A Quick Introduction to Greedy Algorithms

**NOTE:** This section was on last week’s guide as well; it is repeated here for your convenience.

Throughout the rest of the course, we will be discussing a fundamental paradigm called greedy algorithms. Much of these notes are adapted from CLRS Chapter 16.

**Definition** (Greedy Algorithms). A *greedy algorithm* obtains an optimal solution to a problem by making the choice that seems ‘the best’ at the moment. It is a heuristic strategy that does not work all of the time, yet for certain problems, it produces an optimal solution.

Greedy algorithms show up in many parts of computer science. We will see next week how we can use greedy algorithms to perform optimal data compression (Huffman’s Algorithm) and we will soon see how greedy algorithms can be used to find unique graph properties (Dijkstra’s Algorithm for shortest path and Prim’s/Kruskal’s Algorithms to find the minimum spanning tree).

**Greedy-choice Property**

The key ingredient to greedy algorithms is the *greedy-choice property*. This properties states that we can assemble a globally optimal solution by making locally optimal choices. This means that when we are considering a choice in our problem, we will always make the choice that is the best in our current situation without considering any future problems that we may encounter.

You can think of this as a ‘bottoms up’ approach. Greedy algorithms will solve sub problems one by one, choosing what is best at the current iteration, until it finds a globally optimal solution for the entire problem. For any greedy algorithm to be valid, we need to show that a greedy choice at each step yields a globally optimal solution. We can do this with the exchange argument.

**Definition** (The exchange argument). We first examine some globally optimal solution to our problem. We want to show how to modify this solution to substitute a greedy choice for some other choice in the problem that results in a similar but smaller sub problem. If we can show that the optimal solution to our problem includes our greedy choice along with the same optimal solution to a smaller subproblem, then we can ensure our greedy solution is correct.

If you want to learn more about greedy algorithms, please read CLRS Chapter 16.1 and 16.2 for a more in depth analysis.

**Test Your Understanding**

**The Scheduling Problem**

Consider a set of tasks $i_1...i_n$ that need to be run on a single machine. Each task has a required time it takes to run the task denoted as time $t_i$, which must be contiguous, and a distinct deadline time $d_i$ which task $i$ must be completed by. The machine can only run one task at once.
When a task is completed late, we consider the difference in completion time and its deadline to be that task’s ‘lateness’. The goal of our algorithm is to pick some order to process the tasks that minimizes the maximum ‘lateness’ of any single task, call it $L$.

*Solution.* There are multiple greedy strategies which we may think of when we encounter this problem.

**Idea 1:** We could run the tasks in order of increasing length ($t_i$). However we notice quickly that this ordering completely ignores deadlines and will not minimize lateness.

**Idea 2:** We could run tasks in increasing order of $d_i - t_i$. In other words, run the tasks in increasing order of ‘slack time’, the time the task must be started by in order to be completed by the deadline. However, this doesn’t succeed either. Consider task 1 with $t_1 = 1$, $d_1 = 2$, and task 2 with $t_2 = 10$, $d_2 = 10$. We end up running tasks with low slack time earlier, which could cause tasks with early deadlines to be extremely late.

**Idea 3:** Another simple idea is to run the tasks in increasing order of their deadline. This greedy choice turns out to minimize $L$, despite its simplicity. *But why does this work?* It seems like this choice is bound to fail on some input. However, it does actually work, as we will see with the following proof.

**Proof of Correctness by exchange argument:** In order to prove this, we will begin with an optimal solution, $O$ and modify it without changing the fact that it is optimal until it matches the scheduling produced by our algorithm.

**Claim 1:** There is an optimal scheduling with no idle time:
First, we’ll establish that an optimal scheduling has no ‘idle time’, in which no task is currently being worked on. If there was idle time, later tasks could simply be moved earlier by the duration of the idle time, either maintaining or decreasing the total lateness of our schedule, meaning our schedule wouldn’t have been optimal to begin with.

**Construction of $O'$:** As proven earlier, there is some optimal schedule $O$ with no idle time.

If $O$ does not have any inversions, then the schedule would be in strictly increasing order of deadlines, meaning it already matches the solution that our algorithm produces.

However, if $O$ has an inversion, then that means there must be at least one adjacent inversion. That is, there must be some two adjacent tasks $i$ and $j$, $i$ scheduled before $j$, such that $d_j < d_i$.

Suppose $O$ has at least one inversion, meaning it does not quite match the solution that our algorithm produces. Let $(i, j)$ be some inversion between consecutively scheduled tasks that we showed must exist above. Let $O'$ be the schedule formed by starting with $O$ and swapping the scheduling of the tasks $i$ and $j$.

**Claim 2:** $O'$ has a maximum lateness no larger than that of $O$:
Our goal in proving this is to show that when we swap to remove an inversion from an optimal scheduling, the resulting scheduling is also optimal. This would imply that we can transform any optimal scheduling (with up to $\binom{n}{2}$ inversions) into another optimal scheduling with no inversions, meaning the scheduling produced by our algorithm is optimal.

Consider each task $r$ to have a start and finish time, $s(r)$ and $f(r)$ respectively. Let the lateness of task $r$ in $O$ be $l_r$, and let $L = \max l_r$, the maximum lateness among all tasks in $O$. Similarly call the corresponding values from $O'$ $s'(r)$, $f'(r)$, $l'_r$, and $L'$.

Consider again the two adjacent, inverted tasks, $i$ and $j$. The finish time of $j$ before the swap is equal to the finish time of $i$ after the swap, so all other jobs besides $i$ and $j$ do not have their lateness changed. Because $j$ was moved earlier in the schedule, its lateness does not increase, so the only task which can possibly become more late is $i$. Task $i$’s lateness in $O'$ becomes: $l'_i = f'(i) - d_i = f(j) - d_i$.

However, task $i$ can’t possibly have become more late in $O'$ than task $j$ was in $O$, since by our inversion assumption, $d_i > d_j$, so $l'_i = f(j) - d_i < f(j) - d_j = l_j$. Since we are defining $L = \max l_r$, and we know that both of our new lateness times $l'_i$ and $l'_j$ are less than or equal to $l_r$, a value that was included in $L = \max l_r$, we know that the overall lateness is not going to increase. Note that all other lateness remain the same from $O$ to $O'$, so the only ones that matter here are $l'_i$ and $l'_j$.
Thus, we know that $L'$ must be no greater than $L$, and thus the total lateness did not increase from swapping an inversion.

This implies that we can simply take an optimal scheduling, swap all inversions out, maintaining optimality, and obtain a schedule in non-decreasing order of deadline. So, we can clearly see that an ordering of the jobs with no inversions will be an optimal solution. Our algorithm accomplishes this by creating a schedule based on the increasing order of each task’s deadlines, which clearly has no inversions. The combination of these claims prove that this algorithm is optimal using the exchange argument.

Graph Representations

Let $G = (V, E)$ with $|V| = n$, $|E| = m$. In other words, for some graph $G$, it contains $n$ vertices and $m$ edges.

Adjacency Matrix

One way to represent $G$ is with an $n \times n$ matrix $A$ where $A[i][j] = 1$ if there is an edge from vertex $i$ to vertex $j$ and 0 otherwise. The primary advantage of this approach is that you can check whether or not there is an edge connecting two vertices in $O(1)$ time. The disadvantage, however, is that this representation takes up $O(n^2)$ space. When $n$ is large, this might become untenable.

Two things worth noting:

- If $G$ is undirected, then its adjacency matrix is symmetric. That is, flipping the matrix along its main diagonal will produce the same matrix.
- Entries along the diagonal of an adjacency matrix (technically representing the presence of edges from vertices to themselves) are 0 by convention, as our graphs are simple. Non-simple graphs have self-loops, where vertices contain edges to themselves (these will not be dealt with in this course).

Adjacency List

Another way to represent $G$ is to use an adjacency list. Each vertex $u$ is associated to a list $\text{neighbors}(v)$ which contains the nodes $v$ such that $(u, v) \in E$. The advantage of this representation is that we use less space, $O(n + m)$, which is better than $O(n^2)$ of adjacency matrices as long as $m \ll n^2$. The disadvantage, though, is that checking whether $(u, v) \in E$ takes (potentially) linear time.

Graph Traversals

We now look at two ways to traverse a graph.

BFS (Breadth First Search)

In BFS, we begin at a node $v$ (level 0) and explore the graph in “layers.” First we would explore all children of $v$ (level 1), then the children of the nodes in level 1 (these would make up level 2), etc. The key point here is that we explore all nodes at level $i$ before exploring any nodes at level $i + 1$. The output of BFS is called a BFS tree. We typically use a queue to implement this algorithm. For implementation details, see https://en.wikipedia.org/wiki/Breadth-first_search

The running time of BFS is $O(n + m)$, because each vertex is added and removed from the queue once and, in the worst case, we need to traverse every edge to visit each node.
DFS (Depth First Search)

In DFS, we begin at a node \( v \) and examine its neighbors. As soon as we encounter a neighbor that hasn’t been visited, visit it. Once we arrive at a node for which all of its neighbors have been visited, we “backtrack” until we reach a node that has still unvisited neighbors (in the form of returning from recursive visit calls). We typically use a stack. There is also a recursive method to implement this algorithm. Please see both implementation methods in the link below.


The running time analysis for DFS is similar to that of BFS, giving a running time of \( O(n + m) \).

Graph Traversal Questions

**Problem 1.** Design an algorithm to determine whether or not a graph has a cycle.

*Solution.* You can perform a BFS or DFS and just keep track of which elements have been seen. For example, you can run a DFS and store vertices you have seen in a set and just track whether or not any previously seen node is encountered again by checking if it is in the set. Since we are simply doing a BFS or DFS, this algorithm runs in linear time.

**Problem 2.** Design an algorithm to determine whether or not a connected graph has a cycle in \( O(n) \) time.

*Solution.* Perform the same algorithm as problem 1. However, terminate early if you explore at least \( n \) edges. Recall that a tree has exactly \( n - 1 \) edges. An additional edge would signify that two nodes are connected by two independent paths. Thus, there is a cycle in the graph. This algorithm will take \( O(n) \) time to check each vertex and \( O(n) \) to check each edge (since we are checking at most \( n \) edges). Thus, the running time is \( O(n) \).

Note that a graph with \( n \) edges must have a cycle regardless of whether it’s connected. This example is just simple to prove with BFS.

**Problem 3.** Design an algorithm to find the shortest path between nodes \( u \) and \( v \) in a connected, unweighted graph.

*Solution.* Since the graph is unweighted, we can just run BFS starting from \( u \) and for each node \( x \) that we visit, we just keep a pointer to its parent node (the node we visited \( x \) from). When we reach \( v \), we stop and find the shortest path by backtracking through the pointers that we kept (i.e. we could see that \( v \)’s parent was \( d \), \( d \)’s parent was \( c \), and \( c \)’s parent was \( u \), so our path would be \( u \rightarrow c \rightarrow d \rightarrow v \)). We just do a BFS and backtrack no more than \( O(n) \) times (the longest path in a graph is \( n - 1 \) edges), so this algorithm also runs in linear time.