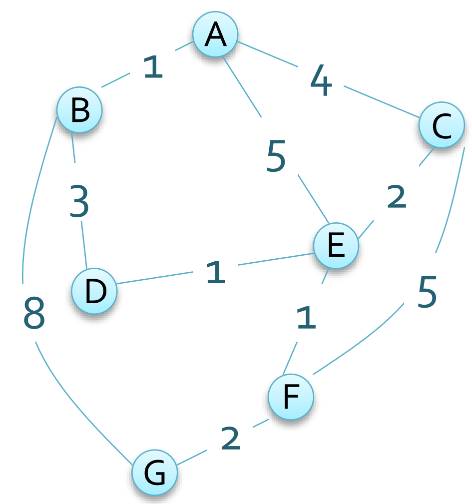
Graphs 2

Shortest Path

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



Shortest path between B and G ?

B–D–E–F–G

not
B–G or B–A–E–F–G

Dijkstra's Algorithm Overview

- Finds the shortest path to all vertices from the start vertex
- 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

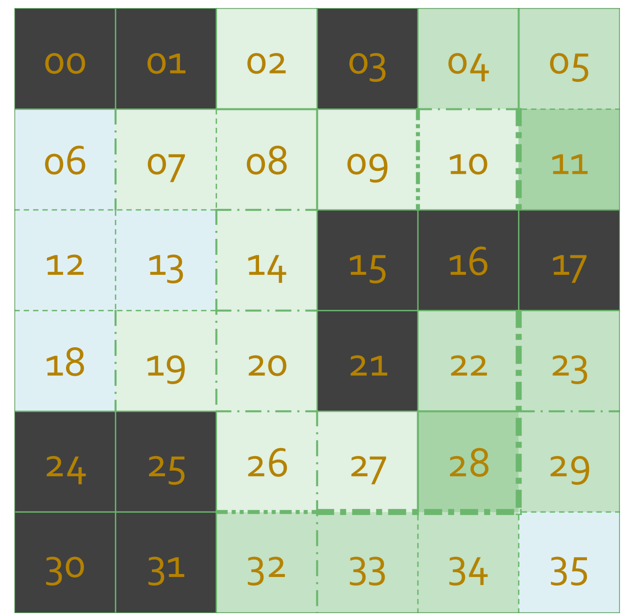
BFS uses a queue

- In a weighted graph we need to assess the edge weights
  - And want choose the path with the least total cost
- Use a priority queue with an entry for each vertex
  - These entries record the cost of the path from the start to the vertex
  - Priority is given to lowest cost paths
  - Costs in the priority queue must be updated if a lower cost path is found
    - Efficiency issue if the priority queue is represented by a heap

Example – Introduction

- Assume the grid represents a map
- Gray shaded areas are inaccessible
  - Mountains?
  - Pits full of demons?

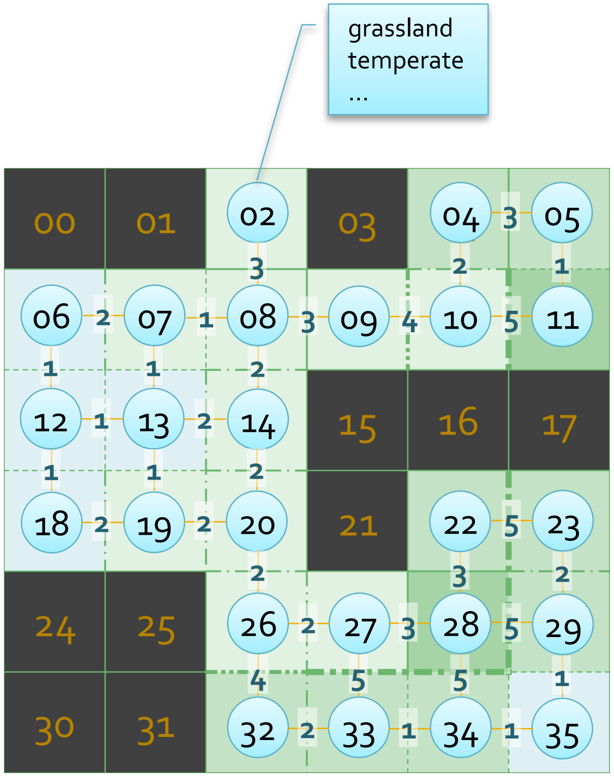
- The cost to move from one location to the next varies between 1 and 5
  - Indicated by the thickness of the border between squares
- Presumably there is additional data about each area on the map
  - Occupants?
  - Terrain type?
  - Main export?



John Edgar

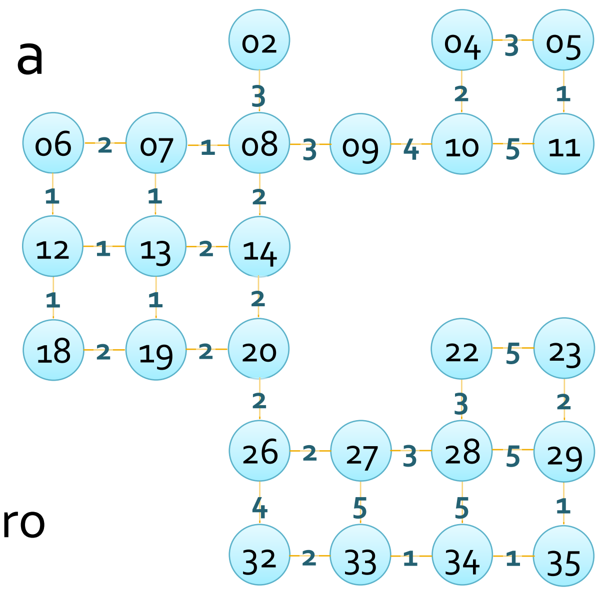
Example – Make a Graph

- Replace the map with a graph
  - Weighted and undirected
- Only need nodes for accessible regions
- The nodes contain only the information that is required for the application
- The edges record the edge weight
  - This graph is relatively sparse
  - So might choose to use an adjacency list
- We can now use the graph to answer questions about the application domain
- Such as – what is the shortest path between vertex 13 and all the other vertices?



Dijkstra’s Algorithm – Initialization

- A record for each vertex is inserted into a priority queue, each record contains
  - The vertex key (label)
  - The cost to reach the vertex from the start
  - The 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
  - Because the cost to reach the start vertex is zero it will be at the front of the priority queue



key	cost	parent
13	0	13
02	∞	13
04	∞	13
05	∞	13
…	…	…

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
  - That allow more efficient look-up than O(*n*)

Dijkstra's Algorithm – Main Loop

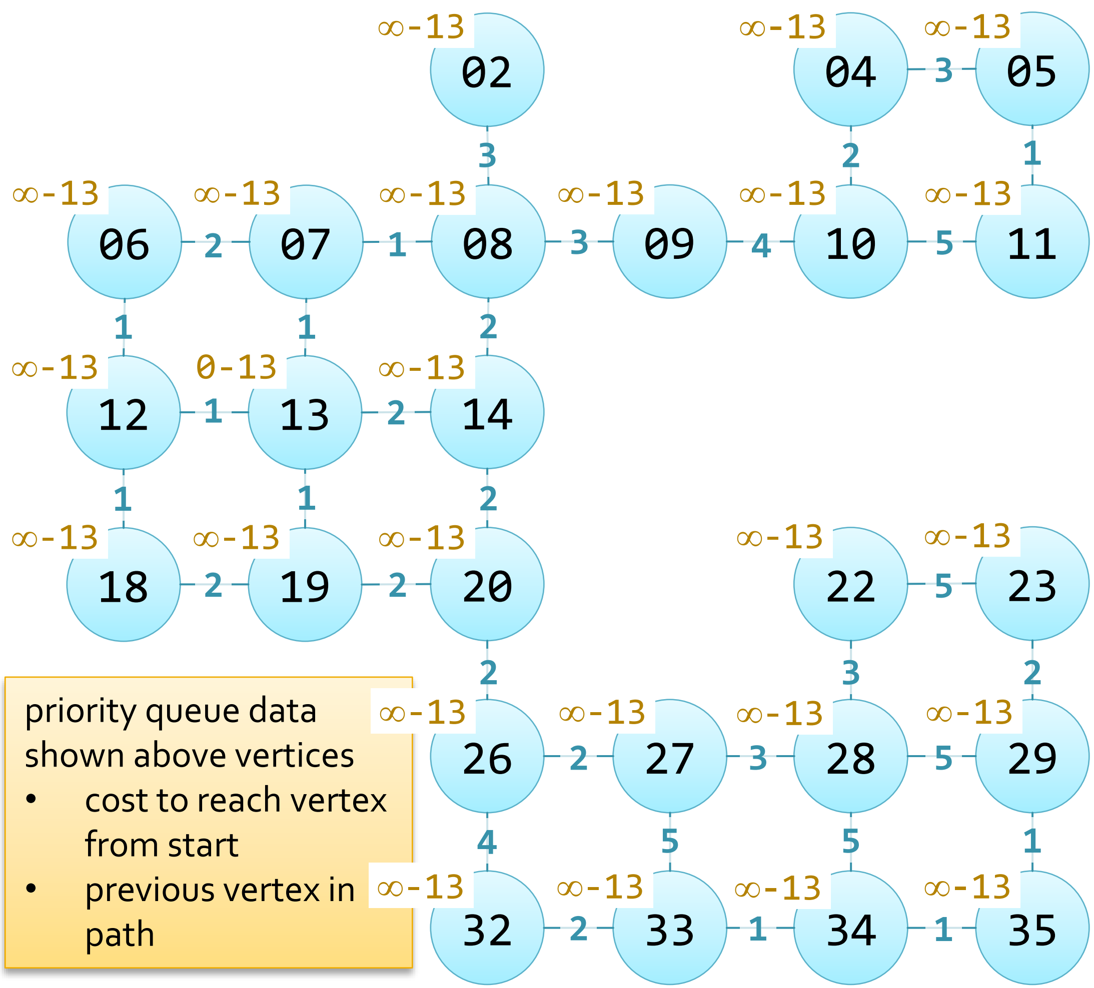
- Until the priority queue is empty
  - Remove the vertex with the least cost and insert it in a *results* list, making it the current vertex
    - The results list should be indexed by the search key of the vertices
  - Search in the adjacency list or matrix for vertices adjacent to the current vertex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

The results list contains {key, cost, parent} data and will need to be searched by key

Final Stage – Finding a Path

- 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 it up in the results list
  - The vertex's parent 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
  - Since the vertices need to be looked up by their key a hash table is a reasonable structure for the results list

Dijkstra's Algorithm Example 1

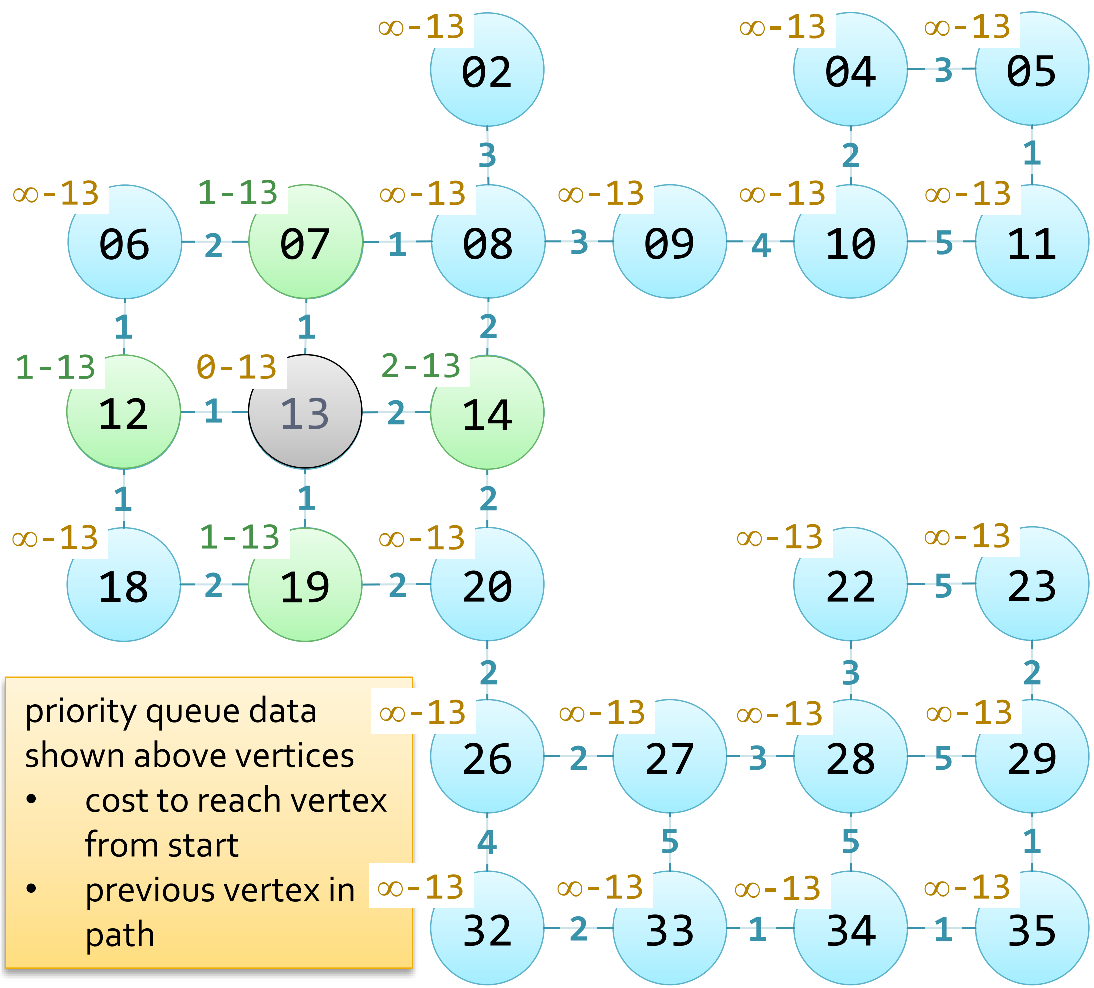


priority queue data shown above vertices

- cost to reach vertex from start
- previous vertex in path

- Initialization
- Create priority queue
  - Entries for each vertex
    - label
    - cost to reach from start
    - previous vertex
- Create empty results list
  - Entries for vertices as priority queue
- Initialize costs in priority queue
  - 0 for start vertex
  - infinity for all others

Dijkstra's Algorithm Example 2



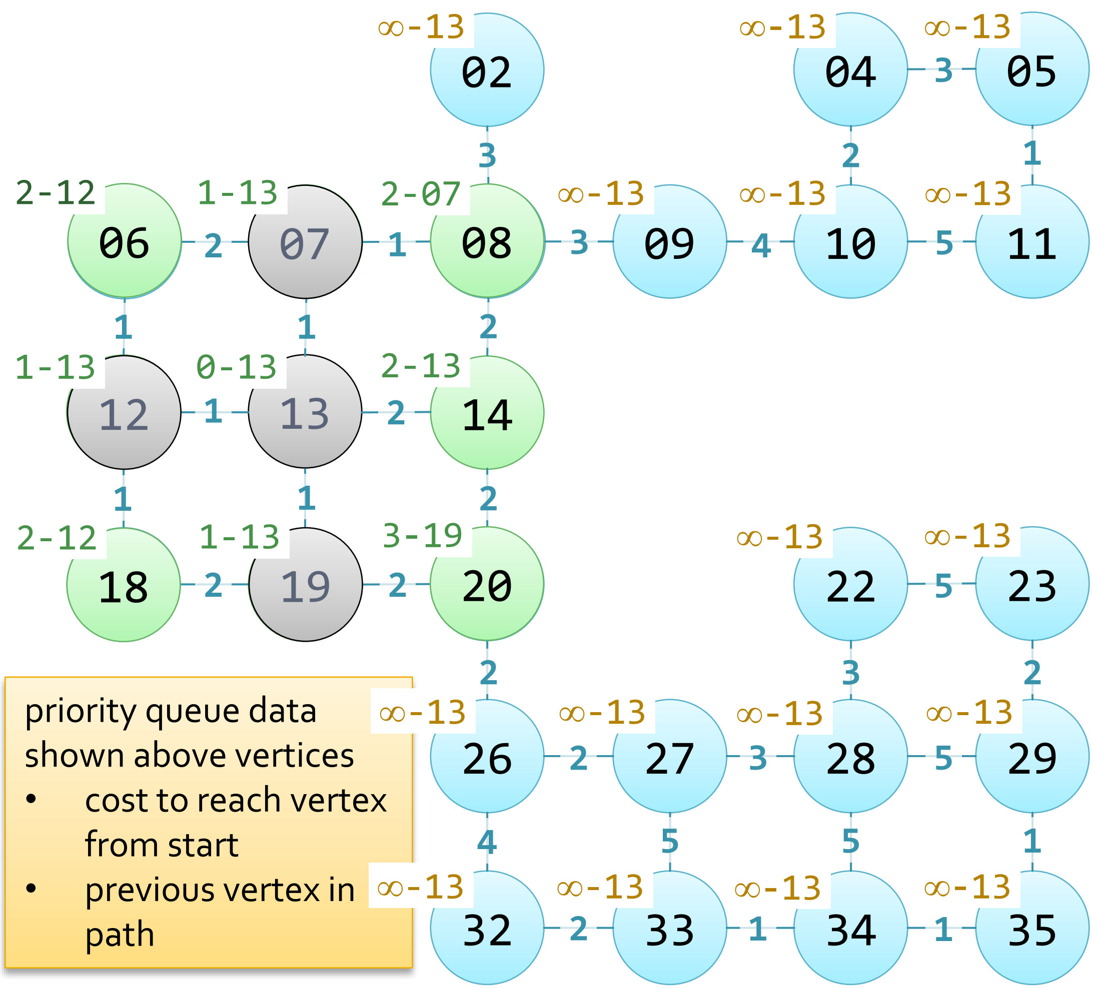
priority queue data shown above vertices

- cost to reach vertex from start
- previous vertex in path

- Remove vertex, *v*, from priority queue
  - Insert in results list
- Compare cost, *c*, to reach each adjacent vertex, *u*
  - If $c_u > (c_v + \text{edge}[v, u])$
  - Update cost and set *u*'s parent to *v*

label	cost	parent	label	cost	parent
13	0	-			

Dijkstra's Algorithm Example 3



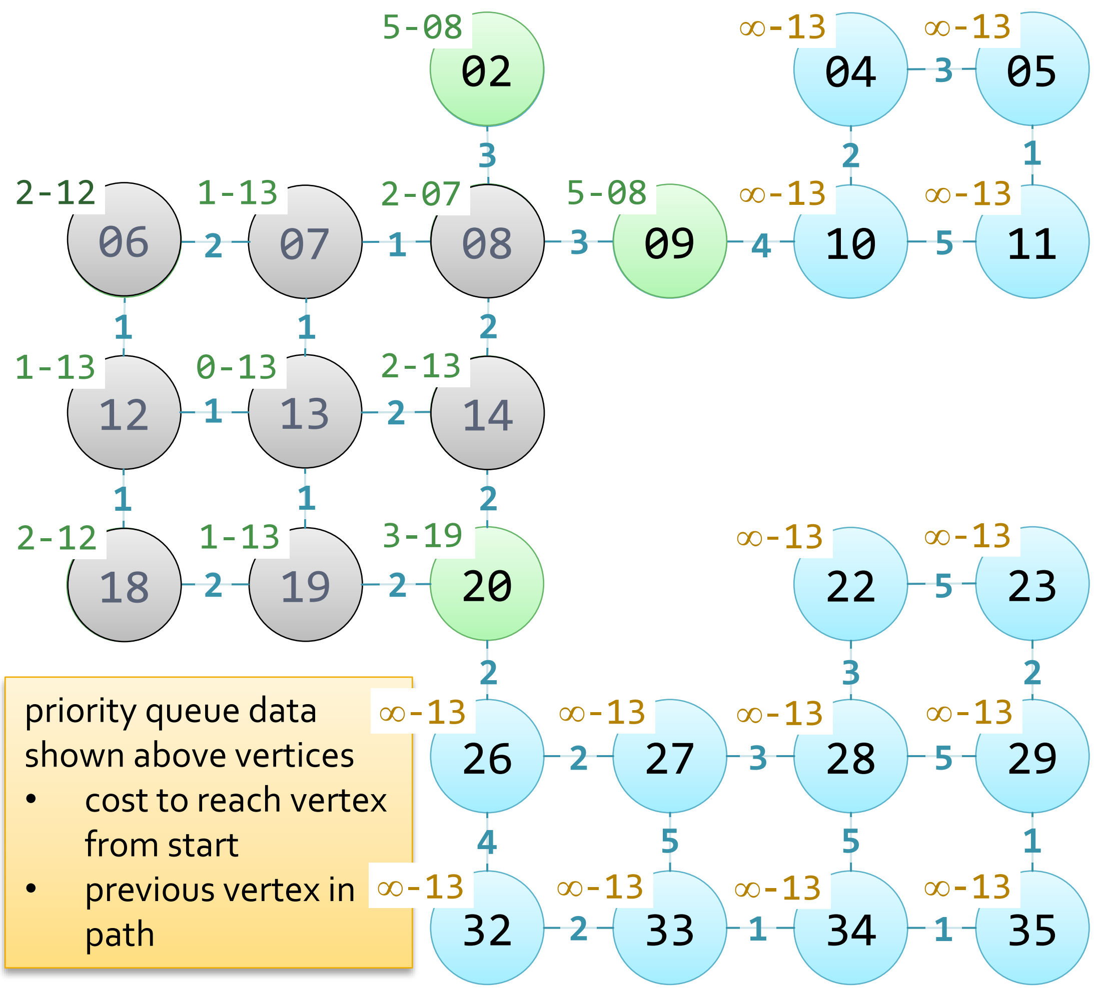
priority queue data shown above vertices
- cost to reach vertex from start
- previous vertex in path

- Remove vertex, *v*, from priority queue
  - Insert in results list
- Compare cost, *c*, to reach each adjacent vertex, *u*
  - If $c_u > (c_v + \text{edge}[v, u])$
  - Update cost and set *u*'s parent to *v*

label	cost	parent
13	0	-
07	1	13
12	1	13
19	1	13

label	cost	parent

Dijkstra's Algorithm Example 4

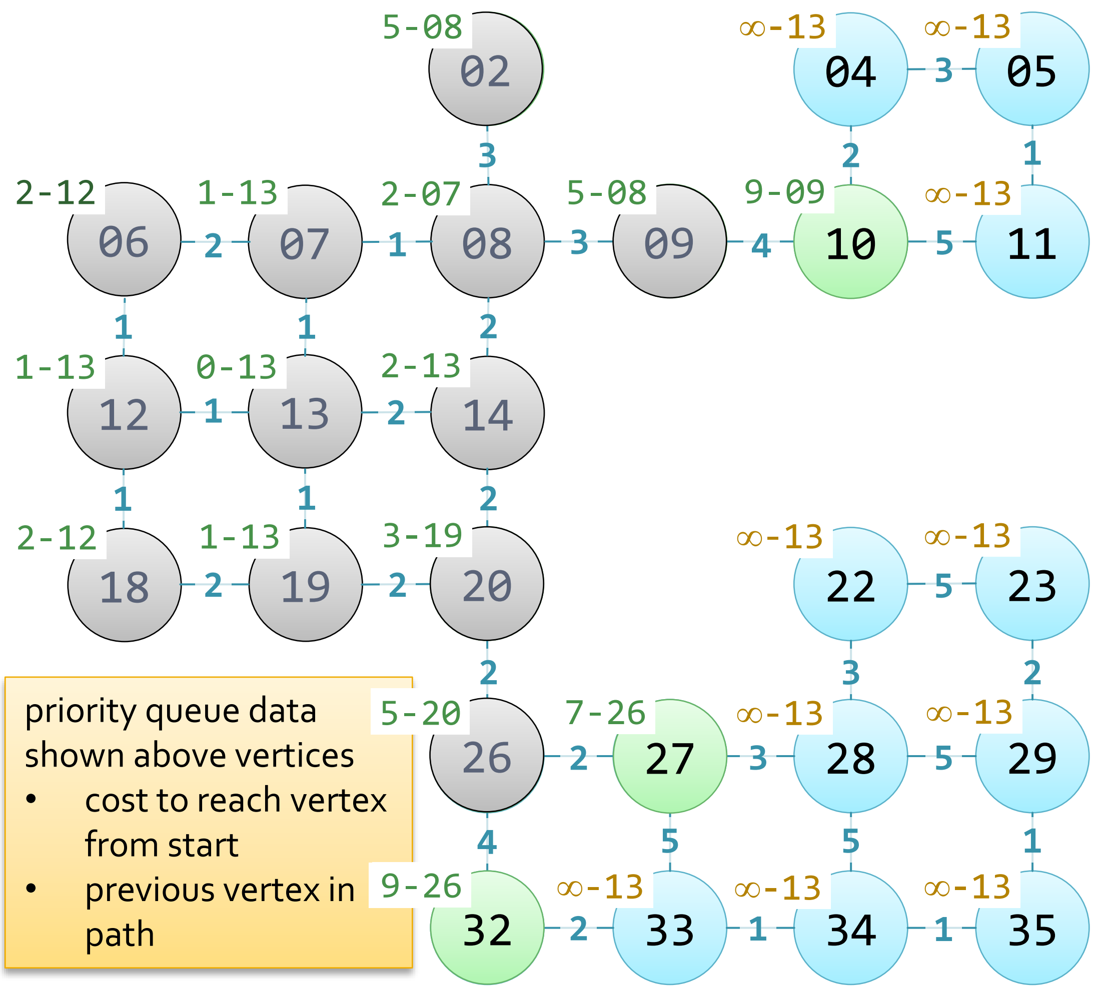


- Remove vertex, *v*, from priority queue
  - Insert in results list
- Compare cost, *c*, to reach each adjacent vertex, *u*
  - If $c_u > (c_v + \text{edge}[v, u])$
  - Update cost and set *u*'s parent to *v*

label	cost	parent
13	0	-
07	1	13
12	1	13
19	1	13
06	2	12
08	2	07
18	2	12
14	2	13

label	cost	parent

Dijkstra's Algorithm Example 5



priority queue data shown above vertices

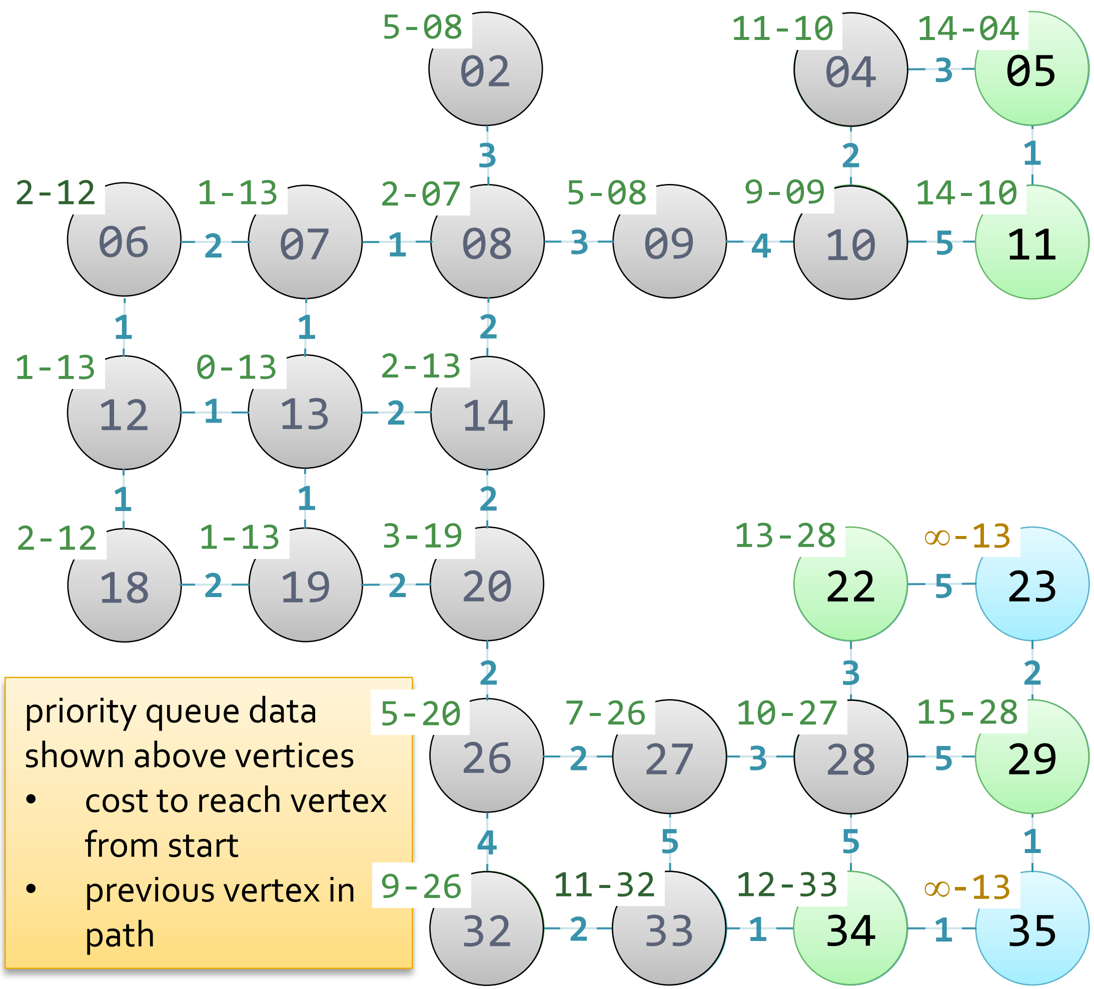
- cost to reach vertex from start
- previous vertex in path

- Remove vertex, *v*, from priority queue
  - Insert in results list
- Compare cost, *c*, to reach each adjacent vertex, *u*
  - If $c_u > (c_v + \text{edge}[v, u])$
  - Update cost and set *u*'s parent to *v*

label	cost	parent
13	0	-
07	1	13
12	1	13
19	1	13
06	2	12
08	2	07
18	2	12
14	2	13
20	3	19
02	5	08
09	5	08
26	5	20

label	cost	parent

Dijkstra's Algorithm Example 6



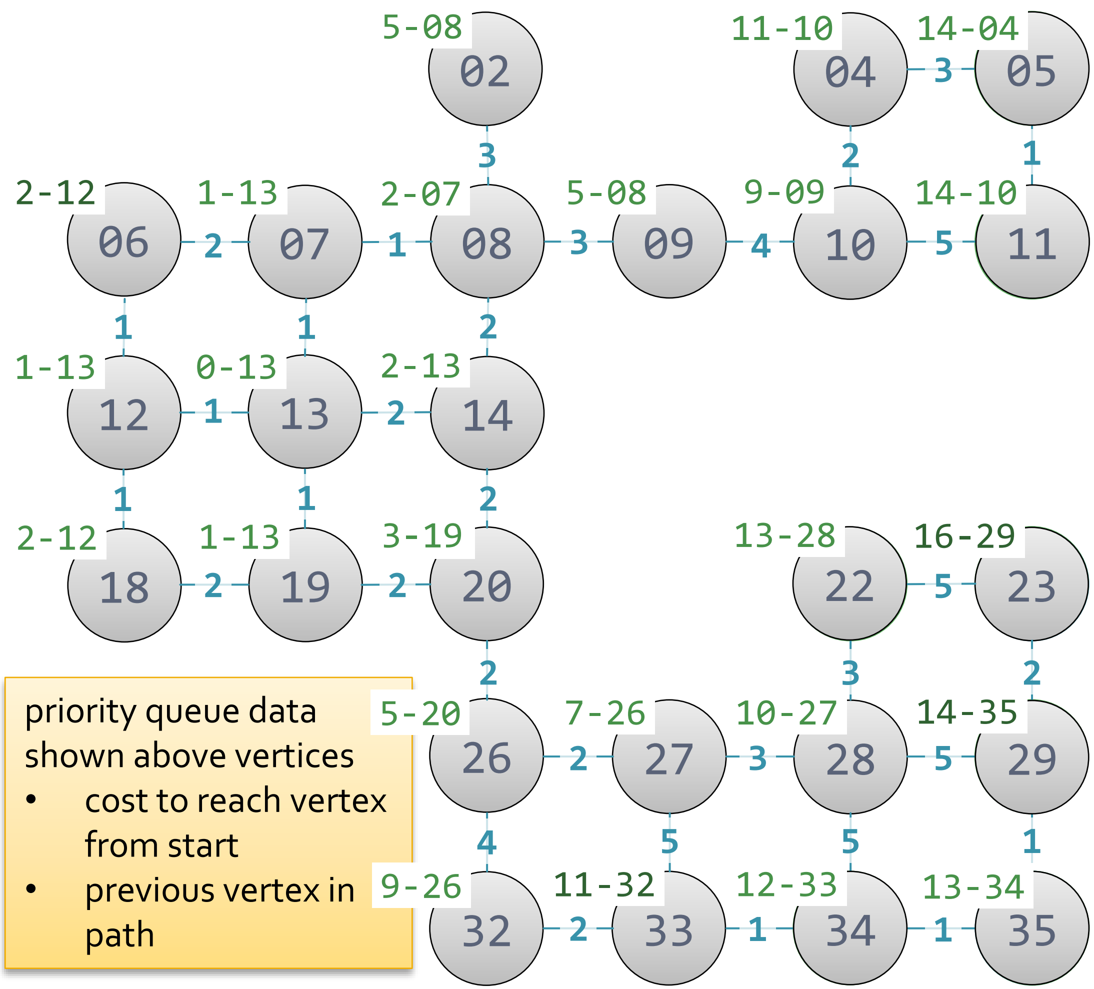
priority queue data shown above vertices
- cost to reach vertex from start
- previous vertex in path

- Remove vertex, *v*, from priority queue
  - Insert in results list
- Compare cost, *c*, to reach each adjacent vertex, *u*
  - If $c_u > (c_v + \text{edge}[v, u])$
  - Update cost and set *u*'s parent to *v*

label	cost	parent
13	0	-
07	1	13
12	1	13
19	1	13
06	2	12
08	2	07
18	2	12
14	2	13
20	3	19
02	5	08
09	5	08
26	5	20
27	7	26

label	cost	parent
10	9	09
32	9	26
28	10	27
04	11	10
33	11	32

Dijkstra's Algorithm Example 7



priority queue data shown above vertices

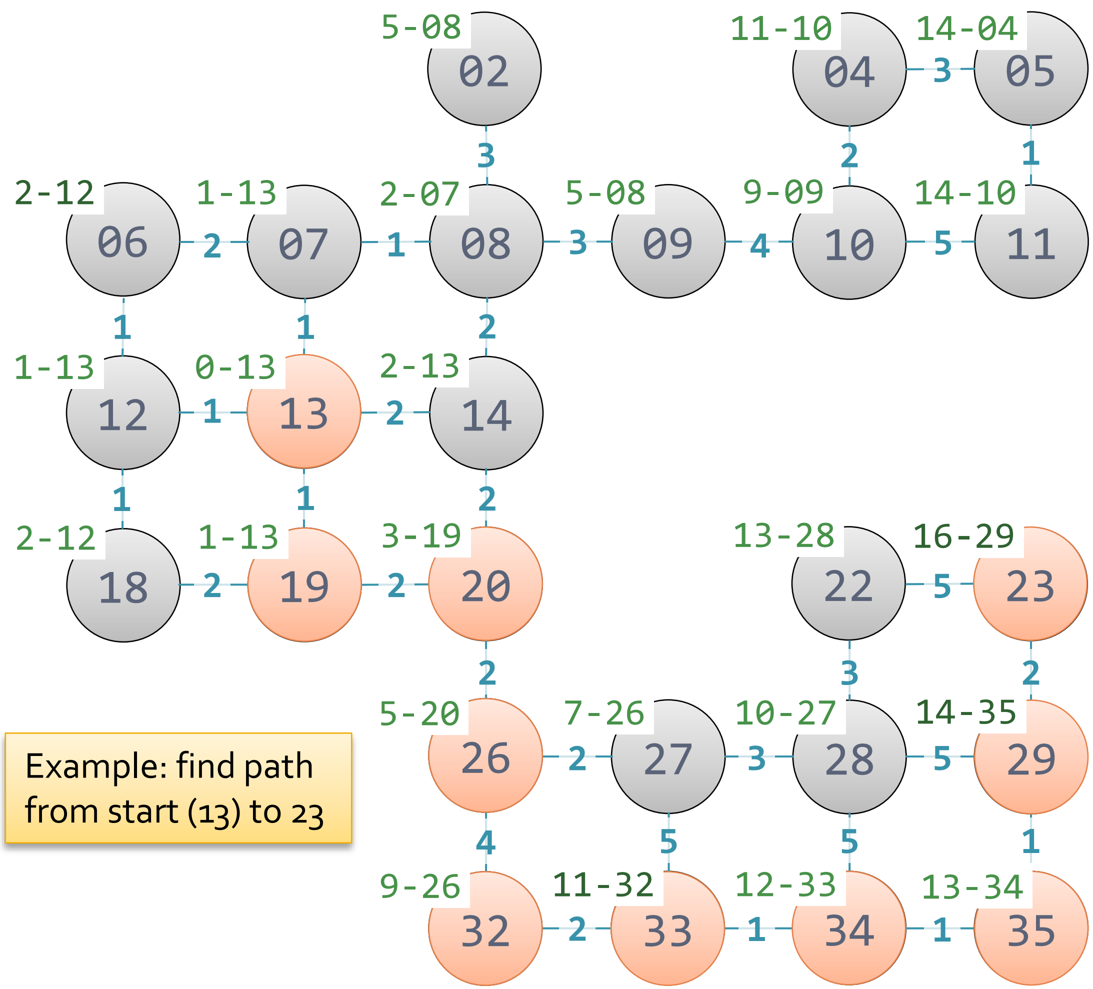
- cost to reach vertex from start
- previous vertex in path

- Remove vertex, *v*, from priority queue
  - Insert in results list
- Compare cost, *c*, to reach each adjacent vertex, *u*
  - If $c_u > (c_v + \text{edge}[v, u])$
  - Update cost and set *u*'s parent to *v*

label	cost	parent
13	0	-
07	1	13
12	1	13
19	1	13
06	2	12
08	2	07
18	2	12
14	2	13
20	3	19
02	5	08
09	5	08
26	5	20
27	7	26

label	cost	parent
10	9	09
32	9	26
28	10	27
04	11	10
33	11	32
34	12	33
22	13	28
35	13	34
05	14	04
11	14	10
29	14	35
23	16	29

Retrieving the Shortest Path



Example: find path from start (13) to 23

- To find a path backtrack through results list
  - Start with end vertex
  - Find parent vertex
  - Repeat until start reached
  - Results list should allow efficient vertex searching

label	cost	parent
13	0	-
07	1	13
12	1	13
19	1	13
06	2	12
08	2	07
18	2	12
14	2	13
20	3	19
02	5	08
09	5	08
26	5	20
27	7	26

label	cost	parent
10	9	09
32	9	26
28	10	27
04	11	10
33	11	32
34	12	33
22	13	28
35	13	34
05	14	04
11	14	10
29	14	35
23	16	29

Dijkstra's Algorithm Operations

- The cost of the algorithm depends on $|E|$ and $|V|$ and the data structures used in the algorithm
  - For the priority queue and results list
- Consider the process
  - Whenever a vertex is removed from the priority queue each of its adjacent edges must be found
  - There are $|V|$ vertices to be removed and
  - For each of $|E|$ edges 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 is 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 each change in cost for an additional cost of O(log|*V*|)
  - Worst case cost is |*V*| + |*V*|*log|*V*| + |*E*|*(|*V*| + log|*V*|)
    - O((|*V*|*log|*V*|) + (|*E*|*|*V*|))
  - If the heap is indexed the cost is O((|*V*| + |*E*|) *log|*V*|)

The implementation we covered

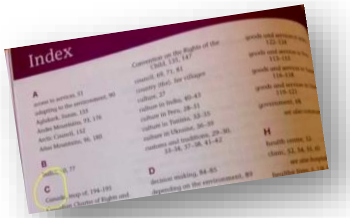
Remember heaps do not support search

Bad!

Best, average case difference

What's an Indexed Heap?

- The cost analysis refers to an *indexed* heap
  - An index is a secondary data structure used to improve access to a primary data structure
    - Like an index …



- Use a second data structure to find a heap node given the vertex label
  - The index records vertex labels and their corresponding indices in the underlying array of the heap
  - The index structure supports fast access by vertex label (its key)
    - Could use a hash table for the index
  - When an element is bubbled up in the heap the index also needs to be modified

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 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 trivial change ...

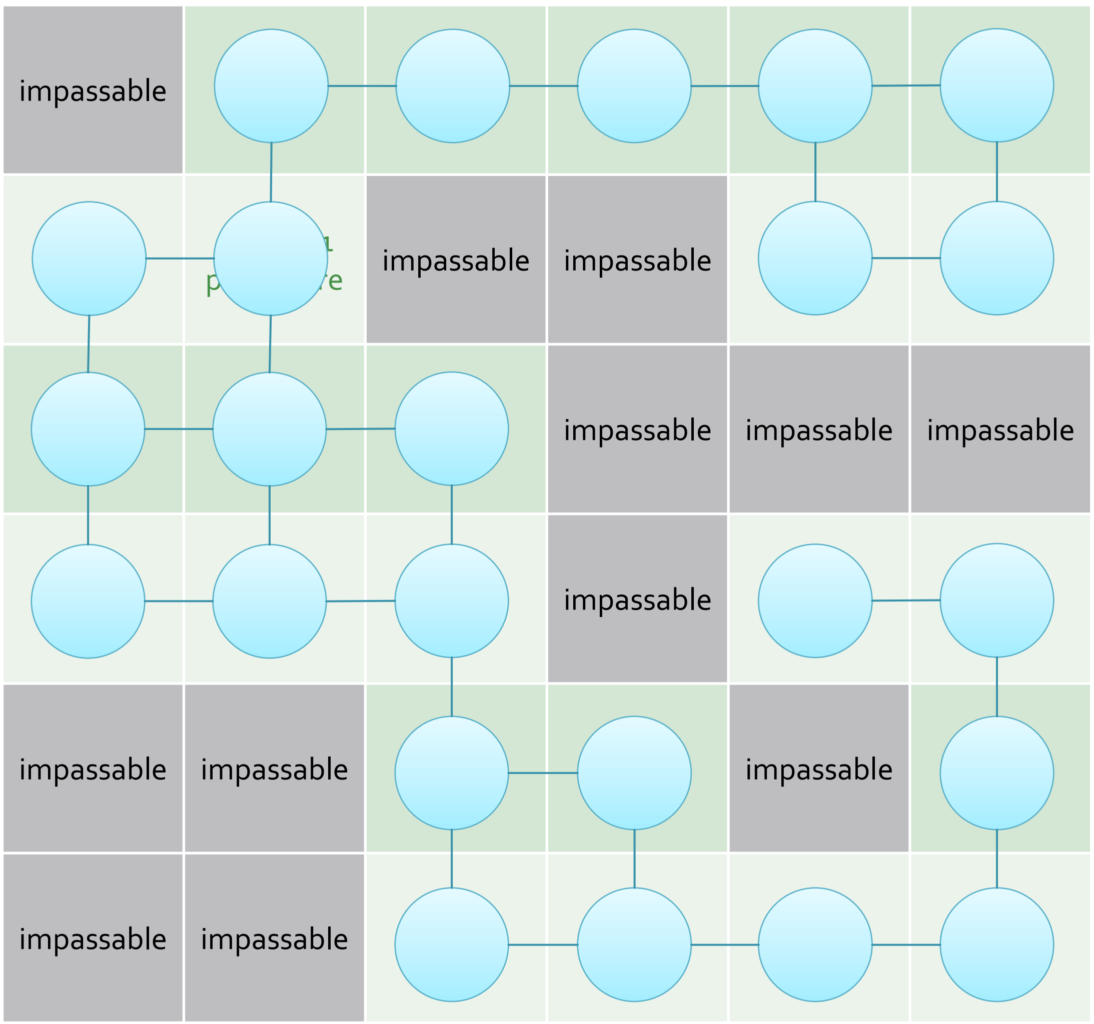
A* Algorithm

- 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

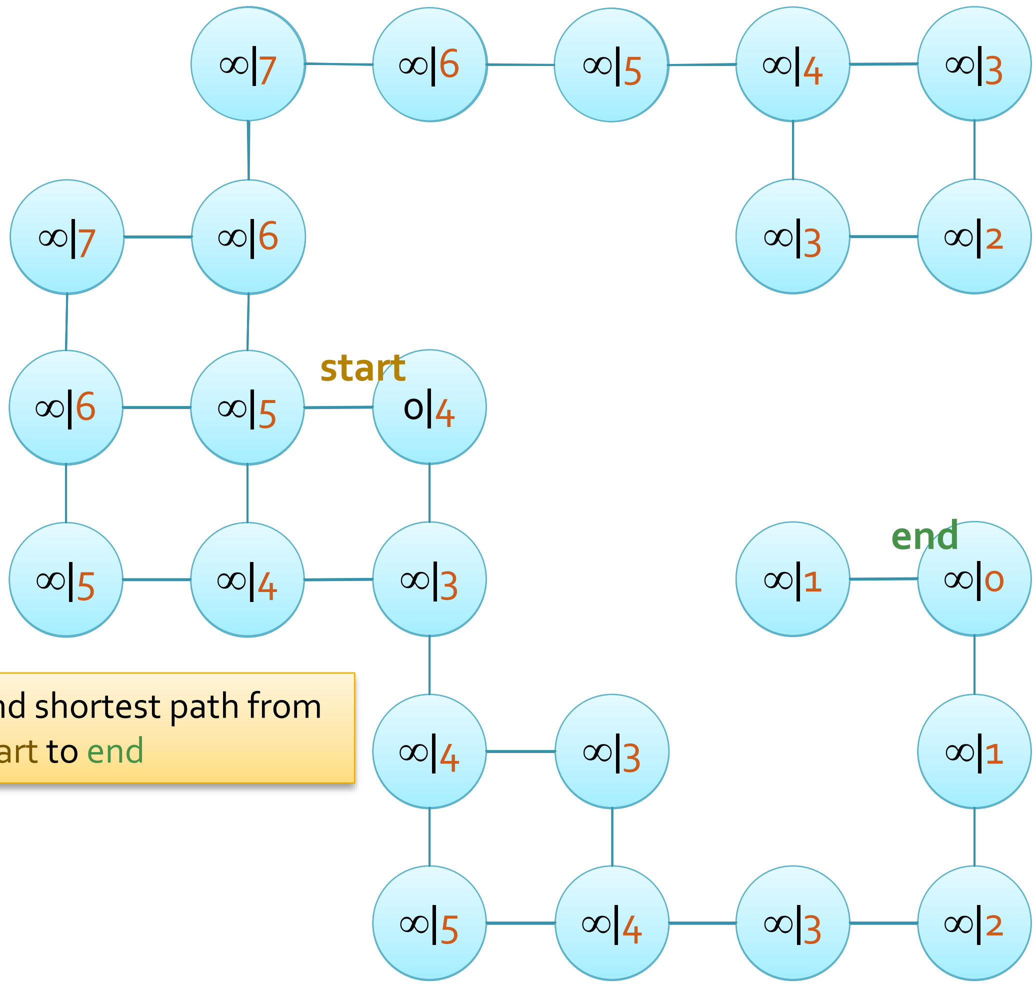
- The key to the efficiency of the A* algorithm is the accuracy of the heuristic, *h*
- To find an optimal path *h* should be *admissible*
  - It should not *overestimate* the cost of the path to the goal
  - Inadmissible heuristics may result in non-optimal paths
    - But may be faster than an inaccurate 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 1



- Imagine a grid gray squares
  - Gaps correspond to some game feature
- The corresponding graph is unweighted
  - Mostly to simplify the example
  - And focus on distinction between *g* and *h*
- *g* and *h* values
  - *g* – path length from start
  - *h* – number of squares to end square on full grid
    - straight-line cost
- Vertices annotated with *g* and *h* values
  - *g*|*h*
  - Grid labels not shown

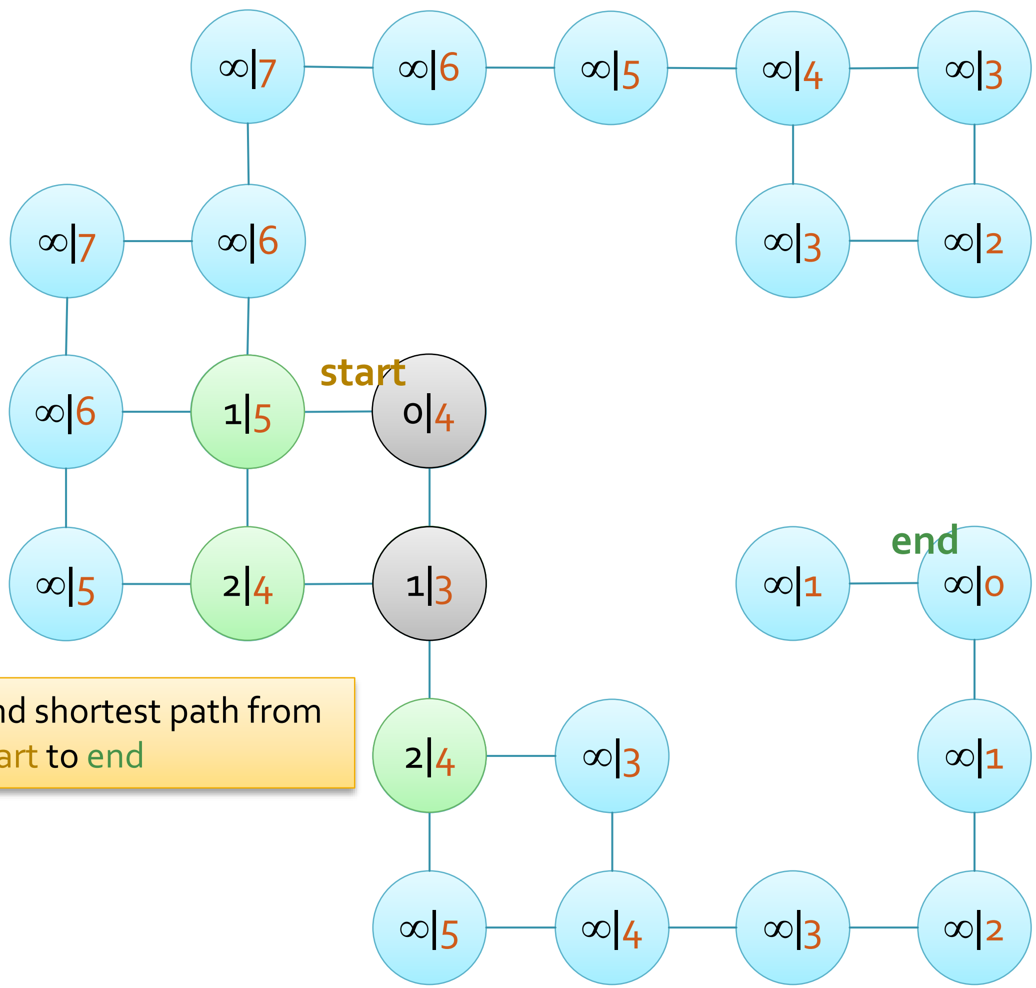
A* Search Example 2



find shortest path from start to end

- The A* algorithm is similar to Dijkstra's except
  - Terminates when end vertex found
  - Values in priority queue based on *f* value
    - Which includes cost estimate to end (*h*)
- Initialization
  - Compute *h* for all vertices
  - Set *g* values
    - 0 for start
    - Infinity for all other vertices

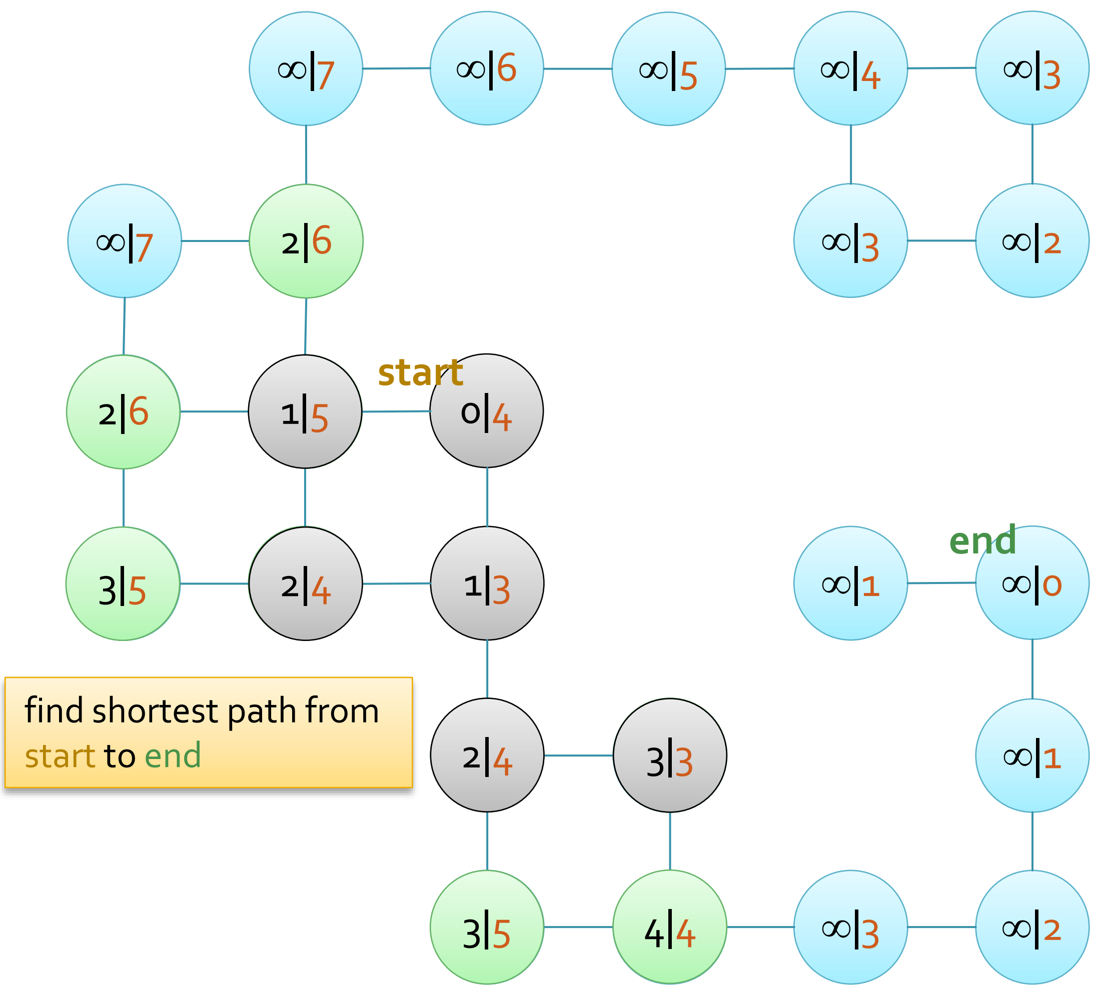
A* Search Example 3



find shortest path from start to end

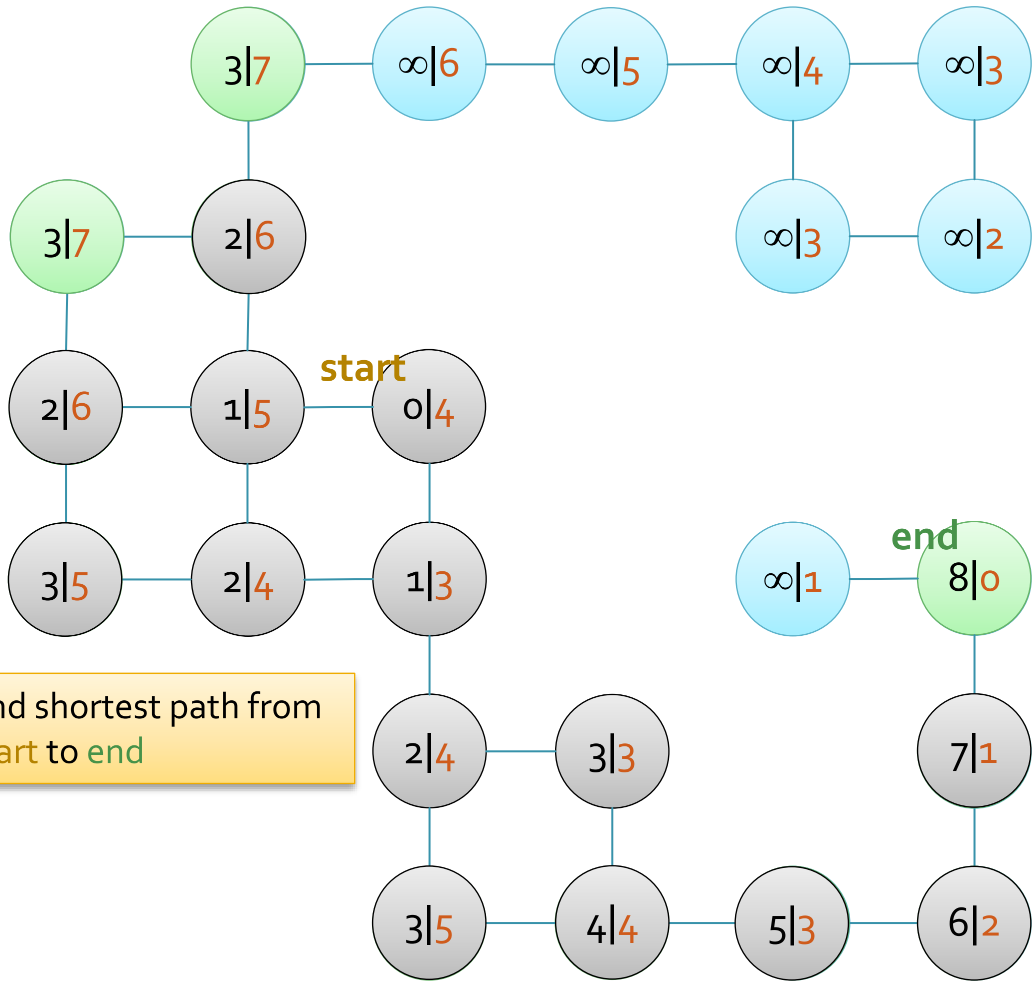
- Remove root from priority queue
  - Adjust *g* values of neighbours
- Remove new root
  - Remember – based on *f*, the sum of *g* + *h*
    - the vertex with *f* = 4

A* Search Example 4



- Repeat
  - Until end removed from priority queue
- Note multiple vertices with $f = 6$
  - Remove whichever happens to be root

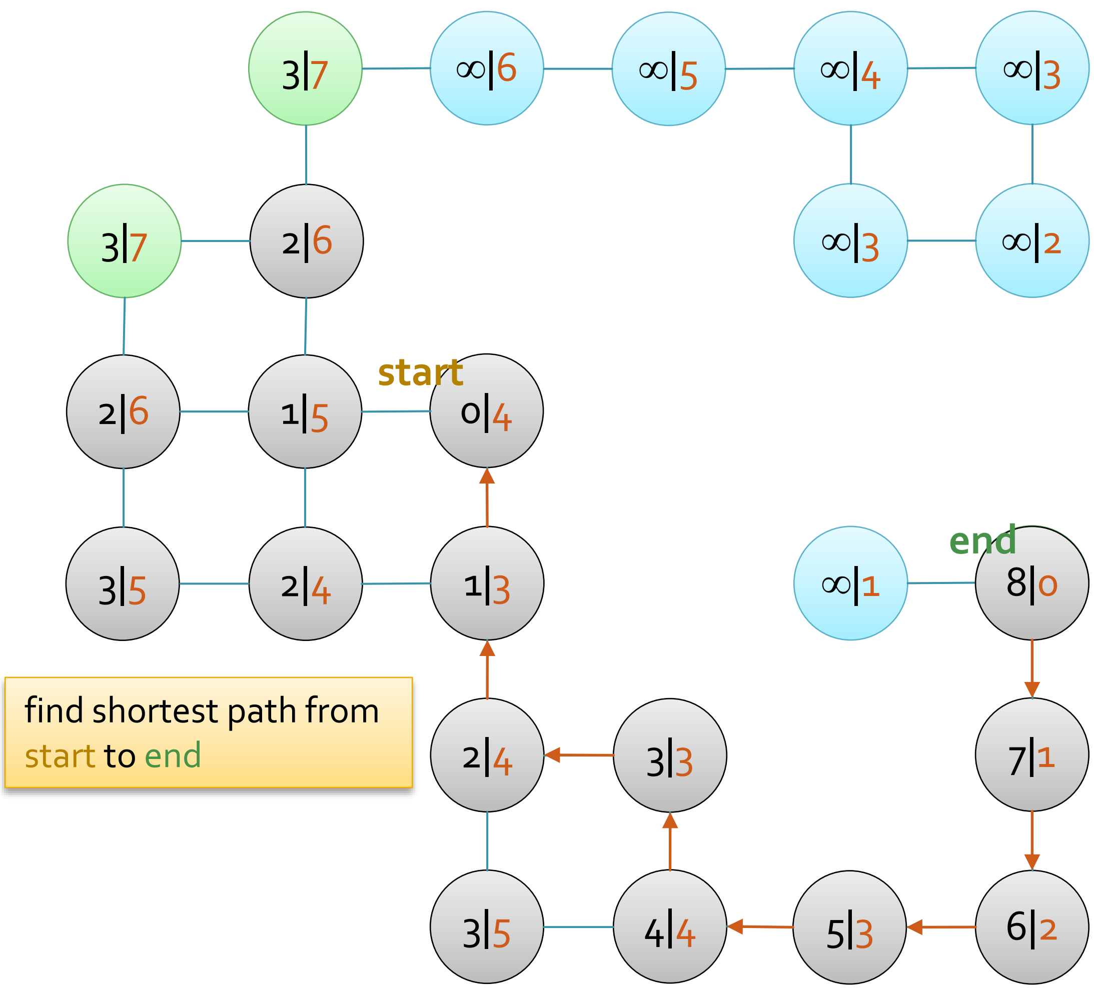
A* Search Example 5



find shortest path from start to end

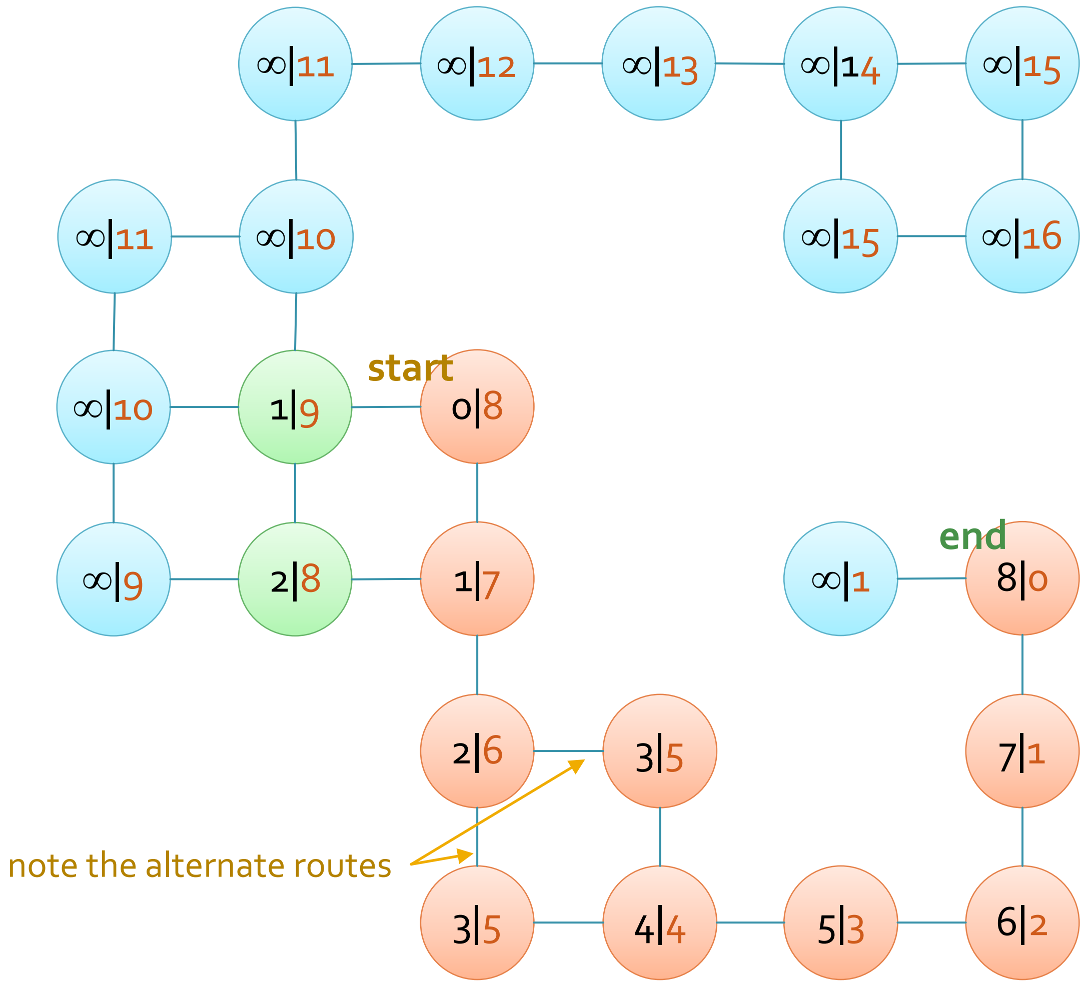
- Repeat
  - Until end removed from priority queue
- Now note multiple vertices with $f = 8$
  - Remove whichever happens to be root

A* Search Example 6



- Repeat
  - Until end removed from priority queue
- To find shortest path
  - Backtrack through results list
  - Like Dijkstra's algorithm

Perfect Heuristic



- A *perfect* heuristic gives correct cost
  - From vertex to end
  - Only vertices on shortest path are removed from the priority queue
- This graph shows a perfect heuristic
- But …
  - There isn't a good general purpose heuristic generator
  - It helps to know about the graph
    - Bottlenecks
    - And other features