.

  Our solution of choice has no inversions by definition and so, it is indeed optimal.

1.3 Tape storage I and II

(*Tape Storage I*)

- **Key points**:
  - You have a list of n files f_i of length ℓ_i stored in a tape.
  - Each file is equally likely to be needed.
  - To retrieve a file, you must start from the beginning and scan it until the file is found.
- **Task**: Order the files on the tape so that the average (expected) retrieval time is minimised.

- **Solution**: If the files are stored in order of length $\ell_1, \ell_2, \dots, \ell_n$, then the expected time is proportional to

$$\ell_1 + (\ell_1 + \ell_2) + (\ell_1 + \ell_2 + \ell_3) + \cdots + (\ell_1 + \ell_2 + \cdots + \ell_n) = n\ell_1 + (n-1)\ell_2 + \cdots + 2\ell_{n-1} + \ell_n.$$

  This is minimised if $\ell_1 \le \ell_2 \le \ell_3 \le \cdots \le \ell_n$.

(*Tape Storage II*)

- **Key points**:
  - You have a list of n files f_i of length ℓ_i stored in a tape.
  - Each file f_i has a probability p_i of being needed such that $\sum_{i=1}^{n} p_i = 1$.
  - To retrieve a file, you must start from the beginning and scan it until the file is found.
- **Task**: Order the files on the tape so that the expected retrieval time is minimised.

- **Solution**: If the files are stored in order of length $\ell_1, \ell_2, \dots, \ell_n$, then the expected time is proportional to

$$p_1\ell_1 + (\ell_1 + \ell_2)p_2 + (\ell_1 + \ell_2 + \ell_3)p_3 + \cdots + (\ell_1 + \ell_2 + \cdots + \ell_n)p_n.$$

  This is minimised if the files are ordered in decreasing order of values of the ratio $\frac{p_i}{\ell_i}$.

- **Optimality**: *Exchange argument.*
  - The expected times before and after the swaps at element k, respectively, are

$$E = \ell_1 p_1 + (\ell_1 + \ell_2)p_2 + \cdots + (\ell_1 + \ell_2 + \cdots + \ell_{k-1} + \ell_k)p_k + (\ell_1 + \ell_2 + \cdots + \ell_{k-1} + \ell_k + \ell_{k+1})p_{k+1} + \cdots + (\ell_1 + \ell_2 + \ell_3 + \cdots + \ell_n)p_n$$
$$E' = \ell_1 p_1 + (\ell_1 + \ell_2)p_2 + \cdots + (\ell_1 + \ell_2 + \cdots + \ell_{k-1} + \ell_{k+1})p_{k+1} + (\ell_1 + \ell_2 + \cdots + \ell_{k-1} + \ell_{k+1} + \ell_k)p_k + \cdots + (\ell_1 + \ell_2 + \ell_3 + \cdots + \ell_n)p_n$$

  - Thus, $E - E' = \ell_k p_{k+1} - \ell_{k+1} p_k$ which is positive if $\ell_k p_{k+1} > \ell_{k+1} p_k \implies \frac{p_k}{\ell_k} < \frac{p_{k+1}}{\ell_{k+1}}$. Consequently, $E > E'$ if and only if $\frac{p_k}{\ell_k} < \frac{p_{k+1}}{\ell_{k+1}}$ which implies that swaps will decrease the expected time of retrieval.
  - Finally, for as long as there are inversions, there will be inversions of consecutive files and swapping them reduces the expected time. As a result, the optimal solution is one with **no** inversions.

1.4 Interval stabbing

(*Interval stabbing*)

- **Key points**:
  - Let X be a set of n intervals on the real line.
  - We say a set P of points *stabs* X if every interval in X contains at least one point in P.
  - We may assume that input consists of two arrays $X_L[1, \dots, n]$ and $X_R[1, \dots, n]$, representing left and right endpoints of the intervals in X.
- **Task**: Describe and analyse an efficient algorithm to compute the smallest set of points that stabs X.

- **Solution**:
  - Look for the interval that ends the earliest and stab it at the very right end of the interval that is stabbed.
  - Remove all of the stabbed intervals and repeat this process until all have been stabbed.
- **Optimality**: *Optimal solution transforms into greedy solution.*
  - Consider an optimal solution that isn't necessarily as per the greedy solution. We can always move the stabbed point closer to the end of the interval that ends the earliest without losing any stabbed intervals and hence, repeating this until all have been used up becomes our greedy solution.

1.5 Knapsack problem

(*0-1 Knapsack problem*)

- **Key points**:
  - You have a list of weights w_i and values v_i for discrete items a_i for $1 \le i \le n$.
  - You also have a maximal weight limit W of your knapsack.
- **Task**: Find a subset S of all items available such that its weight does not exceed W and its value is maximal.

- Greedy strategy of choosing the item with the highest value per unit weight fails! No optimal solution using greedy solution!

1.6 Greedy method applied to graphs

- Given a directed graph $G = (V, E)$ and a vertex v, the **strongly connected component** of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v.
- **Finding a strongly connected component** $C \subseteq V$ containing u:
  - Construct another graph $G_{\text{rev}} = (V, E_{\text{rev}})$ consisting of the same set of vertices V but with the set of edges E_{rev} obtained by reversing the direction of all edges E of G.
  - Apply BFS to find the set $D \subseteq V$ of all vertices in V which are reachable in G from v.
  - Find the set $R \subseteq V$ of all vertices which are reachable in G_{rev} from v.
  - C is just
$$C = D \cap R.$$

- Let S_G be the set of all strongly connected components of a graph G.
- Define a graph $\sum = (S_G, E^*)$ with vertices S_G and directed edges E^* between the strongly connected components so that if $\sigma_1 \in S_G$ and $\sigma_2 \in S_G$ and $\sigma_1 \neq \sigma_2$, then there exists an edge from σ_1 to σ_2 in E^*.
- There exists vertices $u_1 \in \sigma_1$ and $u_2 \in \sigma_2$ such that there is an edge from u_1 to u_2 in E.

2 Lecture 6B – The Greedy method

Date: August 12, 2020

2.1 Shortest paths

- Assume we are given a directed graph $G = (V, E)$ with non-negative weight $w(e) \geq 0$ assigned to each edge $e \in E$.
- We are also given a vertex $v \in V$.
- **Task**: Find, for every $u \in V$ the shortest path from v to u.

2.1.1 Dijkstra's algorithm

- Algorithm builds a set S of vertices for which the shortest path has been already established, starting with a single source vertex $S = \{v\}$ and adding one vertex at a time.
- At each stage of the construction, we add the element $u \in V \setminus S$ which has the shortest path from v to u with all intermediate vertices already in S.

- Assume that this does NOT produce a shortest path. That is, there exists a shorter path from v to u in G. By our choice of u, such a path cannot be entirely in S.
- Let z be the first vertex outside S on such a shortest path.
- Since there are no negative weight edges, the path from v to such z would be shorter than the path from v to u, contradicting our choice of u.

2.2 Minimum Spanning Tree

- **Definition**: A minimum spanning tree T of a connected graph G is a subgraph of G which is a tree, and among all such trees, it is of minimal total length of all edges in T.
- **Lemma**: Let G be a connected graph with all lengths of edges E of G distinct and S a non empty proper subset of the set of all vertices V of G. Assume that $e = (u, v)$ is an edge such that $u \in S$ and $v \notin S$ and is of minimal length among all edges having this property. Then e must belong to every minimum spanning tree T of G.

2.2.1 Kruskal's Algorithm

- Order the edges E in a non-decreasing order with respect to their weights (sort the edges according to their weight).
- Build a tree by adding edges, one at each step of construction.
- An edge e_i is added at a round i of construction whenever its addition does NOT introduce a cycle in the graph constructed so far. If it does introduce a cycle, the edge is discarded.
- The process is terminated when the list of all edges has been exhausted.

- **Claim**: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a minimum spanning tree is unique.

Proof. We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Consider the set S of all vertices w such that there exists a path from u to w using only the subset of edges that have been added by the Kruskal algorithm until just before the edge $e = (u, v)$ has been added.
- Then $u \in S$ but $v \notin S$.
- Until this moment, no edge with one end in S and the other outside S has been considered because otherwise it would have been added, not forming a cycle.
- Consequently, edge $e = (u, v)$ is the shortest one with such a property and by the previous lemma, it must belong to every spanning tree.
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal theorem is clearly cycle-free and it spans the graph, otherwise another edge could be added.

□

2.3 k-clustering of maximum spacing

- **Key points**:
  - You have a complete graph G with weighted edges representing distances between the two vertices.
- **Task**: Partition vertices of G into k disjoint subsets so that the minimal distance between two points belonging to different sets of the partition is as large as possible.

- **Solution**: Sort the edges in an increasing order of weights and start performing the usual Kruskal's algorithm for building a minimal spanning tree, but stop when you obtain k connected components, rather than a single spanning tree.
- **Optimality**: *Exchange argument.*
  - Let d be the distance associated with the first edge of the minimal spanning tree which was not added to our k connected components.
  - It is clearly the minimal distance between two vertices belonging to two of our k connected.
  - Clearly, all the edges included in k many connected components produced by the algorithm of length smaller or equal to d.
  - Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm.
  - This means that there is a connected component produced by our algorithm which contains vertices v_i and v_m such that $v_i \in S_j$ for some $S_j \in \mathcal{S}$ and $v_m \notin S_j$.

- Since v_i and v_m belong to the same connected component, there is a path in that component connecting v_i and v_m.
- Let v_p and v_{p+1} be two consecutive vertices on that path such that v_p belongs to S_j and $v_{p+1} \notin S_j$.
- Thus, $v_{p+1} \in S_n$ for some $n \neq j$.
- Note that $d(v_p, v_{p+1}) \leq d$ which implies that the distance between these two clusters $S_j, S_n \in \mathcal{S}$ is smaller or equal to the minimal distance d between the k connected components produced by our algorithm.
- Thus, such a partition cannot be a more optimal clustering than the one produced by our algorithm.