Graphs

CMPT 225

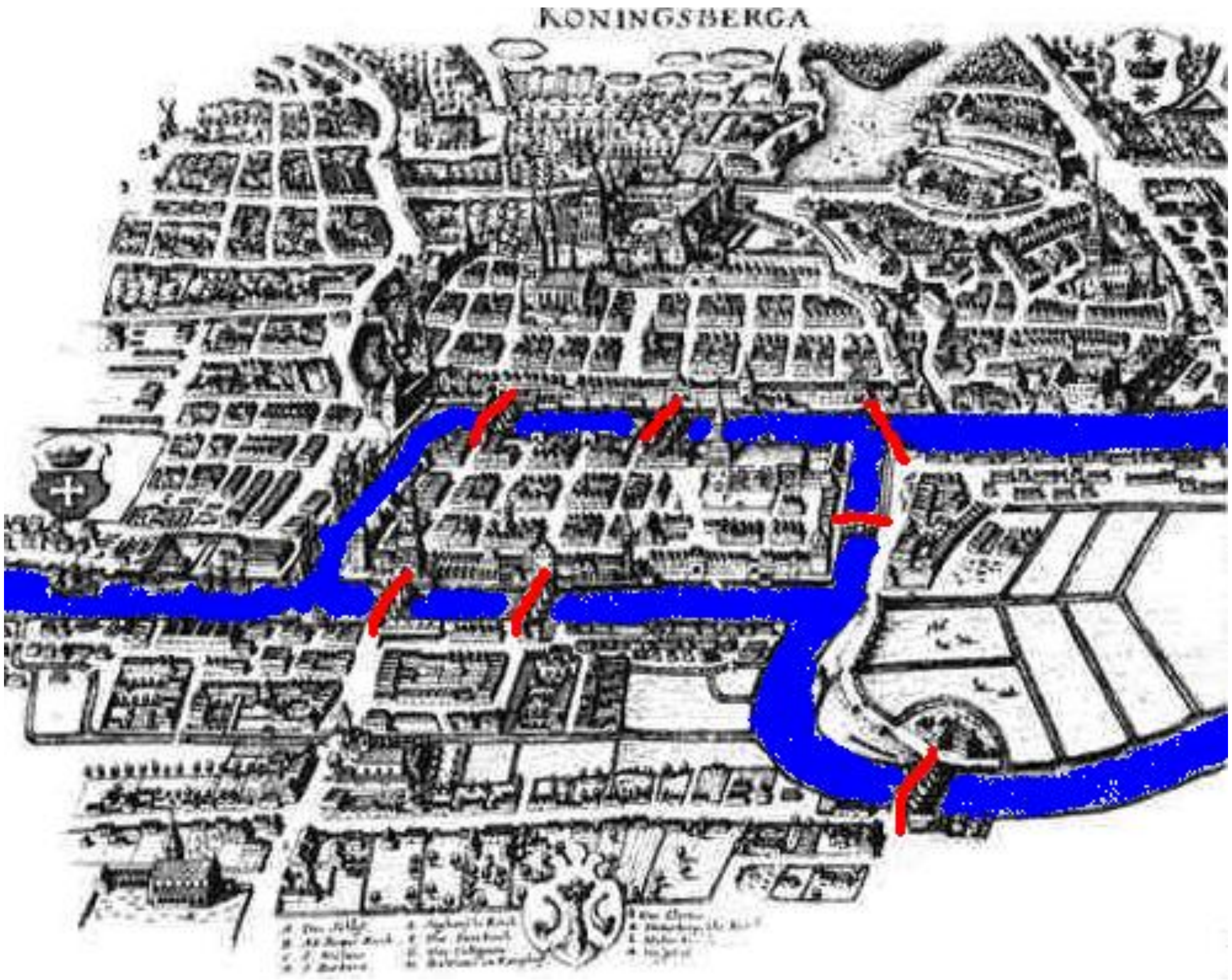
Objectives

- Understand graph terminology
- Implement graphs using
  - Adjacency lists and
  - Adjacency matrices
- Perform graph searches
  - Depth first search
  - Breadth first search
- Perform shortest-path algorithms
  - Disjkstra's algorithm
  - A* algorithm

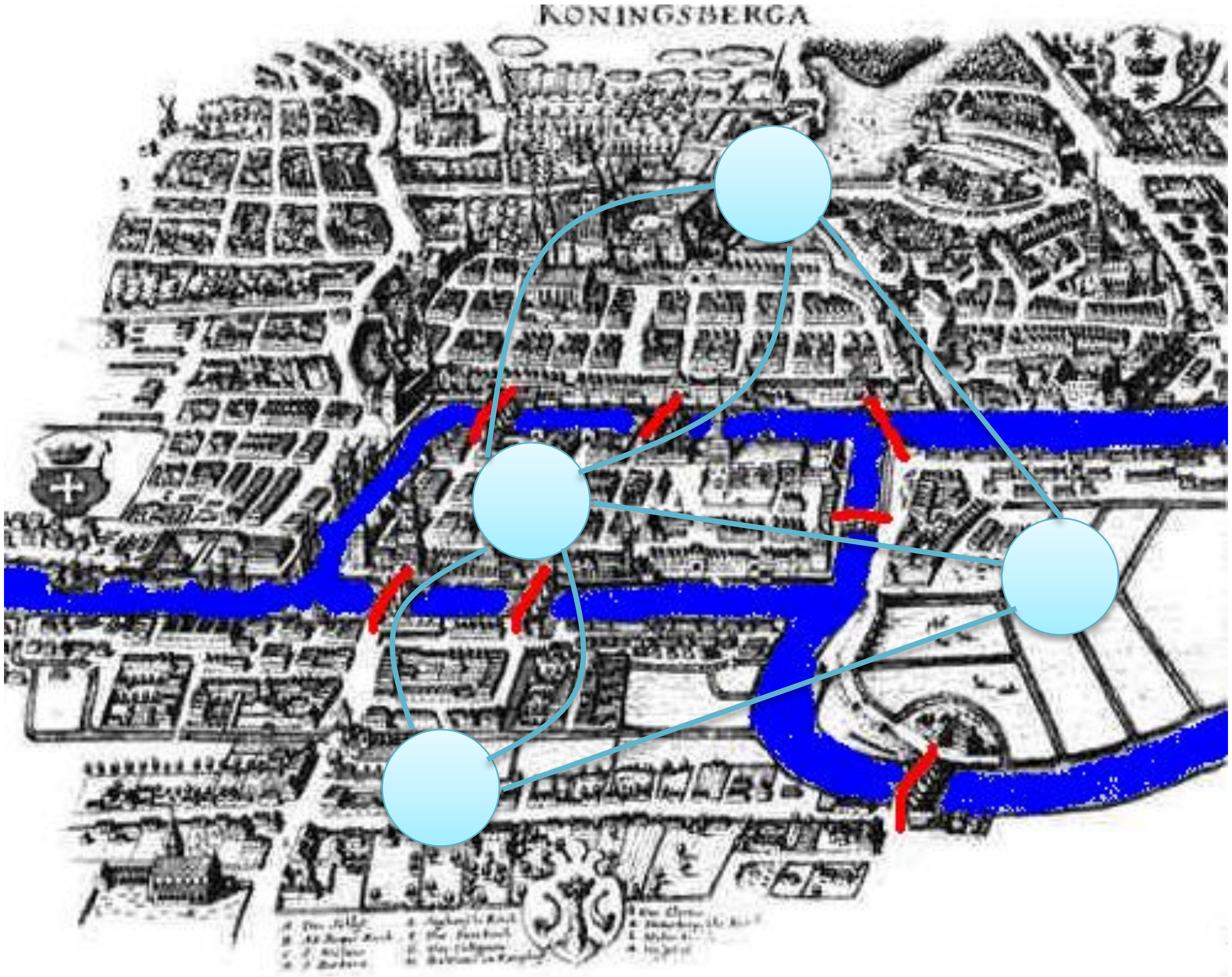
Graph Theory and Euler

- Graph theory is often considered to have been born with Leonhard Euler
  - In 1736 he solved the *Konigsberg bridge problem*
- Konigsberg was a city in Eastern Prussia
  - Renamed Kalinigrad when East Prussia was divided between Poland and Russia in 1945
  - Konigsberg had seven bridges in its centre
    - The inhabitants of Konigsberg liked to see if it was possible to walk across each bridge just once
    - And then return to where they started
  - Euler proved that it was impossible to do this, as part of this proof he represented the problem as a graph

Konigsberg Graph

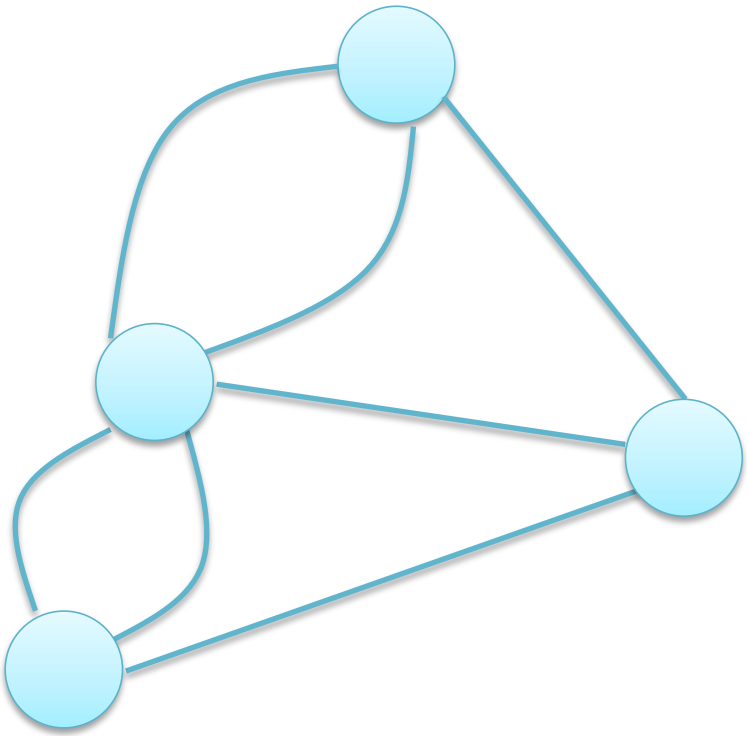


Konigsberg



Multigraphs

- The Konigsberg graph is an example of a *multigraph*
- A multigraph has multiple edges between the same pair of vertices
- In this case the edges represent bridges



Graph Uses

- Graphs are used as representations of many different types of problems
  - Network configuration
  - Airline flight booking
  - Pathfinding algorithms
  - Database dependencies
  - Task scheduling
  - Critical path analysis
  - …

Graph Terminology

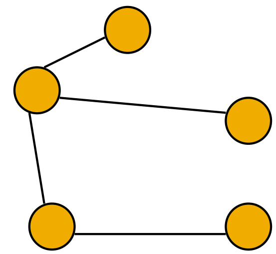
- A graph consists of two sets
  - A set V of *vertices* (or nodes) and
  - A set E of *edges* that connect vertices
  - $|V|$ is the size of V, $|E|$ the size of E
- Two vertices may be connected by a *path*
  - A sequence of edges that begins at one vertex and ends at the other
    - A *simple path* does not pass through the same vertex more than once
    - A *cycle* is a path that starts and ends at the same vertex

Numbers of Vertices and Edges

- If a graph has v vertices, how many edges does it have?
  - If every vertex is connected to every other vertex, and we count each direction as two edges
    - $v^2 - v$
  - If the graph is a tree
    - $v - 1$
  - Minimum number of edges
    - 0

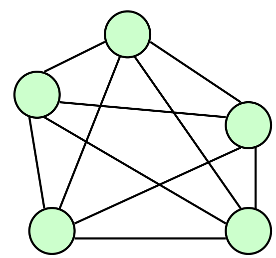
Connected and Unconnected Graphs

- A *connected* graph is one where every pair of distinct vertices has a *path* between them
- A *complete* graph is one where every pair of vertices has an *edge* between them
- A graph cannot have multiple edges between the same pair of vertices
- A graph cannot have *self edges*, an edge from and to the same vertex

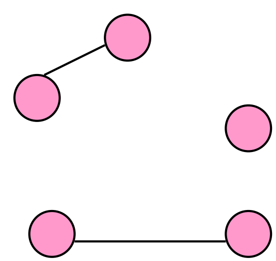


connected graph

and a tree



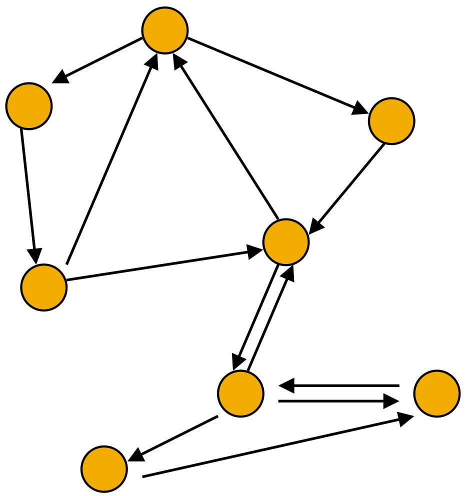
complete graph



unconnected graph

Directed Graphs

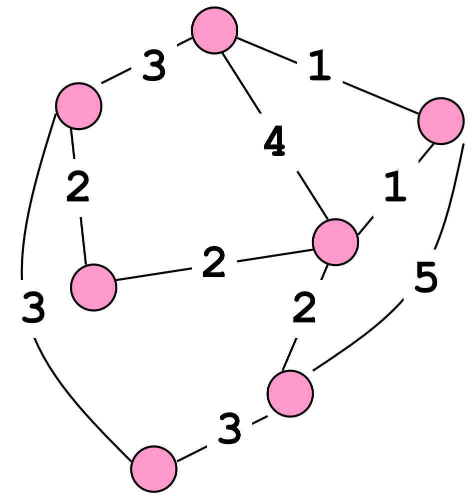
- In a *directed graph* (or digraph) each edge has a direction and is called a directed edge
- A directed edge can only be traveled in one direction
- A pair of vertices in a digraph may have two edges between them, one in each direction



directed graph

Weighted Graphs

- In a *weighted graph* each edge is assigned a weight
  - Edges are labeled with their weights
- Each edge's weight represents the cost to travel along that edge
  - The cost could be distance, time, money or some other measure
  - The cost depends on the underlying problem



weighted graph

Basic Graph Operations

- Create an empty graph
- Test to see if a graph is empty
- Determine the number of vertices in a graph
- Determine the number of edges in a graph
- Determine if an edge exists between two vertices
  - and in a weighted graph determine its weight
- Insert a vertex
  - each vertex is assumed to have a distinct search key
- Delete a vertex, and its associated edges
- Delete an edge
- Return a vertex with a given key

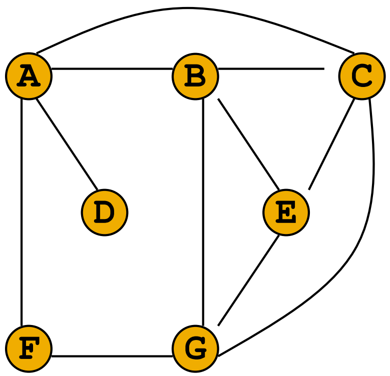
Graph Implementation

- There are two common implementations of graphs
  - Both implementations require a list of all vertices in the set of vertices, *V*
  - The implementations differ in how edges are recorded
- Adjacency matrices
  - Provide fast lookup of individual edges
  - But waste space for sparse graphs
- Adjacency lists
  - Are more space efficient for sparse graphs
  - Can efficiently find all the neighbours of a vertex

Adjacency Matrix

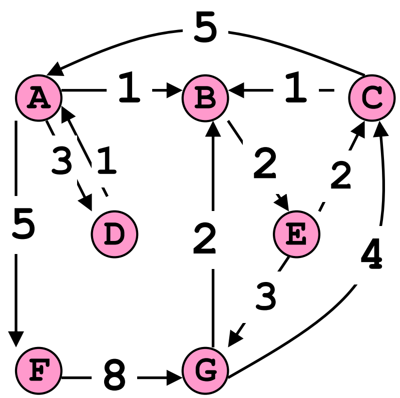
- The edges are recorded in an $|V| * |V|$ matrix
- In an unweighted graph entries in the matrix are
  - 1 when there is an edge between vertices or
  - 0 when there is no edge between vertices
- In a weighted graph entries are either
  - The edge weight if there is an edge between vertices
  - Infinity when there is no edge between vertices
- Adjacency matrix performance
  - Looking up an edge requires O(1) time
  - Finding all neighbours of a vertex requires $O(|\mathbf{V}|)$ time
  - The matrix requires $|V|^2$ space

Adjacency Matrix Examples



line of symmetry

	A	B	C	D	E	F	G
A	0	1	1	1	0	1	0
B	1	0	1	0	1	0	1
C	1	1	0	0	1	0	1
D	1	0	0	0	0	0	0
E	0	1	1	0	0	0	1
F	1	0	0	0	0	0	1
G	0	1	1	0	1	1	0

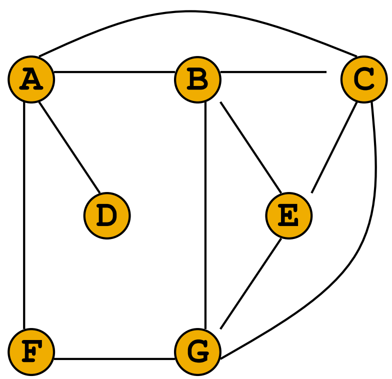


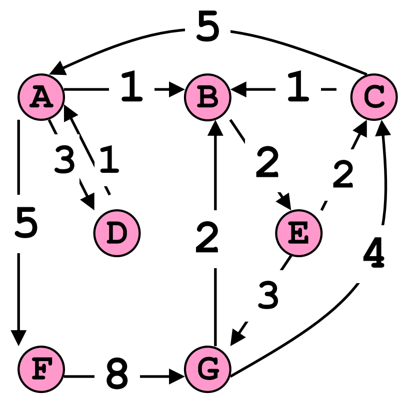
	A	B	C	D	E	F	G
A	∞	1	∞	3	∞	5	∞
B	∞	∞	∞	∞	2	∞	∞
C	5	1	∞	∞	∞	∞	∞
D	1	∞	∞	∞	∞	∞	∞
E	∞	∞	2	∞	∞	∞	3
F	∞	∞	∞	∞	∞	∞	8
G	∞	2	4	∞	∞	∞	∞

Adjacency Lists

- The edges are recorded in an array $|V|$ of linked lists
- In an unweighted graph a list at index *i* records the keys of the vertices adjacent to vertex *i*
- In a weighted graph a list at index *i* contains pairs
  - Which record vertex keys (of vertices adjacent to *i*)
  - And their associated edge weights
- Adjacency List Performance
  - Looking up an edge requires time proportional to the average number of edges
  - Finding all vertices adjacent to a given vertex also takes time proportional to the average number of edges
  - The list requires $O(|E|)$ space

Adjacency List Examples





A	B →	C →	D →	F
B	A →	C →	E →	G
C	A →	B →	E →	G
D	A			
E	B →	C →	G	
F	A →	G		
G	B →	C →	E →	F

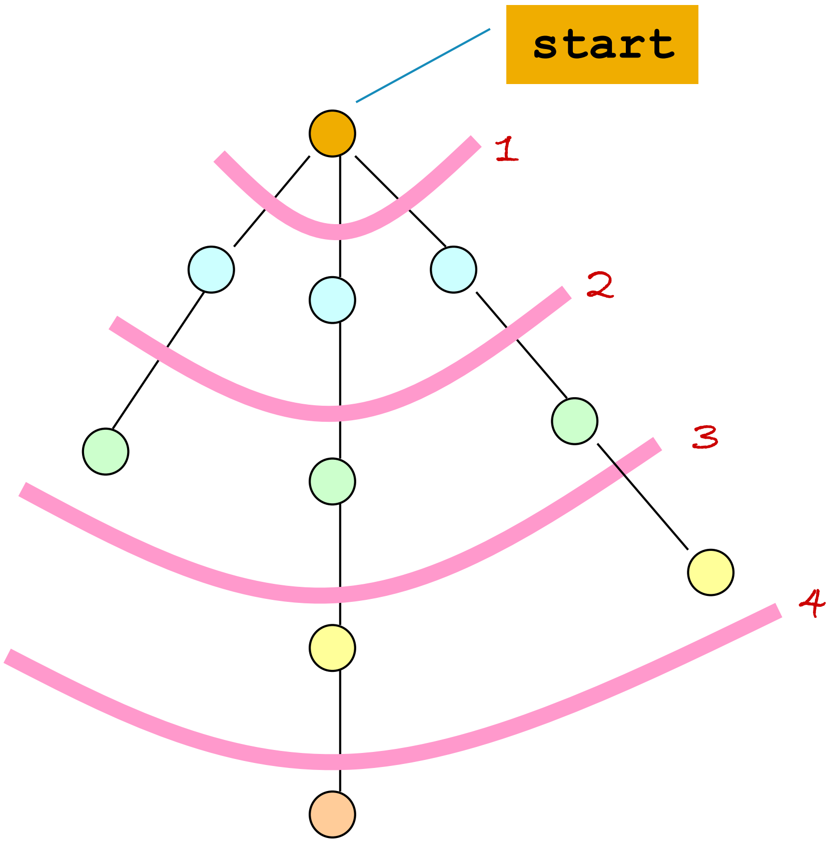
A	B	1 →	D	3 →	F	5
B	E	2				
C	A	5 →	B	1		
D	A	1				
E	C	2 →	G	3		
F	G	8				
G	B	2 →	C	4		

Graph Traversals

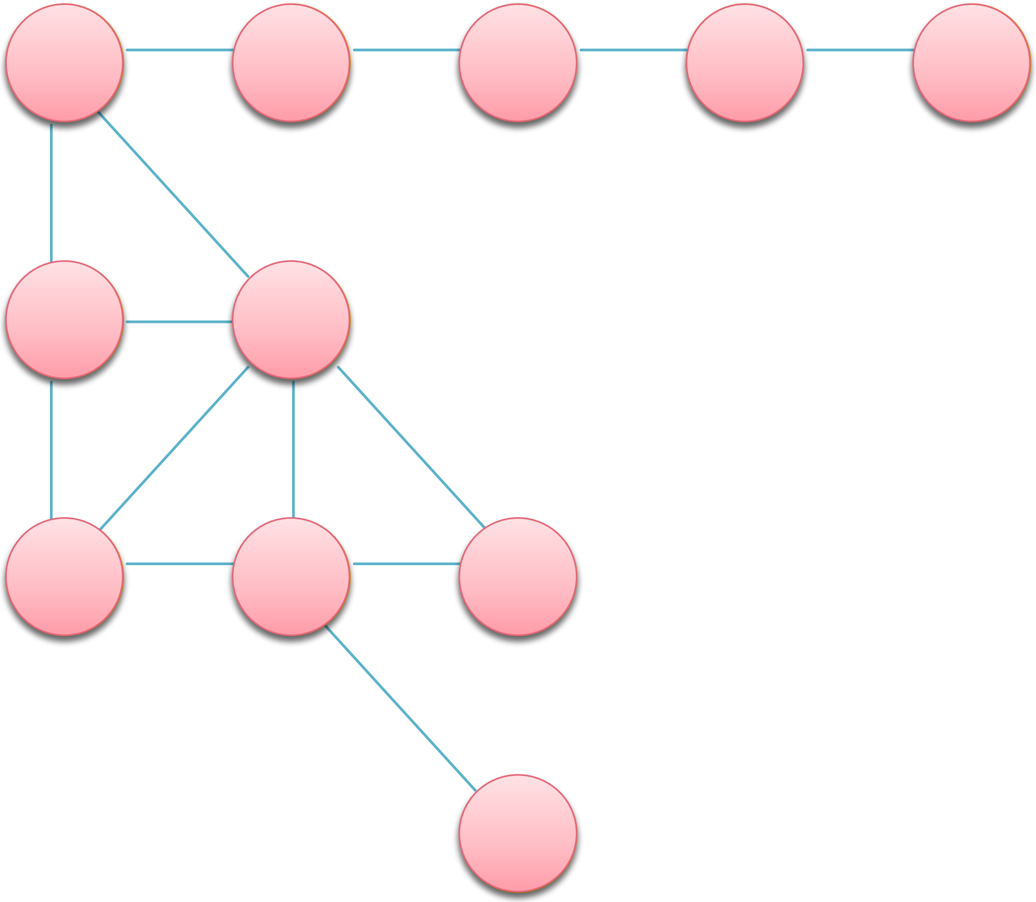
- A graph traversal algorithm visits all of the vertices that can be reached
  - If the graph is not connected some of the vertices will not be visited
  - Therefore a graph traversal algorithm can be used to see if a graph is connected
- Vertices should be marked as *visited*
  - Otherwise, a traversal will go into an infinite loop if the graph contains a cycle

Breadth First Search

- After visiting a vertex, *v*, visit every vertex adjacent to *v* before moving on
- Use a queue to store nodes
  - Queues are FIFO
- BFS:
  - visit and insert start
  - while (*q* not empty)
  - remove node from *q* and make it *current*
  - visit and insert the unvisited nodes adjacent to *current*



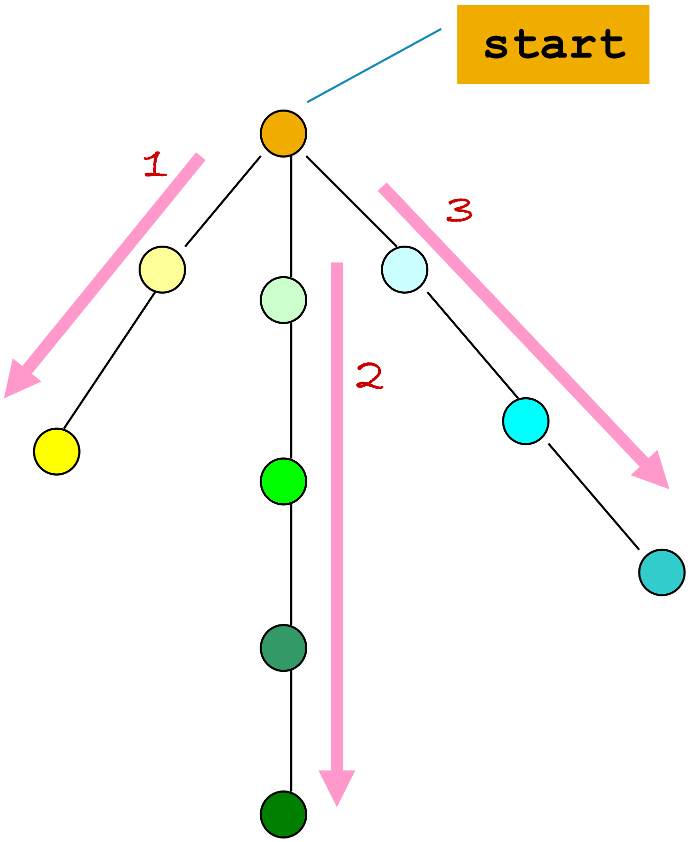
Breadth First Search Example



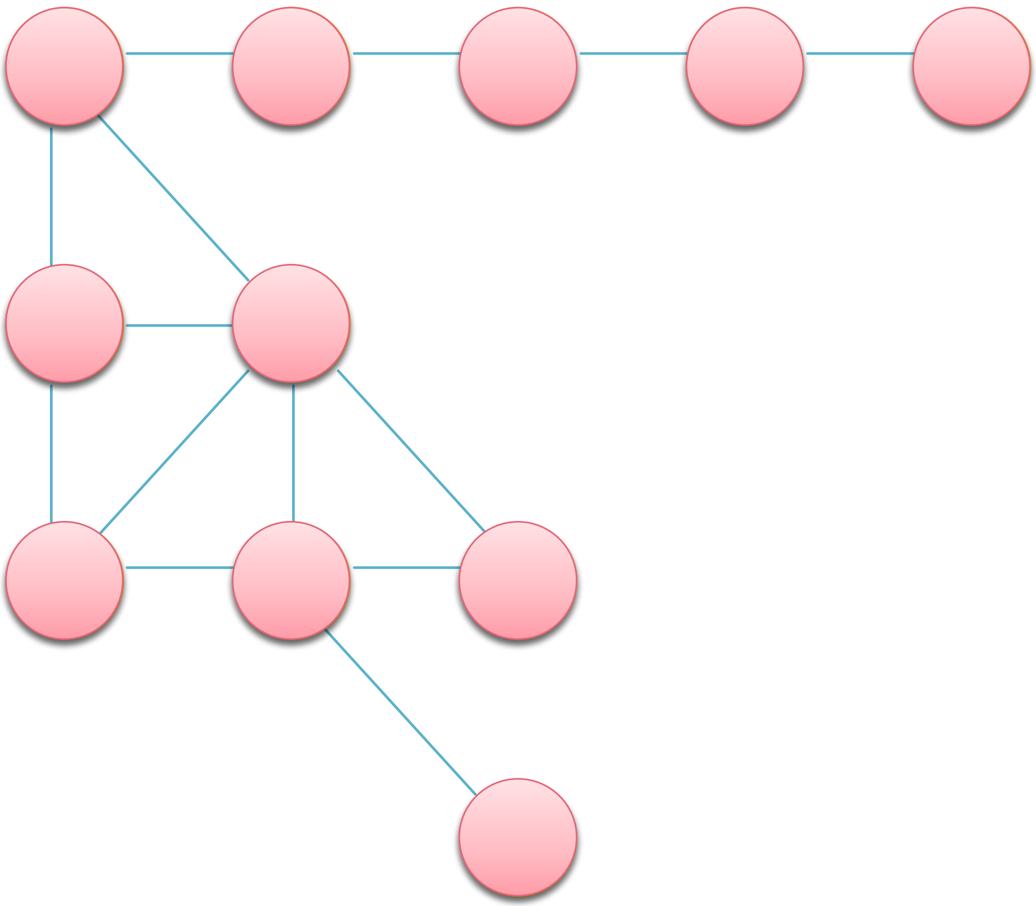
queue	visited
A	A
B	B
F	F
G	G
C	C
H	H
I	I
J	J
D	D
K	K
E	E

Depth First Search

- Visit a vertex, *v*, move from *v* as deeply as possible
- Use a stack to store nodes
  - Stacks are LIFO
- DFS:
  - visit and push start
  - while (*s* not empty)
  - peek at node, *nd*, at top of *s*
  - if *nd* has an unvisited neighbour visit it and push it onto *s*
  - else pop *nd* from *s*



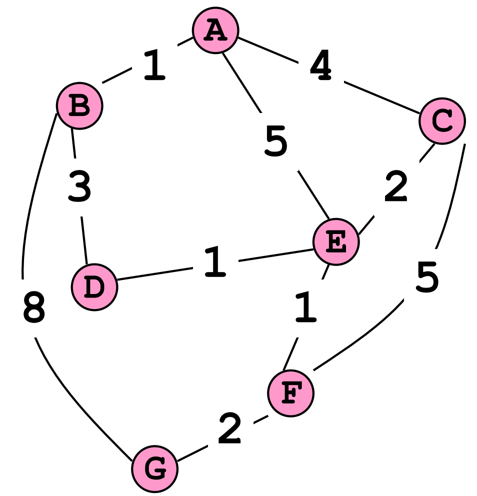
Depth First Search Example



stack		visited
	J K	A
E	I	B
D	H	C
C	G	D
B	F	E
A		F
		G
		H
		I
		J
		K

Shortest Path Problem

- What is the least cost path from one vertex to another?
  - Referred to as the shortest path between vertices
  - For weighted graphs this is the path that has the smallest sum of its edge weights
- Dijkstra's algorithm finds the shortest path between one vertex and all other vertices
  - The algorithm is named after its discoverer, Edgser Dijkstra



The shortest path between B and G is:

B–D–E–F–G and <u>not</u>

B–G (or B–A–E–F–G)

Dijkstra's Algorithm

- Finds the shortest path to all nodes from the start node
- Performs a modified BFS that accounts for edge weights
  - Selects the node with the least cost from the start node
  - In an unweighted graph this reduces to a BFS
- Stores nodes in a *priority queue*
  - In a priority queue the node with the least cost is removed first
  - The queue records the total cost to reach each node from the start node
- The cost in the priority queue is updated when necessary
- The shortest path to any node can be found by *backtracking* from that node's entry in a results list

Initialization

- A record for each vertex is inserted into a priority queue, each record contains
  - It's search key
  - The cost to reach the vertex from the start vertex
  - The search key of the previous vertex in the path
- These values are initially set as follows
  - The cost to reach the start vertex is set to zero
  - The cost to reach all other vertices is set to infinity and the parent vertex is set to the start vertex
  - Because the cost to reach the start vertex is zero it will be at the head of the priority queue

Implementation

- Priority queues can be implemented with a heap
  - It is efficient for removing the highest priority item
    - In this case the element with the least cost
- Using a heap does have one drawback
  - Its elements will need to be accessed to update their costs
  - It is therefore useful to provide an index to its contents
- There are other data structures that can be used instead of a heap

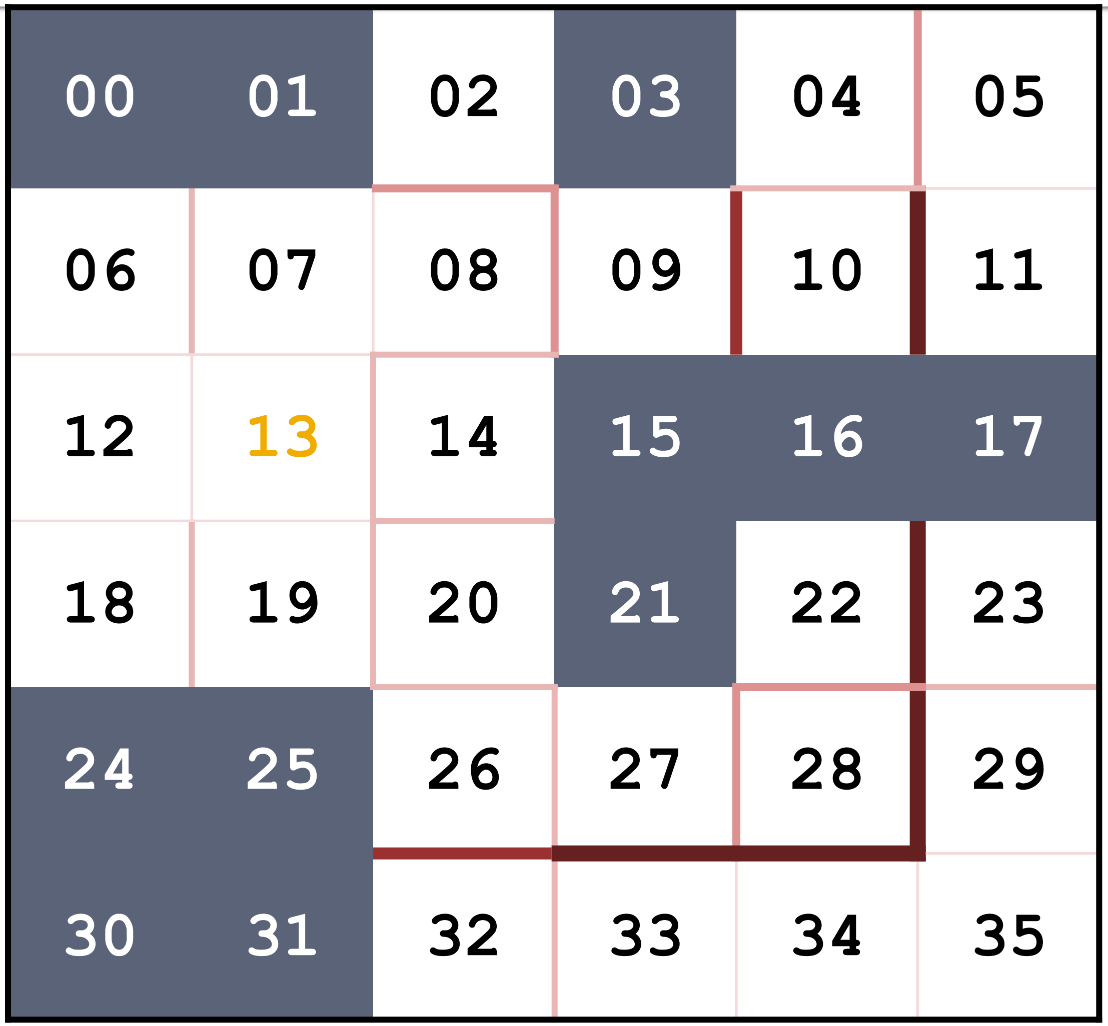
Main Loop

- Until the priority queue is empty
  - Remove the vertex with the least cost and insert it in a *results* list, making it the current vertex
    - The results list should be indexed by the search key of the vertices
  - Search the adjacency list (or matrix) for vertices adjacent to the current vertex
  - For each such vertex, *v*
    - Compare the cost to reach *v* in the priority queue with the cost to reach *v* via the current vertex
    - If the cost via the current vertex is less then change *v*'s entry in the priority queue to reflect this new path

Final Stage

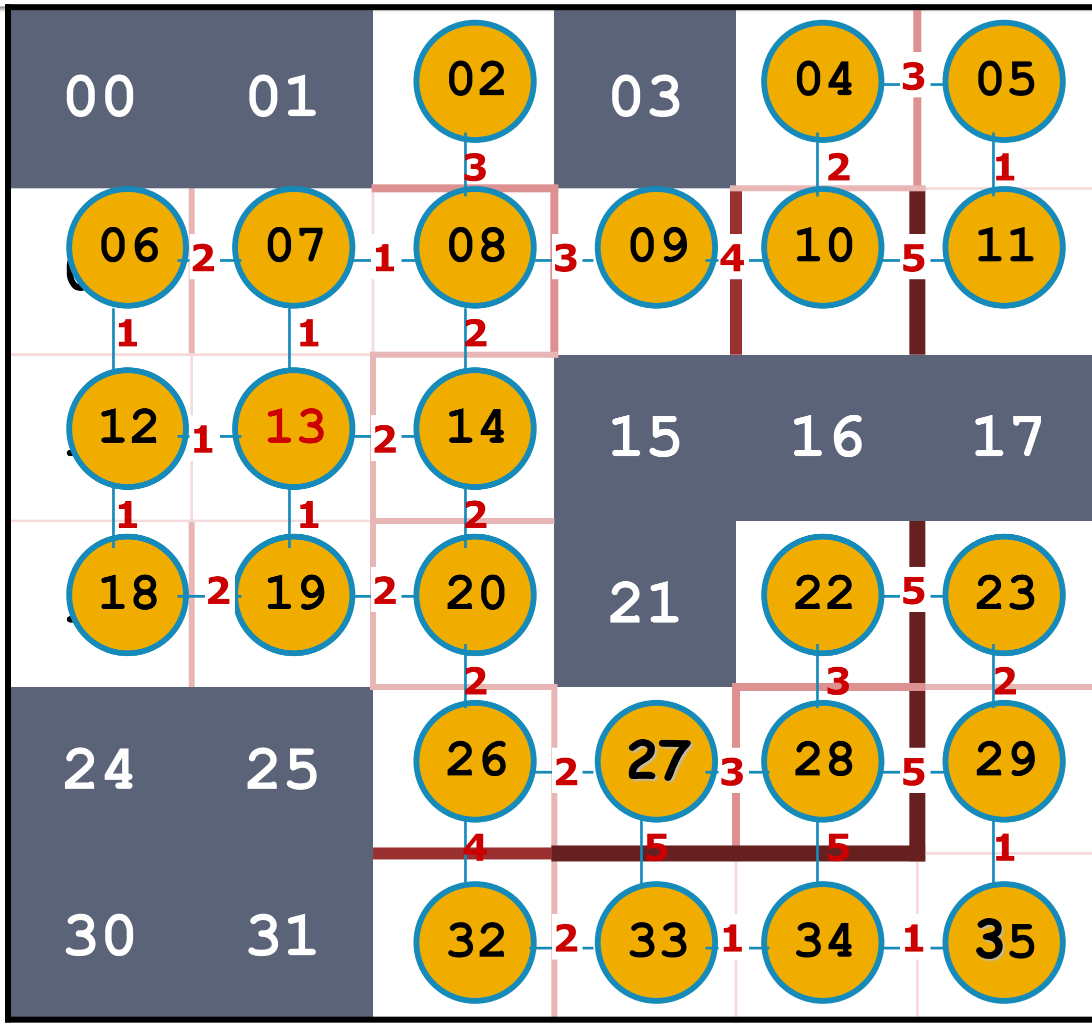
- When the priority queue is empty the results list contains all of the shortest paths from the start
- To find a path to a vertex look up the goal vertex in the results list
  - The vertex's parent vertex represents the previous vertex in the path
  - A complete path can be found by backtracking through all the parent vertices to the start vertex
  - A vertex's cost in the results list represents the total cost of the shortest path from the start to that vertex

Find the Shortest Path …



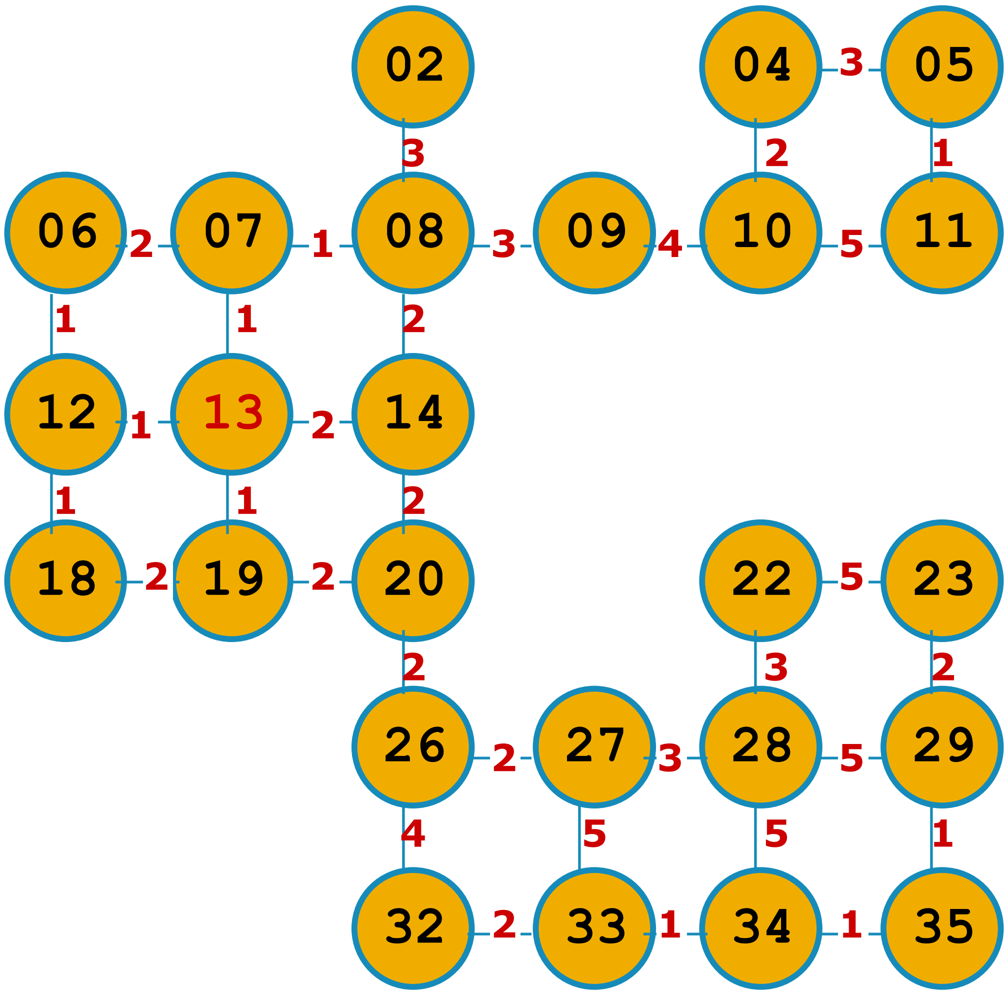
- Shaded squares are inaccessible
- Start at square 13
- Moves can only be made vertically or horizontally and only one square at a time
- The cost to reach an adjacent square is indicated by the width of the walls between squares (from 1 to 5)

Graph Representation



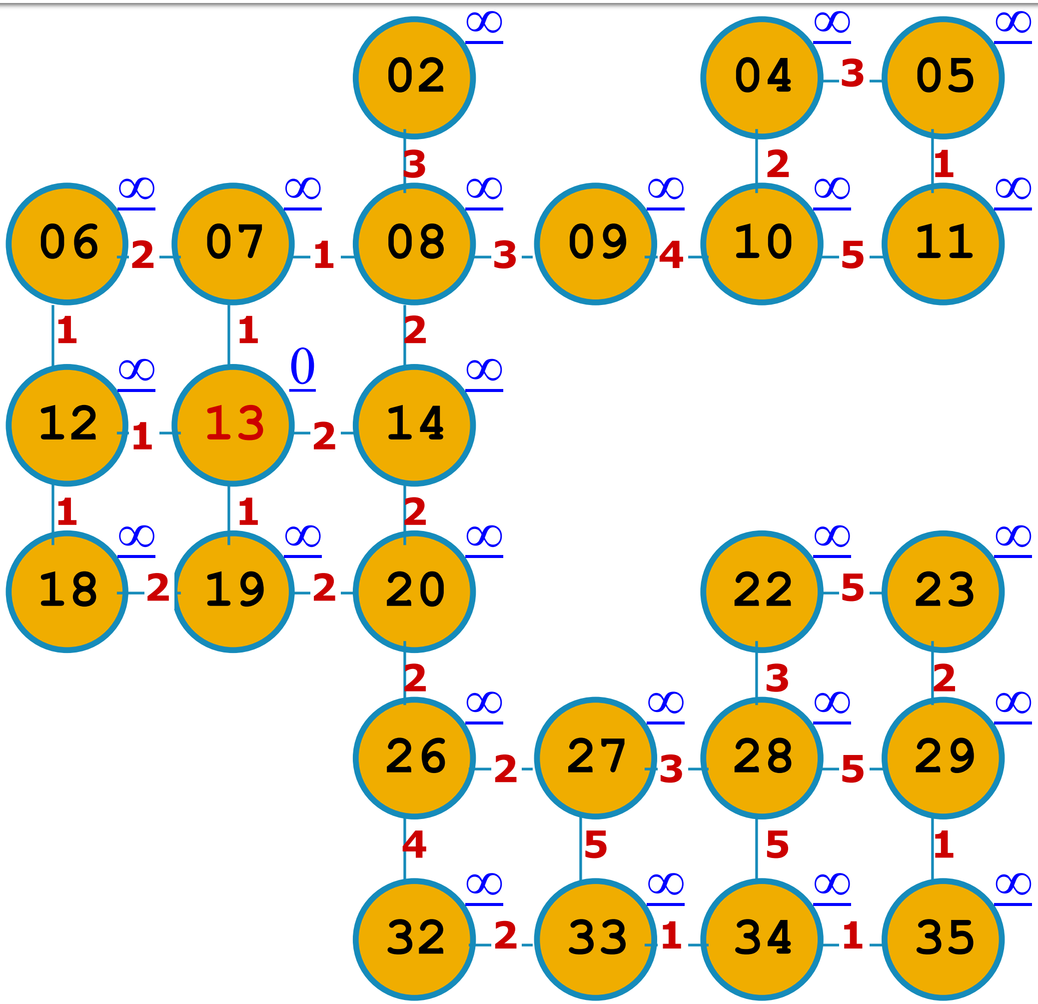
- Only vertices that can be reached are to be represented
- Graph is undirected
- The cost to move from one square to another differs, the graph is weighted
- The graph is fairly sparse, suggesting that the edges should be stored in an adjacency list

Graph Representation



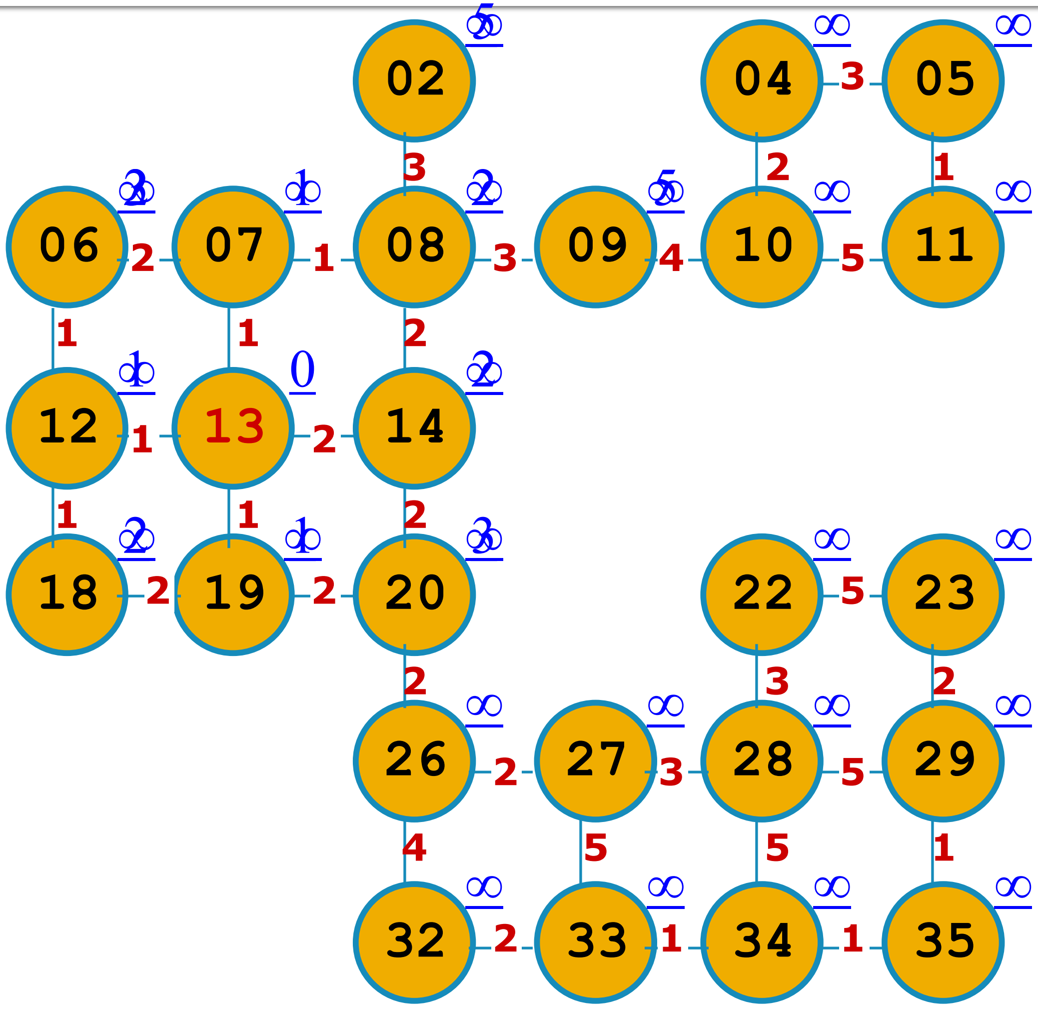
- Only vertices that can be reached are to be represented
- Graph is undirected
- As the cost to move from one square to another differs, the graph is weighted
- The graph is fairly sparse, suggesting that the edges should be stored in an adjacency list

Dijkstra's Algorithm Start



- The cost to reach each vertex from the start (*st*) is set to infinity
  - For vertex *v* let's call this cost *c*[*st*][*v*]
- All nodes are entered in a priority queue, in cost priority
- The cost to reach the start node is set to 0, and the priority queue is updated
- The results list is shown in the sidebar

Dijkstra's Algorithm Demonstration



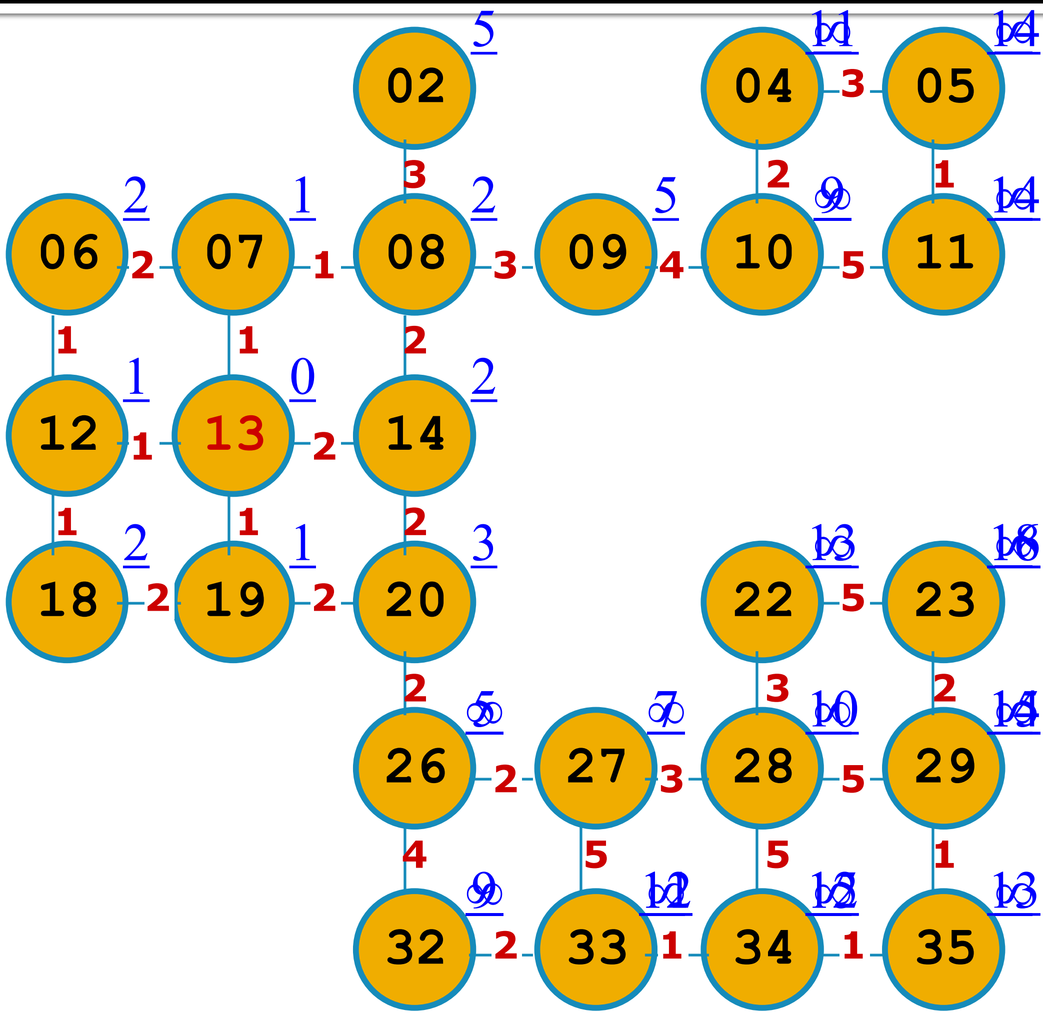
vertex,	cost,	parent
13,	0,	13
07,	1,	13
12,	1,	13
19,	1,	13
14,	2,	13
06,	2,	12
08,	2,	07
18,	2,	12

remove root from prQ

update cost to adjacent vertex, *v*, via removed vertex, *u*, if:

$c[u][v] + c[st][u] < c[st][v]$

Dijkstra's Algorithm Demonstration



vertex, cost, parent

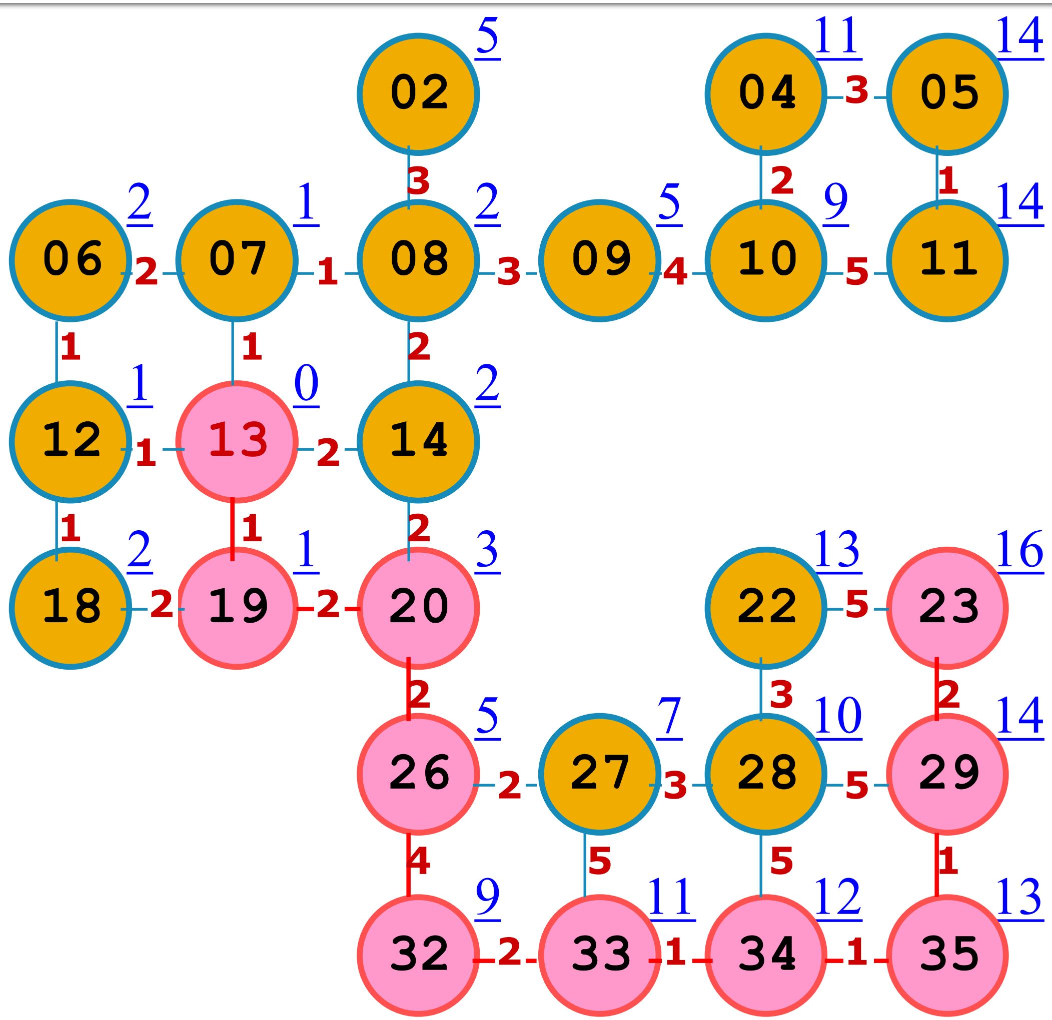
vertex	cost	parent	vertex	cost	parent
13,	0,	13	10,	9,	09
07,	1,	13	32,	9,	26
12,	1,	13	28,	10,	27
19,	1,	13	04,	11,	10
14,	2,	13	33,	11,	32
06,	2,	12	34,	12,	33
08,	2,	07	22,	13,	28
18,	2,	12	35,	13,	34
20,	3,	19	05,	14,	04
02,	5,	08	11,	14,	10
09,	5,	08	29,	14,	35
26,	5,	20	23,	16,	29
27,	7,	26			

Retrieving the Shortest Path

vertex,	cost,	parent	vertex,	cost,	parent
13,	0,	13	10,	9,	09
07,	1,	13	32,	9,	26
12,	1,	13	28,	10,	27
19,	1,	13	04,	11,	10
14,	2,	13	33,	11,	32
06,	2,	12	34,	12,	33
08,	2,	07	22,	13,	28
18,	2,	12	35,	13,	34
20,	3,	19	05,	14,	04
02,	5,	08	11,	14,	10
09,	5,	08	29,	14,	35
26,	5,	20	23,	16,	29
27,	7,	26			

- Once the results array is complete paths from the start vertex can be retrieved
- Done by looking up the end vertex (the vertex to which one is trying to find a path) and backtracking through parent vertices to the start
- For example to find a path to vertex 23 backtrack through:
  - 29, 35, 34, 33, 32, 26, 20, 19, 13
  - Note: there should be some efficient way to search the results array for a vertex

Shortest Path from 13 to 23



vertex	cost	parent	vertex	cost	parent
13,	0,	13	10,	9,	09
07,	1,	13	32,	9,	26
12,	1,	13	28,	10,	27
19,	1,	13	04,	11,	10
14,	2,	13	33,	11,	32
06,	2,	12	34,	12,	33
08,	2,	07	22,	13,	28
18,	2,	12	35,	13,	34
20,	3,	19	05,	14,	04
02,	5,	08	11,	14,	10
09,	5,	08	29,	14,	35
26,	5,	20	23,	16,	29
27,	7,	26			

Dijkstra's Algorithm Operations

- The cost of the algorithm depends on $|E|$ and $|V|$ and the data structure used to implement the priority queue
- Consider how many operations are performed
- Whenever a vertex is removed we have to find each adjacent edge to it
  - There are $|V|$ vertices to be removed and
- For each of $|E|$ edges there it is necessary to
  - Retrieve the edge weight from the matrix or list
  - Look up the cost currently recorded in the priority queue for the edge's destination vertex

Dijkstra's Algorithm Analysis

- Assume a heap is used to implement the priority queue
- Building the heap takes $O(|V|)$ time
- Removing each vertex takes $O(\log|V|)$ time
  - For a total of $O(|V|*\log|V|)$
- Each of $|E|$ edges has to be processed once
  - Looking up (and changing) the current cost of a vertex in a heap takes $O(|V|)$ for an unindexed heap ($O(1)$ if the heap is indexed)
    - The heap property needs to be preserved after a change for an additional cost of $O(\log|V|)$
  - The total cost is $|V| + |V|*\log|V| + |E|*(|V| + \log|V|)$
    - Or, $O(|V|*\log|V| + |E|*|V|)$
  - If the heap is indexed the cost is $O((|V| + |E|) *\log|V|)$

Pathfinding with A*

- There are two drawbacks with Dijkstra's algorithm as a method of pathfinding
  - It finds paths from the start vertex to all other vertices, which results in wasted effort if only one path is required
  - It only measures the cost *so far*, it does not look ahead to judge whether or not a path is likely to be a good one
- The A* algorithm addresses both these issues
  - It returns the path from the start vertex to the target vertex and
  - Uses an estimate of the remaining cost to reach the target to direct its search

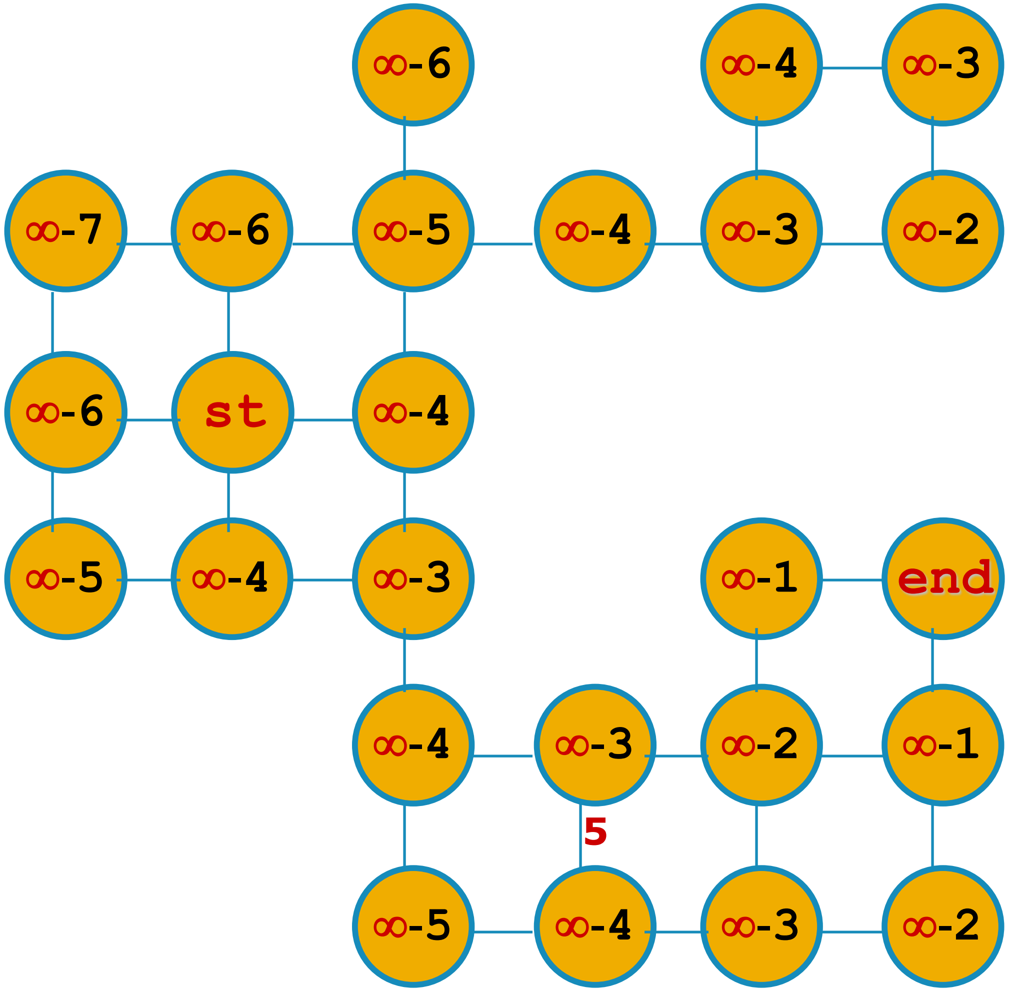
A* Algorithm

- The A* algorithm is similar to Dijkstra's algorithm
  - It performs a modified breadth first search and
  - Uses a priority queue to select vertices
- The A* algorithm uses a different cost metric, f, which is made up of two components
  - g – the cost to reach the current vertex from the start vertex (the same as Dijkstra's algorithm)
  - h – an estimate of the cost to reach the goal vertex from the current vertex
  - $f = g + h$

A* Heuristic – h

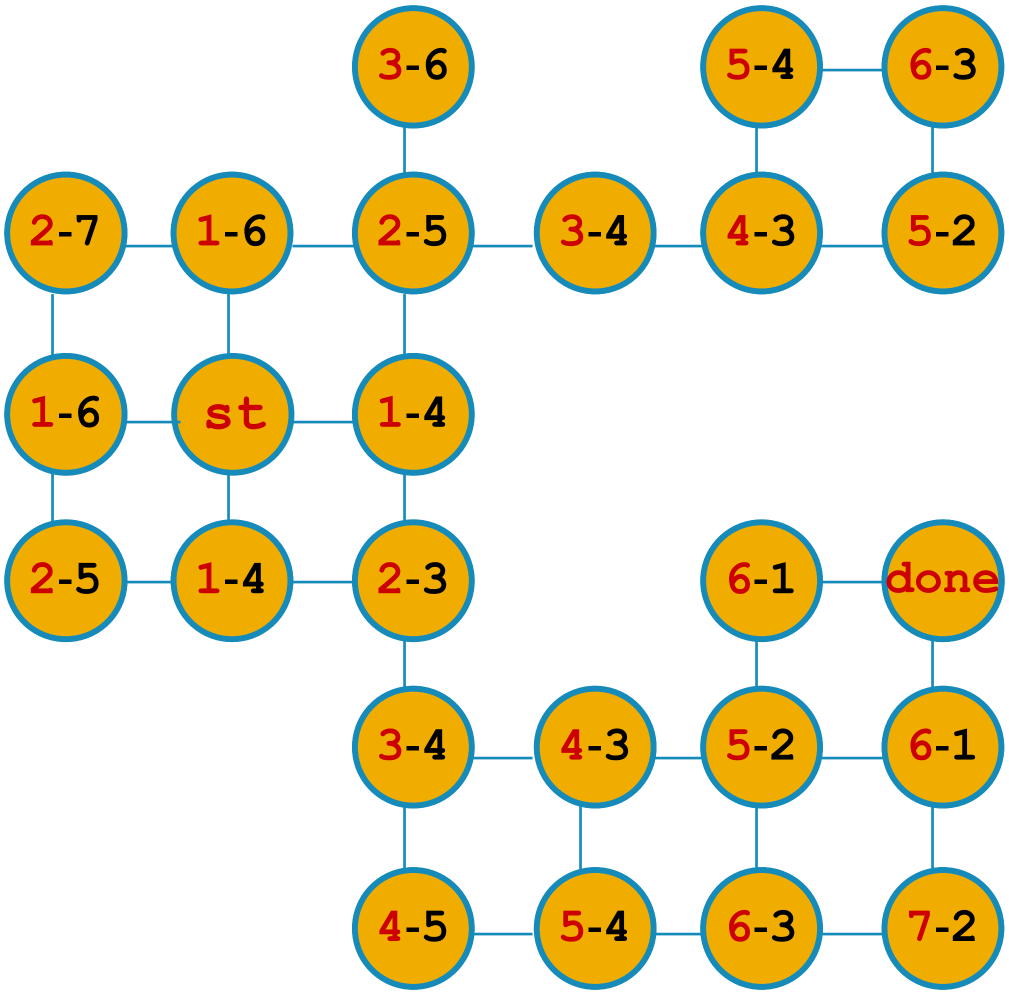
- The key to the efficiency of the A* algorithm is the accuracy of *h*
- To find an optimal path *h* should be *admissible*
  - The heuristic should not overestimate the cost of the path to the goal
  - Inadmissible heuristics may result in non-optimal paths
    - But may be faster than an inaccurate admissible heuristic
    - For a "good enough" solution it may be useful to use an inadmissible heuristic to speed up pathfinding
- If the heuristic is perfect the A* algorithm will find an optimal path with no backtracking

A* Search Example



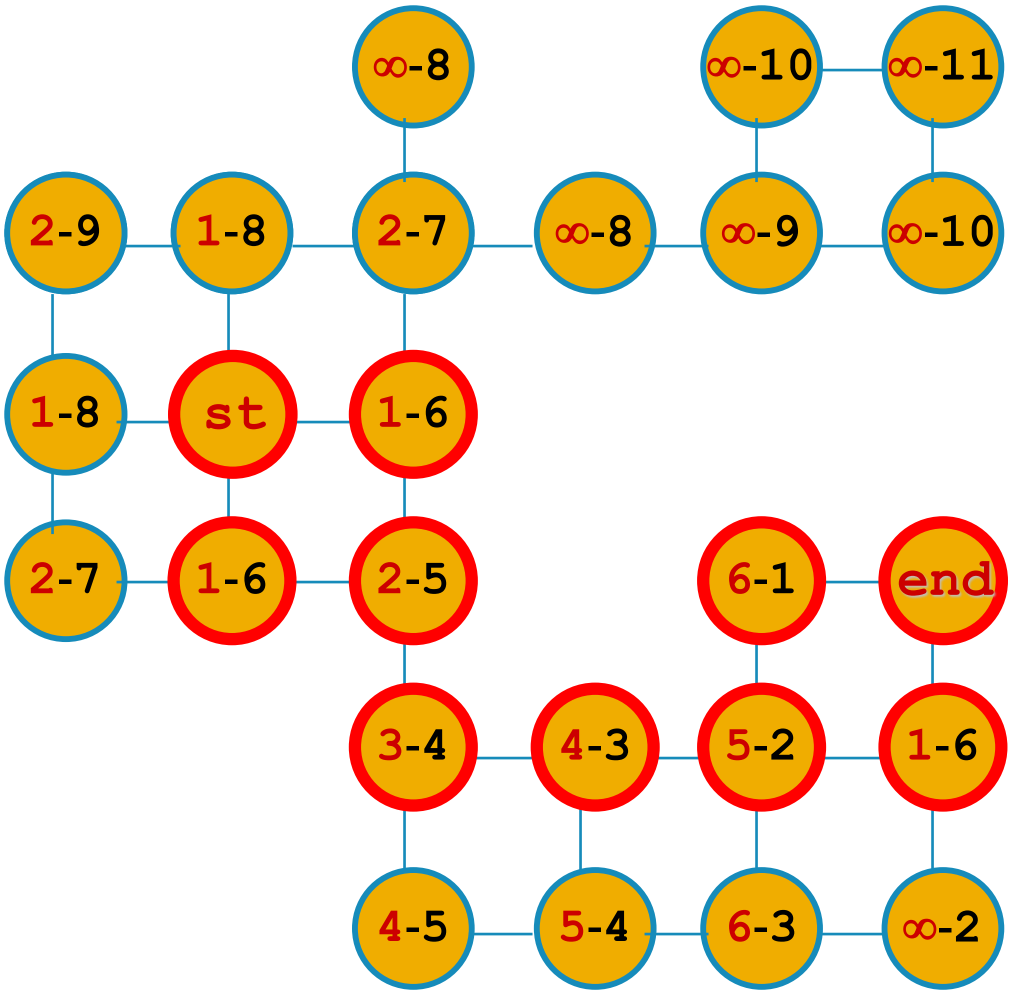
- Edges are unweighted
- The vertices' numbers represent the A* search *h* and *g* values
- *g* (**red**) is the cost to reach the vertex from the start vertex
- *h* (**black**) is the estimated cost to reach the goal from the current vertex
- *h* has been calculated as the straight line cost to reach the goal

A* Search Example



- fading a vertex means it is taken from the prQ
- remove the root (start) from prQ and update the cost to reach adjacent vertices
- remove the new root from prQ – which is ordered by ***f*** (i.e. $h + g$)
- repeat until the goal vertex is reached
- find the path by backtracking through the result away

A* Search – Perfect Heuristic



- in this example the heuristic is perfect
- the final *g* costs at the end of the algorithm are shown
- the vertices that are removed from the prQ during the algorithm are highlighted in red
- note that the vertices correspond to an optimal path, "extra" vertices correspond to choices between paths