7 Greedy Algorithms

In the backtracking algorithms we looked at, we saw algorithms that found decision points and recursed over all options from that decision point. A **greedy algorithm** can be thought of as a backtracking algorithm where at each decision point "the best" option is already known and thus can be picked without having to recurse over any of the alternative options.

The name "greedy" comes from the fact that the algorithms make decisions based on a single criterion, instead of a global analysis that would take into account the decision's effect on further steps. As we will see, such a backtracking analysis will be unnecessary in the case of greedy algorithms, so it is not greedy in the sense of causing harm for only short-term gain.

Unlike backtracking algorithms, greedy algorithms can't be made for every problem. Not every problem is "solvable" using greedy algorithms. Viewing the finding solution to an optimization problem as a hill climbing problem greedy algorithms can be used for only those hills where at every point taking the steepest step would lead to the peak always.

Greedy algorithms tend to be very efficient and can be implemented in a relatively straightforward fashion. Many a times in O(n) complexity as there would be a single choice at every point. However, most attempts at creating a correct greedy algorithm fail unless a precise proof of the algorithm's correctness is first demonstrated. When a greedy strategy fails to produce optimal results on all inputs, we instead refer to it as a heuristic instead of an algorithm. Heuristics can be useful when speed is more important than exact results (for example, when "good enough" results are sufficient).

7.1 Event Scheduling Problem

The first problem we'll look at that can be solved with a greedy algorithm is the event scheduling problem. We are given a set of events that have a start time and finish time, and we need to produce a subset of these events such that no events intersect each other (that is, having overlapping times), and that we have the maximum number of events scheduled as possible.

Here is a formal statement of the problem:

Input: *events*: a set of intervals (s_i, f_i) where s_i is the start time, and f_i is the finish time.

Solution: A subset S of *Events*.

Constraint: No events can intersect (start time exclusive). That is, for all intervals $i = (s_i, f_i), j = (s_j, f_j)$ where $s_i < s_j$ it holds that $f_i \leq s_j$.

Objective: Maximize the number of scheduled events, i.e. maximize the size of the set S.

We first begin with a backtracking solution to the problem:

```
// event-schedule -- schedule as many non-conflicting events as possible
function event-schedule(events array of s[1..n], j[1..n]): set
  if n == 0: return ∅ fi
  if n == 1: return {events[1]} fi
  let event := events[1]
  let S1 := union(event-schedule(events - set of conflicting events), event)
  let S2 := event-schedule(events - {event})
  if S1.size() >= S2.size():
    return S1
  else
    return S2
  fi
end
```

The above algorithm will faithfully find the largest set of non-conflicting events. It brushes aside details of how the set

events - set of conflicting events

is computed, but it would require $O(n)$ time. Because the algorithm makes two recursive calls on itself, each with an argument of size $n-1$, and because removing conflicts takes linear time, a recurrence for the time this algorithm takes is:

$$T(n) = 2 \cdot T(n-1) + O(n)$$

which is $O(2^n)$.

But suppose instead of picking just the first element in the array we used some other criterion. The aim is to just pick the "right" one so that we wouldn't need two recursive calls. First, let's consider the greedy strategy of picking the shortest events first, until we can add no more events without conflicts. The idea here is that the shortest events would likely interfere less than other events.

There are scenarios were picking the shortest event first produces the optimal result. However, here's a scenario where that strategy is sub-optimal:

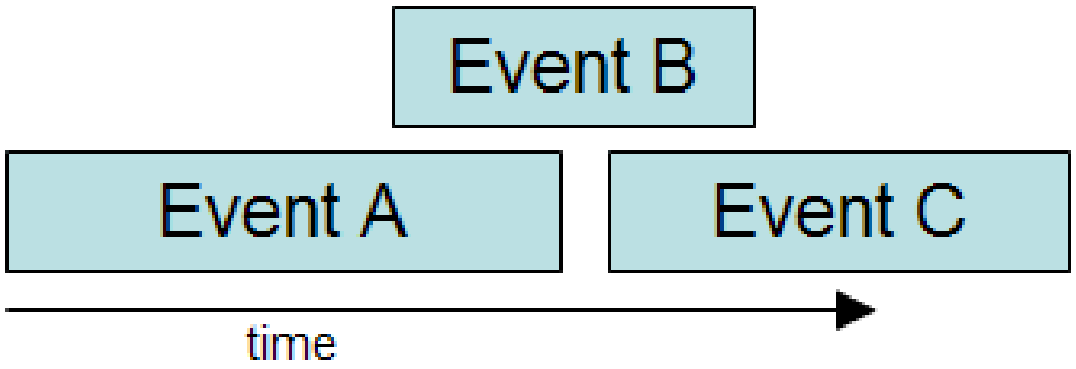

Figure 7

Above, the optimal solution is to pick event A and C, instead of just B alone. Perhaps instead of the shortest event we should pick the events that have the least number of conflicts. This strategy seems more direct, but it fails in this scenario:

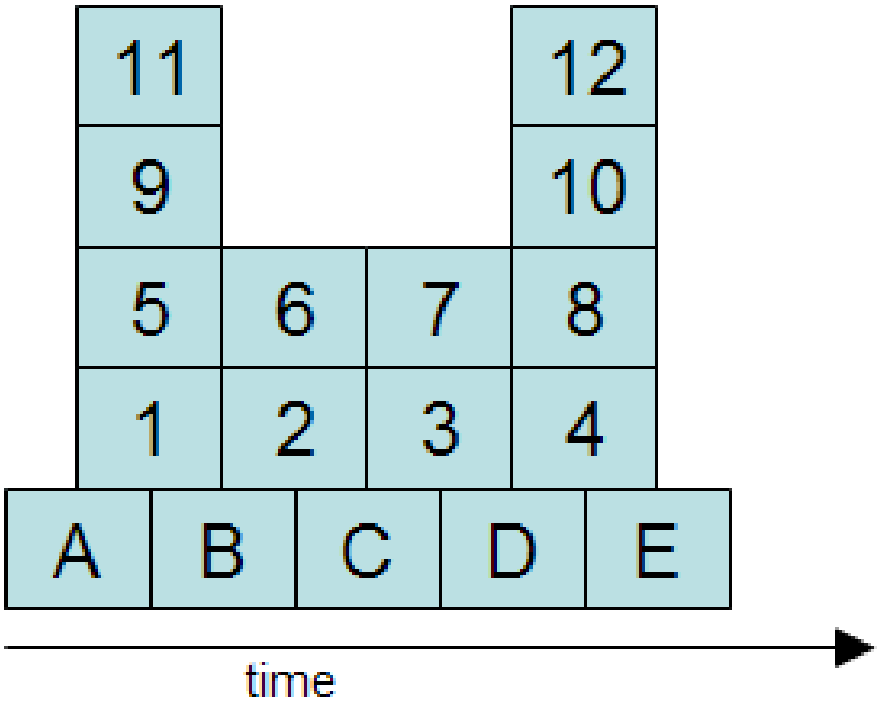

Figure 8

Above, we can maximize the number of events by picking A, B, C, D, and E. However, the events with the least conflicts are 6, 2 and 7, 3. But picking one of 6, 2 and one of 7, 3 means that we cannot pick B, C and D, which includes three events instead of just two.

7.2 Dijkstra's Shortest Path Algorithm

With two (high-level, pseudocode) transformations, Dijsktra's algorithm can be derived from the much less efficient backtracking algorithm. The trick here is to prove the transformations maintain correctness, but that's the whole insight into Dijkstra's algorithm anyway. [TODO: important to note the paradox that to solve this problem it's easier to solve a more-general version. That is, shortest path from s to all nodes, not just to t. Worthy of its own colored box.]

7.3 Minimum spanning tree

w:Minimum spanning tree[1]

1 http://en.wikipedia.org/wiki/Minimum%20spanning%20tree