Design and Analysis of Algorithms

Part 7
Greedy Algorithms

Elias Koutsoupias
with thanks to Giulio Chiribella

Hilary Term 2021
The greedy approach [CLRS 16.2]

Greedy algorithms are typically used to solve optimisation problems.

The solution is constructed step by step. At each step, the algorithm makes the choice that offers *the greatest immediate benefit* (also called the *greedy choice*). A choice made at one step is *not* reconsidered at subsequent steps.

**Example: Dijkstra’s algorithm**

*Optimization problem:* find all shortest paths from the source.

*Construction of the solution:* shortest paths built vertex by vertex.

*Greedy choice:* at each step, choose the closest reachable vertex.

The greedy approach does *not* always work: for some problems, it fails to produce an optimal solution. But *when it does work*, it is attractive:

- It is conceptually simple.
- It does not require us to compare candidate solutions, or to keep a record of them.
Warm up: Coin Changing [CRLS problem 16-1]

Suppose we are in a country with the following coin denominations: quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent).

**Problem**: Assuming an unlimited supply of coins of each denomination, find the minimum number of coins needed to make change for \( n \) cents.

**Example**: \( n = 89 \) cents. What is the optimal solution?
Suppose we are in a country with the following coin denominations: quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent).

**Problem**: Assuming an unlimited supply of coins of each denomination, find the minimum number of coins needed to make change for $n$ cents.

**Example**: $n = 89$ cents. What is the optimal solution?

Answer: 8 coins (3 quarters, 1 dime and 4 pennies).
Suppose we are in a country with the following coin denominations: quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent).

**Problem**: Assuming an unlimited supply of coins of each denomination, find the minimum number of coins needed to make change for \( n \) cents.

**Example**: \( n = 89 \) cents. What is the optimal solution? 
Answer: 8 coins (3 quarters, 1 dime and 4 pennies).

**A greedy algorithm**

- Construct the solution coin by coin, reducing the amount at each step.
- Greedy choice: at each step, choose the coin of the largest denomination that does not exceed the remaining amount.

**Exercise**. Prove that in this case the greedy algorithm yields the optimal solution, and find a choice of coin denominations for which the greedy algorithm does not yield the optimal solution.
Example problem:
☐ A gas company undertakes to supply gas to all villages within a region.
☐ The cost of laying a pipeline between two villages depends on the distance between them, ease of access, etc.

Task: Find the cheapest way to lay pipelines to reach every village.

Graph theoretic formulation

Input: Connected undirected graph $G = (V, E)$ with weights $w: E \rightarrow \mathbb{R}_{\geq 0}$.

Task: Find a connected subgraph that has minimum weight and connects all the vertices of $G$.

Since the weights are nonnegative, there exists an optimal subgraph without cycles, i.e., a spanning tree. Such a tree is called a \textit{minimum spanning tree}. 

Minimum spanning trees [CLRS 23]
Since the graph is *undirected*, it is assumed that the weight function $w$ is *symmetric*, namely $w(u, v) = w(v, u)$ for every $(u, v) \in E$.

A *tree* is an *undirected* graph that is *connected and has no cycles*.

We will often identify a *tree* by its *set of edges* $T \subseteq E$. 
Lemma 1. An undirected graph is a tree iff each pair of vertices are connected by a unique (simple) path.

Proof. A connected undirected graph has a cycle iff there exist two vertices connected by distinct paths.
Lemma 1. An undirected graph is a tree iff each pair of vertices are connected by a unique (simple) path.

Proof. A connected undirected graph has a cycle iff there exist two vertices connected by distinct paths. □

Lemma 2. If a graph $G = (V, E)$ is a tree, then $|E| = |V| - 1$.

The proof is divided in two parts:

□ $G$ connected $\implies |E| \geq |V| - 1$
□ $G$ acyclic $\implies |E| \leq |V| - 1$ (continues on the next slide)
\[ G \text{ connected} \implies |E| \geq |V| - 1 \]

**Proof**

Set \( n = |V| \).

A graph of \( n \) vertices and no edges has \( n \) connected components. Adding an edge reduces the number of connected components by at most 1. To reduce the number of components to 1, we need to add at least \( n - 1 \) edges. \( \square \)
**Proof** By induction on $n = |V|$.  

*Base case.* For $n = 1$, an acyclic graph must have $|E| = 0$.  

*Inductive step.* Suppose that $|E| \leq |V| - 1$ for every acyclic graph with $|V| \leq n$, and consider a graph $G$ with $|V| = n + 1$. There are two possibilities:

1. $G$ has more than one connected component. Then, the induction hypothesis implies $|E_i| \leq |V_i| - 1$ for each component $i$, and therefore, $|E| \leq |V| - 1$.  

2. $G$ has only one connected component.  

Since $G$ is acyclic, removing one edge cuts the graph into two connected components, each satisfying $|E_i| \leq |V_i| - 1$.  

In total:

$$|E| = |E_1| + |E_2| + 1$$

$$\leq (|V_1| - 1) + (|V_2| - 1) + 1$$

$$= |V| - 1.$$  

$\square$
Spanning trees

A spanning tree of a graph $G = (V, E)$ is a subgraph with edge-set $T \subseteq E$ such that

- $T$ is a tree.
- $T$ reaches all the vertices of $G$: for each $u \in V$, there is some $v \in V$ such that $(u, v)$ or $(v, u)$ is in $T$.

**Lemma 3.** Every connected graph has a spanning tree.

**Proof.** Start from $T = \emptyset$.

Take edges from $E$ and add them to $T$ so long as no cycles are formed. This procedure constructs a maximal acyclic subgraph $T$ of $G$.

Now, $T$ must be connected, for if not, since $G$ is connected it would be possible to add another edge to $T$ without making a cycle.

Since $T$ is acyclic and connected, it is a tree. \qed
Definition: Minimum Spanning Tree (MST)

Let $G = (V, E)$ be a weighted graph, i.e. a graph equipped with a function $w : E \rightarrow \mathbb{R}$, assigning each edge $e \in E$ its weight $w(e)$.

If $T \subseteq E$ is a set of edges, the weight of $T$, denoted by $w(T)$, is the sum of the weights of the edges in $T$.

A minimum spanning tree (MST) is a spanning tree of minimum weight i.e. there is no spanning tree $T'$ with $w(T') < w(T)$.

Note: in general, the MST of a graph is not unique. But all MSTs have the same number of edges, equal to $|V| - 1$. 
Example: Two MSTs

The tree indicated by thick edges is an MST.

Replacing \((c, e)\) by \((e, f)\) gives a different MST.
How to build an MST?

**Idea:** build the MST edge by edge.

- Start from $A = \emptyset$.
  By definition $A$ is a (trivial) subset of an MST
- Add edges to $A$,
  *maintaining the property that $A$ is a subset of some MST.*
- Stop when no edge can be added to $A$ anymore.
  At this point, $A$ will be an MST.

**Definition.** Let $A \subseteq E$ be a subset of an MST $T$.
We say that an edge $(u, v)$ is *safe for $A$* iff
$A \cup \{(u, v)\}$ is a subset of *some* MST (not necessarily $T$).

To build an MST, we start from $A = \emptyset$ and we add a safe edge at each step.
Generic MST algorithm [CLRS 23.1]

**Generic-MST** \((V, E, w)\)

1. \(A = \emptyset\)
2. **while** \(A\) is not a spanning tree
3. find an edge \((u, v)\) that is *safe* for \(A\)
4. \(A = A \cup \{ (u, v) \}\)
5. **return** \(A\).

**Loop invariant:** \(A\) is safe i.e. a subset of some MST.

**Initialization:** The invariant is trivially satisfied by \(A = \emptyset\).

**Termination:** All edges added to \(A\) are in an MST, so upon termination, \(A\) is a spanning tree that is also an MST.

**Maintenance:** Since only safe edges are added, \(A\) remains a subset of some MST.
How to find safe edges: preliminary definitions

Let $G = (V, E)$ be an undirected graph.

- A **cut** is a partition of the vertex-set into two subsets $S$ and $V \setminus S$.
- An edge $(u, v) \in E$ **crosses** a cut $(S, V \setminus S)$ if one endpoint is in $S$ and the other in $V \setminus S$.
- A cut **respects** $A \subseteq E$ if no edge in $A$ crosses the cut.
- An edge is a **light edge crossing a cut** if its weight is minimum over all edges that cross the cut.

**Example**

$S = \{a, b, c, e, f\}$  
$A = \{(a, b), (c, e)\}$  
light edge: $(f, h)$
How to find safe edges: the Cut Lemma

**Lemma 4 (Cut).** Let $A$ be a subset of some MST. If $(S, V \setminus S)$ is a cut that respects $A$, and $(u, v)$ is a light edge crossing the cut, then $(u, v)$ is safe for $A$. 
How to find safe edges: the Cut Lemma

**Lemma 4 (Cut).** Let $A$ be a subset of some MST. If $(S, V \setminus S)$ is a cut that respects $A$, and $(u, v)$ is a light edge crossing the cut, then $(u, v)$ is safe for $A$.

**Example:** Assume that $A = \{(a, b), (c, e)\}$ is included in some MST. $(f, h)$ is a light edge crossing the cut. Hence, $A' = \{(a, b), (c, e), (f, h)\}$ is included in an MST.
Proof of the Cut Lemma

Let $T$ be a MST that includes $A$. Since $T$ is a tree, it contains a unique path $p$ between $u$ and $v$.

Path $p$ must cross the cut $(S, V - S)$ at least once. Let $(x, y)$ be an edge of $p$ that crosses the cut.

Adding $(u, v)$ and removing $(x, y)$ creates a tree $T'$.

□ Why is $T'$ a tree? Because by adding $(u, v)$, we create a cycle that consists of $p$ and $(u, v)$. This cycle contains edge $(x, y)$, which when removed leaves the graph connected with $|V| - 1$ edges.

□ Why is it minimum? Because the weight of $T'$ is $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$.

□ Does $T'$ include $A$? Yes, because $A$ was included in $T$, and $A$ did not contain $(x, y)$, the only edge we removed from $T$. 
Kruskal’s algorithm [CLRS 23.2]

**Idea:** Start from $A = \emptyset$

At every step, pick the edge with the smallest weight and add it to $A$, *if it does not create cycles.*

**Example:**

```
Idea: Start from A = ∅
At every step, pick the edge with the smallest weight and add it to A, if it does not create cycles.

Example:
```
How to avoid cycles

Kruskal’s algorithm is simple and intuitive, but how does the computer check whether adding an edge introduces a cycle?

**Idea:** keep track of the connected components. At each step, the set $A \subseteq E$ divides $V$ into connected components. We can add an edge only if it connects two distinct components.

To implement Kruskal’s algorithm we need a data structure that
- tells us whether two vertices $u$ and $v$ are in the same connected component
- merges two components when we put an edge between them.

This data structure is the *disjoint-set data structure.*
Disjoint-set data structure [CLRS 21]

Disjoint-set data structure

- Maintains a collection \( S = \{ S_1, \cdots, S_k \} \) of disjoint dynamic sets (i.e. disjoint sets changing over time).
- Each set is identified by a representative, a member of the set. It does not matter which member is the representative.

Three basic operations:

1. MAKE-SET\( (x) \): Makes a new set \( \{ x \} \) and add it to \( S \).
2. UNION\( (x, y) \): Removes \( S_x \) and \( S_y \) from \( S \), and adds the new set \( S_x \cup S_y \) to the collection \( S \).
3. FIND-SET\( (u) \): Returns the representative of the set containing \( u \).

Example: Consider the following sequence of operations:
MAKE-SET\( (a) \), MAKE-SET\( (b) \), UNION\( (a, b) \), MAKE-SET\( (c) \), \( x = \) FIND-SET\( (a) \), UNION\( (x, c) \). After these operations, \( S \) is \( \{ \{ a, b, c \} \} \).
Running times of different implementations

**Running time analysis:** given in terms of two numbers, \( m \) and \( n \).
- \( m \) = total number of operations
- \( n \) = number of MAKE-SET operations.

**Running times of different implementations**

1. **Linked-list:** \( O(m + n^2) \) time.
2. **Weighted linked-list:** \( O(m + n \log n) \) time.
3. **Disjoint-set forest:** \( O(m \alpha(n)) \) time,
   where \( \alpha(n) \) is an extremely slow-growing function
   (for all practical purposes, \( \alpha(n) \) can be treated as a constant).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \alpha(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>from 0 to 2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>from 4 to 7</td>
<td>2</td>
</tr>
<tr>
<td>from 8 to 2047</td>
<td>3</td>
</tr>
<tr>
<td>from 2048 to ( A_4(1) \gg 10^{80} )</td>
<td>4</td>
</tr>
</tbody>
</table>
Kruskal’s algorithm

\[
\text{Kruskal}(V, E, w)
\]

1. \( A = \emptyset \)
2. \textbf{for each} \( v \in V \)
3. \hspace{1em} \text{MAKE-SET}(v)
4. Sort \( E \) into increasing order by weight \( w \)
5. \textbf{for} each edge \((u, v)\) taken from the sorted list
6. \hspace{1em} \textbf{if} \( \text{FIND-SET}(u) \neq \text{FIND-SET}(v) \)
7. \hspace{2em} \( A = A \cup \{ (u, v) \} \)
8. \hspace{2em} \text{UNION}(u, v)
9. \textbf{return} \( A \).
Example

<table>
<thead>
<tr>
<th>Iter.</th>
<th>Edge</th>
<th>Add to A?</th>
<th>Connected Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$(c, f)$</td>
<td>yes</td>
<td>$a b c f d e g h i$</td>
</tr>
<tr>
<td>2</td>
<td>$(g, i)$</td>
<td>yes</td>
<td>$a b c f d e g i h$</td>
</tr>
<tr>
<td>3</td>
<td>$(c, e)$</td>
<td>yes</td>
<td>$a b c e f d g i h$</td>
</tr>
<tr>
<td>4</td>
<td>$(e, f)$</td>
<td>no</td>
<td>$a b c e f d g i h$</td>
</tr>
<tr>
<td>5</td>
<td>$(d, h)$</td>
<td>yes</td>
<td>$a b c e f d h g i$</td>
</tr>
<tr>
<td>6</td>
<td>$(f, h)$</td>
<td>yes</td>
<td>$a b c d e f h g i$</td>
</tr>
<tr>
<td>7</td>
<td>$(d, e)$</td>
<td>no</td>
<td>$a b c d e f h g i$</td>
</tr>
<tr>
<td>8</td>
<td>$(b, d)$</td>
<td>yes</td>
<td>$a b c d e f h g i$</td>
</tr>
<tr>
<td>9</td>
<td>$(d, g)$</td>
<td>yes</td>
<td>$a b c d e f g h i$</td>
</tr>
<tr>
<td>10</td>
<td>$(b, c)$</td>
<td>no</td>
<td>$a b c d e f g h i$</td>
</tr>
<tr>
<td>11</td>
<td>$(g, h)$</td>
<td>no</td>
<td>$a b c d e f g h i$</td>
</tr>
<tr>
<td>12</td>
<td>$(a, b)$</td>
<td>yes</td>
<td>$a b c d e f g h i$</td>
</tr>
</tbody>
</table>
Running time of Kruskal’s algorithm

- Initializing $A$ takes $O(1)$.
- First for-loop uses $|V|$ MAKE-SET operations.
- Sorting $E$ takes $O(|E| \cdot \log |E|)$.
- Second for-loop takes $2|E|$ FIND-SET and $|V| - 1$ UNION operations.

Hence, $n = |V|$ and $m = \Theta(|V| + |E|) = \Theta(|E|)$ (because the graph is connected).

**Overall running time:**
- $O(|E| \log |E| + |V|^2)$ linked-list implementation
- $O(|E| \log |E|)$ weighted linked-list implementation
- $O(|E| \log |E|)$ disjoint-set forest implementation.
Prims’s algorithm [CLRS 23.2]

**Idea:** Pick a vertex \( r \in V \) and grow the tree from that vertex.
Set \( S = \{ r \} \) and \( A = \emptyset \).
At every step, find a light edge \((u, v)\) connecting \( u \in S \) to \( v \in V \setminus S \).
Update \( S \) to \( S \cup \{ v \} \) and \( A \) to \( A \cup \{ (u, v) \} \).

**Example:**

![Diagram of Prims's algorithm](image-url)
How to find the light edge?

Construct a **priority queue** $Q$, such that

- $Q = V \setminus S$.
- The key of $v$ is the minimum weight of any edge $(u, v)$ where $u \in S$ (If $v$ is not adjacent to any vertex in $S$, set $key[v] = \infty$).

To find a light edge crossing the cut $(S, V \setminus S)$, **extract the minimum from the queue**.

If $v = \text{EXTRACT-MIN}(Q)$, then exists a light edge $(u, v)$ for some $u \in S$.

The vertex $u$ can be retrieved by a **backpointer**: when the key of $v$ is set to $key(v) = w(u, v)$, we define $\pi[v] = u$. 
Prim’s algorithm

**Prim**(*V, E, w, r*)

1. \( Q = \emptyset \)
2. for each \( u \in V \)  // Initializes key values and backpointers
   3. \( key[u] = \infty \)
   4. \( \pi[u] = \text{NIL} \)
   5. **INSERT**(*Q*, *u*)
   6. **DECREASE-KEY**(*Q*, *r*, 0)
7. while \( Q \neq \emptyset \)
   8. \( u = \text{EXTRACT-MIN}(Q) \)  // finds light edge for cut \((Q, V \setminus Q)\)
   9. for each \( v \in \text{Adj}[u] \)  // updates keys and backpointers
      10. if \( v \in Q \) and \( w(u, v) < key[v] \)
          11. \( \pi[v] = u \)
      12. **DECREASE-KEY**(*Q*, *v*, \( w(u, v) \))
Running time

- Initializing $Q$ to $\emptyset$ takes $O(1)$.
- Initializing $key[v]$ and $\pi[v]$ for every vertex takes $O(|V|)$.
- Each \textsc{Decrease-Key} operation takes $O(\log |V|)$ (assuming a min-heap implementation of the min-priority queue).
- While-loop takes $|V|$ \textsc{Extract-Min} and at most $|E|$ \textsc{Decrease-Key} operations.

Hence overall running time is $O(|E| \cdot \log |V|)$.

\textbf{Note.} Since $\log |E| = \Theta(\log |V|)$ for a connected graph, \textsc{Prim} and \textsc{Kruskal} have the same asymptotic running time.

\textbf{Note.} The running time of \textsc{Prim} can be improved to $O(|E| + |V| \log |V|)$ using a Fibonacci heap implementation of the min-priority queue.
Columns have $\text{key}[u], \pi[u]$ for $u \in Q$, and $-$ for $u \notin Q$. 
The *property of optimal substructure* —that an optimal solution to a problem is composed of optimal solutions to some of its subproblems — must hold in order to solve an optimisation problem using a dynamic programming algorithm or a greedy algorithm.

In a *dynamic programming algorithm* we typically solve the problem bottom-up: we solve smaller subproblems first, and use their solutions to obtain an optimal solution to a larger subproblem.

In a *greedy algorithm* we typically solve the problem top-down: we make a greedy choice at each step and then solve the resulting smaller subproblem.