COMP202
Complexity of Algorithms

Fundamental Solution Techniques

Greedy Algorithms

[See Chapter 5 in Goodrich and Tamassia.]

Learning Outcomes

At the conclusion of this set of lecture notes, you should:

1. Understand the idea of greedy algorithms.
2. Be familiar with some of the classical problems for which a greedy solution exists.
3. Realize that greedy algorithms are not always appropriate, and know some examples where some seemingly natural greedy approaches fail to find the best solution.

Fundamental Techniques

We want to examine some general algorithmic tools that can be applied to a wide variety of different problems.

These tools include

- the Greedy Method,
- Divide and Conquer (seen already in some of the sorting algorithms), and
- Dynamic Programming.

The first one we consider is the Greedy Method.

Greedy Method

An *optimization problem* is a problem that involves searching for a configuration that *minimizes* or *maximizes* some *objective function*.

The *greedy method* solves (or tries to solve) a given optimization problem by going through a sequence of (feasible) choices.

The sequence starts from a well-understood starting configuration, and then iteratively makes the decision that *seems best* from all those that are currently possible.

Problems that have a greedy solution are said to possess the *greedy-choice property*.

Greed Works!

There are many problems that can be solved using a greedy method, such as

- Fractional Knapsack Problem
- Interval Scheduling
- Task Scheduling
- Minimum Spanning Trees (e.g. Kruskal's Algorithm, Prim's Algorithm)
- Shortest Paths (e.g. Dijkstra's Algorithm, Bellman-Ford Algorithm)
- Change Making (for *some* (but not all) sets of coins)
- Maximum Spacing k-clustering
- ...

Greed Works (well, sometimes)

- A word of warning! The greedy approach does *not always* lead to an optimal solution. (And we will see some problems for which it doesn't work.)

The greedy method is also sometimes used for some *hard* (i.e. difficult to solve) problems in order to generate *approximate solutions*.

Fractional Knapsack Problem (FKP)

Let S be a set of n items, where each item i has a *positive benefit* b_i and a positive *weight* w_i.

Goal: Find the *maximum benefit* subset that does not exceed total weight W.

In a FKP we are allowed to take *arbitrary fractions*, x_i, of each item, i.e. the solution is a set of values x_i such that

$$0 \leq x_i \leq w_i \text{ for all } i, \text{ and}$$

$$\sum_{i \in S} x_i \leq W.$$

The *total benefit* of the items taken is determined by the sum

$$\sum_{i \in S} b_i (x_i / w_i).$$

Fractional Knapsack Problem (cont.)

The general method for the FKP is to compute the *value index* for each item i, where this is defined as

$$v_i = \frac{b_i}{w_i}.$$

Then we select items to include in the knapsack, starting with the *highest value index*.

Using a heap (storing the *maximum* at the root), we can compute the value indices and build the heap in $O(n \log n)$ time.

Then, each greedy choice (which removes an item with the greatest remaining value index) requires $O(\log n)$ time.

```
FRACTIONALKNAPSACK(S, W)
      ▷ Input: Set S of items, item i has weight w_i and benefit b_i,
        maximum total weight W.
      ▷ Output: Amount x_i of each item to maximize the total benefit.
 1  for i ← 1 to |S| do
 2          x_i ← 0
 3          v_i ← b_i/w_i
 4          Insert (v_i, i) into a heap H (max value index at root).
 5  w ← 0
 6  while w < W do
 7          Remove the max value from H.
 8          a ← min{w_i, W − w}
 9          x_i ← a
10          w ← w + a
```

Fractional Knapsack Problem (cont.)

FKP satisfies the *greedy-choice property* (why?), hence

Theorem: Given an instance of FKP with set S of n items, we can construct a maximum benefit subset of S (allowing for fractional amounts of items) in $O(n \log n)$ time.

The $\{0, 1\}$-Knapsack Problem

The $\{0, 1\}$-Knapsack Problem is very similar to the Fractional Knapsack Problem.

The significant difference is that in the $\{0, 1\}$ version, we are not allowed to take fractional amounts of the items, either we take the entire item, or leave it behind.

A Greedy Method *does not* (in general) find an optimal solution of the $\{0, 1\}$-Knapsack Problem.

In COMP 108, you were given examples to illustrate that a Greedy Method does not necessarily give you the best solution in this case. You also discussed the method of Dynamic Programming, which can be used to find a solution to the $\{0, 1\}$-Knapsack Problem.

Interval Scheduling

We now consider a problem sometimes referred to as the *Interval Scheduling Problem*.

Here we have a collection of tasks (or intervals of requested time to use a room, etc.), and a single machine that can process these tasks (or a single room in which meetings can be held, etc.).

Goal: We want to select a subset of the tasks in order to maximize the *number* of tasks that we can schedule on the machine.

Interval Scheduling (cont.)

More formally (and keeping with the task/machine terminology), suppose we are given a set T of n tasks, where each task i has a start time s_i and a finish (or completion) time f_i.

Two tasks i and j are *non-conflicting* (or *compatible*) if

$$f_i \leq s_j \text{ or } f_j \leq s_i.$$

So two tasks can be executed on the machine only if they are *non-conflicting*.

Goal: Select a *maximum size* subset of non-conflicting tasks.

Interval scheduling (cont.)

Can we find a greedy algorithm for finding this solution?

There are several methods we might happen to try in our search for a method that works.

Interval Scheduling (cont.)

Suppose, for example, we always pick the first available task with the *earliest start time*.

This idea doesn't work, so we might try to *accept "short" intervals first*, i.e. take ones for which $f_i - s_i$ is smallest first.

Or, maybe we should first *accept tasks that don't "collide" with many other tasks*.

Hmmm, we seem to be running out of ideas...

Interval Scheduling (cont.)

How about accepting the task that finishes first? In other words, we want to "free up" the machine as soon as possible to start another task.

This idea works!

Order the tasks in terms of their *completion times*. Then select the task that finishes first, remove all tasks that conflict with this one, and repeat until we're done.

INTERVALSCHEDULE(T)

▷ Input: A set, T, of tasks with start times and end times.
▷ Output: A maximum-size subset, A, of non-conflicting tasks.

1 $A \leftarrow \emptyset$
2 **while** $T \neq \emptyset$
3 **do**
4 Remove the task t with earliest completion time.
5 Add t to A
6 Delete all tasks from T that conflict with t
7 Return the set A as the set of scheduled tasks

Interval Scheduling (cont.)

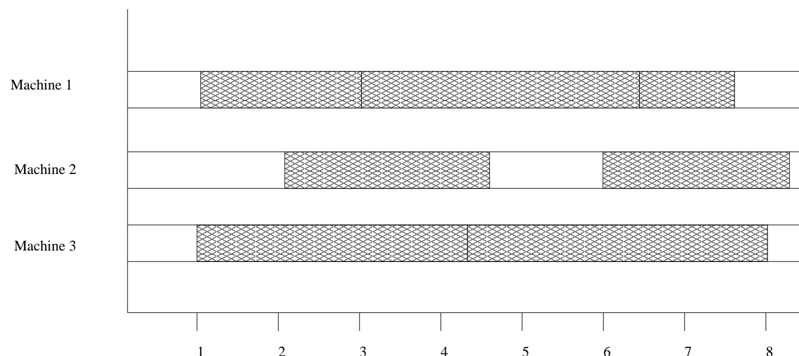
Theorem: The algorithm *IntervalSchedule* returns a set of non-compatible tasks having maximum size.

Furthermore, we can make this algorithm run in time $O(n \log n)$ (the main time will lie in sorting the tasks by their completion times).

Task Scheduling

Now we consider a different, yet related problem. Suppose we still have a set T of n tasks with start and completion times as before.

Now we want to schedule *all* of the tasks using *as few machines as possible* (in a non-conflicting way).



Task Scheduling - Idea for algorithm

In this case, a greedy algorithm will still work to solve this problem.

- This time we consider the tasks ordered by their *start times*.

Then for each task i, if we have the machine that can handle task i, it is scheduled on that machine.

Otherwise, we *allocate* a new machine, schedule task i on it, and repeat the greedy selection process until we have considered all tasks in T.

Task Scheduling - Algorithm

```
TaskSchedule(T)
    ▷ Input: A set, T, of tasks with start times and end times.
    ▷ Output: A non-conflicting schedule of the T tasks.
1   m ← 0
2   while T ≠ ∅
3       do
4           removeMin(T)
5           if ∃ machine j with no conflicts
6             then Schedule(i, j)
7           else
8                   m ← m + 1
9                   Schedule(i, m)
```

Task scheduling - Result

Theorem: Given an instance of the Task Scheduling Problem with a set of n tasks, the algorithm *TaskSchedule* produces a schedule of the tasks with the *minimum* number of machines in $O(n \log n)$ time.

Why does this greedy method work here?

We want to show that this greedy algorithm produces a schedule that uses the smallest number of machines.

Suppose that our method finds a schedule that uses k machines, but that there is some other (non-conflicting) schedule that uses at most $k-1$ machines.

Let k denote the last machine allocated by our algorithm, and let i be the *first* task that we scheduled on machine k.

Now, from how our algorithm works, when we scheduled task i, we used the new machine k because each of the other tasks had tasks already scheduled on them. Because we consider the tasks ordered by their start times, each of these other tasks have a start time that began before s_i, and since they conflict with task i, each of them has a finish time after s_i.

Proof of correctness (cont.)

This means that each of these other $k-1$ tasks not only conflict with task i, they conflict with each other too!

So this means we have a set of k tasks in T that all conflict with each other, hence it is impossible to schedule them using $k-1$ machines (contradicting our assumption that there is a schedule that uses only $k-1$ machines).

Hence, k is the minimum number of machines needed to schedule all tasks in T, and our greedy method gives us an optimal solution.

Clustering

Clustering arises when we try to group together or classify a collection of objects (photographs, documents, microorganisms, etc) based on the idea of *distance*.

For example, we might characterize the “distance” between two documents using the *edit distance*, i.e. the number of characters that you have to add, delete, or alter to transform one document into another.

For photographs, we might use the number of pixels at which their color differs by some threshold (of the wavelength of light), or by the intensity of the light, etc.

For microorganisms (or animals), the distance could be the number of years since they diverged on the evolutionary scale.

Clustering (cont.)

Regardless of what we use for distance, we will assume that we have a collection of n objects $p_1, p_2, \ldots, p_n$, together with a distance function, i.e. a function that satisfies the three properties that

1. $d(p_i, p_i) = 0$ for all i,
2. $d(p_i, p_j) > 0$ for $i \neq j$, and
3. $d(p_i, p_j) = d(p_j, p_i)$,

Suppose that we want to group the objects into k (non-empty) clusters, $C_1, \ldots, C_k$.

Maximum Spacing Clustering

In particular, given a clustering $C_1, \ldots, C_k$, define the *spacing* of the clustering to be the minimum distance between any pair of points lying in different clusters.

Our goal is, given the n objects and the value k, find a k-clustering with maximum spacing.

How do we do this?

Maximum Spacing Clustering

We can consider the idea of building a graph, where the objects $p_1, \ldots, p_n$ are the vertices of the graph.

The connected components of the graph that we build will be the clusters. We want to try to bring objects that are close together into the same cluster, as quickly as possible (so that they don’t end up in different clusters).

So, we take the two closest objects, and add an edge between them (as the first edge in our graph).

Then take the next two closest objects and add an edge between them.

Maximum Space Clustering (cont.)

We continue, considering the distances between pairs in increasing order. So we are growing a graph on the n objects. If we are about to add an edge between two objects that are already in the same cluster, then we don't bother doing so (as this gives us no additional information about the clustering).

This means that during our clustering process, we never form a cycle in our graph. Adding an edge corresponds to merging two clusters into one.

(We can consider starting the process with n clusters, each object belonging to its own cluster.)

Maximum Spacing Clustering (cont.)

If you think about this process, the procedure we are following is exactly the same as that for Kruskal's algorithm for finding a minimum spanning tree of a graph.

We add edges in increasing order of the edge length, without ever creating a cycle.

In this case, we can stop Kruskal's algorithm when we have a graph that has exactly k connected components. Or, equivalently, if we run Kruskal's algorithm, and then delete the last $k-1$ edges that were added, this will result in a k-clustering of maximum spacing.

Maximum Space Clustering (cont.)

Theorem

Given n objects, and the collection of their pairwise distances, a maximum spacing k-clustering can be found in time $O(n^2 \log n)$.

(The main bottleneck is to perform the sort of the pairwise distances.)