Chapter B1: Greedy Algorithms and Spanning Trees

In a greedy algorithm, the optimal solution is built up one piece at a time. At each stage the best feasible candidate is chosen as the next piece of the solution. There is no back-tracking.

These notes are based on the discussion in Brassard and Bratley.

B1.1 The Generic Greedy Algorithm

The elements of a greedy algorithm are:

1. A set $C$ of candidates
2. A set $S$ of selected items
3. A solution check: does the set $S$ provide a solution to the problem (ignoring questions of optimality)?
4. A feasibility check: can the set $S$ be extended to a solution to the problem?
5. A select function which evaluates the items in $C$
6. An objective function

**Example.** How do you make change in South Africa with the minimum number of coins? (The coins are 1c, 2c, 5c, 10c, 20c, 50c.) Answer: Repeatedly add the largest coin that doesn’t go over.

The set $C$ of candidates is the infinite collection of coins $\{1, 2, 5, 10, 20, 50\}$. The feasibility check is that the next coin must not bring the total to more than that which is required. The select function is the value of the coin. The solution check is whether the selected coins reach the desired target.

However, a greedy algorithm does not work for every monetary system. Give an example!

In general, one can describe the greedy algorithm as follows:
Greedy(C:set)
   S := []
   while not Solution(S) and C nonempty do {
      x := element of C that maximizes Select(x)
      C := C \ [x]
      if Feasible(S + [x]) then S += [x]
   }
   if Solution(S) then return S
   else return "no solution"

There are greedy algorithms for many problems. Unfortunately most of those do not work! It is not simply a matter of devising the algorithm—one must prove that the algorithm does in fact work.

One useful concept for proving the correctness of greedy algorithms is the definition of a promising set. This is a set that can be extended to an optimal solution. It follows that:

Lemma. If S is promising at every step of the Greedy procedure and the procedure returns a solution, then the solution is optimal.

B1.2 Graphs and Minimum Spanning Trees

A graph is a collection of nodes some pairs of which are joined by edges. In a weighted graph each edge is labeled with a weight. In an unweighted graph each edge is assumed to have unit weight. The number of nodes is \( n \) and the number of edges is \( a \).

A walk in a graph is a sequence of edges where the end of one edge is the start of the next. A cycle in a graph is a walk of distinct edges that takes you back to where you started without any repeated intermediate node. A graph without a cycle is called a forest. A graph is connected if there is a walk between every pair of nodes. A tree is a connected forest.

A subgraph of a given graph is a graph which contains some of the edges and some
of the nodes of the given graph. A subgraph is a \textit{spanning} subgraph if it contains all the nodes of the original graph. A \textit{spanning tree} is a spanning subgraph that is a tree. The \textit{weight} of a subgraph is the sum of the weights of the edges. The \textit{minimum spanning tree} is the spanning tree with the minimum weight. For the following picture, a spanning tree would have 5 edges; for example, the edges $BC$, $AB$, $BE$, $EF$ and $DF$. But this is not optimal.

![Graph](image)

Trees have some useful properties:

\textbf{Lemma.} (a) If a tree has $n$ nodes then it has $n - 1$ edges.
(b) If an edge is added to a tree, a unique cycle is created.

The proof is left as an exercise.

\subsection{B1.3 Prim’s Minimum Spanning Tree Algorithm}

Prim provided a greedy algorithm to find a minimum spanning tree (though this had been previously published by Jarník). One starts at an arbitrary node and maintains a tree throughout. At each step, we add one node to the tree.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Prim} \\
\hline
\text{- Candidates = edges} \\
\text{- Feasibility Test = no cycles} \\
\text{- Select Function = weight of edge if incident with current tree, else } \infty. \\
\text{(Note we minimize Select(x))} \\
\hline
\end{tabular}
\end{center}

\textbf{Example.} For the graph in the previous picture, suppose we started at node $A$. Then we would add edges to the tree in the order: $AB$, $BC$, $BE$, $DE$, $EF$.

We need to discuss (1) validity (2) running time.
✓ **Validity**

A collection of edges is **promising** if it can be completed to a minimum spanning tree. An edge is said to **extend** a set $B$ of nodes if precisely one end of the edge is in $B$.

**Lemma.** Let $G$ be a weighted graph and let $B$ be a subset of the nodes. Let $P$ be a promising set of edges such that no edge in $P$ extends $B$. Let $e$ be any edge of largest weight that extends $B$. Then $P \cup \{e\}$ is promising.

**Proof.** Let $U$ be a minimum spanning tree that contains $P$. ($U$ exists since $P$ is promising.) If $U$ contains $e$ then we are done. So suppose that $U$ does not contain $e$. Then adding $e$ to $U$ creates a cycle. There must be at least one other edge of this cycle that extends $B$. Call this edge $f$. Note that $e$ has weight at most that of $f$ (look at the definition of $e$), and that $f$ is not in $P$ (look at the definition of $P$).

Now let $U'$ be the graph obtained from $U$ by deleting $f$ and adding $e$. The subgraph $U'$ is a spanning tree. (Why?) And its weight is at most that of $U$. Since $U$ was minimum, this means that $U'$ is a minimum spanning tree. Since $U'$ contains $P$ and $e$, it follows that $P \cup \{e\}$ is promising. 

If we let $B$ be the set of nodes inside our tree, then it is a direct consequence of the lemma that at each stage of the algorithm, the tree constructed so far is promising. When the tree reaches full size, it must therefore be optimal.

✓ **Running time**

By a bit of thought, this algorithm can be implemented in time $O(n^2)$. One stores

- for each node outside $B$, the smallest weight edge from $B$ to it

This is stored in an array called $\text{minDist}$. When a node is added to $B$, one updates this array.

The pseudocode given below uses that idea and assumes one starts at node 1. However, the running time can be improved using a priority queue such as a (Fibonacci) heap.
B1.4 Kruskal’s Minimum Spanning Tree

Kruskal also found a greedy algorithm. This time a forest is maintained throughout.

The validity is left as an exercise. Use the above lemma.
Example. For the graph on page 3, the algorithm would proceed as follows.
*DE, BC, BE, CE* not taken, *AB, CD* not taken, *EF*.

◊ Running time

One obvious idea is to pre-sort the edges.

Each time we consider an edge, we have to check whether the edge is feasible or not. That is, would adding it create a cycle. It seems reasonable to keep track of the different components of the forest. Then, each time we consider an edge, we check whether the two ends are in the same component or not. If they are we discard the edge. If the two ends are in separate components, then we merge the two components.

We need a data structure. Simplest idea: number each node and use an array $\text{Comp}$ where, for each node, the entry in $\text{Comp}$ gives the smallest number of a node in the same component. Testing whether the two ends of an edge are in different components involves comparing two array entries: $O(1)$ time. Total time spent querying: $O(a)$. This is insignificant compared to the $O(a \log a)$ needed for sorting the edges.

However, merging two components takes $O(n)$ time (in the worst case). Fortunately, we only have to do a merge $n - 1$ times. So total time is $O(n^2)$.

The running time of Kruskal’s algorithm, as presented, is

$$\max\{O(n^2), O(a \log a)\}$$

which is no improvement over the simple implementation of Prim. But actually, we can use a better data structure and bring it down. See the section on “disjoint set” data structure later.

Exercises

1. Prove that the greedy algorithm works for U.S. coinage. Concoct an example monetary system where it doesn’t work.

2. In what order are the edges of a minimum spanning tree chosen in the following graph, using (a) Prim (b) Kruskal?
3. Write a couple of pages on spanning tree algorithms, explaining why they work and how to implement them. Include at least one algorithm not described here.

4. In a coloring of a graph, one assigns colors to the nodes such that any two nodes connected by an edge receive different colors. An optimal coloring is one which uses the fewest colors. For example, here is an optimal coloring of these two graphs.

Dr I.B. Greedy proposes the following algorithm for finding an optimal coloring in a graph where the nodes are numbered 1 up to \( n \): Use as colors the positive integers and color the nodes in increasing order, each time choosing the smallest unused color of the neighbors colored so far.

(a) How long does this algorithm take? Justify.

(b) Does the algorithm work? Justify.