AI for games

Lecture 4

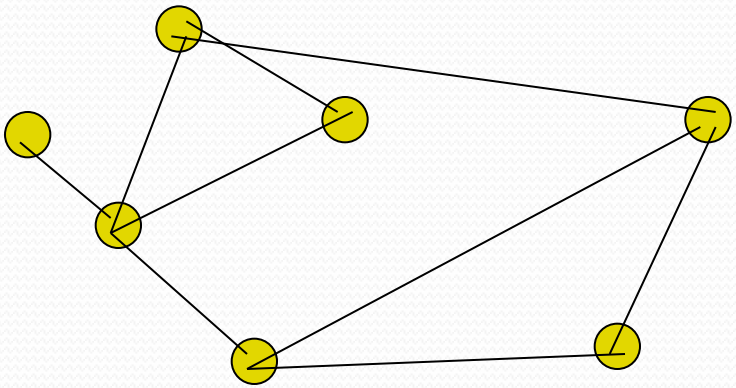
Pathfinding

Organiser

- Graphs
- Storing a sparse graph
- Pathfinding
  - Dijkstra’s algorithm
  - A*
- Path smoothing
- Optimisation issues

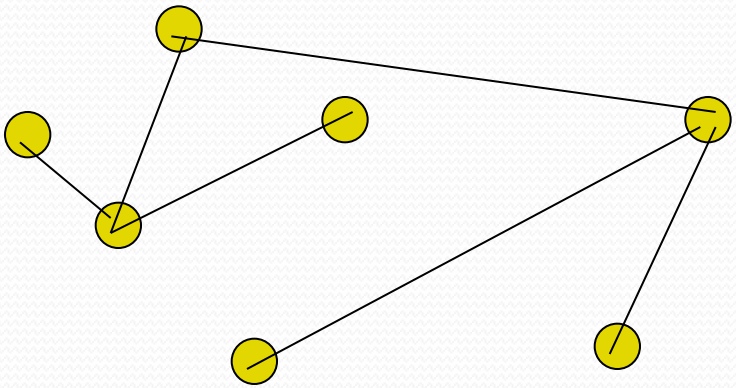
Graphs

- To a mathematician (and an AI guy), a graph is:
  - A set of “nodes” linked by “edges”.
  - Very important for a wide range of maths uses.



Trees

- Like graphs, but no cycles.
  - ie Tech Trees are really graphs



Graphs

- They are important for a whole range of AI applications.
  - Not just pathfinding.
- E.g.
  - Minimax trees
  - Artificial neural networks
  - Decision trees
  - Goal trees
  - Technology trees

Graphs

- We'll keep to pathfinding for now.
- In this context:
  - A "node" is one of the waypoints we found last week.
  - An "edge" is a line between two waypoints.
- Our graphs are "weighted", meaning that the length of the edges matters to us.
  - Often called the "cost".

Digraphs

- Note that most of the time, the “cost” to go from A to B is the same as the cost to go from B to A.
- If this is not the case (hilly terrain, one-way doors), we need to store the cost of two edges for each pair of nodes.
  - Twice as much memory.
  - Called a “digraph”.

Storing a graph

- One obvious way to store a navigation graph is a simple table:

	A	B	C	D
A	0	7.6	1.5	-
B	7.6	0	-	4.2
C	1.5	-	0	1.9
D	-	4.2	1.9	0

Visibility table

Node	A	B	C	D	E	F	G
A	-	12	20	-	-	-	9
B	12	-	-	46	-	-	-
C	20	-	-	12	-	-	-
D	-	46	12	-	9	72	-
E	-	-	-	9	-	12	-
F	-	-	-	72	12	-	12
G	9	-	-	-	-	12	-

Storing a graph

- If it is not a digraph, we can reduce the memory cost.

	B	C	D
A	7.6	1.5	-
B		-	4.2
C			1.9

Storing a graph

- Even so, a map with 300 nodes is going to take a lot of memory.
- But the graph will be very sparse.
- A more efficient way to store the data will be by giving each node a list of the edges connected to it.

Storing a graph

```
class Edge
{
  int fromIndex;
  int toIndex;
  double cost;
};

class Node
{
  int indexNumber;
  list<Edge> edgeList;
  Vector2D position;
};
```

Storing a graph

```
class Graph
{
  vector<Node> NodeVector;
};
```

- Why a vector for the nodes and a list for the edges?
  - Because there is little point accessing a sparse collection by index

Storing a graph

Node	Edges		
1	1-3 (12.2)	1-7 (6.2)	1-9 (2.7)
2	2-4 (12.2)	2-9 (3.2)	
3	3-1 (12.2)	3-4 (1.8)	3-12 (8.8)
4			
etc	3-1 (12.2)		

Storing a graph

- You then just need to add
  - methods to the Node to set and access edges
  - methods to the graph to set and access edges via the vector of nodes

- Buckland's book has a more involved (and faster) method that also allows you to "delete" nodes and edges easily. The code includes a customised iterator and other fancy stuff.

Storing a graph

- Ultimately, you want to be able to:
  - Build a list of nodes and edge.
  - Quickly ask each node for a list of connected nodes.
  - Quickly ask for the cost between two nodes.
  - Find the closest (accessible) node to any specified location.

Pathfinding

- Next we need a method that gives us a route from any node to any other node.
- To find a path we need to:
  - Find the closest node to the start point.
  - Find the closest node to the end point.
  - Search the graph to find the optimal path between the two nodes.

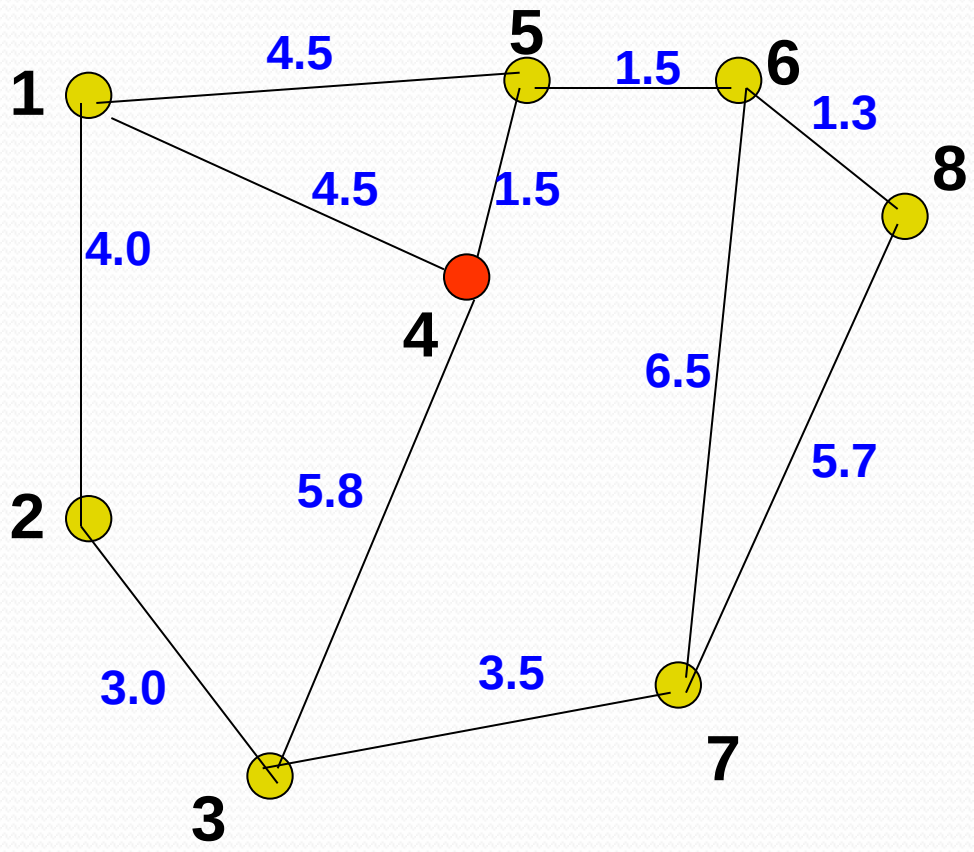
Pathfinding

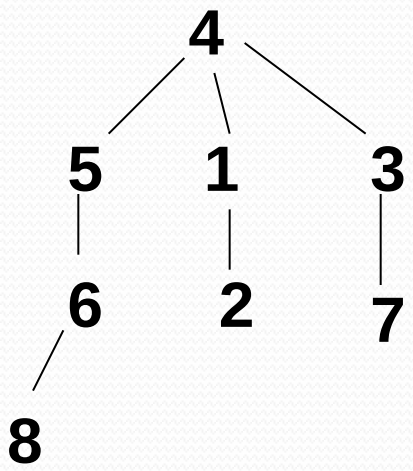
- The resultant path is a (STL) list of…..
  - Edges?
  - Nodes?
  - Positions?
- It is sometimes handy to have “extra” edges or nodes to mark the end and start positions.
  - Good idea to add these after the algorithm has generated the list, since you don’t want the graph class to consider anything except the known nodes.
  - Buckland has a slightly different approach.

Dijkstra's algorithm

- To search a graph to find an optimal path, we use the concept of a "shortest path tree".
- This is a tree that shows the shortest path to any node on the graph (from the current node).
- Consider:

Dijkstra's algorithm





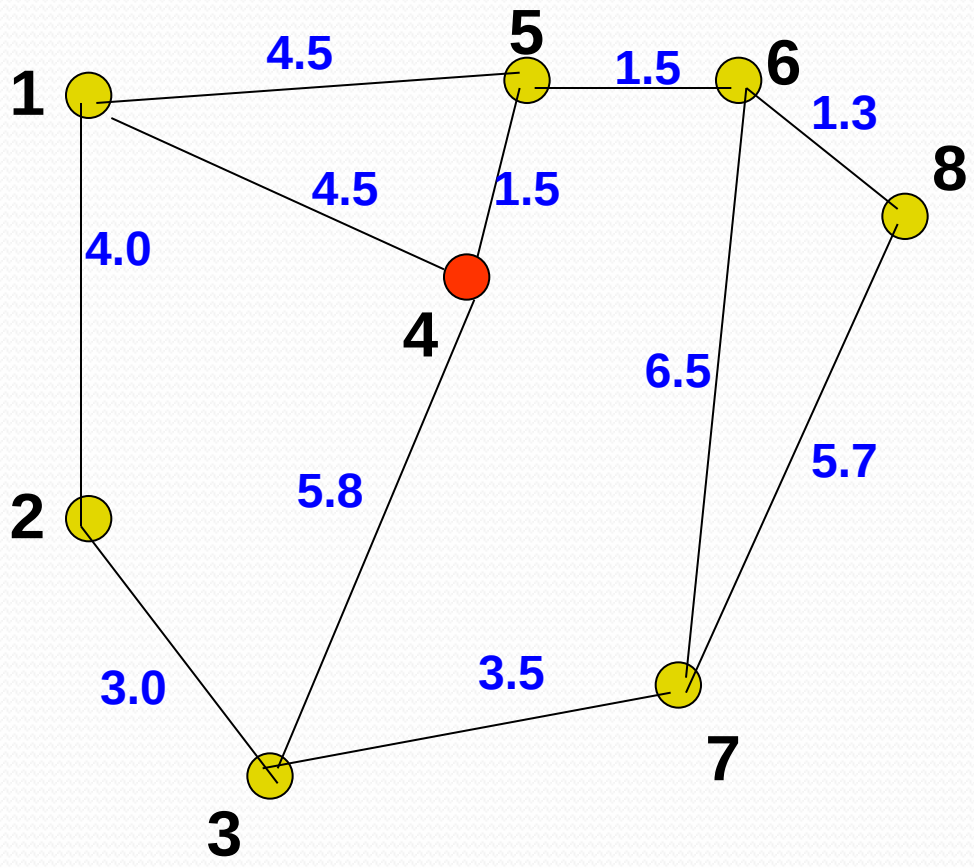
Dijkstra's algorithm

- The aim is to build this tree.
- But you don't need to build all of it.
  - Just enough to be sure you have the best path.
- Dijkstra's algorithm builds the shortest path tree efficiently and does not have to build the whole tree to be sure the path found is optimal.

Dijkstra's algorithm

- The basic iterative principal for constructing a shortest path tree:
  1. Given a partial shortest path tree
  2. Find the node that is closest to the leaves of the tree
  3. And add it
  4. Repeat 1-3 until the tree is complete

Dijkstra's algorithm

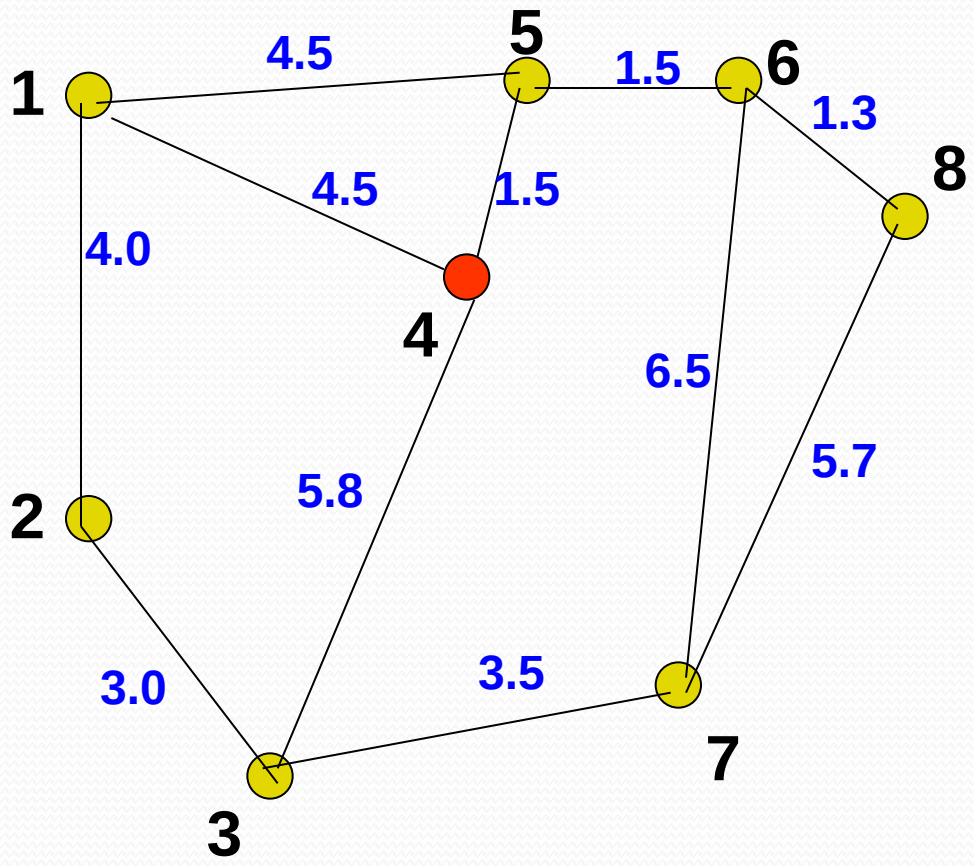


- Consider going from 4 to 7
- Put 4 in the list. Add 5, 1 and 3 to the *frontier*.

5(1.5)	1(4.5)	3(5.8)

4

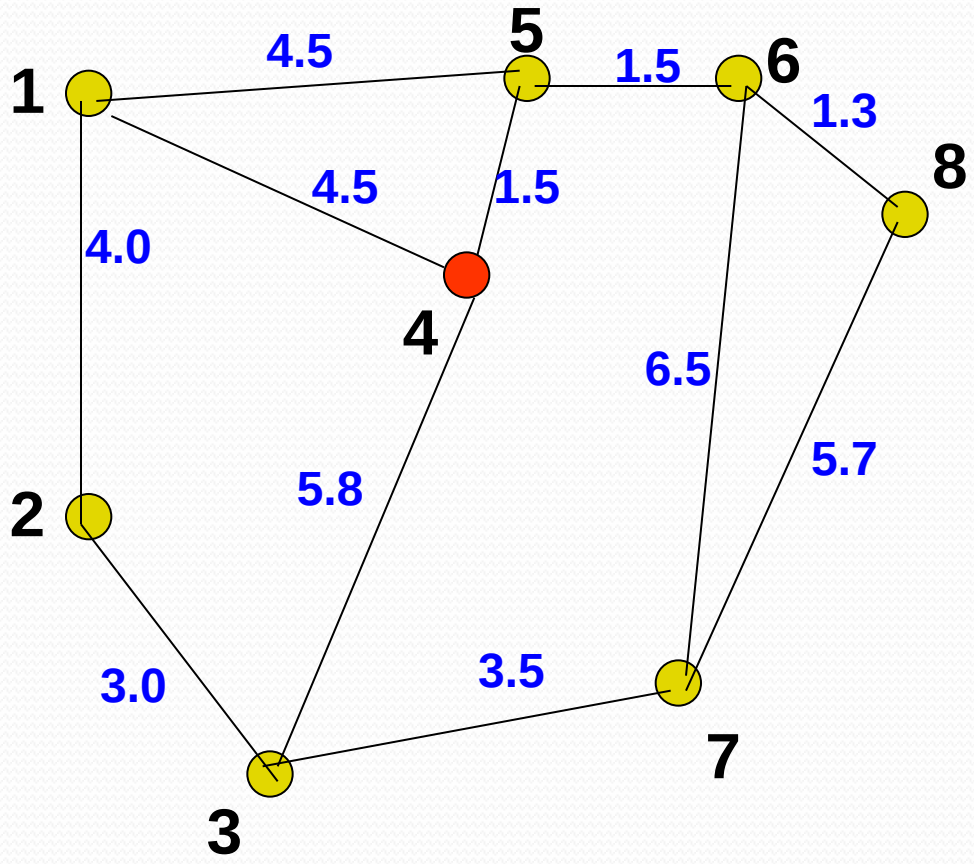
Dijkstra's algorithm



- Closest is 5. Add to tree. Add 6 to frontier.

1(4.5)	3(5.8)	6(3.0)

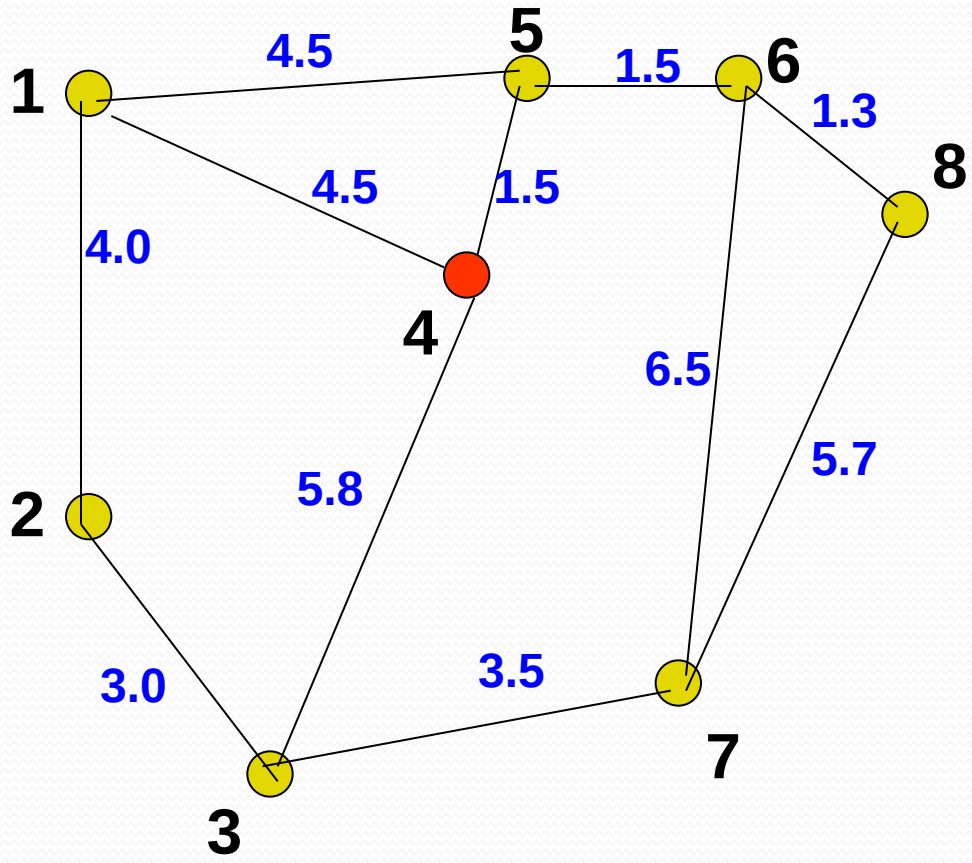
Dijkstra's algorithm



- Closest is 6. Add to tree. Add 8 and 7 to frontier.

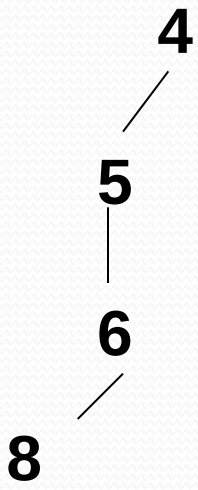
1(4.5)	3(5.8)	8(4.3)
7(9.5)		

Dijkstra's algorithm

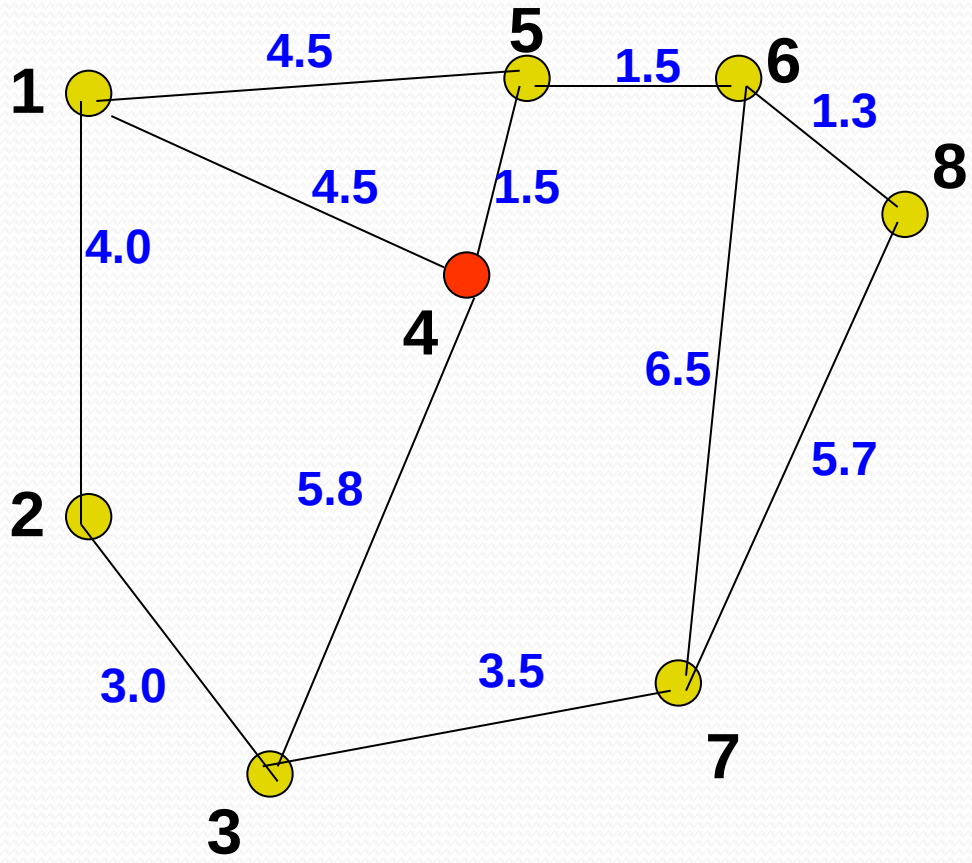


- Closest is 8. Add to tree. 7 already on frontier. This path is not closer, so do not update.

1(4.5)	3(5.8)
7(9.5)	

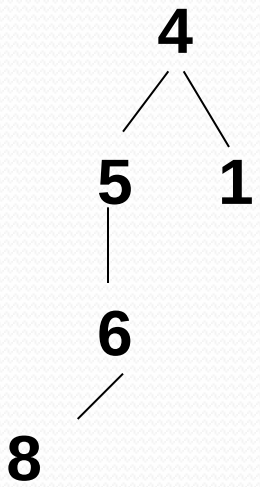


Dijkstra's algorithm

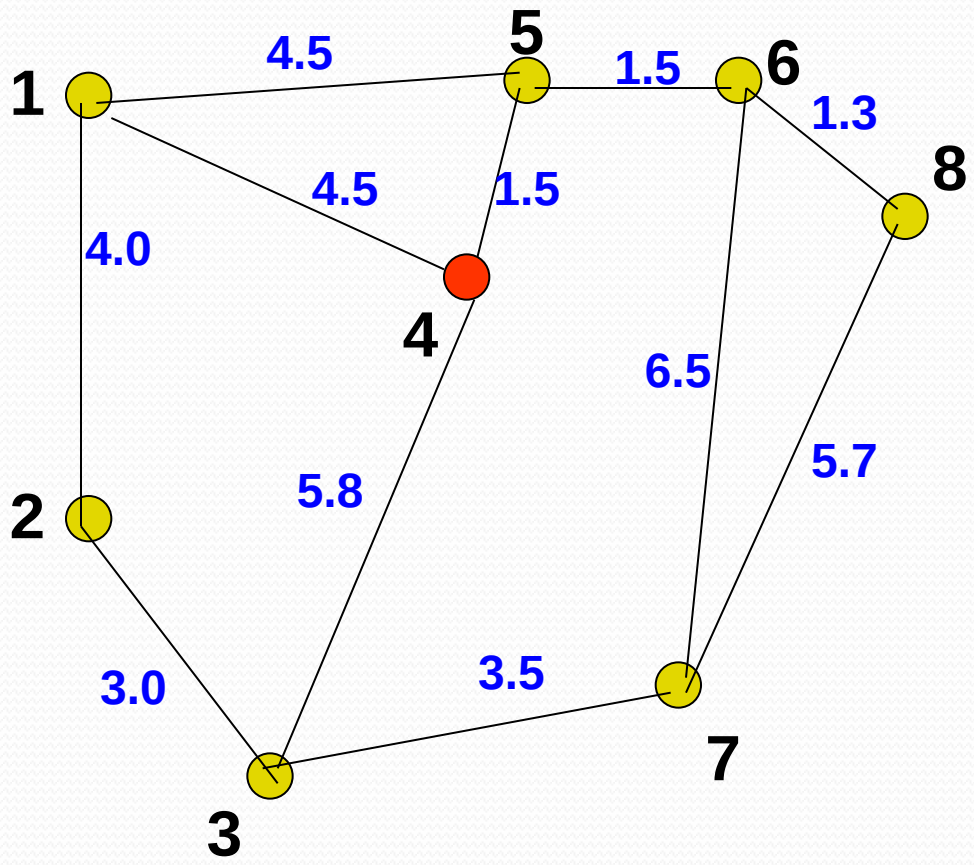


- Closest is 1. Add to tree. Add 2 to frontier. New route to 5 is not closer. No update.

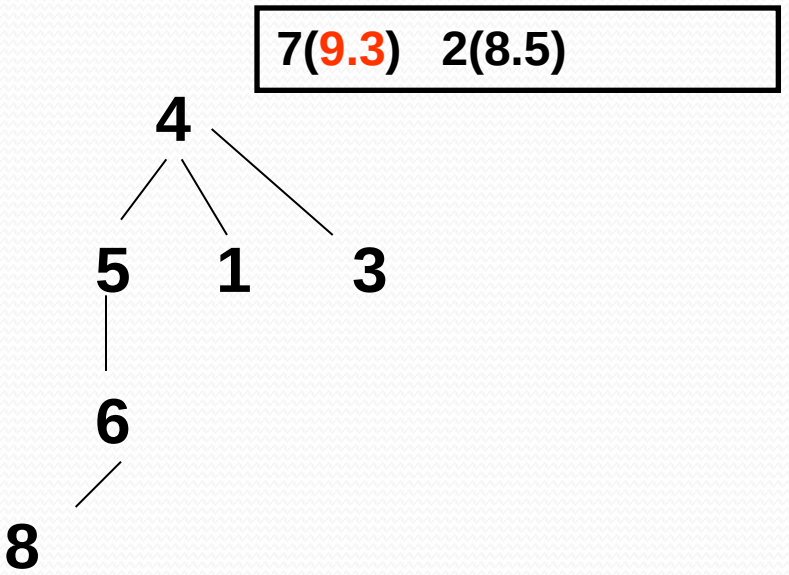
3(5.8)	7(9.5)	2(8.5)



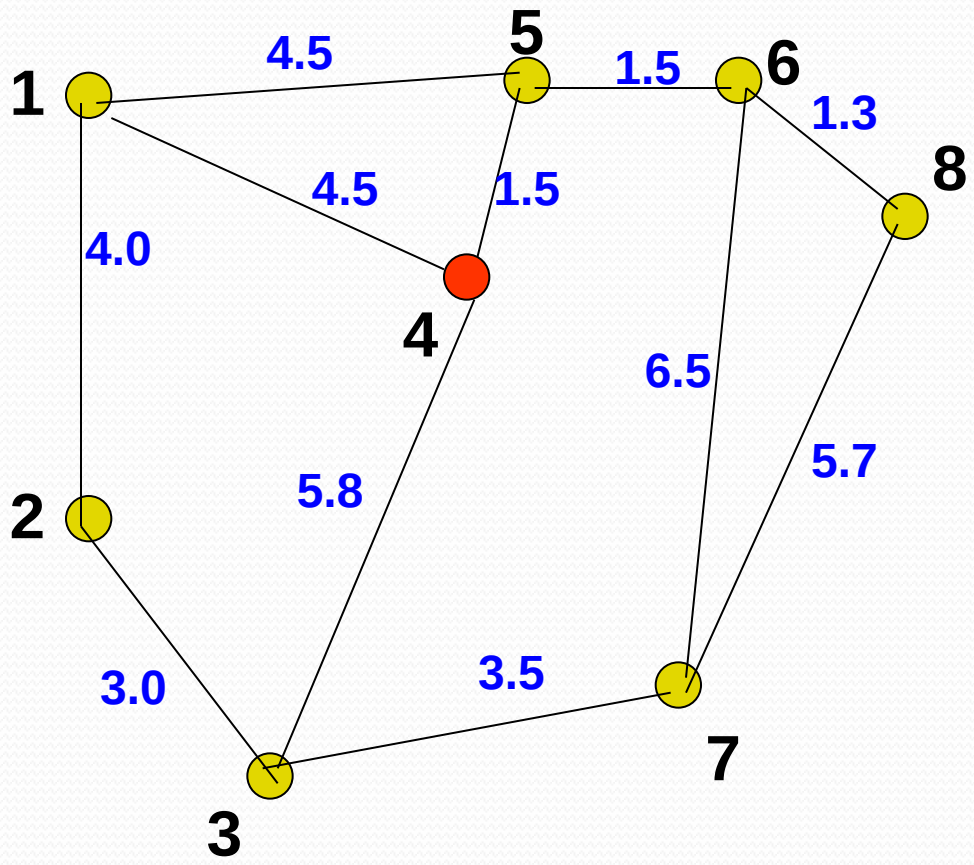
Dijkstra's algorithm



- Closest is 3. Add to tree. New route to 2 is not closer. No update. New route to 7 is closer. Update.

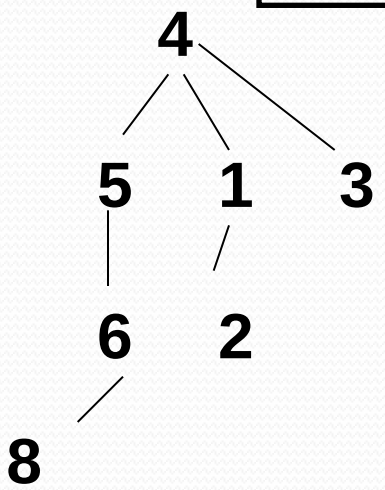


Dijkstra's algorithm

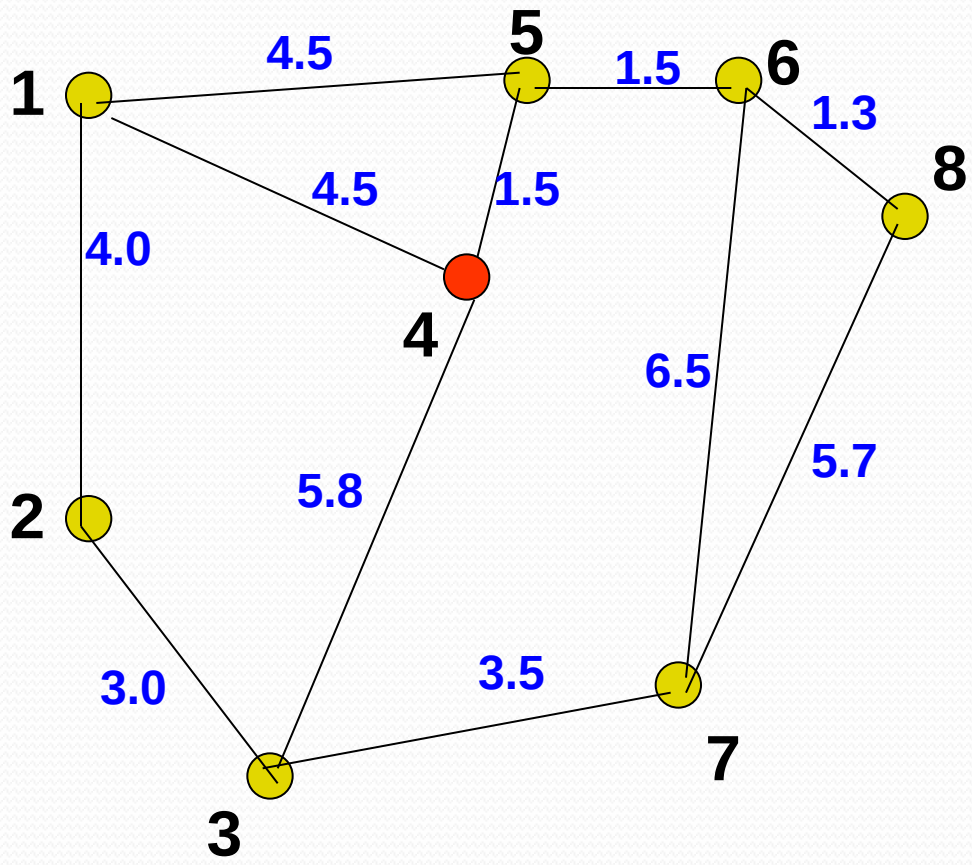


- Closest is 2. No new nodes or updates.

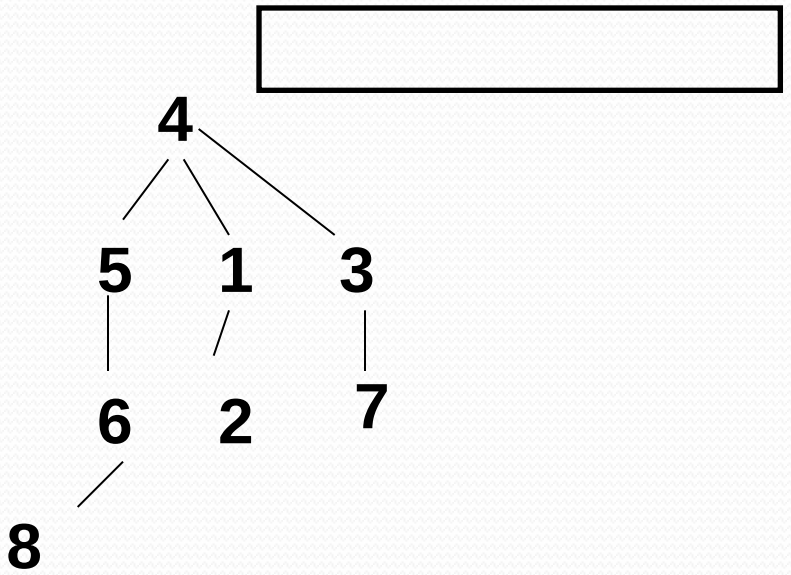
7(9.3)



Dijkstra's algorithm



- Closest is 7. No new nodes or updates. This must be shortest route.



Dijkstra's algorithm

- This example explored all nodes, but try it for route 4 to 2.
  - Some nodes not explored.
  - But still sure this is closest route.
- Note the concept of "edge relaxation".
  - Edges of frontier have "shortest known route", but this can be reduced by exploring from new edges on tree.

A*

- A-star adds the understanding that the nodes are in Euclidean space.
- Rather than exploring the closest node on the frontier, it goes for the closest node that seems to be in the right direction.

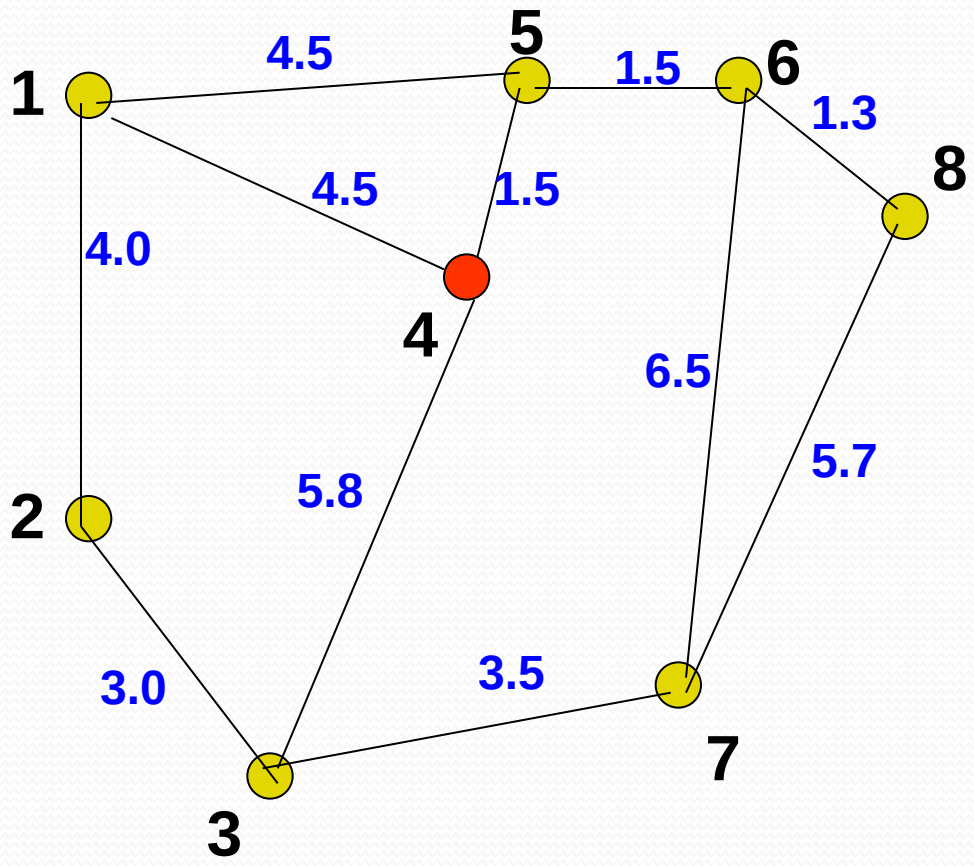
A*

- “Closest node” is no longer the node in the frontier with the shortest known path, but the shortest:
  - F = G + H where
  - G = Known path
  - H = Estimate of remaining distance

A*

- The estimate is called the “heuristic”, H.
- As long as the estimate is never larger than the actual distance, A* will produce an optimal path.
  - Use the straight-line distance?
  - Square root!!!

A*



- Put 4 in the list. Add 5, 1 and 3 to the frontier.

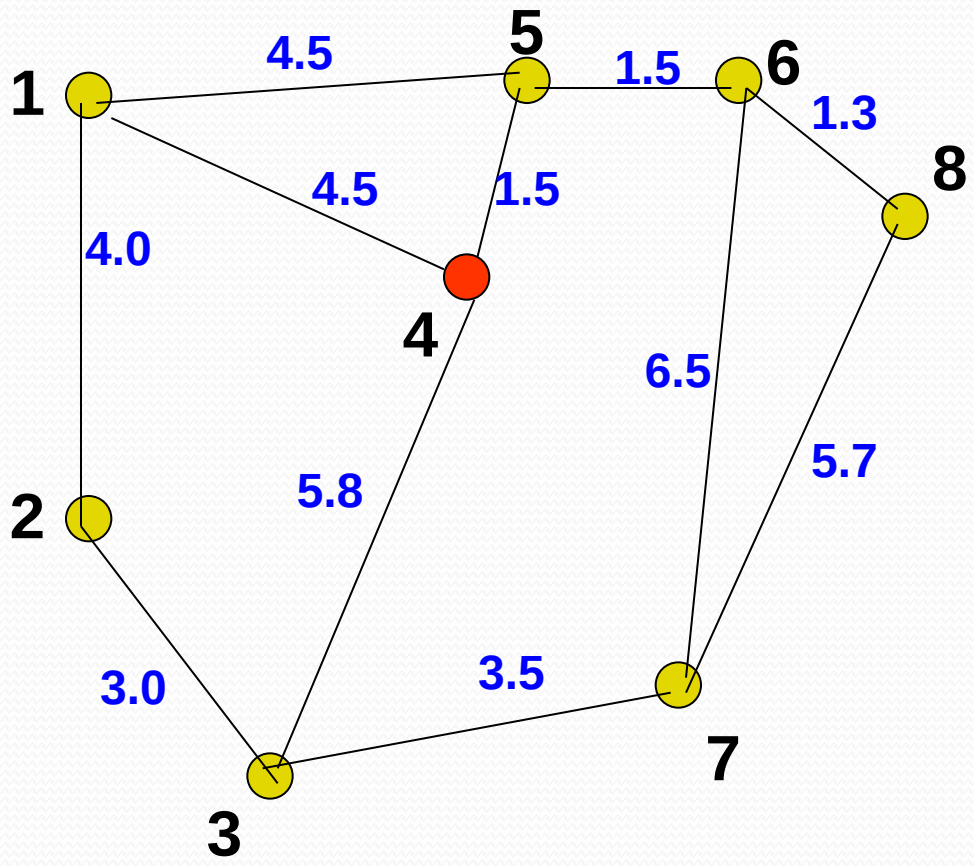
4

5(1.5 + 7.5?=9?)

1(4.5 + 9.5?=14?)

3(5.8 + 3.5?=9.3?)

A*



- 5 may be closest. Add 6 to frontier. No updates.

1(4.5 + 9.5?=14?)

3(5.8 + 3.5?=9.3?)

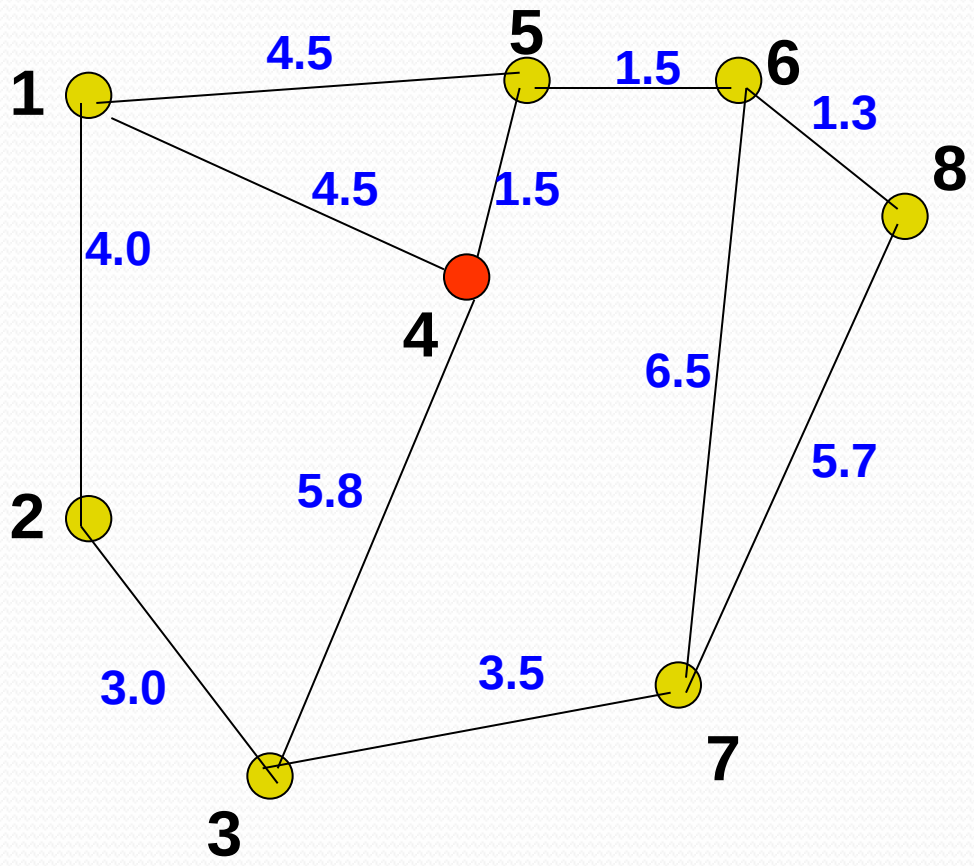
6(3.0 + 6.5?=9.5?)

4

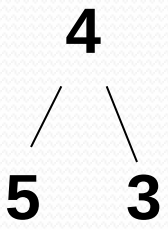
/

5

A*



- 3 looks closest. Add 2 and 7 to frontier. No updates.



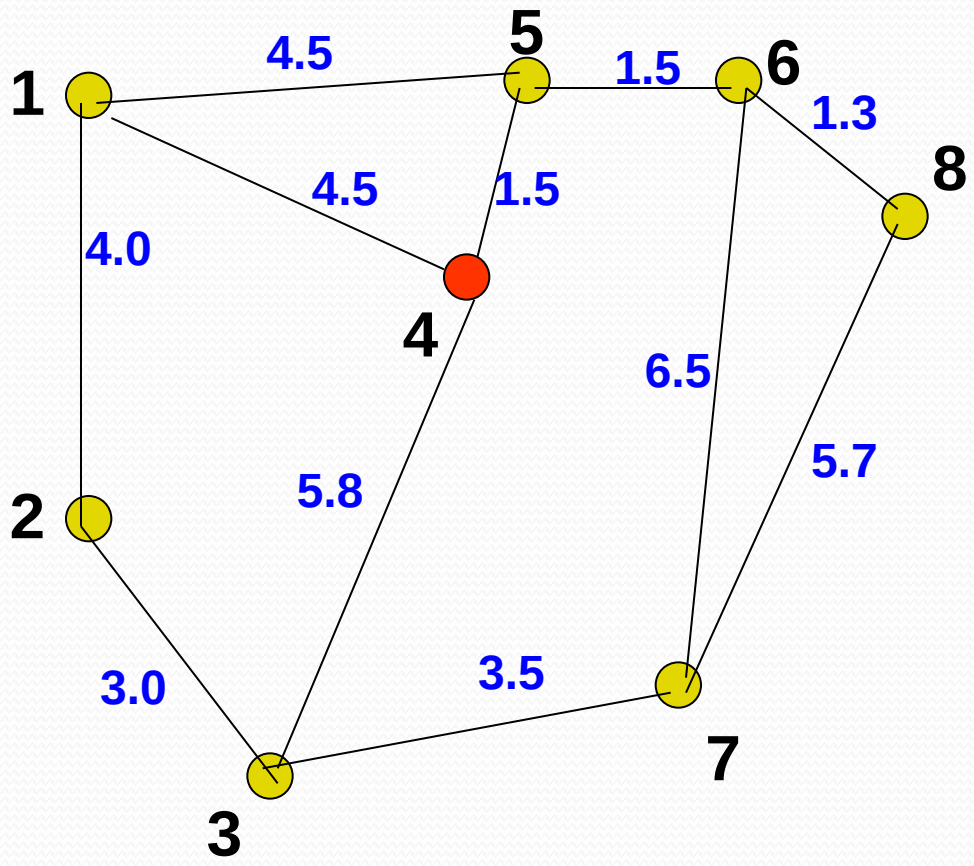
1(4.5 + 9.5?=14?)

6(3.0 + 6.5?=9.5?)

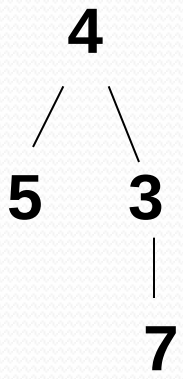
2(8.8 + 7.0?=15.8?)

7(9.3 + 0.0=9.3)

A*



- 7 is closest. Job done.



1(4.5 + 9.5?)

6(3.0 + 6.5?)

2(8.8 + 7.0?)

A*

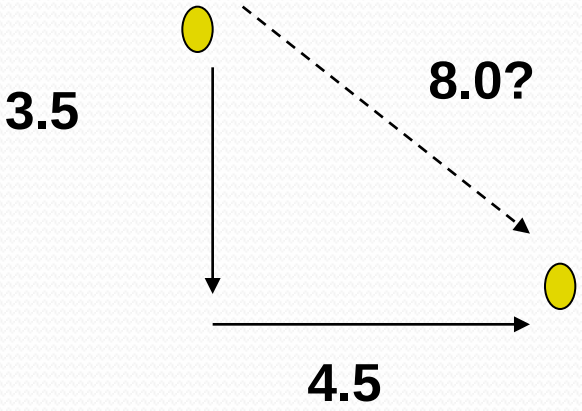
- Note that the algorithm is not finished when it finds the end point.
  - The end point must be the node in the Frontier with the smallest (g + h)
  - There may be shorter routes that we have not yet found (remember edge relaxation)

A*

- Many (most) explanations of A* refer to:
  - The “Closed List” – the list of nodes on the tree.
  - The “Open List” – the list of nodes in the Frontier.

A*

- To avoid a square root, programmers often use a “Manhattan distance” as the heuristic.



A*

- This is often an overestimate, but still gives very good paths, very fast.
- Is not an overestimate for some grid-based games.

Implementing A*

- The STL library does have structures that can manage trees, but I usually find it less confusing to just do it myself.
- A “node” in an A* search is much like a Graph node, but also needs variables to store f, g and h, plus pointers to attach to other nodes in the tree and linked list.

Implementing A*

```
class Node
{
  int indexNumber;
  list<Edge> edgeList;
  Vector2D position;
  double f,g,h;
  Node* parent;
  list<Node*> children;
};
```

- Subclass, or just put in the original?

Implementing A*

```
list<Node*> openList;
list<Node*> closedList;
Node* CurrentPtr;
Node* NextPtr;
```

Implementing A*

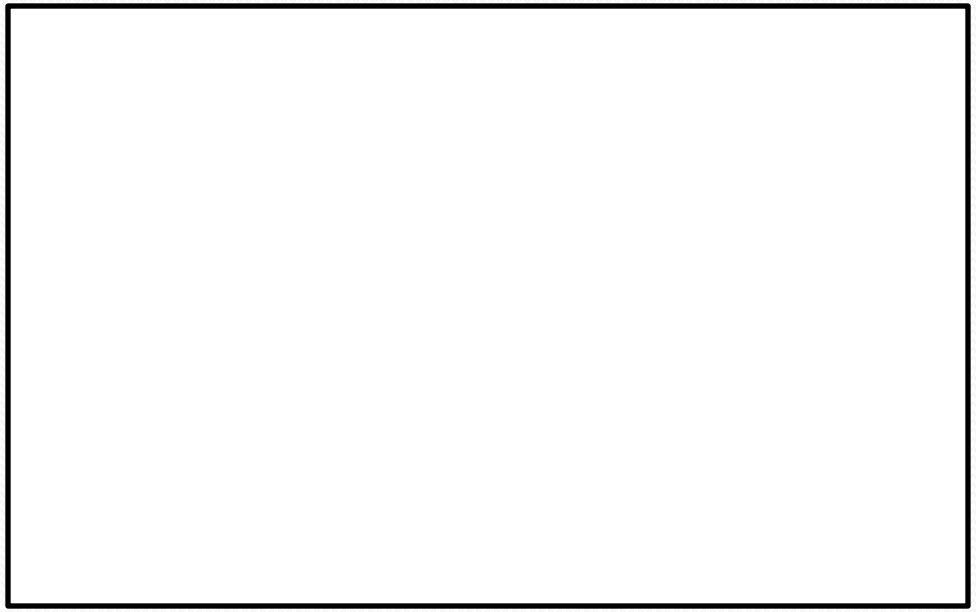
```
add the starting node to the open list
while open list is not empty...
        current node = node from open list with lowest cost
           if current node = goal node then path complete
         else
            move current node to the closed list
            examine each adjacent node to the current node
            for each adjacent node
                if it isn't on the open list
                and it isn't on the closed list
                and it isn't an obstacle then
                    move it to open list and calculate cost
                    set its parent pointer

return a (STL) list of the nodes in the path
```

Implementing A*

- Once you find the end point, you can make a list of waypoints by following the parent pointers backwards.
  - (Probably as a list<int> of Node indices)
  - Return this to the ‘Bot.

Implementing A*

- What is the best way to implement the open and closed lists?
- What operations do we need?

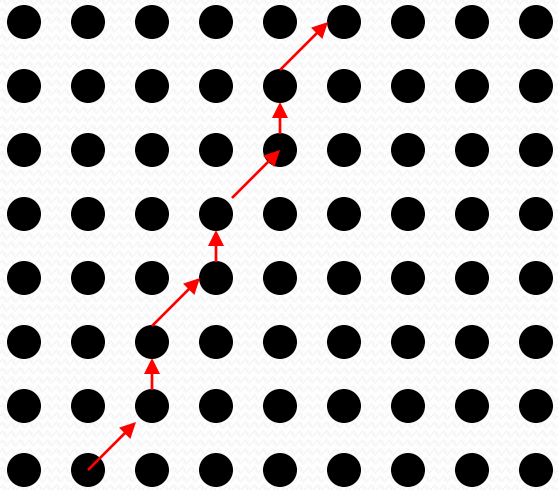


Implementing A*

- Sometimes you don't want to go to a particular point, but to several possible points. (E.g. you want to find the closest health pickup.)
- Don't want to repeat A* for each possible node.
- In this case, use a heuristic of zero.
  - A* now becomes Dijkstra's algorithm.
  - End algorithm when reach any of the target nodes.
  - (Often easiest to add extra data to nodes at the start of the game. E.g. "I am the node closest to a health pack")

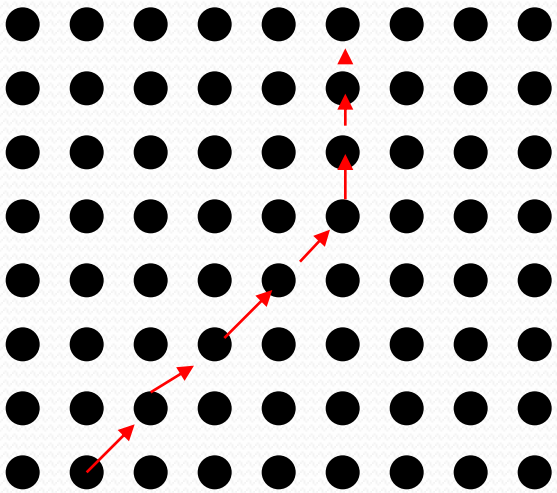
Smoothing

- The paths produced by A* in a grid will often zigzag.



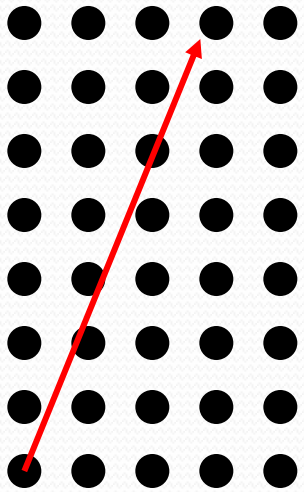
Smoothing

- One way to avoid this is to add a cost penalty whenever the new edge has a different direction to the previous edge.



Smoothing

- This still has a bit of a zig, but not as bad.
- Problem is that comparing “directions” can be slow.
- What we really want is to skip unneeded waypoints.



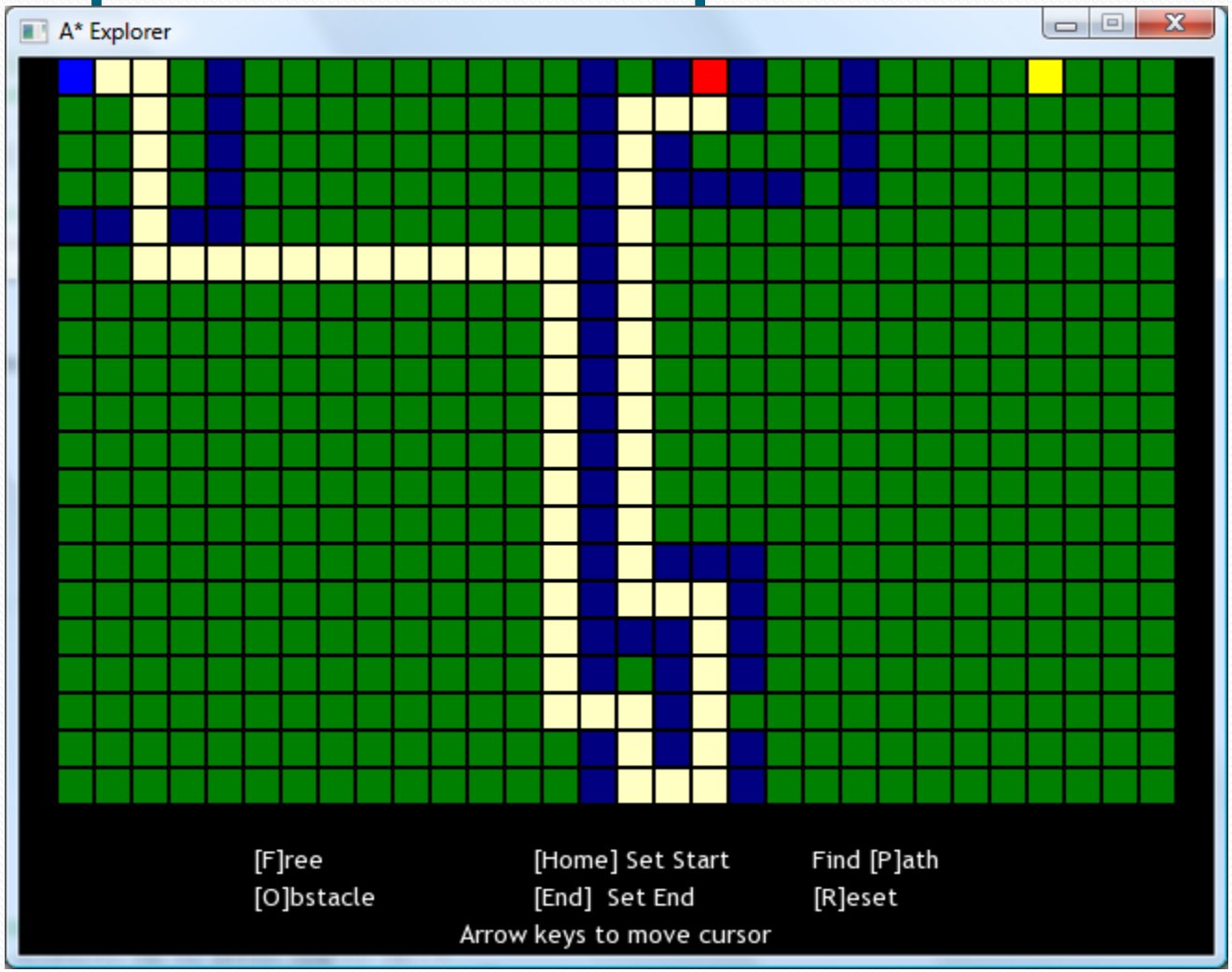
Smoothing

- A quick way to do this is to run through the waypoint list and see if the next node but one is walkable from the current node. If it is, skip the intervening node, and check the next one.
- Continue until you find a node you can’t walk to. Advance to that node and repeat.
  - Not perfect, but good 90% of the time and fast.
  - See Buckland for a better algorithm.

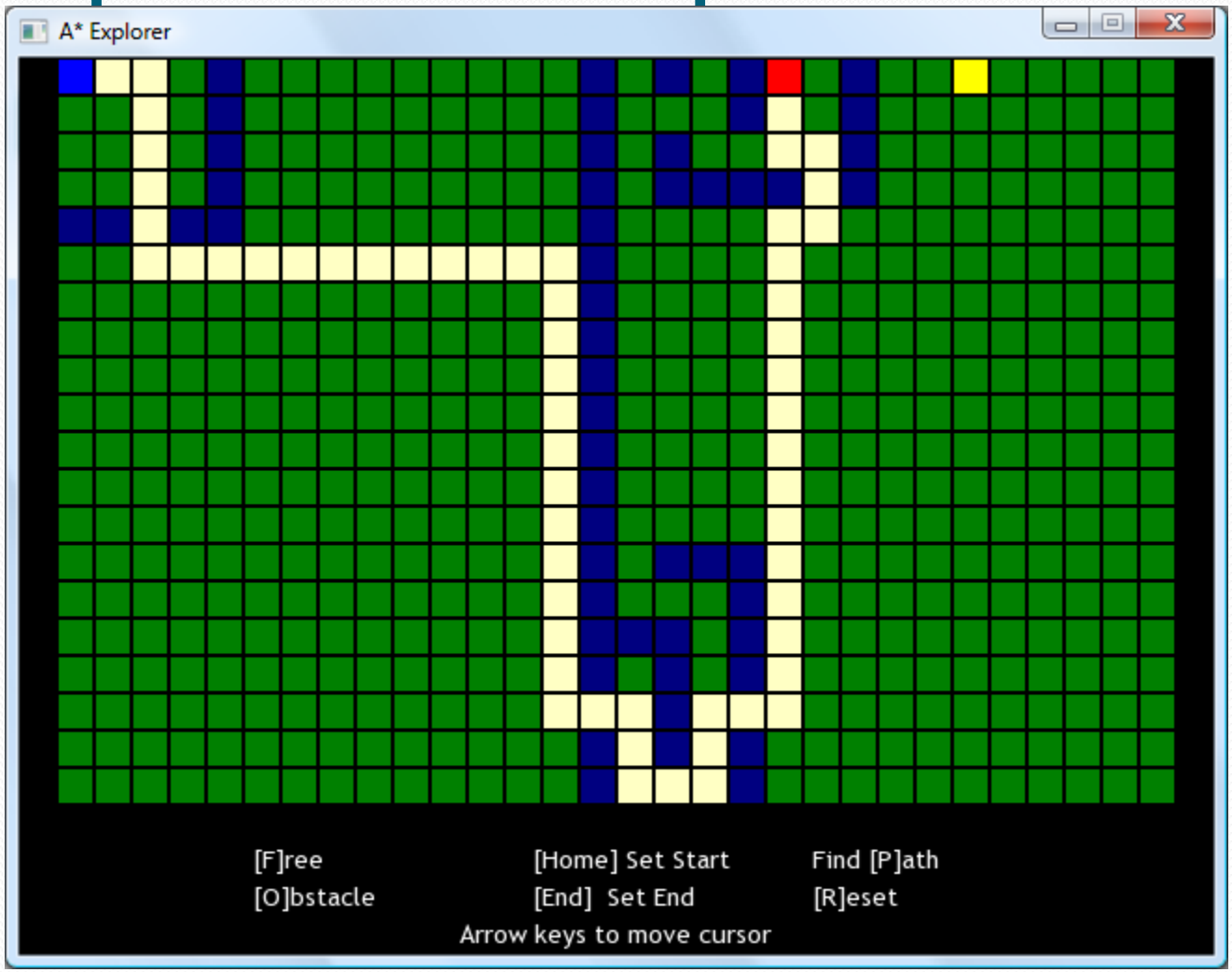
Following a path

- Once a path has been generated, it's easy to get your 'bot to follow it.
- But:
  - Don't follow to the end. Once you can see the target, just head for it directly.
  - Check that the 'bot is actually making progress. If not, it may be stuck somewhere.
  - Re-evaluate the tactical situation to see if the objective is no longer relevant (or has a bigger gun.)

Example: AStarExplorer



Example: AStarExplorer



Summary

- Graphs
- Storing a sparse graph
- Dijkstra's algorithm
  - Optimality
- A*
  - Heuristics
  - Optimality
  - Open and closed lists
  - Pseudocode
  - Manhattan distances
- Path smoothing
- Following the waypoints