# Algorithms & Complexity: Lecture 7, Greedy Algorithms I

Sam Barrett

March 4, 2021

## 1 What are greedy algorithms?

There is no formal definition for greedy algorithms, we can actually design multiple different greedy algorithms to solve the same problem. They are categorised as algorithms that make local decisions to improve a solution, working step-by-step with no concern for what they have done previously or might do later. Often this short-sighted approach does not help in finding an optimal solution, however, some can still approximate an optimal solution.

There are advantages and disadvantages of greedy algorithms.

Advantages:

- They are intuitive
- This leads to them being easy to explain and implement
- Most heuristics are based on *greedy* choices
- They can be shown to sometimes be approximately correct

Disadvantages:

- There are different notions of greedy, no formal definition
- Local correct steps do not guarantee a globally correct approach
- Often do not result in optimal solutions

Our 2-approximation for the vertex cover problem discussed in the previous lecture is an example of a greedy algorithm, it ran in polynomial time.

## 2 Dijkstra's Algorithm

This is an algorithm that finds the shortest paths in a directed graph from a fixed vertex $s$.

The running time of this algorithm is $O(mn)$ where $n, m$ are the number of vertices and edges respectively. This is the case as the while loop runs at most $n$ times and each iteration takes $m$ steps.

**Algorithm 1:** Dijkstra's Algorithm

```
1 Let $S$ be the set of explored vertices
2 For each $u \in S$ we store the distance of the shortest $s \to u$ path in
    $\texttt{dist}(u)$
3 Initialise $S = \{s\}$ and $\texttt{dist}(s) = 0$
4 while $S \neq V$ do
5     Select a vertex $v \notin S$ which minimises
        $\texttt{temp-dist}(v) = \min_{(u,v) \in E, u \in S} (\texttt{dist}(u) + \texttt{length}(u, v))$
6     Add $v$ to $S$ and set $\texttt{dist}(v) = \texttt{temp-dist}(v)$
7 end
```

## 2.1 Correctness of Dijkstra's algorithm

**Theorem 1** *Consider the set $S$ at any point in the running of the algorithm. For each $u \in S$ the quantity $\texttt{dist}(u)$ stores the value of the shortest $s \to u$ path.*

We can prove this by induction:

- Base case: $|S| = 1$ and $\texttt{dist}(s) = 0$
- Inductive hypothesis: Suppose that the theorem holds for $|S| = k$
- Inductive step: Suppose we now grow $S$ by one more vertex by adding some vertex $v$.

  By line 5 we know: $\texttt{temp-dist}(v) = \texttt{dist}(u) + \texttt{length}(u, v)$ for some $u$ already in $S$

  Suppose that there is a path $P$ $s \to v$ that is shorter than $\texttt{dist}(v)$

  Let $P$ leave $S$ via an edge $(x, y)$ for some $x \in S, y \notin S$

  **Contradiction**

# 3 Prim's algorithm

Prim's algorithm is an algorithm for finding the minimum spanning tree of a given graph $G$.

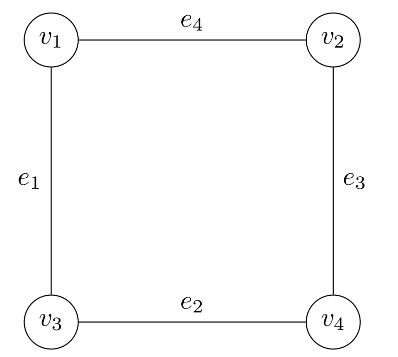
## 3.1 Minimum spanning trees

Let $G$ be a undirected, connected graph $G = (V, E)$ with $n$ vertices.

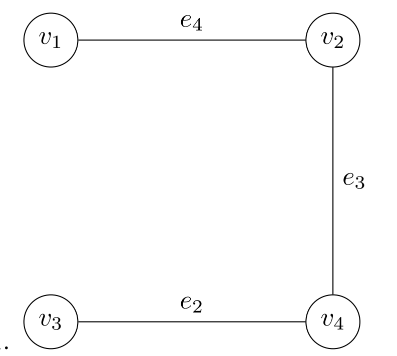
A subgraph $T = (V', E')$ of $G$ is said to be a **spanning tree** if:

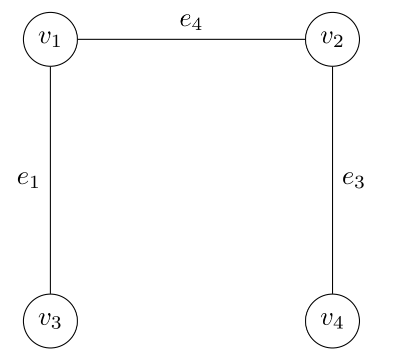
- spanning $\to V' = V$
- tree $\to |E'| = n - 1$

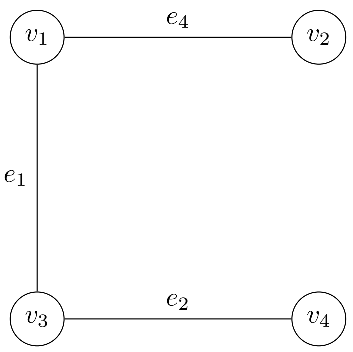
For instance, the graph:



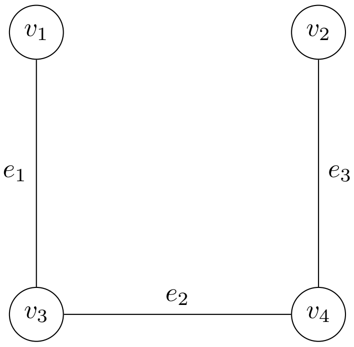
Has 4 different spanning trees:

1. 

2. 



3.



4.

**Problem 1** *(Minimum Spanning Tree (MST) problem)*
*Given an undirected, connected graph $G = (V, E)$ with edge costs given by $\texttt{cost} : E \to \mathbb{R}^+$, find a spanning tree $T = (V, E')$ such that $\sum_{e\in E'} \texttt{cost}(e)$ is minimised*

**Algorithm 2:** Finding MST - Prim's algorithm

```
1 Let $S$ be the set of explored vertices
2 Initialise $S = \{s\}$ where $s$ is any vertex
3 Initialise $E' = \emptyset$
4 while $S \neq V$ do
5     Select a vertex $v \notin S$ which minimises $\min_{e=u-v, u\in S} \texttt{cost}(e)$
6     Add $v$ to $S$ and $e$ to $E'$
7 end
```

Running time of this algorithm is $O(mn)$ where $n, m$ are the number of vertices and edges respectively. The while loop runs at most $n$ times with each loop requiring $m$ time.

## 3.2 Correctness of Prim's algorithm

We first make an assumption that all edge costs are **distinct**.

**Theorem 2** *For any $S \subset V$, let $e$ be the edge of minimum cost having one end point in $S$ and one end point in $V \backslash S$. Then every MST contains the edge $e$*

First suppose that there is a MST $T$ which does not contain this edge $e$ whose endpoints are $v \in S$ and $w \notin S$. We will now find an edge $e'$ in $T$ s.t. $\texttt{cost}(e') > \texttt{cost}(e)$. Therefore, replacing $e'$ with $e$ gives a spanning tree of lower cost, thus deriving a contradiction.

## 4 Kruskal's algorithm

**Algorithm 3:** Kruskal's algorithm

```
1  Order the edges of $E$ as $e_1, e_2, \dots, e_m$ in order of cost (increasing)
2  Initialise $E' = \emptyset$ and $i = 1$
3  while $i \leq m$ do
4      if adding $e_i$ to $E'$ does not create a cycle then
5          Add $e_i$ to $E'$
6      else
7          Do not add $e_i$ to $E'$
8      end
9      $i++$
10 end
```

This algorithms also runs in $O(mn)$ time where $n, m$ are the number of vertices and edges respectively.

The notable difference with this algorithm is that we do **not** consider all edges

### 4.1 Correctness of Kruskal's algorithm

We again make the assumption that all edge-costs are distinct.

**Theorem 3** *(Same as for Prim's) For any $S \subset V$, let $e$ be the edge of minimum cost having one end point in $S$ and one end point in $V \backslash S$. Then every MST contains the edge $e$*

We consider the algorithm at some arbitrary step.

Suppose Kruskal's algorithm adds the edge $v - w$ at this step.

Let $S$ be the set of all vertices to which $v$ had a path to before this step. We can see that $w \notin S$ as otherwise we would have a cycle, breaking the algorithm at line 4.

By the definition of $S$ no edges from $S$ to $V \backslash S$ have been added before this stage.

Since $v \in S$ and $w \notin S$ and as Kruskal's algorithm adds edges in increasing order of cost, it follows that $v - w$ is the *cheapest* edge with one endpoint in $S$ and the other in $V \backslash S$

## 5 Reverse-delete algorithm for finding MSTs

We can also approach this problem from the opposite direction. Instead of adding edges until we cannot avoid creating a cycle, we start and remove (the most expensive) edges until no cycles are left.