Lecture 8: Greedy algorithms

- Example: Making change
- Minimum weight spanning trees
- Data structures for MST algorithms
- Eulerian trails and circuits
  - Only borderline greedy, but otherwise important
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:
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 (1956) 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 of the vertices of $G$, such that no edge in $X$ crosses between $S$ and $V \setminus S$, 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 $T' = T \cup \{e\} \setminus \{e'\}$. 


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:

1. \textbf{function} \text{MST}(G);

   \textbf{Input:} Weighted, connected graph $G = (V, E, w)$

   \textbf{Output:} A minimal weight spanning tree $X$ for $G$.

2. set $X \leftarrow \emptyset$;

3. \textbf{while} $|X| < |V| - 1$ \textbf{do}

4. \hspace{1em} pick $S \subseteq V$ so that $X$ has no edges between $S$ and $V \setminus S$;

5. \hspace{1em} let $e$ be a minimum-weight edge between $S$ and $V \setminus S$;

6. \hspace{1em} set $X \leftarrow X \cup \{e\}$

7. \textbf{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** (1957),\(^2\) 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.

\(^{2}\)Actually Jarník’s algorithm (1930).
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 $H$ ($\text{DeleteMin}(H)$), and decreasing the key values of the remaining elements as more information becomes available ($\text{DecreaseKey}(H, (u, x))$).
- These are supported by the priority queue data structure, commonly implemented as a heap.$^3$

---

$^3$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] ← ∞;
4     prev[u] ← ⊥ {Father in MST under construction};
5 end

6 pick any initial vertex \( s \);
7 cost[s] ← 0;

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

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

\footnote{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   \textbf{if} FIND($u$) \neq FIND($v$) \textbf{then}
7     $X \leftarrow X \cup \{(u, v)\}$;
8     UNION($u$, $v$);
9   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 ⇒ $O(|V| \log |V|)$
  - |V| DeleteMin operations ⇒ $O(|V| \log |V|)$
  - |E| DecreaseKey operations ⇒ $O(|E| \log |V|)$

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

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

This yields a total time complexity of $O(|E| \log |E|) = O(|E| \log |V|)$. 
8.4 Eulerian trails and circuits

Figure: L. Euler, The Seven Bridges of Königsberg (1735)
Euler’s Theorem (1/3)

A trail in a graph $G$ (either directed or undirected) is a walk in which no edge is repeated. A circuit is a closed trail, i.e. one where the starting and ending vertex coincide.

A trail (resp. circuit) is Eulerian if it covers every edge of $G$.

**Theorem E (L. Euler 1735)** An undirected graph $G$ contains an Eulerian circuit if and only if:

(i) $G$ is connected; and

(ii) every vertex in $G$ is of even degree.

*Note:* Actually, the first complete proof of the “$\Leftarrow$” direction is due to C. Hierholzer (1873).
Euler’s Theorem (2/3)

Proof.

(⇒) Clear. (Observe that for every vertex v in G, an Eulerian circuit will enter and exit v equally many times.)

(⇐) The proof is by induction on the number of edges e in G. Assume the claim holds for all graphs G′ that have e′ < e edges, and let G = (V, E), |E| = e, be (i) a connected graph where (ii) every vertex has even degree.

By property (ii), G contains at least one circuit C ⊆ E. (Pursue arbitrary non-repeating walk until it terminates at initial vertex.)

Remove C from G and consider the graph G′ = (V, E \ C). G′ still has property (ii). G′ may not be connected, but every one of its components G′₁, . . . , G′ₖ is. By induction, every component G′ᵢ contains an Eulerian circuit. These can be “glued” together with C to constitute an Eulerian circuit for G.
Euler’s Theorem (3/3)

Figure: Gluing a core circuit and three component circuits together
Versions of Euler’s Theorem

Euler’s Theorem has the following common variants, with similar proofs:

**Theorem E’** An undirected connected graph $G$ contains an Eulerian trail if and only if it has either 0 or 2 vertices of odd degree.

**Theorem E”** A directed strongly connected graph $G$ contains an Eulerian circuit if and only if for every vertex $v$ in $G$, $\text{indegree}(v) = \text{outdegree}(v)$.

**Theorem E”’** A directed strongly connected graph $G$ contains an Eulerian trail if and only if either for every vertex $v$ in $G$, $\text{indegree}(v) = \text{outdegree}(v)$, or for exactly one vertex $u$ $\text{indegree}(u) = \text{outdegree}(u) - 1$, and for exactly one vertex $v$ $\text{indegree}(v) = \text{outdegree}(v) + 1$. 
An algorithm for Eulerian circuits (1/3)

Hierholzer’s constructive proof for the existence of Eulerian circuits can be turned into an efficient algorithm:

**Input:** A connected undirected graph \( G = (V, E) \) in adjacency list representation.

**Output:** An Eulerian circuit \( C \) in \( G \), if such exists, represented as a linked list of edges.

**Method:** At the base record for the neighbour list of each vertex \( v \), maintain a pointer to the first “unused” edge out of \( v \).

Starting from an arbitrary vertex \( v_0 \), include edges in sequence to list \( C \), following an arbitrary trail on \( G \).

Whenever an edge is included in \( C \), it is marked as “used”.
An algorithm for Eulerian circuits (2/3)

The list $C$ also contains a pointer $p$ indicating to what point it has been validated as corresponding to the eventual Eulerian circuit.

When the edge-inclusion trail terminates at $v_0$ without any “unused” exit edges along which to continue (as it must if $G$ is Eulerian), start a traversal of list $C$ from the beginning, advancing pointer $p$ along the way, until either a branch vertex $v$ with “unused” exit edges is encountered, or pointer $p$ reaches the end of list $C$ and the algorithm terminates.

In case a branch vertex $v$ is found, a new trail starting and ending at $v$ is initiated, and the circuit of edges $C'$ discovered in this excursion are inserted in $C$ after the “old” edge indicated by pointer $p$.

The validation traversal of $C$ then continues from edge $p$ – now in the direction of the newly-inserted circuit $C'$. 
**Time complexity:** Every edge in $G$ is considered twice: once when it is included in $C$ and marked as “used”, and then when it is traversed by pointer $p$ in the validation traversal of $C$.

Hence the time complexity is $O(|E|)$. 