Teoria dos Grafos e Computabilidade

— Greedy graph algorithms —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Feb 2025

Teoria dos Grafos e Computabilidade

— Graphs —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Feb 2025

Graphs

- Model pairwise relationships (edges) between objects (nodes or vertices).
- **Undirected graph** $G = (V, E)$: set V of nodes and set E of edges, where $E \subseteq V \times V$. Elements of E are unordered pairs.
- **Directed graph** $G = (V, E)$: set V of nodes and set E of edges, where $E \subseteq V \times V$. Elements of E are ordered pairs.

Applications of Graphs

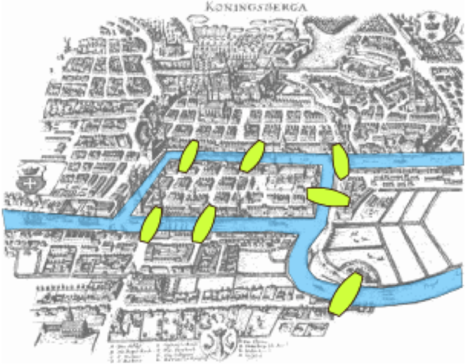
- Useful in a large number of applications:

Applications of Graphs

- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ...
- Problems involving graphs have a rich history dating back to Euler.

Applications of Graphs

- Useful in a large number of applications: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, ...
- Problems involving graphs have a rich history dating back to Euler.



Teoria dos Grafos e Computabilidade

— Shortest Paths —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Feb 2025

Shortest Path Problem

- $G = (V, E)$ is a connected directed graph. Each edge e has a length $l_e \geq 0$.
- V has n nodes and E has m edges.
- Length of a path P is the sum of lengths of the edges in P.
- Goal is to determine the shortest path from some start node s to each node in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

Shortest Path Problem

- $G = (V, E)$ is a connected directed graph. Each edge e has a length $l_e \geq 0$.
- V has n nodes and E has m edges.
- Length of a path P is the sum of lengths of the edges in P.
- Goal is to determine the shortest path from some start node s to each node in V.
- Aside: If G is undirected, convert to a directed graph by replacing each edge in G by two directed edges.

Shortest Paths

INSTANCE	A directed graph $G(V, E)$, a function $l : E \rightarrow \mathbb{R}^+$, and a node $s \in V$
SOLUTION	A set $\{P_u, u \in V\}$, where P_u is the shortest path in G from s to u.

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

Algorithm: Shortest path algorithm – Dijkstra

---

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;

while $S \neq V$ **do**
  Select a node $v \notin S$ with at least one edge from S for which
    $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
  Add v to S and define d[v] = d'[v];
end

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

Dijkstra's Algorithm

- Maintain a set S of explored nodes: for each node $u \in S$, we have determined the length $d(u)$ of the shortest path from s to u.
- Greedily add a node v to S that is closest to s.

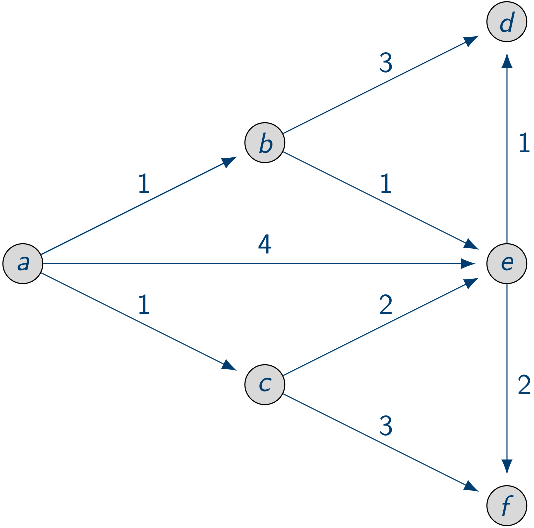
Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

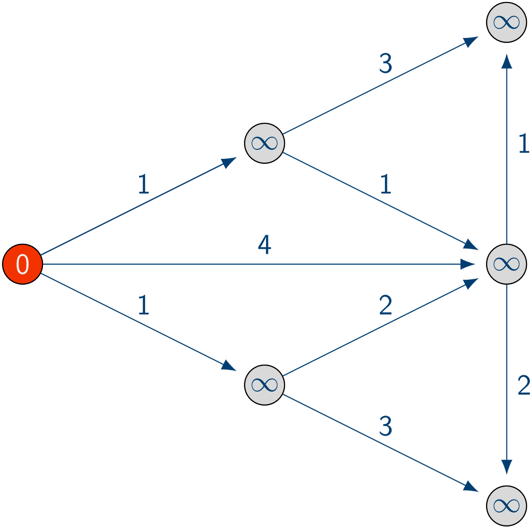
Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;
while $S \neq V$ **do**
  Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
  Add v to S and define d[v] = d'[v];
end

- Can modify algorithm to compute the shortest paths themselves: **record the predecessor** u that minimizes $d'(v)$.

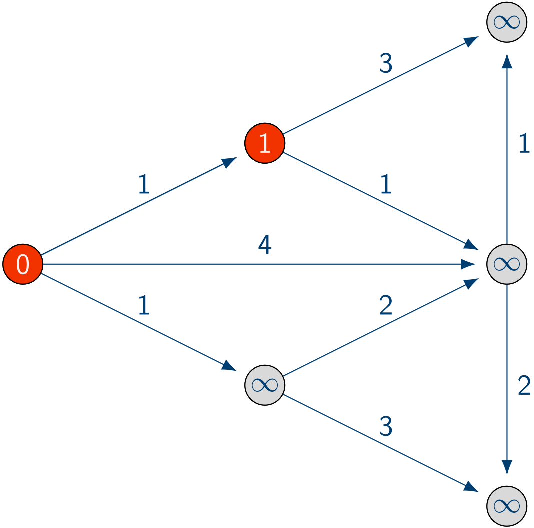
Example of Dijkstra's Algorithm



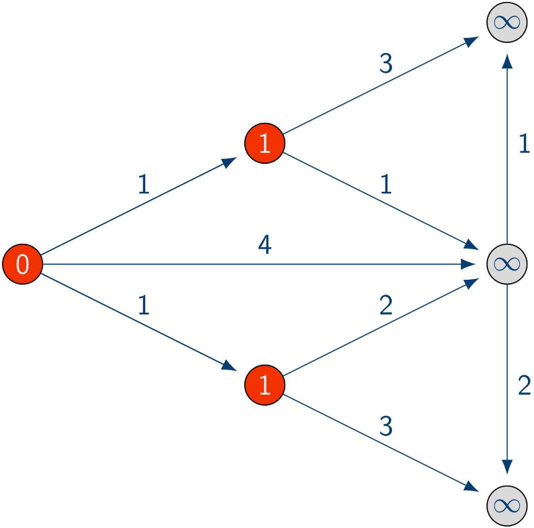
Example of Dijkstra's Algorithm



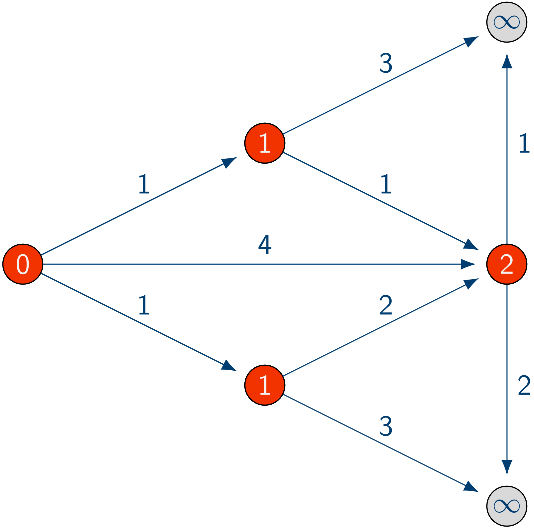
Example of Dijkstra's Algorithm



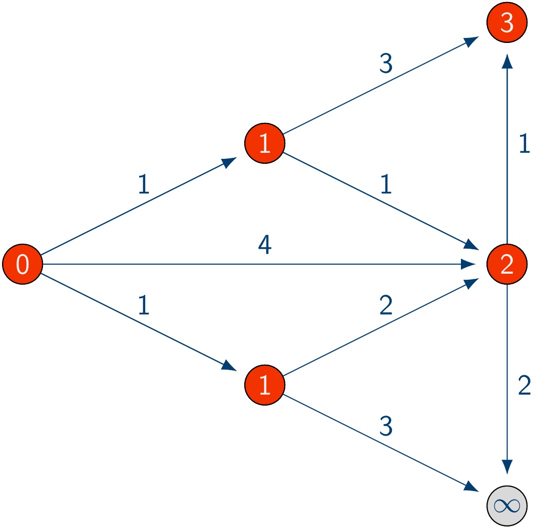
Example of Dijkstra's Algorithm



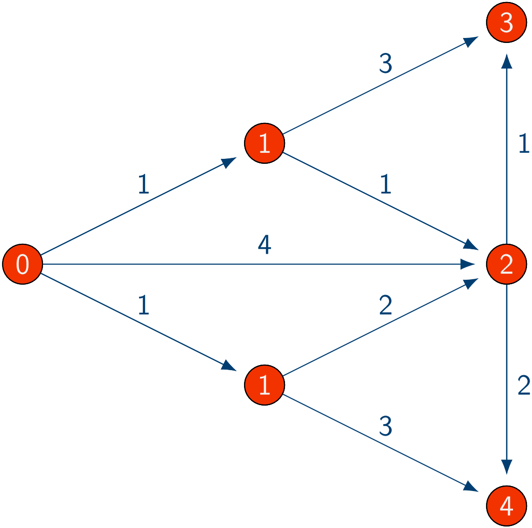
Example of Dijkstra's Algorithm



Example of Dijkstra's Algorithm



Example of Dijkstra's Algorithm

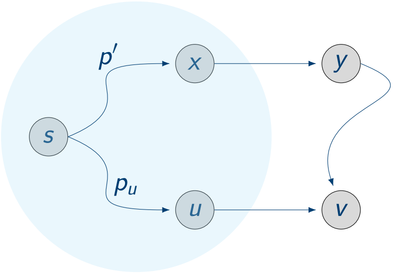


Proof of Correctness

- Let P_u be the shortest path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive step: we add the node v to S. Let u be the v's predecessor on the path P_v. Could there be a shorter path P from s to v?

Proof of Correctness

- Let P_u be the shortest path computed for a node u.
- Claim: P_u is the shortest path from s to u.
- Prove by induction on the size of S.
  - Base case: $|S| = 1$. The only node in S is s.
  - Inductive step: we add the node v to S. Let u be the v's predecessor on the path P_v. Could there be a shorter path P from s to v?



The alternate $s - v$ path P through x and y already too long by the time it had left the set S

Comments about Dijkstra's Algorithm

- Algorithm **cannot handle negative** edge lengths.
- Union of shortest paths output form a tree. Why?

Implementing Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

Implementing Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;

while $S \neq V$ **do**
  Select a node $v \notin S$ with at least one edge from S for which
    $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
  Add v to S and define d[v] = d'[v];
end

- How many iterations are there of the while loop? .

Implementing Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;

while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

- How many iterations are there of the while loop? $n - 1$.

Implementing Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

- How many iterations are there of the while loop? $n - 1$.
- In each iteration, for each node $v \notin S$, compute

$$\min_{e=(u,v),u \in S} d(u) + l_e.$$

Implementing Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;

while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

- How many iterations are there of the while loop? $n - 1$.
- In each iteration, for each node $v \notin S$, compute

$$\min_{e=(u,v),u \in S} d(u) + l_e.$$

- Running time **per iteration** is .

Implementing Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

- How many iterations are there of the while loop? $n - 1$.
- In each iteration, for each node $v \notin S$, compute
$$\min_{e=(u,v),u \in S} d(u) + l_e.$$
- Running time **per iteration** is $O(m)$, yielding an overall running time of $O(nm)$.

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;
while $S \neq V$ **do**
  Select a node $v \notin S$ with at least one edge from S for which
    $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
  Add v to S and define d[v] = d'[v];
end

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;
while $S \neq V$ **do**
    Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
    Add v to S and define d[v] = d'[v];
end

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a **priority queue**.
- Determine the next node v to add to S using ExtractMin.
- After adding v, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$, update w's key using ChangeKey.

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;

while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a **priority queue**.
- Determine the next node v to add to S using ExtractMin.
- After adding v, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$, update w's key using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked? .

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

Let S be the set of explored nodes;
foreach $u \in S$ **do** store distance $d[u] = \infty$;
Initially $d[s] = 0$ and $S = s$;

while $S \neq V$ **do**
  Select a node $v \notin S$ with at least one edge from S for which $d'(v) = \min_{e=(u,v):u \in S} d[u] + W(e)$ is as small as possible;
  Add v to S and define d[v] = d'[v];
end

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a **priority queue**.
- Determine the next node v to add to S using ExtractMin.
- After adding v, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$, update w's key using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked? $n - 1$ and m times, respectively. .

A Faster implementation of Dijkstra's Algorithm

Algorithm: Shortest path algorithm – Dijkstra

input : A graph $G = (V, E)$, a weight map W and a source node s.
output: The distances of the vertices from s

```
Let S be the set of explored nodes;
foreach u ∈ S do store distance d[u] = ∞;
Initially d[s] = 0 and S = s;
while S ≠ V do
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d[u] + W(e) is as small as possible;
    Add v to S and define d[v] = d'[v];
end
```

- Observation: If we add v to S, $d'(w)$ changes only for v's neighbours.
- Store the minima $d'(v)$ for each node $v \in V - S$ in a **priority queue**.
- Determine the next node v to add to S using ExtractMin.
- After adding v, for each neighbour w of v, compute $d(v) + l_{(v,w)}$.
- If $d(v) + l_{(v,w)} < d'(w)$, update w's key using ChangeKey.
- How many times are ExtractMin and ChangeKey invoked? $n - 1$ and m times, respectively. Total running time is $O(m \log n)$.

Teoria dos Grafos e Computabilidade

— Minimum Spanning Trees —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Network Design

- Connect a set of nodes using a set of edges with certain properties.
- Input is usually a graph and the desired network (the output) should use subset of edges in the graph.
- Example: connect all nodes using a cycle of shortest total length.

Network Design

- Connect a set of nodes using a set of edges with certain properties.
- Input is usually a graph and the desired network (the output) should use subset of edges in the graph.
- Example: connect all nodes using a cycle of shortest total length. This problem is the NP-complete traveling salesman problem.

Minimum Spanning Tree (MST)

- Given an undirected graph $G = (V, E)$ with a cost $c_e > 0$ associated with each edge $e \in E$.
- Find a subset T of edges such that the graph (V, T) is connected and the cost $\sum_{e \in T} c_e$ is as small as possible.

Minimum Spanning Tree (MST)

- Given an undirected graph $G = (V, E)$ with a cost $c_e > 0$ associated with each edge $e \in E$.
- Find a subset T of edges such that the graph (V, T) is connected and the cost $\sum_{e \in T} c_e$ is as small as possible.

MINIMUM SPANNING TREE

INSTANCE An undirected graph $G = (V, E)$ and a function $c : E \to \mathbb{R}^+$

SOLUTION A set $T \subseteq E$ of edges such that (V, T) is connected and the $\sum_{e \in T} c_e$ is as small as possible.

Minimum Spanning Tree (MST)

- Given an undirected graph $G = (V, E)$ with a cost $c_e > 0$ associated with each edge $e \in E$.
- Find a subset T of edges such that the graph (V, T) is connected and the cost $\sum_{e \in T} c_e$ is as small as possible.

MINIMUM SPANNING TREE

INSTANCE An undirected graph $G = (V, E)$ and a function $c : E \to \mathbb{R}^+$

SOLUTION A set $T \subseteq E$ of edges such that (V, T) is connected and the $\sum_{e \in T} c_e$ is as small as possible.

- Claim: If T is a minimum-cost solution to this network design problem then (V, T) is a tree.
- A subset T of E is a spanning tree of G if (V, T) is a tree.

Greedy Algorithm for the MST Problem

- Template: process edges in some order. Add an edge to T if tree property is not violated.

Greedy Algorithm for the MST Problem

- Template: process edges in some order. Add an edge to T if tree property is not violated.

  Increasing cost order *Process edges in increasing order of cost. Discard an edge if it creates a cycle.*

  Dijkstra-like *Start from a node s and grow T outward from s: add the node that can be attached most cheaply to current tree.*

  Decreasing cost order *Delete edges in order of decreasing cost as long as graph remains connected.*

Greedy Algorithm for the MST Problem

- Template: process edges in some order. Add an edge to T if tree property is not violated.

  Increasing cost order *Process edges in increasing order of cost. Discard an edge if it creates a cycle.*

  Dijkstra-like *Start from a node s and grow T outward from s: add the node that can be attached most cheaply to current tree.*

  Decreasing cost order *Delete edges in order of decreasing cost as long as graph remains connected.*

- Which of these algorithms works?

Greedy Algorithm for the MST Problem

- Template: process edges in some order. Add an edge to T if tree property is not violated.

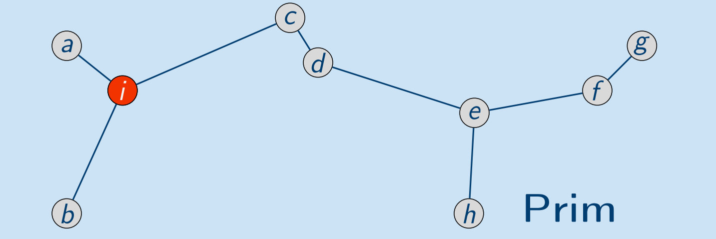
  Increasing cost order *Process edges in increasing order of cost. Discard an edge if it creates a cycle.* *Kruskal's algorithm*

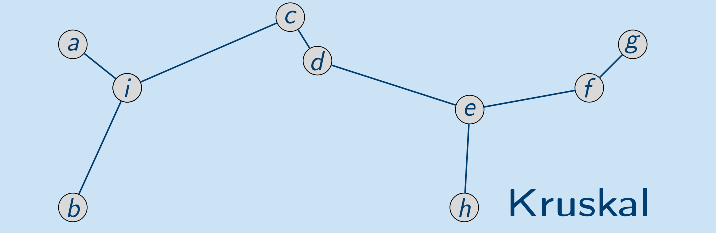
  Dijkstra-like *Start from a node s and grow T outward from s: add the node that can be attached most cheaply to current tree.* *Prim's algorithm*

  Decreasing cost order *Delete edges in order of decreasing cost as long as graph remains connected.* *Reverse-Delete algorithm*

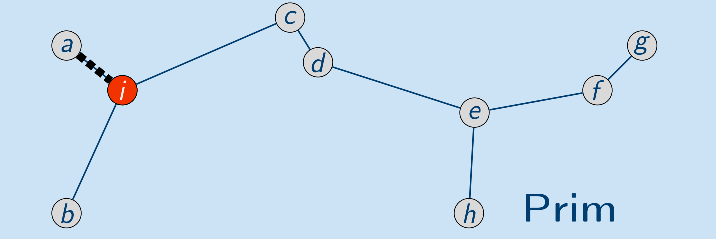
- Which of these algorithms works? All of them!

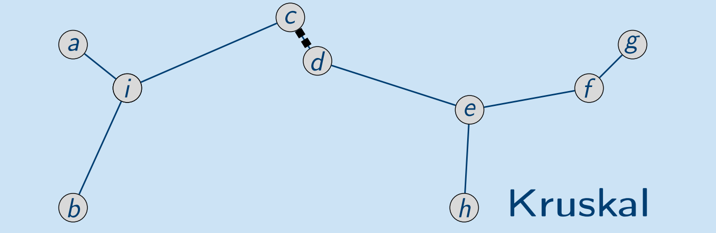
Example of Prim's and Kruskal's Algorithms



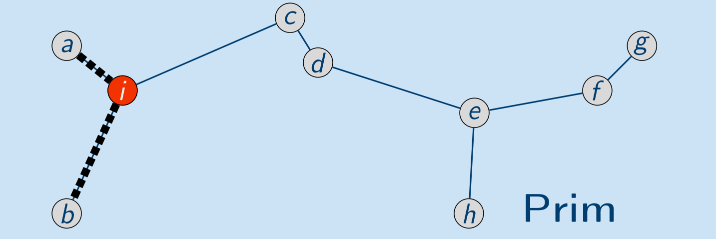


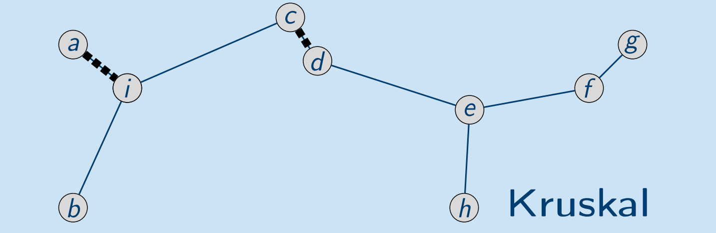
Example of Prim's and Kruskal's Algorithms



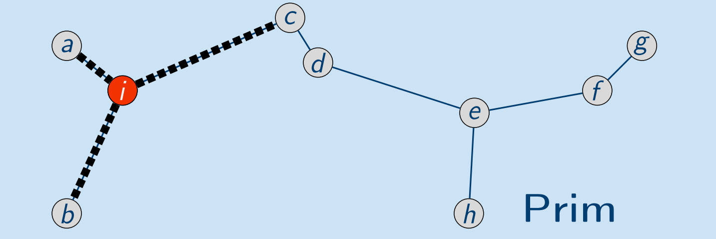


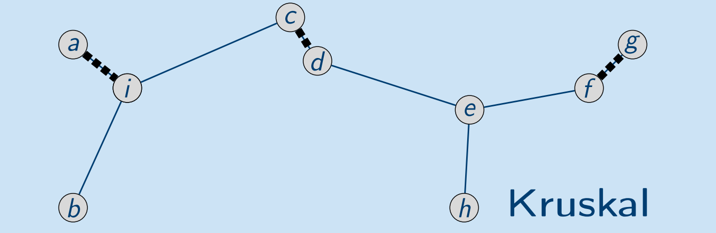
Example of Prim's and Kruskal's Algorithms



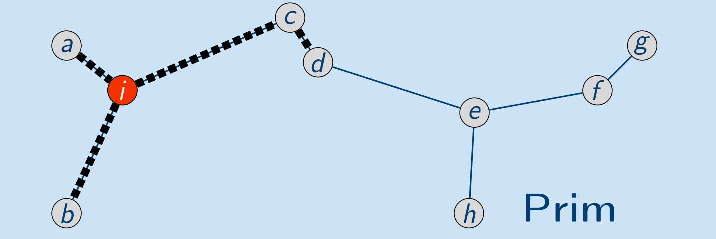


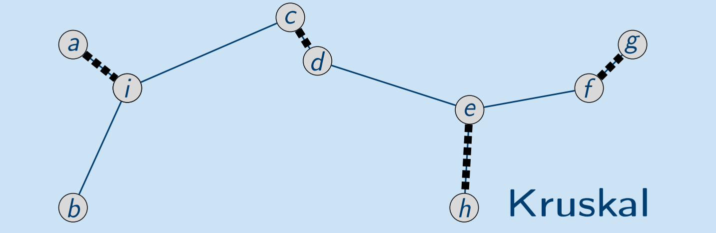
Example of Prim's and Kruskal's Algorithms



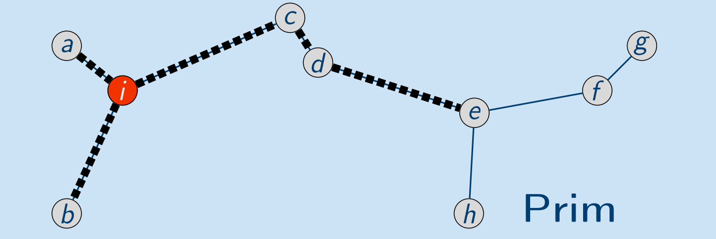


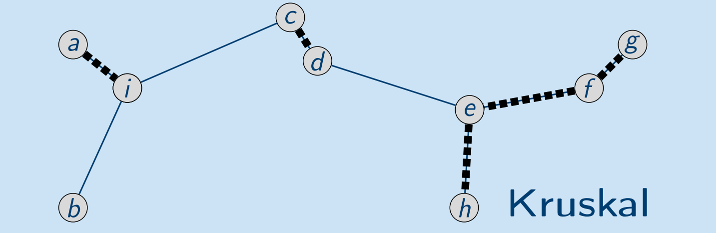
Example of Prim's and Kruskal's Algorithms





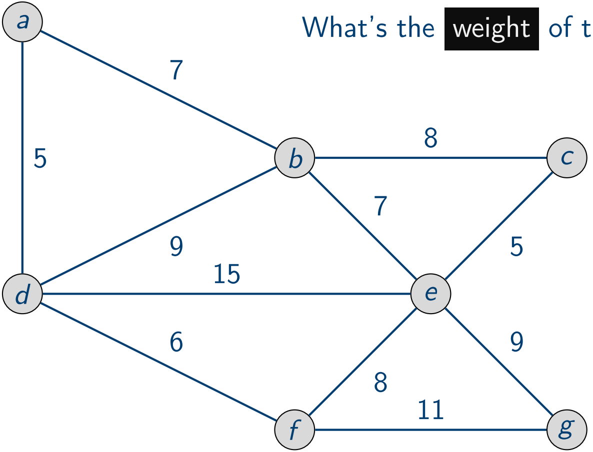
Example of Prim's and Kruskal's Algorithms





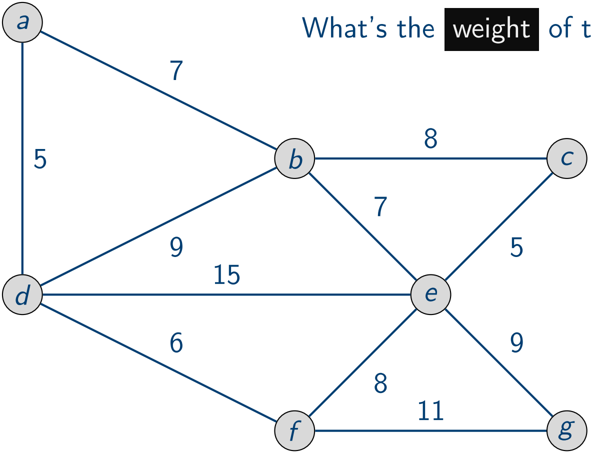
Example of Prim's Algorithm

What's the weight of the MST?



Example of Kruskal's Algorithm

What's the weight of the MST?

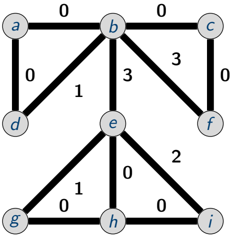


Graph Cuts

- A cut in a graph $G = (V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).

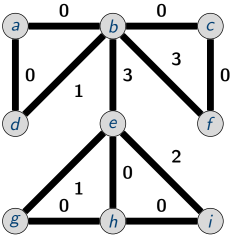
Graph Cuts

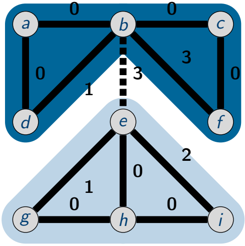
- A cut in a graph $G = (V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).



Graph Cuts

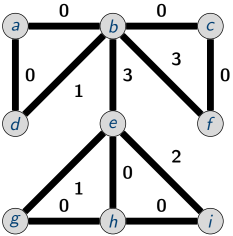
- A cut in a graph $G = (V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).

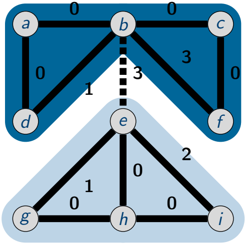




Graph Cuts

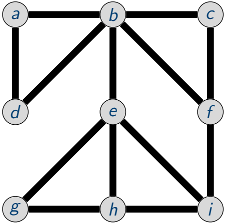
- A cut in a graph $G = (V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).
- Every set $S \subset V$ (S cannot be empty or the entire set V) has a corresponding cut: $\mathrm{cut}(S)$ is the set of edges (v, w) such that $v \in S$ and $w \in V - S$.





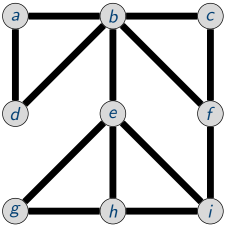
Graph Cuts

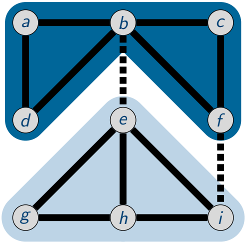
- A cut in a graph $G = (V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).
- Every set $S \subset V$ (S cannot be empty or the entire set V) has a corresponding cut: $\text{cut}(S)$ is the set of edges (v, w) such that $v \in S$ and $w \in V - S$.



Graph Cuts

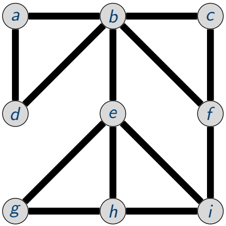
- A cut in a graph $G = (V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).
- Every set $S \subset V$ (S cannot be empty or the entire set V) has a corresponding cut: $\text{cut}(S)$ is the set of edges (v, w) such that $v \in S$ and $w \in V - S$.

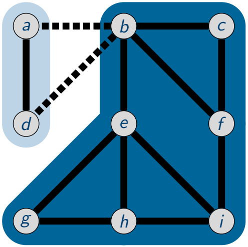




Graph Cuts

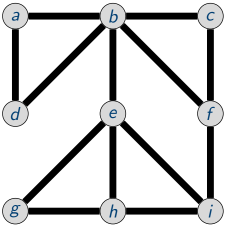
- A cut in a graph $G = (V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).
- Every set $S \subset V$ (S cannot be empty or the entire set V) has a corresponding cut: $\text{cut}(S)$ is the set of edges (v, w) such that $v \in S$ and $w \in V - S$.

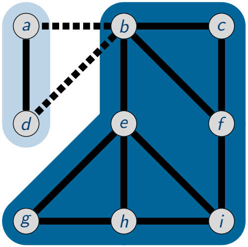




Graph Cuts

- A cut in a graph $G = (V, E)$ is a set of edges whose removal disconnects the graph (into two or more connected components).
- Every set $S \subset V$ (S cannot be empty or the entire set V) has a corresponding cut: cut(S) is the set of edges (v, w) such that $v \in S$ and $w \in V - S$.
- cut(S) is a cut because deleting the edges in cut(S) disconnects S from $V - S$.





Cut Property

- When is it **safe** to include an edge in an MST?

Cut Property

- When is it **safe** to include an edge in an MST?
- Assume all edge costs are distinct.
- Let $S \subset V$, S is not empty or equal to V.
- Let e be the cheapest edge in cut(S).
- Claim: every MST contains e.

Cut Property

- When is it **safe** to include an edge in an MST?
- Assume all edge costs are distinct.
- Let $S \subset V$, S is not empty or equal to V.
- Let e be the cheapest edge in cut(S).
- Claim: every MST contains e.
- Proof: exchange argument. If a supposed MST T does not contain e, show that there is a tree with smaller cost than T that contains e.

Using the Cut Property

- Let F be the set of all edges that satisfy the cut property.
- Is the graph induced by F connected?
- Can the graph induced by F contain a cycle?
- How many edges can F contain?

Using the Cut Property

- Let F be the set of all edges that satisfy the cut property.
- Is the graph induced by F connected? **Yes.**
- Can the graph induced by F contain a cycle? **No.**
- How many edges can F contain? $n - 1$

Using the Cut Property

- Let F be the set of all edges that satisfy the cut property.
- Is the graph induced by F connected? **Yes.**
- Can the graph induced by F contain a cycle? **No.**
- How many edges can F contain? $n - 1$
- F is the unique MST.
- Kruskal's and Prim's algorithms compute F **efficiently**.

Optimality of Kruskal's Algorithm

- Kruskal's algorithm:
  - Start with an empty set T of edges.
  - Process edges in E in non decreasing order of cost.
  - Add the next edge e to T only if adding e does not create a cycle. Discard e if it creates a cycle.
- Claim: Kruskal's algorithm outputs an MST.

Optimality of Kruskal's Algorithm

- Kruskal's algorithm:
  - Start with an empty set T of edges.
  - Process edges in E in non decreasing order of cost.
  - Add the next edge e to T only if adding e does not create a cycle. Discard e if it creates a cycle.
- Claim: Kruskal's algorithm outputs an MST.
  1. For every edge e added, demonstrate the existence of S and $V - S$ such that e and S satisfy the cut property.
  2. Prove that the algorithm computes a spanning tree.

Optimality of Prim's Algorithm

- Prim's algorithm: Maintain a tree (S, U)
  - Start with an **arbitrary** node $s \in S$ and $U = \emptyset$.
  - Add the node v to S and the edge e to U that **minimize**

$$\min_{e=(u,v), u \in S, v \notin S} c_e \equiv \min_{e \in \text{cut}(S)} c_e.$$

  - Stop when $S = V$.
- Claim: Prim's algorithm outputs an MST.

Optimality of Prim's Algorithm

- Prim's algorithm: Maintain a tree (S, U)
  - Start with an **arbitrary** node $s \in S$ and $U = \emptyset$.
  - Add the node v to S and the edge e to U that **minimize**

$$\min_{e=(u,v), u \in S, v \notin S} c_e \equiv \min_{e \in \mathrm{cut}(S)} c_e.$$

  - Stop when $S = V$.
- Claim: Prim's algorithm outputs an MST.
  1. Prove that every edge inserted satisfies the cut property.
  2. Prove that the graph constructed is a spanning tree.

Cycle Property

- When can we be sure that an edge cannot be in *any* MST?

Cycle Property

- When can we be sure that an edge cannot be in *any* MST?
- Let C be any cycle in G and let $e = (v, w)$ be the most expensive edge in C.
- Claim: e does not belong to any MST of G.

Cycle Property

- When can we be sure that an edge cannot be in *any* MST?
- Let C be any cycle in G and let $e = (v, w)$ be the most expensive edge in C.
- Claim: e does not belong to any MST of G.
- Proof: exchange argument. If a supposed MST T contains e, show that there is a tree with smaller cost than T that does not contain e.

Optimality of the Reverse-Delete Algorithm

- Reverse-Delete algorithm: Maintain a set E' of edges.
  - Start with $E' = E$.
  - Process edges in non increasing order of cost.
  - Delete the next edge e from E' only if (V, E') is connected after removal.
  - Stop after processing all the edges.
- Claim: the Reverse-Delete algorithm outputs an MST.

Optimality of the Reverse-Delete Algorithm

- Reverse-Delete algorithm: Maintain a set E' of edges.
  - Start with $E' = E$.
  - Process edges in non increasing order of cost.
  - Delete the next edge e from E' only if (V, E') is connected after removal.
  - Stop after processing all the edges.
- Claim: the Reverse-Delete algorithm outputs an MST.
  1. Show that every edge deleted belongs to no MST.
  2. Prove that the graph remaining at the end is a spanning tree.

Comments on MST Algorithms

- To handle **multiple edges** with the **same weight**, perturb each length by a random infinitesimal amount.
- **Any** algorithm that constructs a spanning tree by including edges that satisfy the cut property and deleting edges that satisfy the cycle property will yield an **MST**!

Teoria dos Grafos e Computabilidade

— Implementation —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Feb 2025

Implementing Prim's Algorithm

- Maintain a tree (S, U).
  - Start with an arbitrary node $s \in V$ and $U = \emptyset$.
  - Add the node v to S and the edge e to U that minimize

$$\min_{e \in \mathrm{cut}(S)} c_e.$$

  - Stop when $S = V$.

Implementing Prim's Algorithm

- Maintain a tree (S, U).
  - Start with an arbitrary node $s \in V$ and $U = \emptyset$.
  - Add the node v to S and the edge e to U that minimize

$$\min_{e \in \mathrm{cut}(S)} c_e.$$

  - Stop when $S = V$.
- Sorting edges takes $O(m \log n)$ time.
- Implementation is very similar to Dijkstra's algorithm.
- Maintain S and store attachment costs $a(v) = \min_{e \in \mathrm{cut}(S)} c_e$ for every node $v \in V - S$ in a priority queue.
- At each step, extract minimum v from priority queue and update the attachment costs of the neighbours of v.
- Total of $n - 1$ ExtractMin and m ChangeKey operations, yielding a running time of $O(m \log n)$.

Implementing Kruskal's Algorithm

- Start with an empty set T of edges.
- Process edges in E in increasing order of cost.
- Add the next edge e to T only if adding e does not create a cycle.

Implementing Kruskal's Algorithm

- Start with an empty set T of edges.
- Process edges in E in increasing order of cost.
- Add the next edge e to T only if adding e does not create a cycle.
- Sorting edges takes $O(m \log n)$ time.
- Key question: "Does adding $e = (u, v)$ to T create a cycle?"
  - Maintain set of connected components of T.
  - FIND(u): return the name of the connected component of T that u belongs to.
  - UNION(A, B): merge connected components A and B.
- Answering the question: Adding e creates a cycle if and only if FIND(u) = FIND(v). If not, execute UNION(FIND(u), FIND(v)).

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need?

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need? $2m$.
- How many UNION invocations does Kruskal's algorithm need?

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need? $2m$.
- How many UNION invocations does Kruskal's algorithm need? $n - 1$.

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need? $2m$.
- How many UNION invocations does Kruskal's algorithm need? $n - 1$.
- We will show two implementations of UNION-FIND:
  - Each FIND takes $O(1)$ time, k invocations of UNION take $O(k \log k)$ time in total.
  - Each FIND takes $O(\log n)$ time and each invocation of UNION takes $O(1)$ time.

Analysing Kruskal's Algorithm

- How many FIND invocations does Kruskal's algorithm need? $2m$.
- How many UNION invocations does Kruskal's algorithm need? $n-1$.
- We will show two implementations of UNION-FIND:
  - Each FIND takes $O(1)$ time, k invocations of UNION take $O(k \log k)$ time in total.
  - Each FIND takes $O(\log n)$ time and each invocation of UNION takes $O(1)$ time.
- Total running time of Kruskal's algorithm is $O(m \log n)$.

Teoria dos Grafos e Computabilidade

— Huffman code —

Silvio Jamil F. Guimarães

Graduate Program in Informatics – PPGINF
Laboratory of Image and Multimedia Data Science – IMScience
Pontifical Catholic University of Minas Gerais – PUC Minas

Feb 2025

Data compression

Given a text that uses 32 symbols (26 different letters, space, and some punctation characters), how can we **encode** this text in bits?

Data compression

Given a text that uses 32 symbols (26 different letters, space, and some punctation characters), how can we encode this text in bits?

We can encode 2^5 different symbols using a fixed length of 5 bits per symbol. This is called fixed length encoding.

Data compression

Given a text that uses 32 symbols (26 different letters, space, and some punctation characters), how can we encode this text in bits?

We can encode 2^5 different symbols using a fixed length of 5 bits per symbol. This is called fixed length encoding.

Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?

Data compression

Given a text that uses 32 symbols (26 different letters, space, and some punctation characters), how can we encode this text in bits?

We can encode 2^5 different symbols using a fixed length of 5 bits per symbol. This is called fixed length encoding.

Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?

Encode these characters with fewer bits, and the others with more bits

Data compression

Given a text that uses 32 symbols (26 different letters, space, and some punctation characters), how can we encode this text in bits?

We can encode 2^5 different symbols using a fixed length of 5 bits per symbol. This is called fixed length encoding.

Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?

Encode these characters with fewer bits, and the others with more bits

How do we know when the next symbol begins?

Data compression

Given a text that uses 32 symbols (26 different letters, space, and some punctation characters), how can we encode this text in bits?

We can encode 2^5 different symbols using a fixed length of 5 bits per symbol. This is called fixed length encoding.

Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?

Encode these characters with fewer bits, and the others with more bits

How do we know when the next symbol begins?

Use a separation symbol (like the pause in Morse), or make sure that there is no ambiguity by ensuring that no code is a prefix of another one

Data compression

A prefix code for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x, y \in S$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$.

Data compression

A **prefix code** for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x, y \in S$, $x \neq y$, $c(x)$ is **not a prefix** of $c(y)$.

c(a) $= 11$

c(e) $= 01$

c(k) $= 001$

c(l) $= 10$

c(u) $= 000$

Data compression

A prefix code for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x, y \in S$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$.

c(a) $= 11$
c(e) $= 01$
c(k) $= 001$
c(l) $= 10$
c(u) $= 000$

What is the meaning of 1001000001 ?

Suppose frequencies are known in a text of 1G characters: $f_a = 0.4$, $f_e = 0.2$, $f_k = 0.2$, $f_l = 0.1$, $f_u = 0.1$. What is the size of the encoded text?

Data compression

A prefix code for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x, y \in S$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$.

c(a) $= 11$

c(e) $= 01$

c(k) $= 001$

c(l) $= 10$

c(u) $= 000$

What is the meaning of 1001000001 ?

Suppose frequencies are known in a text of 1G characters: $f_a = 0.4$, $f_e = 0.2$, $f_k = 0.2$, $f_l = 0.1$, $f_u = 0.1$. What is the size of the encoded text?

2*fa + 2*fe + 3*fk + 2*fl + 4*fu = 2.4 G bits

How to **greedly** compute a prefix tree to encode an alphabet?

Data compression: how to create a prefix tree

How to greedly compute a prefix tree to encode an alphabet?

Suppose frequencies are known:
$f_a = 0.32$, $f_e = 0.25$, $f_k = 0.20$, $f_l = 0.18$, $f_u = 0.05$.

How to create an encoding to minimize the size of a text?

Data compression: how to create a prefix tree

How to **greedly** compute a prefix tree to encode an alphabet?

Suppose frequencies are known:
$f_a = 0.32, f_e = 0.25, f_k = 0.20, f_l = 0.18, f_u = 0.05.$

How to create an encoding to **minimize** the size of a text?

Data compression: how to create a prefix tree

How to **greedly** compute a prefix tree to encode an alphabet?

Suppose frequencies are known:
$f_a = 0.32, f_e = 0.25, f_k = 0.20, f_l = 0.18, f_u = 0.05.$

How to create an encoding to **minimize** the size of a text?

Algorithm: Huffman code

input : A set S of elements with their frequencies.
output: A prefix tree

```
if S = 2 then
    return a tree with root and 2 leaves;
else
    let y and z be lowest-frequency letters in S;
    S' = S;
    remove y and z from S';
    insert new letter w in S' with f_w = f_y + f_z;
    T'= Huffman(S');
    T = add two children y and z to leaf w from T';
    return T;
end
```