Principles of Algorithmic Techniques
T-79.4202

Pekka Orponen
Department of Computer Science
Aalto University

Autumn 2015
Lecture 8: Greedy algorithms

- Example: Making change
- Minimum weight spanning trees
- Data structures for MST algorithms
- Greedy approximation: the Set Cover problem
8.1 Greedy algorithms: the idea

- Basic idea: construct the solution to a problem instance as a sequence of locally optimal choices of components.
- In some cases this approach provably leads to globally optimal solutions, in other cases such “greedy” local choices may lead to an impasse later on, and force the algorithm to conclude with a suboptimal final result.
Example: Making change

- Consider the task of representing a given sum of money with as few coins as possible in the European currency system.
- The greedy approach, which in this case is also optimal, is to always extend a partial collection of coins by a coin that has as large a denomination as possible, without exceeding the target sum.
- For instance, for a target sum of 1.70 €, this method leads to the solution 170 ¢ = 100 ¢ + 50 ¢ + 20 ¢ (3 coins).
- If, however, a new 1.25 € coin was introduced, then the greedy heuristic would break down: e.g. for the target of 1.70 € it would lead to the suboptimal solution 170 ¢ = 125 ¢ + 20 ¢ + 20 ¢ + 5 ¢ (4 coins).
8.2 Minimum spanning trees

- Let $G = (V, E, w)$ be a connected, weighted graph.
- Consider the task of finding a minimum-weight subset of $E' \subseteq E$ of edges that still keeps $G$ connected.
- One observes the following property:
  
  **Property 1.** Removing a cycle edge cannot disconnect a connected graph.

- Consequently, the minimum-weight subset $E'$ cannot contain cycles, i.e. it determines a tree, a minimum (weight) spanning tree (MST) of $G$.
- A graph and one of its minimum-weight spanning trees:

![Graph and Minimum Spanning Tree](image)
Some general definitions and properties on trees

- A graph $G = (V, T)$ is a **tree** if it is acyclic and connected.
- **Property 2.** A tree on $n$ vertices has $n - 1$ edges.$^1$
- **Property 3.** A connected graph on $n$ vertices must have at least $n - 1$ edges. If the number of edges is exactly $n - 1$, then the graph is a tree.
- **Property 4.** A graph is a tree if and only if there is a unique path between any pair of vertices.
- A **spanning tree** of a connected graph $G = (V, E)$ is a subgraph $(V, T)$, $T \subseteq E$, that is a tree and contains all the vertices $V$ of $G$.

---

$^1$For proofs of these little lemmas, see p. 129 of the textbook Dasgupta et al., *Algorithms*. 

**Kruskal’s algorithm**

- A greedy procedure for constructing (as it turns out) minimum-weight spanning trees.
- Given a connected graph \( G = (V, E) \), start with edge set \( X_0 = \emptyset \), and then at each stage \( k \geq 1 \) let

\[
X_k = X_{k-1} \cup \{e\},
\]

where \( e \) is a least-weight edge in \( E \setminus X_{k-1} \) that does not induce a cycle in \((V, X_k)\).

- **Claim.** After \( n - 1 \) extension stages, \((V, T) = (V, X_{n-1})\) is a minimum-weight spanning tree of \( G \).
The cut (edge-exchange) property

- It is easy to see that Kruskal’s algorithm produces a spanning tree of the input graph $G$, but why do the greedy local choices it makes lead to a globally minimal solution?

- *The cut property.* Let $G = (V, E)$ be a connected graph, and suppose edge set $X \subseteq E$ can be extended to an MST of $G$. Let $(S, V \setminus S)$ be some cut (partition) of the vertices of $G$, so that no edge in $X$ crosses between $S$ and $V \setminus S$ (that is, has one end in $S$ and the other in $V \setminus S$), and let $e$ be a lightest-weight edge between $S$ and $V \setminus S$. Then also $X \cup \{e\}$ can be extended to an MST of $G$.

- Since in Kruskal’s algorithm initially $X_0 = \emptyset$ can be extended to an MST, and by the cut property each insertion of an edge to $X$ preserves this condition, the eventual result $X_{n-1} = T$ is an MST of $G.$
Proof of the cut property (1/2)

Let $T$ be an MST extending $X$. If also $e \in T$ then there is nothing to prove, so assume $e \notin T$.

Add edge $e = (u, v)$ to $T$. Because $T$ is a tree, it already contains a path connecting $u$ and $v$, and adding $e$ induces a cycle in $T \cup \{e\}$.

Let $e' \neq e$ be some other edge along this cycle crossing from $S$ to $V \setminus S$, and consider the edge set $T' = T \cup \{e\} \setminus \{e'\}$. 

Figure 5.2 $T \cup \{e\}$. The addition of $e$ (dotted) to $T$ (solid lines) produces a cycle. This cycle must be $T'$ as well, and it has $\text{weight}(T') = \text{weight}(T)$ and that $T'$ is also an MST.

Figure 5.3 shows an example of the cut property. Which edge is $e'$?
Proof of the cut property (2/2)

Now $T'$ is a tree extending $X \cup \{e\}$; we claim that it is in fact an MST of $G$.

The weight of $T'$ is clearly

$$\text{weight}(T') = \text{weight}(T) + w(e) - w(e').$$

Since $e$ was by selection the lightest edge crossing from $S$ to $V \setminus S$, it holds that $w(e) \leq w(e')$, and so $\text{weight}(T') \leq \text{weight}(T)$.

But since $T$ is an MST, it has minimal weight, and so in fact $\text{weight}(T') = \text{weight}(T)$, and $T'$ is also an MST of $G$. 
The cut property at work

Figure 5.3 The cut property at work. (a) An undirected graph. (b) Set $X$ has three edges, and is part of the MST $T$ on the right. (c) If $S = \{A, B, C, D\}$, then one of the minimum-weight edges across the cut $(S, V - S)$ is $e = \{D, E\}$. $X \cup \{e\}$ is part of MST $T'$, shown on the right.

5.1.3 Kruskal’s algorithm

We are ready to justify Kruskal’s algorithm. At any given moment, the edges it has already chosen form a partial solution, a collection of connected components each of which has a tree structure. The next edge $e$ to be added connects two of these components; call them $T_1$ and $T_2$. Since $e$ is the lightest edge that doesn’t produce a cycle, it is certain to be the lightest edge between $T_1$ and $V - T_1$ and therefore satisfies the cut property.

Now we fill in some implementation details. At each stage, the algorithm chooses an edge to add to its current partial solution. To do so, it needs to test each candidate edge $u - v$ to see whether the endpoints $u$ and $v$ lie in different components; otherwise the edge produces a cycle. And once an edge is chosen, the corresponding components need to be merged. What kind of data structure supports such operations?

We will model the algorithm’s state as a collection of disjoint sets, each of which contains the nodes of a particular component. Initially each node is in a component by itself:

- makeset($x$): create a singleton set containing just $x$.
- find($x$): to which set does $x$ belong?
A general scheme for MST algorithms

By the cut property, any greedy MST algorithm conforming to the following edge-selection scheme works correctly:

```plaintext
1 function MST(G);
   Input: Weighted, connected graph $G = (V, E, w)$
   Output: A minimal weight spanning tree $X$ for $G$.
2 set $X \leftarrow \emptyset$;
3 while $|X| < |V| - 1$ do
4     pick $S \subseteq V$ so that $X$ has no edges between $S$ and $V \setminus S$;
5     let $e$ be a minimum-weight edge between $S$ and $V \setminus S$;
6     set $X \leftarrow X \cup \{e\}$
7 end
```
Prim’s algorithm

- In Kruskal’s algorithm, the edge set $X$ has at every stage the structure of a **spanning forest** of all of $G$, and each added edge $e$ connects two trees in the forest into one.
- Another alternative is Prim’s algorithm, which maintains a **single tree** $X$ spanning a part of $G$, and extends this at each stage by the minimum-weight edge connecting $X$ to some vertex not yet spanned.
8.3 Data structures for MST algorithms

**Prim’s algorithm**

- The structure of Prim’s algorithm is very similar to Dijkstra’s algorithm. (See pseudocode below.)
- The basic set operations needed to support the algorithm are extracting the element with smallest key value in a set \( H \) (DeleteMin(\( H \))), and decreasing the key values of the remaining elements as more information becomes available (DecreaseKey(\( H \), (\( u \), \( x \)))).
- These are supported by the priority queue data structure, commonly implemented as a heap.\(^2\)

\(^2\)To be discussed at Lecture 16.
Prim’s MST algorithm in pseudocode (1/2)

1 function Prim($G, w$);

Input: Graph $G = (V, E)$ with edge weights $w(e), e \in E$.  
Output: An MST for $G$ defined by the array prev.

2 for all $u \in V$ do
3  cost[$u$] $\leftarrow \infty$;
4  prev[$u$] $\leftarrow \perp$ {Father in MST under construction};
5 end

6 pick any initial vertex $s$;
7 cost[$s$] $\leftarrow 0$;

(Continued on next slide.)
Prim’s MST algorithm in pseudocode (2/2)

1. $H \leftarrow \text{MAKEQUEUE}(\langle V, \text{cost}\rangle)$;
2. \textbf{while} $H$ is not empty \textbf{do}
   3. \quad $u \leftarrow \text{DELETEMIN}(H)$;
   4. \quad \textbf{for} all edges $(u, v) \in E$ \textbf{do}
   5. \quad \quad \textbf{if} $\text{cost}[v] > w(u, v)$ \textbf{then}
   6. \quad \quad \quad \text{cost}[v] \leftarrow w(u, v);
   7. \quad \quad \quad \text{prev}[v] \leftarrow u;
   8. \quad \quad \quad \text{DECREASEKEY}(H, (v, \text{cost}[v]));
   9. \quad \textbf{end}
10. \textbf{end}
11. \textbf{end}
Kruskal’s algorithm

- The data structure requirements of Kruskal’s algorithm are somewhat different. One needs to keep track of the trees in the spanning forest induced by the algorithm at each stage, and merge two trees into one as needed.

- These requirements are supported by the disjoint sets data structure, usually implemented by the ingenious union-find trees.

- The basic operations in this structure are the following:
  - \texttt{MAKESET}(x): Create and name a singleton set containing only element \( x \).
  - \texttt{FIND}(x): Return the name of the set into which element \( x \) currently belongs.
  - \texttt{UNION}(x, y): Merge the (disjoint) sets containing elements \( x \) and \( y \) into one. The name of the new set is either one of the names of the old sets.

\[\text{To be discussed at Lecture 16.}\]
Kruskal’s MST algorithm in pseudocode

1 function Kruskal(G, w);

   Input: Graph $G = (V, E)$ with edge weights $w(e)$, $e \in E$. 
   Output: An MST for $G$ determined by the edge set $X$.

2 for all $u \in V$ do MakeSet($u$)

3 $X \leftarrow \emptyset$;

4 sort the edges in $E$ by weight;

5 for all edges $(u, v) \in E$ in increasing order of weight do

   6 if FIND($u$) $\neq$ FIND($v$) then

   7 $X \leftarrow X \cup \{(u, v)\}$;

   8 UNION($u, v$);

   end

10 end
Time complexity of MST algorithms

The exact time complexities of the algorithms depend on the details of how the underlying data structures are implemented. However, using standard implementations:

- Prim’s algorithm requires:
  - 1 MakeQueue operation $\Rightarrow O(|V| \log |V|)$
  - $|V|$ DeleteMin operations $\Rightarrow O(|V| \log |V|)$
  - $|E|$ DecreaseKey operations $\Rightarrow O(|E| \log |V|)$

This yields a total time complexity of $O((|V| + |E|) \log |V|) = O(|E| \log |V|)$.

- Kruskal’s algorithm requires:
  - 1 sorting of edges $\Rightarrow O(|E| \log |E|)$
  - $|V|$ MakeSet operations $\Rightarrow O(|V|)$
  - 2$|E|$ Find operations $\Rightarrow O(|E| \log |V|)$
  - $|V| - 1$ Union operations $\Rightarrow O(|E| \log |V|)$

This yields a total time complexity of $O(|E| \log |E|) = O(|E| \log |V|)$. 


8.4 Greedy approximation: the Set Cover problem

- In many cases where exact algorithms are difficult to find, or all known ones are inefficient, the greedy approach can provide a simple and effective heuristic, or approximate solution method.

- Usually such greedy heuristics for hard problems are too simple to have good performance, but there are some exceptions.

- As an example, we shall consider a greedy approximation algorithm for the important Set Cover problem.
The Set Cover problem

- **Set Cover Optimisation Problem** (SC-O): Given a base set \( B \) and a collection of finite subsets \( S = \{ S_1, \ldots, S_m \} \) of \( B \), determine the smallest number of sets in \( S \) whose union covers all of \( B \).

- A candidate solution \( \{ S'_1, \ldots, S'_k \} \subseteq S \) to a given instance of the SC-O problem is feasible if \( \bigcup_{i=1}^{k} S'_i = B \), and its cost is the number of sets picked, i.e. \( k \). An optimal solution is a feasible solution with minimum cost.

- The Set Cover problem is **NP-complete**,\(^4\) so there are not likely to exist any efficient exact solution algorithms for it.

\(^4\)To be discussed at Lecture 10.
A greedy approximation heuristic

- A straightforward greedy heuristic for the Set Cover problem is the following:

  Repeat until all elements in $B$ are covered:
  
  Pick set $S_i$ with most uncovered elements.

- This does not always yield optimal solutions. (Example!)

- However, one can establish the following bound:

- **Claim.** Suppose $B$ contains $n$ elements and the optimal cover consists of $k$ sets. Then the greedy algorithm will pick at most $k \ln n$ sets.

- I.e. the greedy algorithm is guaranteed to be always at most a factor of $\ln n$ worse than the optimum. We say that it has an approximation bound of $\ln n$.\(^5\)

\(^5\)In fact, if $P \neq NP$, there exists a constant $0 < c < 1$ such that no polynomial-time approximation algorithm can have an approximation bound of $c \ln n$. [Raz & Safra 1997; Alon, Moshkovitz & Safra 2006]
Proof of approximation bound

- Let $n_t$ be the number of elements still uncovered after $t$ steps of the greedy algorithms. Initially $n_0 = n$.
- These $n_t$ elements are covered by the optimal $k$ sets, so some set in the optimal cover must contain $\geq n_t/k$ of them.
- By the greedy set selection rule, then:

$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \left(1 - \frac{1}{k}\right).$$

- By repeated application of this inequality one obtains:

$$n_t \leq n_0 \left(1 - \frac{1}{k}\right)^t < n_0 \left(e^{-1/k}\right)^t = ne^{-t/k}.$$

- At $t = k \ln n$, this inequality results in $n_t < ne^{-\ln n} = 1$. i.e. after this many stages of the greedy algorithm no more elements remain to be covered.