COSC1285/2123: Algorithms & Analysis

Greedy Techniques

Jeffrey Chan

RMIT University
Email : jeffrey.chan@rmit.edu.au

Lecture 9

Overview

Levitin – The design and analysis of algorithms

This week we will be covering the material from Chapter 9.

Learning outcomes:

- Understand and be able to apply the greedy approach to solving problems.
- Examples:
  - spanning tree – Prim's algorithm
  - spanning tree – Kruskal's algorithm
  - single source shortest-path – Dijkstra's algorithm
  - data compression

Outline

1. Overview
2. Prim's Algorithm
3. Kruskal's Algorithm
4. Dijkstra's Algorithm
5. Data Compression
6. Summary

Overview

Greedy Algorithms build up a solution piece by piece, always choosing the next piece that offers the most immediate and obvious benefit.

Sometimes such an approach can lead to an inferior solution, but in other cases it can lead to a simple and optimal solution.

Overview

Minimum Spanning Tree

Spanning Tree Problem

A spanning tree of a connected graph is a connected acyclic subgraph (i.e. tree) which contains all the vertices of the graph and a subset of edges from the original graph.

Minimum Spanning Tree

Spanning Tree Problem

A spanning tree of a connected graph is a connected acyclic subgraph (i.e. tree) which contains all the vertices of the graph and a subset of edges from the original graph.

Minimum Spanning Tree Problem

A minimum spanning tree of a weighted connected graph is the spanning tree of the smallest total weight (sum of the weights on all of the tree's edges).

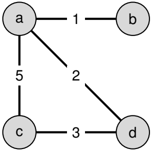
Minimum Spanning Tree

Spanning Tree Problem

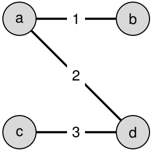
A spanning tree of a connected graph is a connected acyclic subgraph (i.e. tree) which contains all the vertices of the graph and a subset of edges from the original graph.

Minimum Spanning Tree Problem

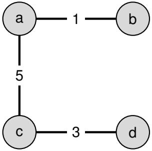
A minimum spanning tree of a weighted connected graph is the spanning tree of the smallest total weight (sum of the weights on all of the tree's edges).



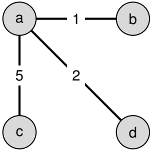
Graph



$w(T_1) = 6$



$w(T_2) = 9$



$w(T_3) = 8$

Applications of Minimum Spanning Tree

- Designing networks (phones, computers etc): Want to connect up a series of offices with telephone or wired lines, but want to minimise cost.
- Approximate solutions to hard problems: travelling salesman
- Generation of perfect mazes

Prim's Algorithm – Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Prim's Algorithm – Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

1. Start with one arbitrary selected vertex and add this to tree.

Prim's Algorithm – Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

1. Start with one arbitrary selected vertex and add this to tree.
2. Then at each iteration add a neighbouring vertex to the tree that has minimum edge weight to one of the vertices in the current tree. It must not be in the tree.

Prim's Algorithm – Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

1. Start with one arbitrary selected vertex and add this to tree.
2. Then at each iteration add a neighbouring vertex to the tree that has minimum edge weight to one of the vertices in the current tree. It must not be in the tree.
3. When all vertices added to tree, we are done.

Prim's Algorithm – Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

1. Start with one arbitrary selected vertex and add this to tree.
2. Then at each iteration add a neighbouring vertex to the tree that has minimum edge weight to one of the vertices in the current tree. It must not be in the tree.
3. When all vertices added to tree, we are done.

Note:

- Use a min priority queue to quickly find this neighbouring vertex with minimum edge weight (in literature, the neighbour set is sometimes called the frontier set).

Prim's Algorithm – Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

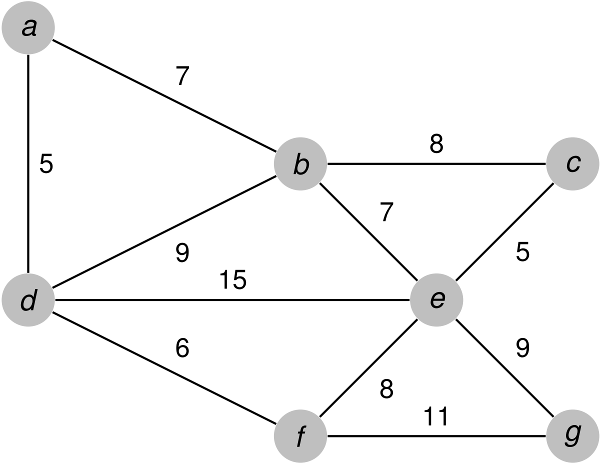
Idea: Select one vertex at a time and add to tree.

1. Start with one arbitrary selected vertex and add this to tree.
2. Then at each iteration add a neighbouring vertex to the tree that has minimum edge weight to one of the vertices in the current tree. It must not be in the tree.
3. When all vertices added to tree, we are done.

Note:

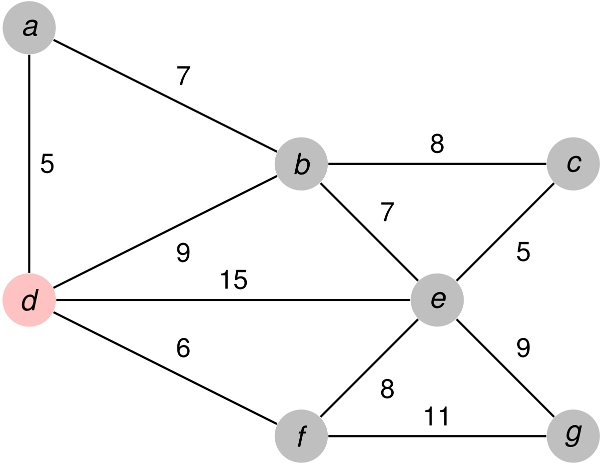
- Use a min priority queue to quickly find this neighbouring vertex with minimum edge weight (in literature, the neighbour set is sometimes called the frontier set).
- When adding, we may need to update the smallest edge weight to a vertex in neighbour set as there may be a smallest edge weight from updated tree to new neighbour set.

Prim's Algorithm – Example



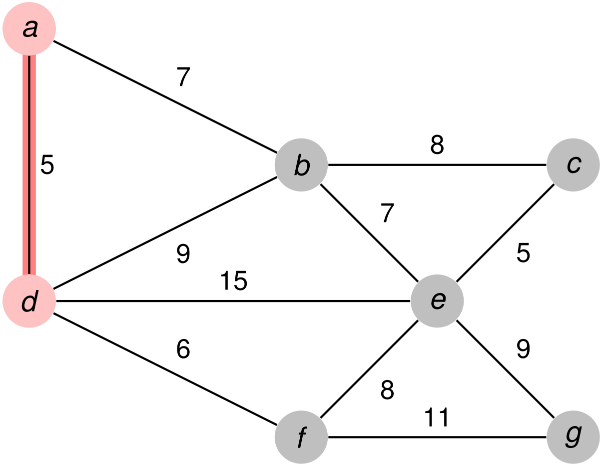
$V_T = \{\}$, $PQ = \{\}$

Prim's Algorithm – Example



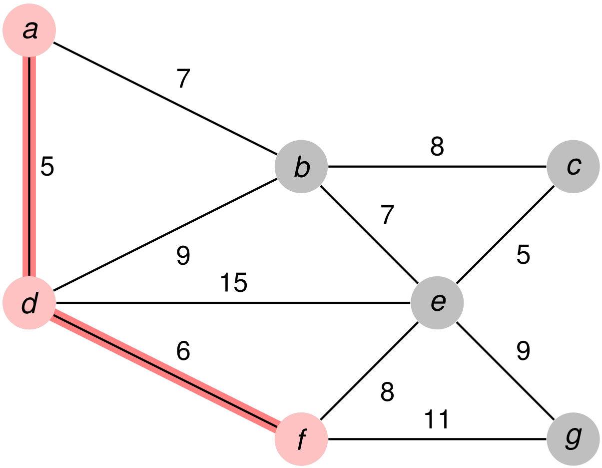
$V_T = \{d\}$, $PQ = \{(a, 5), (f, 6), (b, 9), (e, 15)\}$

Prim's Algorithm – Example



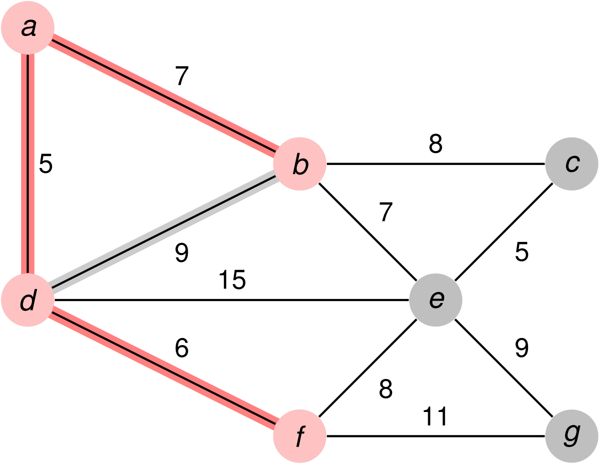
$V_T = \{d, a\}$, $PQ = \{(f, 6), (b, 7), (b, 9), (e, 15)\}$

Prim's Algorithm – Example



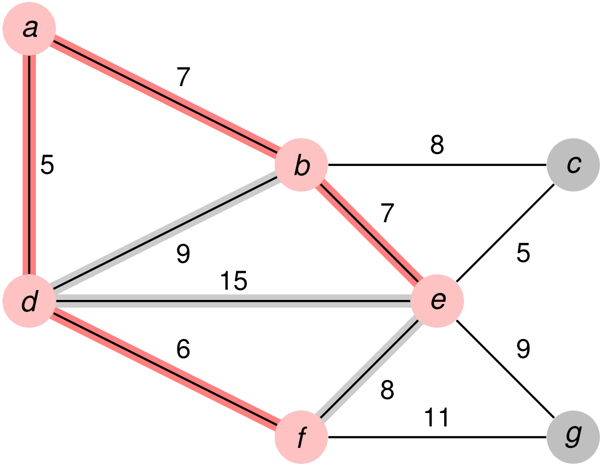
$V_T = \{d, a, f\}$, $PQ = \{(b, 7), (e, 8), (g, 11), (e, 15)\}$

Prim's Algorithm – Example



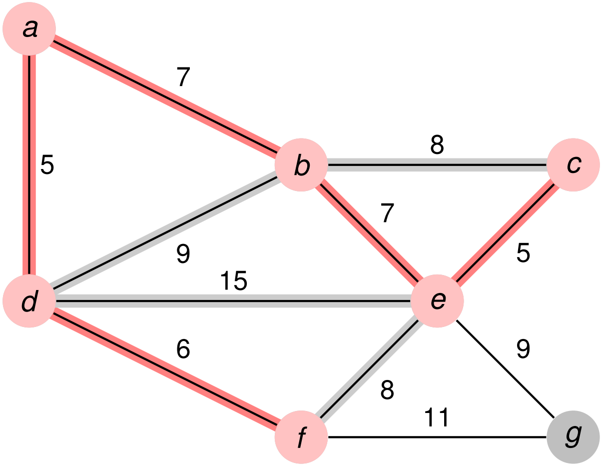
$V_T = \{d, a, f, b\}$, $PQ = \{(e, 7), (e, 8), (c, 8), (g, 11)\}$

Prim's Algorithm – Example



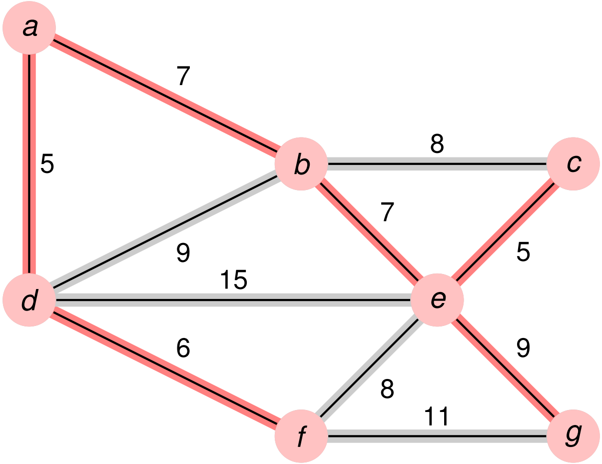
$V_T = \{d, a, f, b, e\}$, $PQ = \{(c, 5), (c, 8), (g, 9), (g, 11)\}$

Prim's Algorithm – Example



$V_T = \{d, a, f, b, e, c\}$, $PQ = \{(g, 9)\}$

Prim's Algorithm – Example



$V_T = \{d, a, f, b, e, c, \textcolor{red}{g}\}$, $PQ = \{\}$

Prim's Algorithm – Summary

- The algorithm always gives the optimal solution, regardless of where you start. See Levitin Chp 9.1 for the proof.

Prim's Algorithm – Summary

- The algorithm always gives the optimal solution, regardless of where you start. See Levitin Chp 9.1 for the proof.
- The efficiency of the algorithm using a min-heap and an adjacency list is $O(|E| \log |V|)$.
- The efficiency of the algorithm using a Fibonacci heap and an adjacency list is $O(|E| + |V| \log |V|)$.

Overview

Kruskal's Algorithm – Sketch

Krushal's algorithm is an alternative method to finding minimum spanning trees. (We will explore why have two algorithms at end of this section)

Kruskal's Algorithm – Sketch

Krushal's algorithm is an alternative method to finding minimum spanning trees. (We will explore why have two algorithms at end of this section)

Idea: Select one edge at a time and add to tree.

1. Start with empty tree.

Kruskal's Algorithm – Sketch

Krushal's algorithm is an alternative method to finding minimum spanning trees. (We will explore why have two algorithms at end of this section)

Idea: Select one edge at a time and add to tree.

1. Start with empty tree.
2. Add one edge at a time to the current tree. Edge selected is the one with minimum weight and doesn't create a cycle.

Kruskal's Algorithm – Sketch

Krushal's algorithm is an alternative method to finding minimum spanning trees. (We will explore why have two algorithms at end of this section)

Idea: Select one edge at a time and add to tree.

1. Start with empty tree.
2. Add one edge at a time to the current tree. Edge selected is the one with minimum weight and doesn't create a cycle.
3. Terminate when $|V| - 1$ such edges have been added. (A tree of $|V|$ nodes has $|V| - 1$ edges).

Kruskal's Algorithm – Sketch

Krushal's algorithm is an alternative method to finding minimum spanning trees. (We will explore why have two algorithms at end of this section)

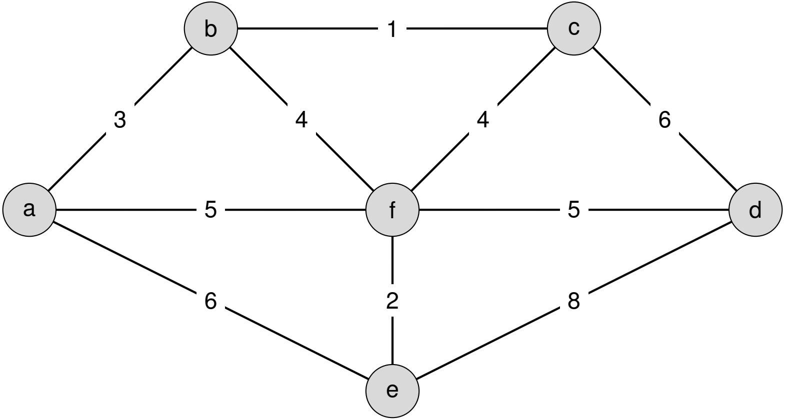
Idea: Select one edge at a time and add to tree.

1. Start with empty tree.
2. Add one edge at a time to the current tree. Edge selected is the one with minimum weight and doesn't create a cycle.
3. Terminate when $|V| - 1$ such edges have been added. (A tree of $|V|$ nodes has $|V| - 1$ edges).

Note:

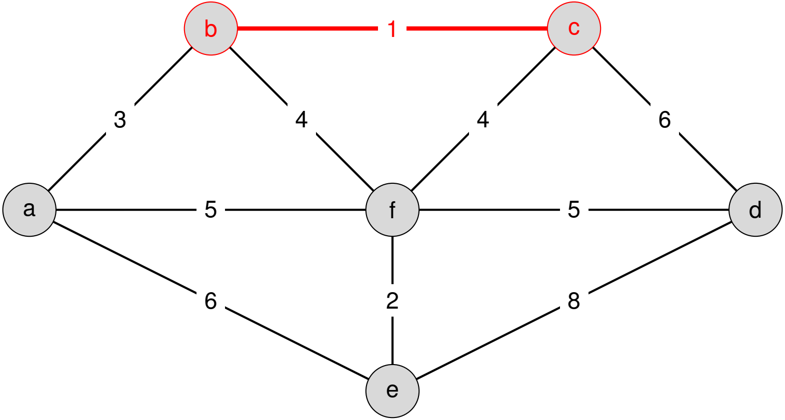
- To be fast in edge selection, we initially sort all edges from smallest to largest by weight.
- Note that the nodes are not always connected in the intermediate stages of the algorithm.

Kruskal's Algorithm – Example



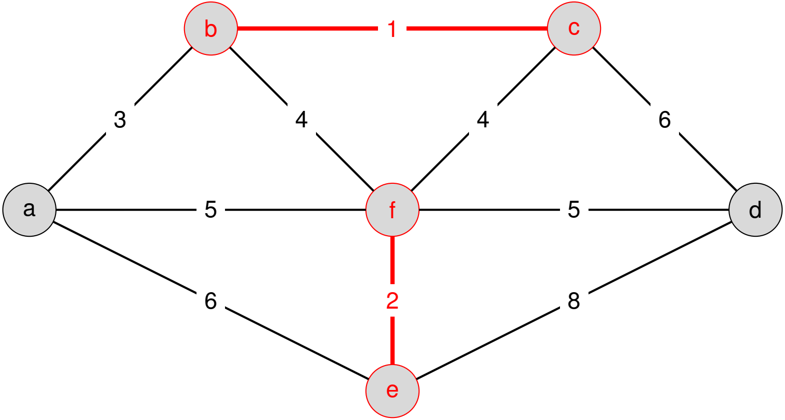
All Edges :	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
Tree Edges :	$\emptyset$									
	$\emptyset$									

Kruskal's Algorithm – Example



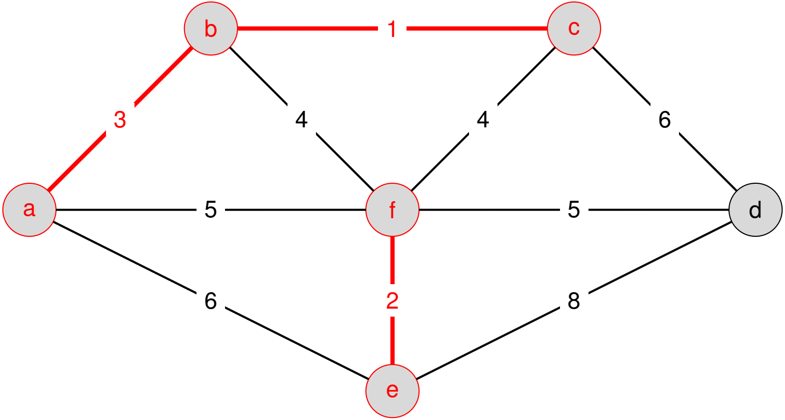
All Edges :	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
Tree Edges :	$\emptyset$									
	$\emptyset$									

Kruskal's Algorithm – Example



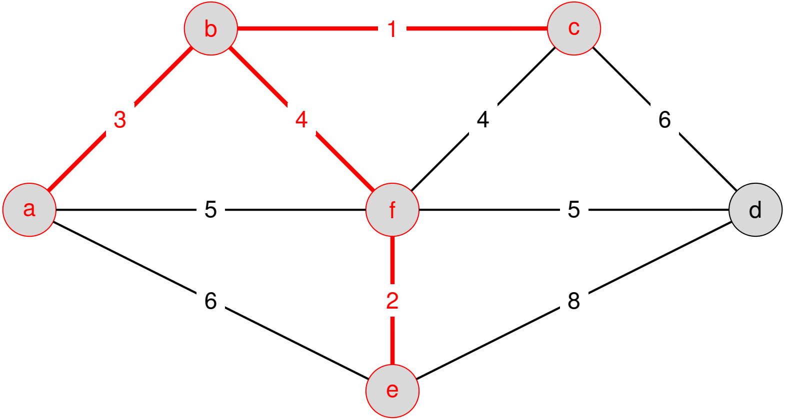
All Edges :	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
Tree Edges :	bc									
	1									

Kruskal's Algorithm – Example



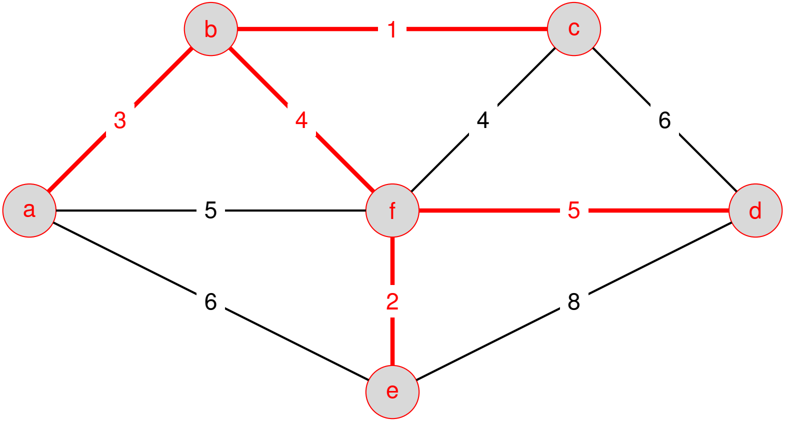
All Edges :	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
Tree Edges :	bc	ef								
	1	2								

Kruskal’s Algorithm – Example



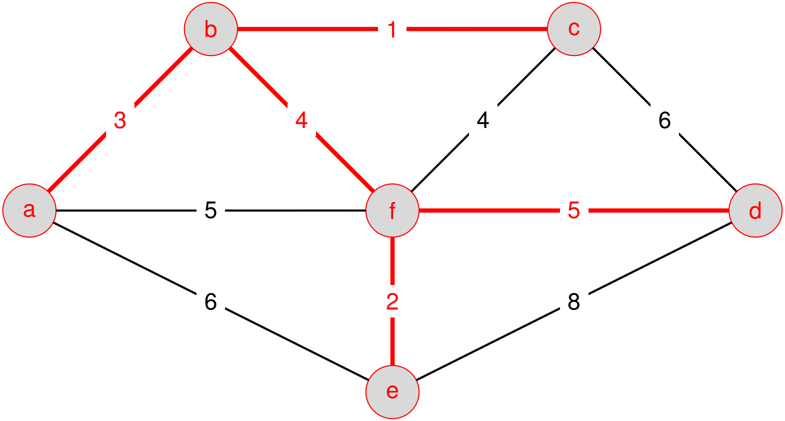
All Edges :	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
Tree Edges :	bc	ef	ab							
	1	2	3							

Kruskal's Algorithm – Example



All Edges :	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
Tree Edges :	bc	ef	ab	bf						
	1	2	3	4						

Kruskal's Algorithm – Example



All Edges :	bc	ef	ab	bf	cf	af	df	ae	cd	de
	1	2	3	4	4	5	5	6	6	8
Tree Edges :	bc	ef	ab	bf	df					
	1	2	3	4	5					

Kruskal's Algorithm – Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.

Kruskal's Algorithm – Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.
- It is harder to implement because checking for cycles requires us to maintain a forest of trees on the intermediate steps.

Kruskal's Algorithm – Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.
- It is harder to implement because checking for cycles requires us to maintain a forest of trees on the intermediate steps.
- This is essentially a case of **Union-Find** algorithm.

Kruskal's Algorithm – Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.
- It is harder to implement because checking for cycles requires us to maintain a forest of trees on the intermediate steps.
- This is essentially a case of **Union-Find** algorithm.
- With an efficient Union-find algorithm, Kruskal's algorithm is slightly less efficient that Prim's algorithm, requiring $O(|E| \log |E|)$ time (dominated by the time to sort the edges).

Kruskal's Algorithm – Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.
- It is harder to implement because checking for cycles requires us to maintain a forest of trees on the intermediate steps.
- This is essentially a case of **Union-Find** algorithm.
- With an efficient Union-find algorithm, Kruskal's algorithm is slightly less efficient that Prim's algorithm, requiring $O(|E| \log |E|)$ time (dominated by the time to sort the edges).
- If do not have to sort edges, runs in $O(|E| \log |V|)$.

Kruskal's Algorithm – Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.
- It is harder to implement because checking for cycles requires us to maintain a forest of trees on the intermediate steps.
- This is essentially a case of **Union-Find** algorithm.
- With an efficient Union-find algorithm, Kruskal's algorithm is slightly less efficient that Prim's algorithm, requiring $O(|E| \log |E|)$ time (dominated by the time to sort the edges).
- If do not have to sort edges, runs in $O(|E| \log |V|)$.
- Note if graph is sparse, $|E|$ same order of magnitude as $|V|$, than Krushal's algorithm can be faster than Prim's (recall $O(|E| + |V| \log |V|)$). Otherwise, Prim's algorithm can be faster.

Minecraft Maze Generation

https://www.youtube.com/watch?v=5mnOClCO_9o

Overview

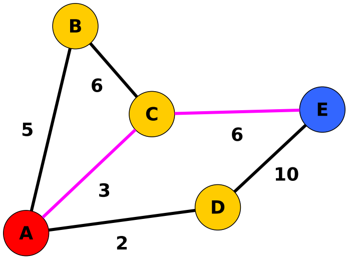
Shortest Paths in Graphs

Problem

Given a weighted connected graph, the shortest-path problem asks to find the shortest path from a starting source vertex to a destination target vertex.

Problem

Given a weighted connected graph, the shortest-path problem asks to find the shortest path from a starting source vertex to a destination target vertex.



Problem

Given a weighted connected graph, the single-source shortest-paths problem asks to find the shortest path to all vertices given a single starting source vertex.

Idea:

- At all times, we maintain our best estimate of the shortest-path distances from source vertex to all other vertices.

Dijkstra's Algorithm

Problem

Given a weighted connected graph, the single-source shortest-paths problem asks to find the shortest path to all vertices given a single starting source vertex.

Idea:

- At all times, we maintain our best estimate of the shortest-path distances from source vertex to all other vertices.
- Initially we do not know, so all these distance estimates are ∞.

Problem

Given a weighted connected graph, the single-source shortest-paths problem asks to find the shortest path to all vertices given a single starting source vertex.

Idea:

- At all times, we maintain our best estimate of the shortest-path distances from source vertex to all other vertices.
- Initially we do not know, so all these distance estimates are ∞.
- But as the algorithm explores the graph, we update our estimates, which converges to the true shortest path distance.

Dijkstra's Algorithm – Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

Dijkstra's Algorithm – Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

1. Initially S is empty. Initialise distance estmates to ∞ for all non-source vertices. Distance of source vertex is 0.

Dijkstra's Algorithm – Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

1. Initially S is empty. Initialise distance estmates to ∞ for all non-source vertices. Distance of source vertex is 0.
2. Select the vertex v not in S with the minimum shortest-path estimate.

Dijkstra's Algorithm – Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

1. Initially S is empty. Initialise distance estmates to ∞ for all non-source vertices. Distance of source vertex is 0.
2. Select the vertex v not in S with the minimum shortest-path estimate.
3. Add v to S.

Dijkstra's Algorithm – Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

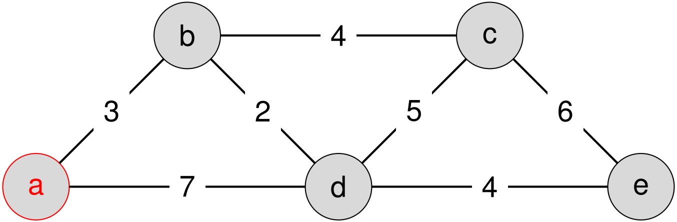
1. Initially S is empty. Initialise distance estmates to ∞ for all non-source vertices. Distance of source vertex is 0.
2. Select the vertex v not in S with the minimum shortest-path estimate.
3. Add v to S.
4. Update our distance estimates to neighbouring vertices that are not in S.

Dijkstra's Algorithm – Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

1. Initially S is empty. Initialise distance estmates to ∞ for all non-source vertices. Distance of source vertex is 0.
2. Select the vertex v not in S with the minimum shortest-path estimate.
3. Add v to S.
4. Update our distance estimates to neighbouring vertices that are not in S.
5. Repeat from step 2, until all vertices have been added to S.

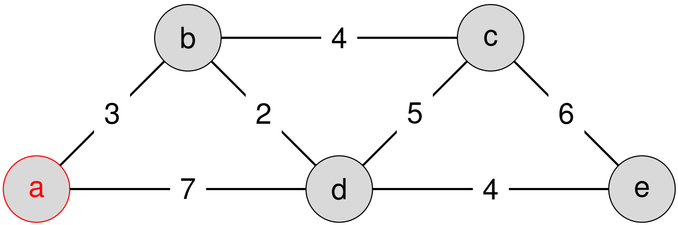
Dijkstra’s Algorithm – Example



a(a,0) b(-,∞) c(-,∞) d(-,∞) e(-,∞)

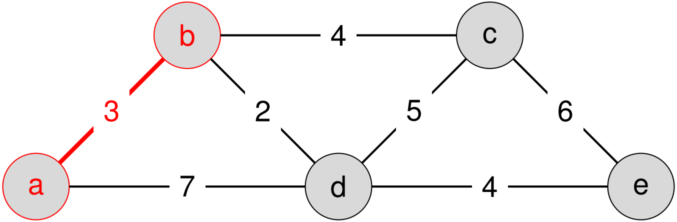
S = { }

Dijkstra's Algorithm – Example



b(a,3) c(-,∞) d(a,7) e(-,∞)

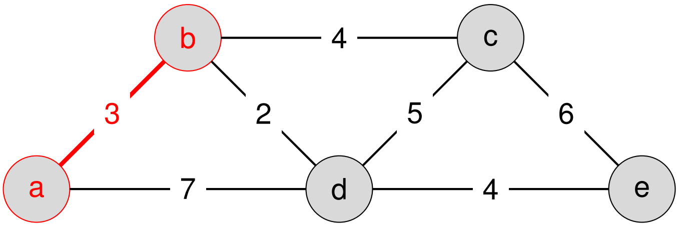
S = {a(a,0)}



b(a,3) c(-,∞) d(a,7) e(-,∞)

S = {a(a,0)}

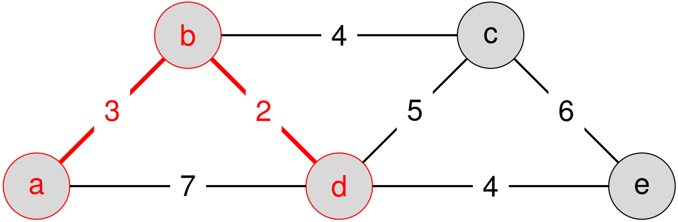
Dijkstra's Algorithm – Example



c(b,3 + 4) d(b,3 + 2) e(-,∞)

S = {a(a,0), b(a,3)}

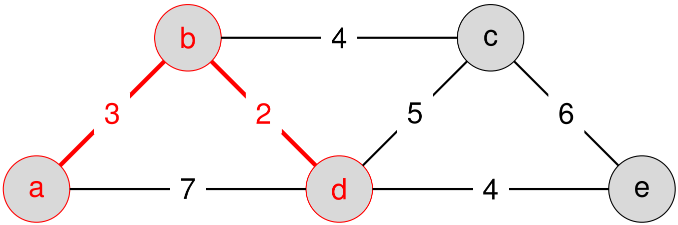
Dijkstra's Algorithm – Example



c(b,7) d(b,5) e(-,∞)

S = {a(a,0), b(a,3)}

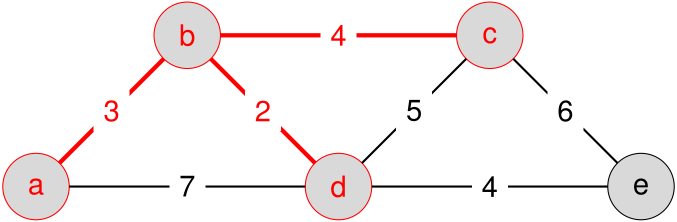
Dijkstra's Algorithm – Example



c(b,7) e(d,5+4)

S = {a(a,0), b(a,3), d(b,5)}

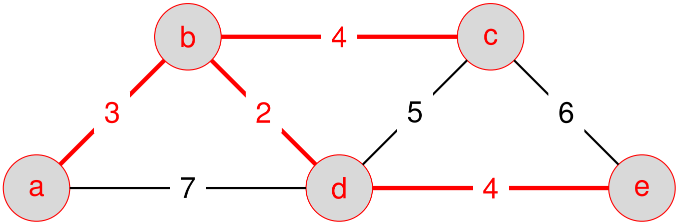
Dijkstra's Algorithm – Example



e(d,9)

S = {a(a,0), b(a,3), d(b,5), c(b,7)}

Dijkstra's Algorithm – Example



S = {a(a,0), b(a,3), d(b,5), c(b,7), e(d,9)}

Dijkstra's Algorithm – Example

So, we have the following distances from vertex *a*:

a(a,0) b(a,3) d(b,5) c(b,7) e(d,9)

Dijkstra's Algorithm – Example

So, we have the following distances from vertex *a*:

a(a,0)   b(a,3)   d(b,5)   c(b,7)   e(d,9)

Which gives the following shortest paths:

Length	Path
3	a - b
5	a - b - d
7	a - b - c
9	a - b - d - e

Dijkstra's Algorithm – Summary

- Dijkstra's algorithm is guaranteed to always return the optimal solution. This is not necessarily true for all greedy algorithms.

Dijkstra's Algorithm – Summary

- Dijkstra's algorithm is guaranteed to always return the optimal solution. This is not necessarily true for all greedy algorithms.
- If we use an adjacency list and a min-heap, the algorithm runs in $\Theta(|E| \log |V|)$ time.

Overview

What is data compression?

- Data compression is the process of representing a data source in a reduced form. If there is no loss of information, it is called lossless compression.

What is data compression?

- Data compression is the process of representing a data source in a reduced form. If there is no loss of information, it is called lossless compression.

The three principle components of a compression system: modelling, probability estimation, and coding.

Fixed Length Codes

The simplest encoding/decoding approach is to create a mapping from source alphabet to strings (codewords), where these codewords are fixed in length.

Fixed Length Codes

The simplest encoding/decoding approach is to create a mapping from source alphabet to strings (codewords), where these codewords are fixed in length.

Example (Fixed Length Coding)

Given $\mathcal{S} = \{a, c, g, t\}$, $\Sigma = \{0, 1\}$, and the encoding scheme,

$$\begin{aligned} a &\mapsto 00, \\ c &\mapsto 01, \\ g &\mapsto 10, \\ t &\mapsto 11, \end{aligned}$$

then $\phi(gattaca) = 10001111000100$.

Fixed Length Codes – ASCII

ASCII

American Standard Code for Information Interchange is a fixed length character encoding scheme over an alphabet of 128 characters.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

```
char* string = "AARDVARK"
```

A	A	R	D	V	A	R	K
0x41	0x41	0x52	0x44	0x56	0x41	0x52	0x4B
1000001	1000001	1010010	1000100	1010110	1000001	1010010	1001011

Variable Length Codes

- Fixed length codewords are not the optimal in average bits per source symbol.
- Why? The frequency of appearance of each member of the source alphabet may not be uniformly distributed.

Variable Length Codes

- Fixed length codewords are not the optimal in average bits per source symbol.
- Why? The frequency of appearance of each member of the source alphabet may not be uniformly distributed.
- Consider the letters `e` and `z` in natural language text, and using the same length codewords to represent both.
- e.g., "zee"
- Using ascii where each letter represented by 7 bits, this is 3 * 7 = 21 bits

Character Frequency

Character	Frequency	Probability
e	24,600,752	0.0880
t	18,443,242	0.0660
a	17,379,446	0.0621
⋮	⋮	⋮
j	671,765	0.0024
q	264,712	0.0009
z	186,802	0.0007

The frequency of appearance of characters from the English alphabet extracted from a 267 MB segment of SGML-tagged newspaper text drawn from the *WSJ* component of the TREC data set.

Variable Length Codes

Solution?

- A variable length code maps each member of a source alphabet to a codeword string, but the length of codewords is no longer fixed.
- E.g., use a shorter codeword for ’e’ and a larger one for ’z’.

Variable Length Codes

Solution?

- A variable length code maps each member of a source alphabet to a codeword string, but the length of codewords is no longer fixed.
- E.g., use a shorter codeword for ’e’ and a larger one for ’z’.
- “zee” (hypothetically use 2 bit code for ’e’ and ’10’ bit code for z, this is 10 + 2*2 = 14 bits)

Variable Length Codes

Solution?

- A variable length code maps each member of a source alphabet to a codeword string, but the length of codewords is no longer fixed.
- E.g., use a shorter codeword for 'e' and a larger one for 'z'.
- "zee" (hypothetically use 2 bit code for 'e' and '10' bit code for z, this is 10 + 2*2 = 14 bits)
- However, not all possible variable length coding schemes are decodeable.

Variable Length Codes – Decoding

Symbol	a	b	c	d	e	f	g
Frequency	25	12	9	4	3	2	1

Symbol	Codeword	ℓ_i
a	0	1
b	1	1
c	00	2
d	01	2
e	10	2
f	11	3
g	110	3

Decode: `001010001000011001011`

Variable Length Codes – Decoding

Symbol	a	b	c	d	e	f	g
Frequency	25	12	9	4	3	2	1

Symbol	Codeword	ℓ_i
a	`0`	1
b	`1`	1
c	`00`	2
d	`01`	2
e	`10`	2
f	`11`	3
g	`110`	3

Decode: `0010100010000111001011`

Variable length codes must be chosen so text is uniqly decodeable.

Variable Length Codes – Prefix codes

Prefix Codes: Variable length codewords where no codeword is a prefix of any other codeword. Prefix codes are uniquely decodeable.

Symbol	Codeword	ℓ_i
a	0	1
b	100	3
c	110	3
d	111	3
e	1010	4

Huffman's Code – Sketch

Huffman Algorithm generates prefix codes that are optimal in the average number of bits per symbol.

Huffman's Code – Sketch

Huffman Algorithm generates prefix codes that are optimal in the average number of bits per symbol.

Idea: Build prefix tree bottom up. Read the codes from this prefix tree.

Huffman's Code – Sketch

Huffman Algorithm generates prefix codes that are optimal in the average number of bits per symbol.

Idea: Build prefix tree bottom up. Read the codes from this prefix tree.

1. For each symbol, calculate the probability of appearance. Construct a leaf node for it.

Huffman's Code – Sketch

Huffman Algorithm generates prefix codes that are optimal in the average number of bits per symbol.

Idea: Build prefix tree bottom up. Read the codes from this prefix tree.

1. For each symbol, calculate the probability of appearance. Construct a leaf node for it.
2. Put these leaf nodes to the set of candidate nodes (to merge).

Huffman's Code – Sketch

Huffman Algorithm generates prefix codes that are optimal in the average number of bits per symbol.

Idea: Build prefix tree bottom up. Read the codes from this prefix tree.

1. For each symbol, calculate the probability of appearance. Construct a leaf node for it.
2. Put these leaf nodes to the set of candidate nodes (to merge).
3. Select the two nodes with the lowest probability (from candidate nodes) and combine them in a "bottom-up" tree construction.

Huffman's Code – Sketch

Huffman Algorithm generates prefix codes that are optimal in the average number of bits per symbol.

Idea: Build prefix tree bottom up. Read the codes from this prefix tree.

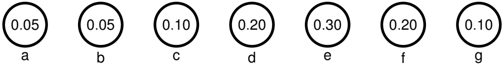
1. For each symbol, calculate the probability of appearance. Construct a leaf node for it.
2. Put these leaf nodes to the set of candidate nodes (to merge).
3. Select the two nodes with the lowest probability (from candidate nodes) and combine them in a "bottom-up" tree construction.
4. The new parent has a probability equal to the sum of the two child probabilities, and replaces the two children in the set of candidate nodes. Add links to the children, one link with labelled '0', the other '1'.

Huffman's Code – Sketch

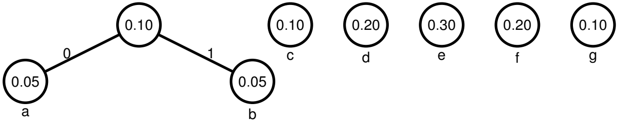
Huffman Algorithm generates prefix codes that are optimal in the average number of bits per symbol.

Idea: Build prefix tree bottom up. Read the codes from this prefix tree.

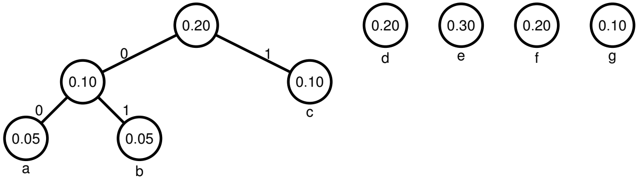
1. For each symbol, calculate the probability of appearance. Construct a leaf node for it.
2. Put these leaf nodes to the set of candidate nodes (to merge).
3. Select the two nodes with the lowest probability (from candidate nodes) and combine them in a “bottom-up” tree construction.
4. The new parent has a probability equal to the sum of the two child probabilities, and replaces the two children in the set of candidate nodes. Add links to the children, one link with labelled ’0’, the other ’1’.
5. When only one candidate node remains, a tree has been formed, and codewords can be read from the edge labels of a tree.



Construct a Huffman Tree from the Alphabet
$\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

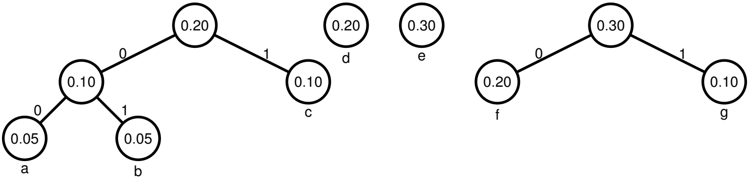


Construct a Huffman Tree from the Alphabet
$\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.



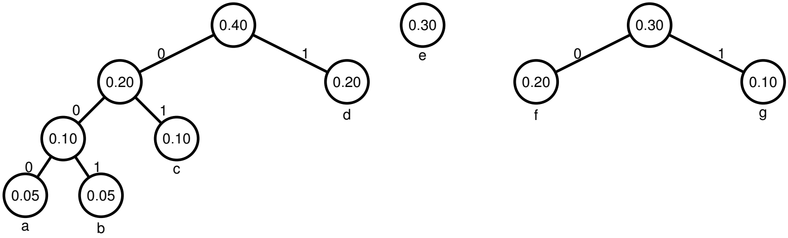
Construct a Huffman Tree from the Alphabet
$\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees – Example



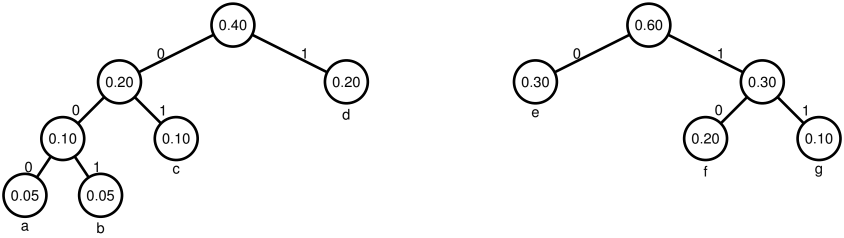
Construct a Huffman Tree from the Alphabet
$\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees – Example



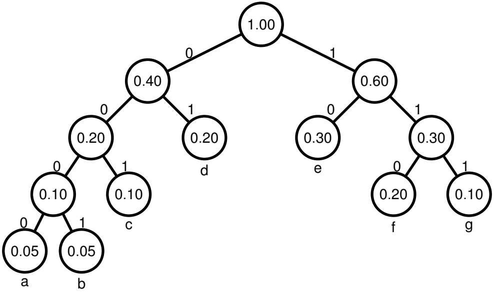
Construct a Huffman Tree from the Alphabet
$\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees – Example



Construct a Huffman Tree from the Alphabet
$\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees – Example



Construct a Huffman Tree from the Alphabet
$\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Codes

Symbol	Codeword	ℓ_i
a	`0000`	4
b	`0001`	4
c	`001`	3
d	`01`	2
e	`10`	2
f	`110`	3
g	`111`	3

The Huffman Codes and the corresponding codeword lengths.

Huffman Codes

- This approach requires $\Theta(n \log n)$ time if a min heap (priority queue) is used to manage the set of candidates and their weights.
- If the input list is already sorted by their probabilities, then the codes can be constructed in $\Theta(n)$ time.

Summary

- Understand and be able to apply the greedy approach to solving problems.
- Examples:
  - spanning tree – Prim's algorithm
  - spanning tree – Kruskal's algorithm
  - single source shortest-path – Dijkstra's algoirthm
  - data compression