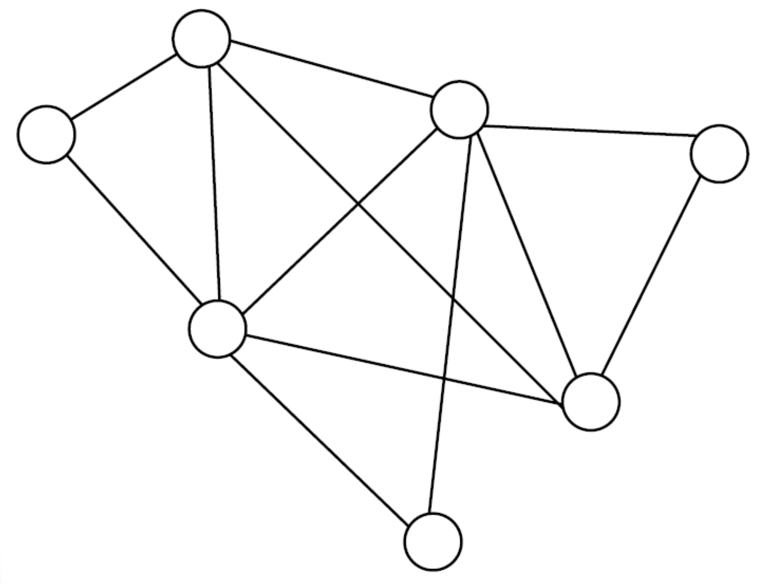
Pathfinding

Overview

- Graphs
- Pathfinding algorithms
  - Dijkstra
  - A*
- Pathfinding networks
- Advance techniques
  - Group path-finding
  - Hierarchical path-finding

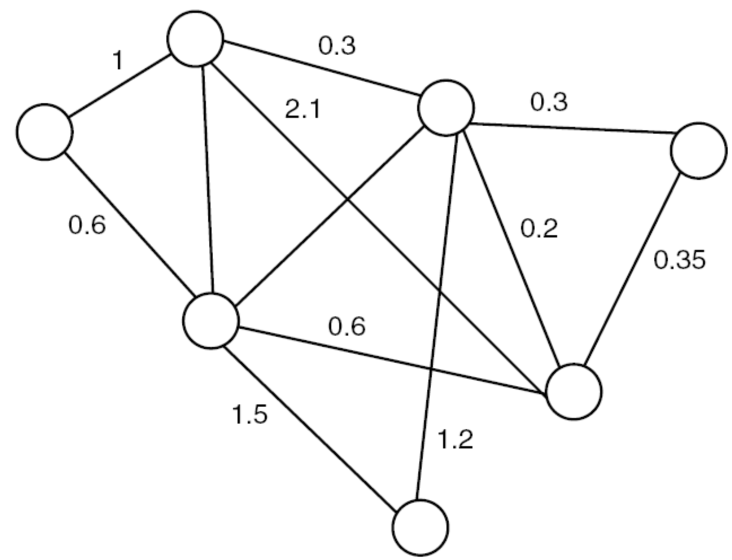
The Pathfinding Graph

- A graph consists of
  - ➢ Nodes
  - ➢ Connections
    - An unordered pair of nodes
- Pathfinding algorithms use
  - ➢ Non-negative weighted graph
  - ➢ A path consists of
    - Zero or more connections



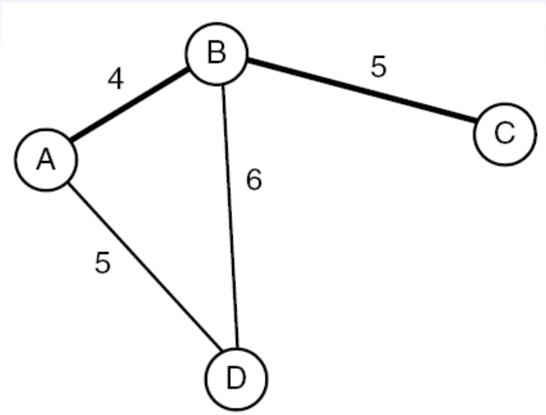
The Pathfinding Graph

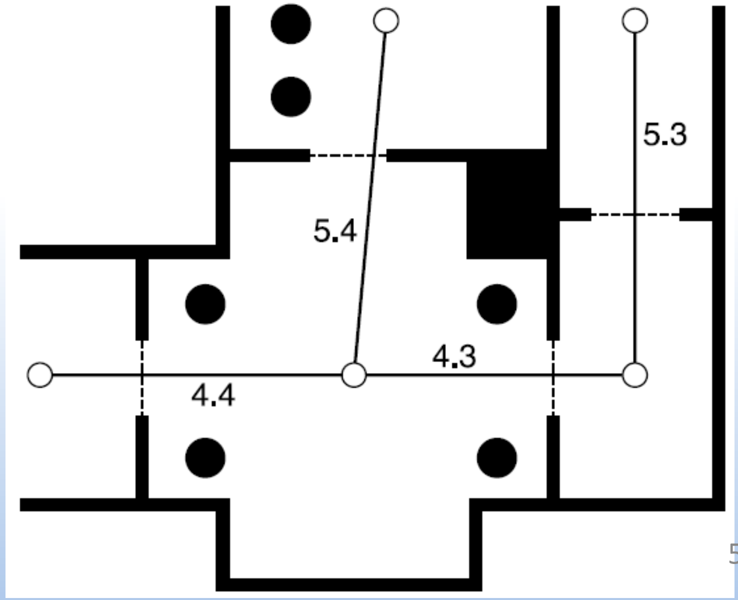
- Weighted graphs
  - ➢ Connections have weights/costs
- Costs
  - ➢ In games often represent
    - Time or distance
    - Can also be other factors
  - ➢ Non-negative



The Pathfinding Graph

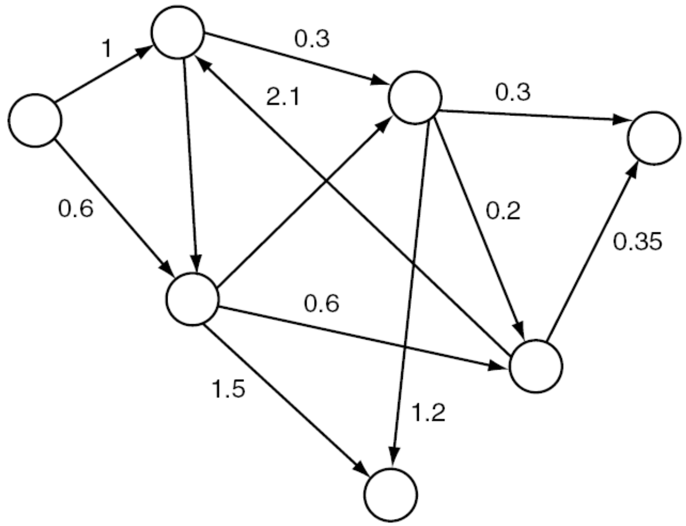
- Total path cost
  - ➢ Sum connection costs
- Nodes are representative points
  - ➢ Can be centre of a region
  - ➢ Or some other position





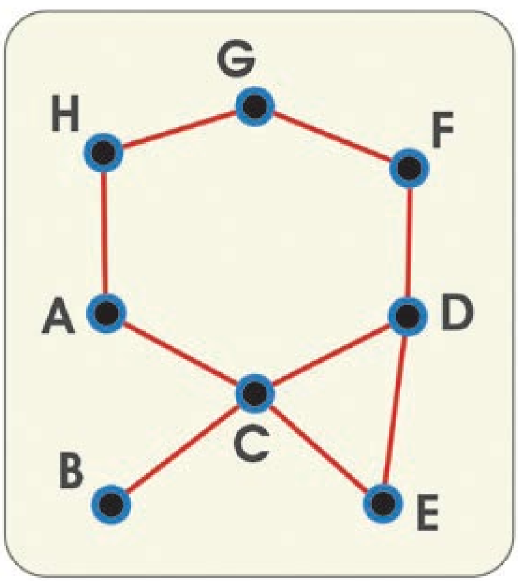
The Pathfinding Graph

- Directed weighted graphs
  - Connections and costs in one direction only
    - If can return, considered as two connections
  - Connections now ordered
    - E.g., from node 1 to node 2



The Pathfinding Graph

- Representation
  - Need to represent graph for pathfinding algorithms
  - Lookup table
    - Easy to implement but impractical for applications with a large number of nodes



From \ To	A	B	C	D	E	F	G	H
A	X	C	C	C	C	H	H	H
B	C	X	C	C	C	C	C	C
C	A	B	X	D	E	D	D	A
D	C	C	C	X	E	F	F	F
E	C	C	C	D	X	D	D	C
F	G	D	D	D	D	X	G	G
G	H	H	H	F	F	F	X	H
H	A	A	A	G	A	G	G	X

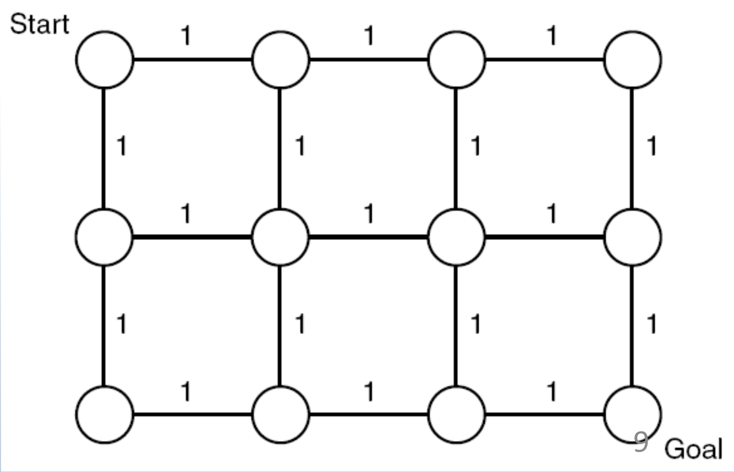
The Pathfinding Graph

- Representation
  - Algorithms need to find outgoing connections from any given node

```
class Connection:
  // Returns the non-negative cost of the connection
  def getCost()

  // Returns the node that this connection came from
  def getFromNode()

  // Returns the node that this connection leads to
  def getToNode()

class Graph:
  // Returns an array of connections (of class Connection)
     outgoing from the given node
  def getConnections(fromNode)
```

Dijkstra

- Dijkstra's algorithm
  - Designed to solve the 'shortest path' problem
  - Simpler version of the A* algorithm
- Problem statement
  - Given a directed non-negative weighted graph, start and goal nodes, generate a path such that the total path cost is minimal among all possible paths
  - May be more than one with same minimal cost
    - If so, any one will do

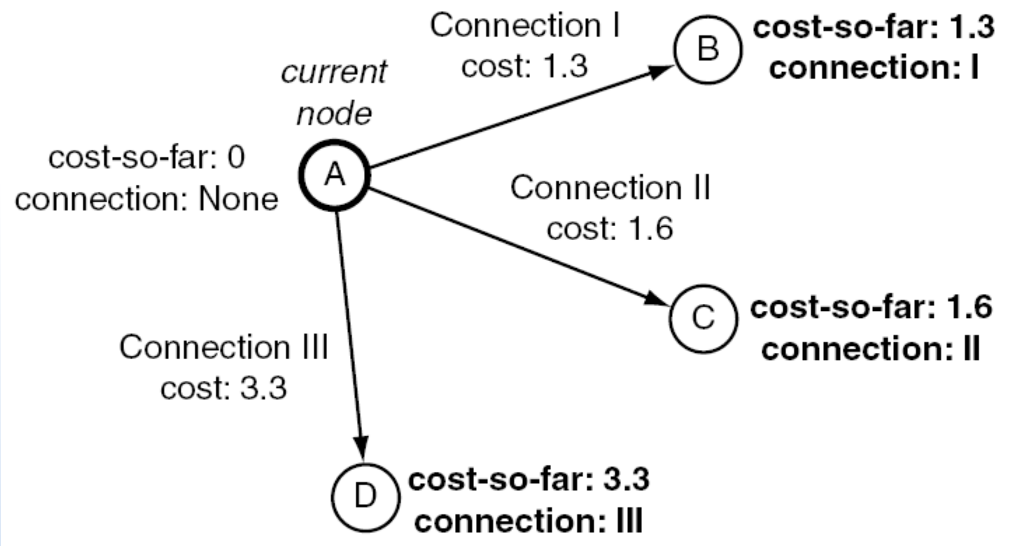


Dijkstra

- In general, the path returned
  - Consists of a set of connections, not nodes
    - Two nodes may be linked by more than one connection
    - Each connection may have a different cost
- The algorithm
  - Works in iterations
    - Each iteration, considers one node (i.e., the "**current node**") and follows outgoing connections
    - First iteration, considers the **start node**

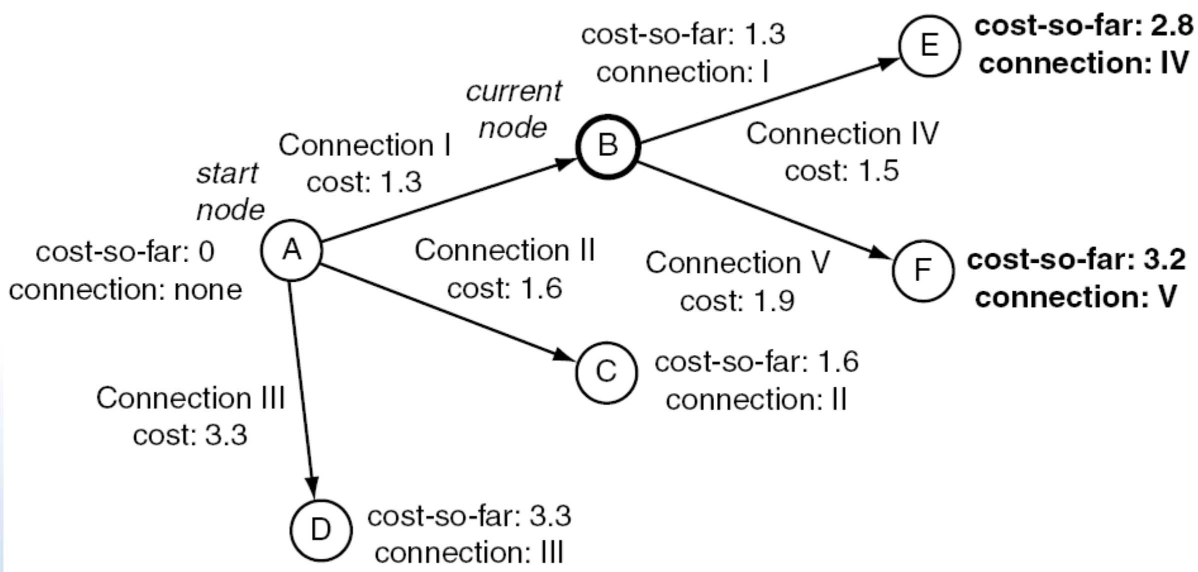
Dijkstra

- Processing the current node
  - Trace all the outgoing connections from the current node
  - For each connection, find the end node and store
    1. Total cost of path thus far ('**cost-so-far**'), and
    2. The **connection** it arrived there from
  - Need to record the cost-so-far and connection for each node



Dijkstra

- For iterations after the first
  - Cost-so-far for the end node of each connection, is the
  - Sum of **connection cost + cost-so-far** of the current node

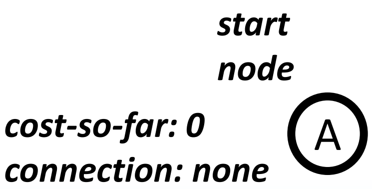


Dijkstra

- Node lists
  - Keeps track of all nodes seen so far in two lists, “open” list and “closed” list
  - Can think of nodes in three categories
    - Closed: has been **processed in its own iteration**
    - Open: all nodes **seen but not processed**
    - Unvisited (in neither list)
  - Each iteration
    - Choose a node from the open list with the **smallest cost-so-far**
      - Set it as the current node
    - Process the current node
      - Once processed, remove the current node from the open list and place it in the closed list

Dijkstra

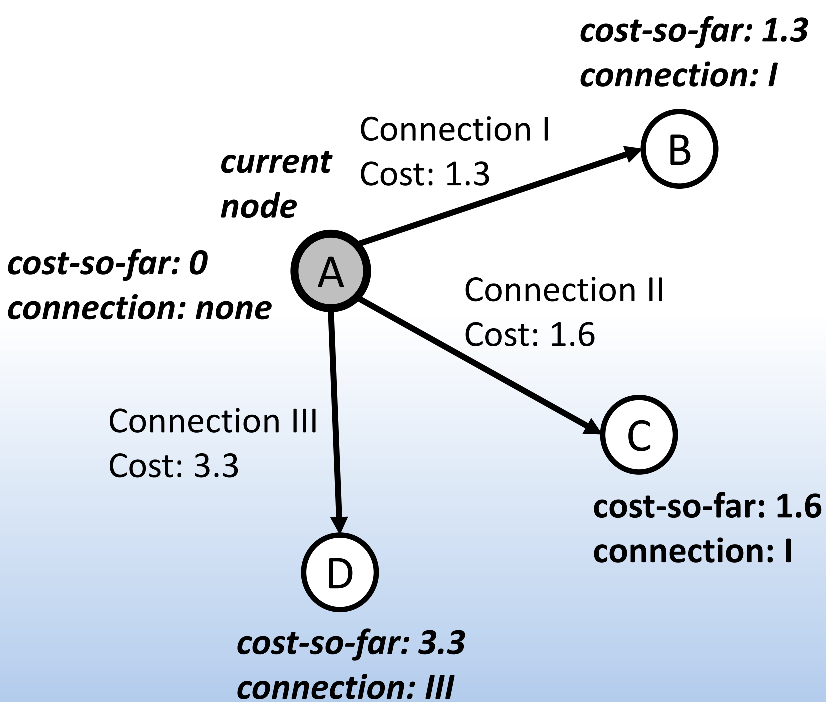
- Example
  - At the start, only the start node will be on the open list
  - For the first iteration, select the start node as the current node



Open List	Closed List
A	-

Dijkstra

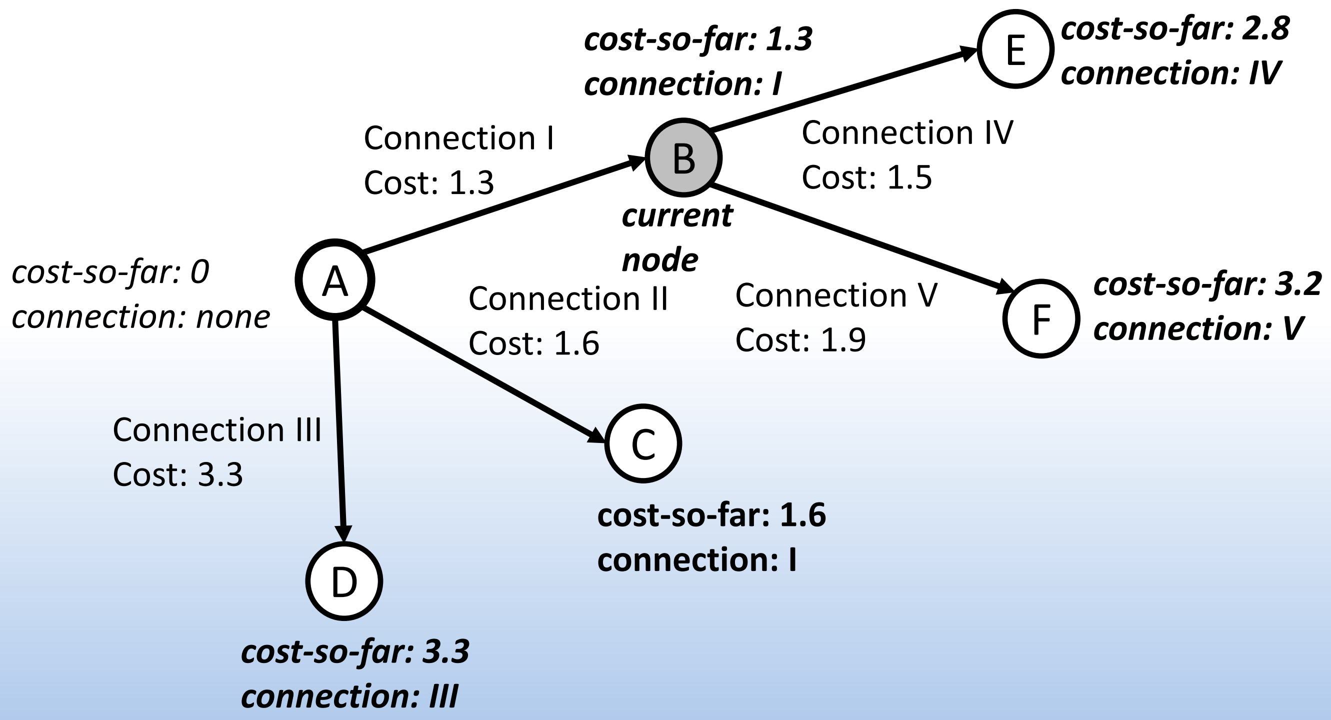
- Process the current node
  - Trace outgoing connections, find their end nodes, put these on the open list and update their node records
  - Once complete, remove the current node from the open list and put it in the closed list



Open List	Closed List
B, C, D	**A**

Dijkstra

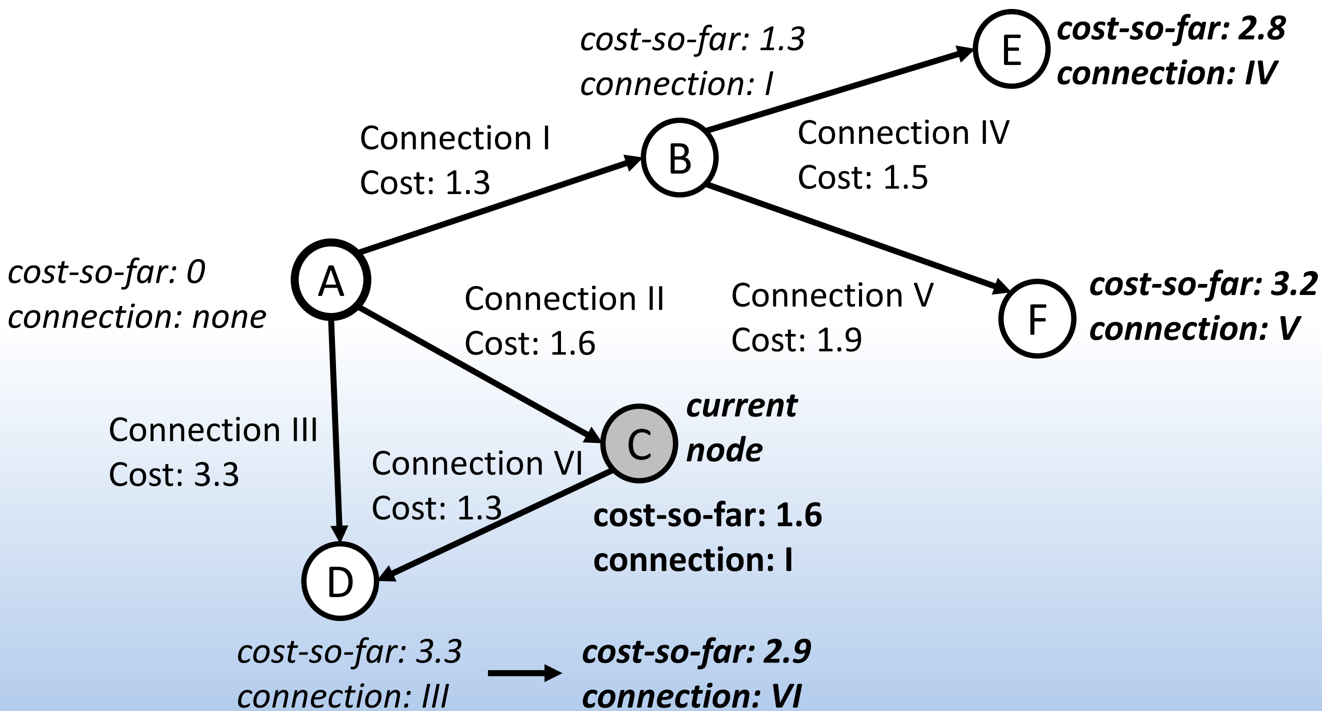
- Next iteration
  - Select the node with the lowest cost-so-far from the open list to the the current node
  - Process the current node



Open List	Closed List
C, D, E, F	A, **B**

Dijkstra

- Potential complication: end node already has a node record
  - Check for a better route
    - » If the cost-so-far of the current route is higher than the recorded value, don't do anything
    - » Otherwise, update the node records and put the node back onto the open list (if it was on the closed list)



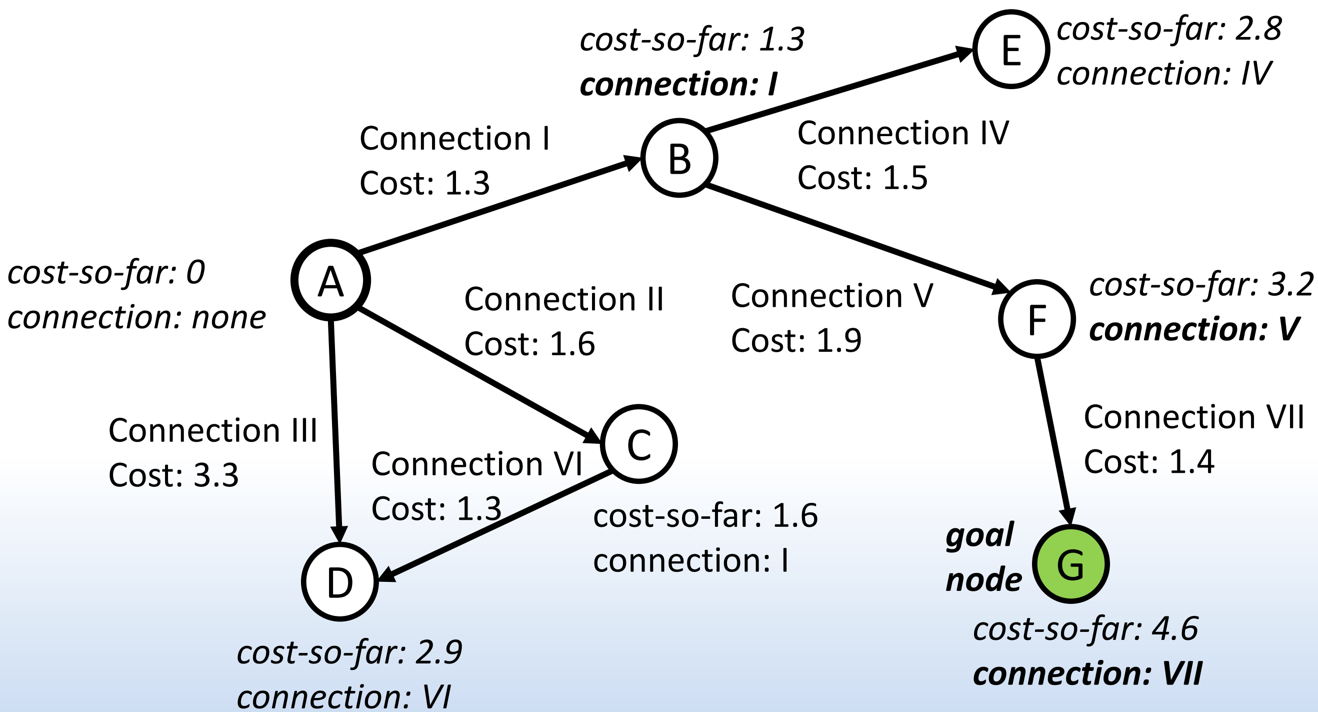
Open List	Closed List
D, E, F	A, B, **C**

Dijkstra

- Terminating the algorithm
  - The basic algorithm terminates when the open list is empty
  - In pathfinding
    - Only interested in reaching goal node, may stop earlier
    - Can terminate when the goal node has the smallest cost-so-far on the open list
  - Why don’t we stop as soon as the goal node is found?
    - No guarantee that this is the shortest path

Dijkstra

- Retrieving the path
  - Start at the goal node, work backward through the connections until the start node, then reverse the order

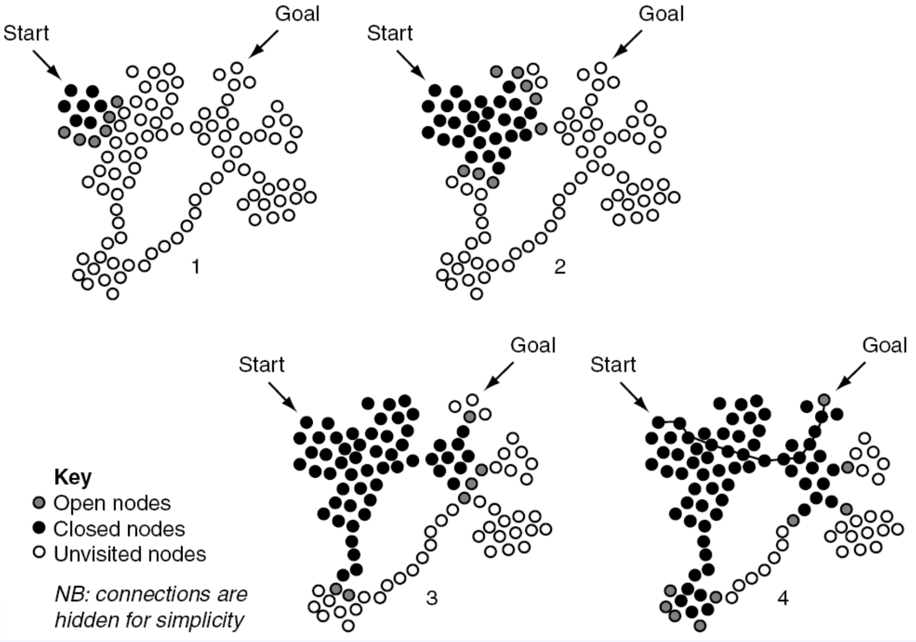


Connections working back from the goal: **VII, V, I**

Final path: **I, V, VII**

Dijkstra

- **Weaknesses**
  - Designed to search the whole graph indiscriminately
    - To work out shortest possible route
  - Inefficient for ‘point-to-point’ path-finding
    - In general, want to consider as few nodes as possible
    - Number of nodes considered but not explored, called the ‘fill’ of the algorithm



A* Pathfinding

- A* algorithm
  - Designed for point-to-point path-finding and not to solve shortest path problem
  - Works very similar to Dijkstra
    - But rather than always considering node on open list with the lowest cost-so-far
    - Choose the node ‘most likely’ to lead to the shortest overall path
    - Notion of ‘most likely’ controlled by a **heuristic**
      - If heuristic accurate, efficient
      - If heuristic terrible, can perform worse than Dijkstra
      - If heuristic is all 0, then exactly the same as Dijkstra

A* Pathfinding

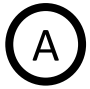
- The algorithm
  - ➢ Like Dijkstra, works in iterations
  - ➢ Processing current node
    - Difference from Dijkstra? Store another value called the "**estimated-total-cost**"
      - – Estimate of total cost for a path from the start node, through the current node, onto the goal
      - – Estimated-total-cost = **cost-so-far + heuristic**
      - – The heuristic is an estimate of the cost from that node to the goal
        - » Cannot be negative
      - – This estimate is not part of the algorithm and is generated by a separate piece of code

A* Pathfinding

- Example
  - At the start, only the start node will be on the open list
  - For the first iteration, select the start node as the current node

heuristic: 4.2
cost-so-far: 0
connection: none
estimated-total-cost: 4.2

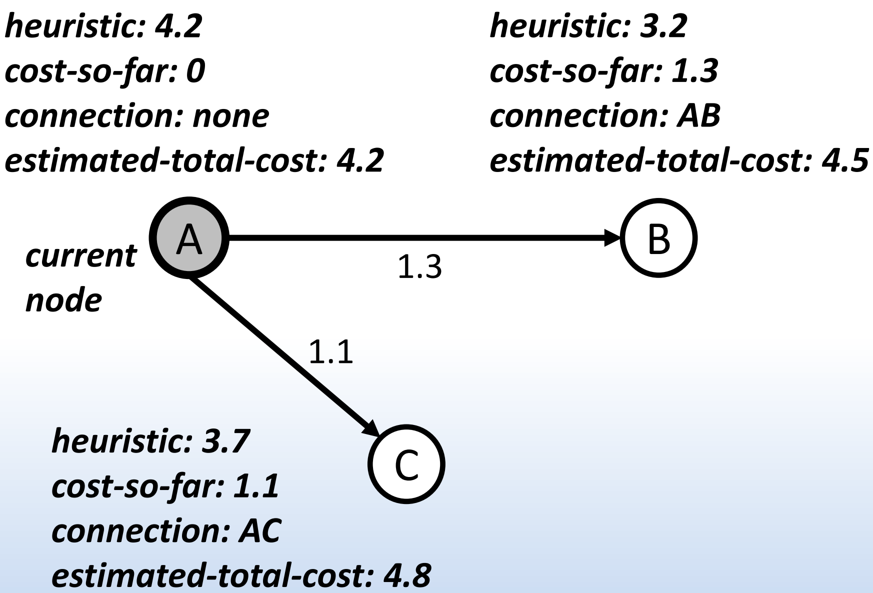
start node



Open List	Closed List
A	-

A* Pathfinding

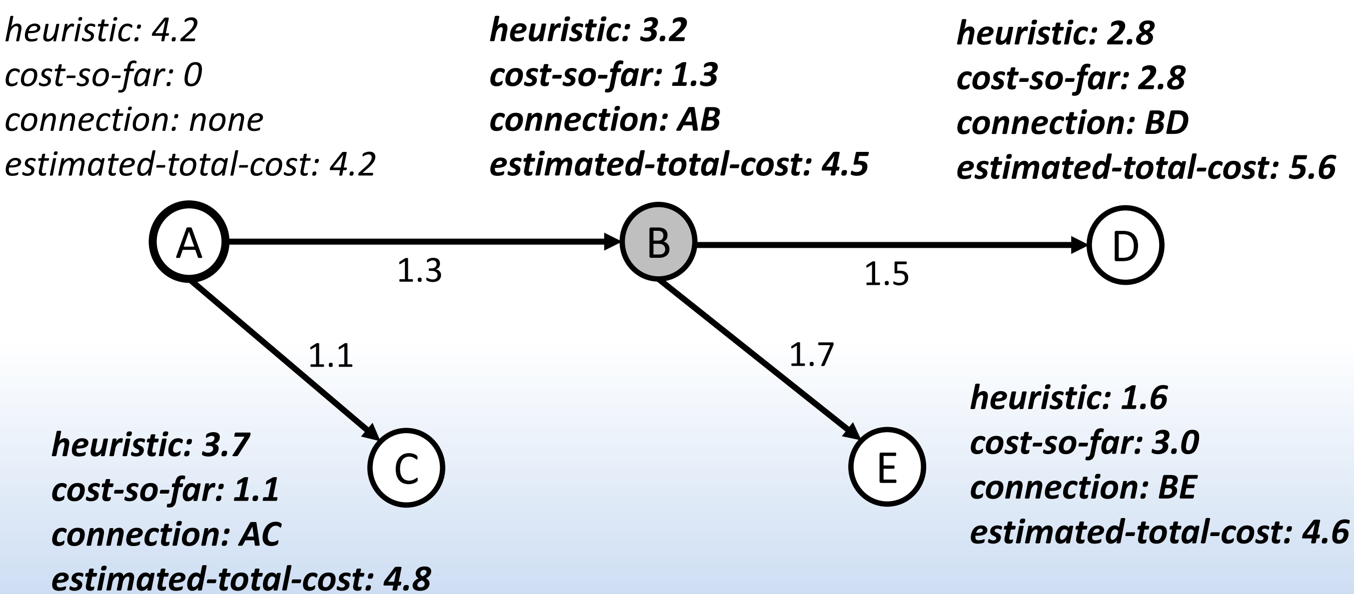
- Process the current node
  - Trace outgoing connections, find their end nodes, put these on the open list and update their node records
  - Once complete, remove the current node from the open list and put it in the closed list



Open List	Closed List
B, C	**A**

A* Pathfinding

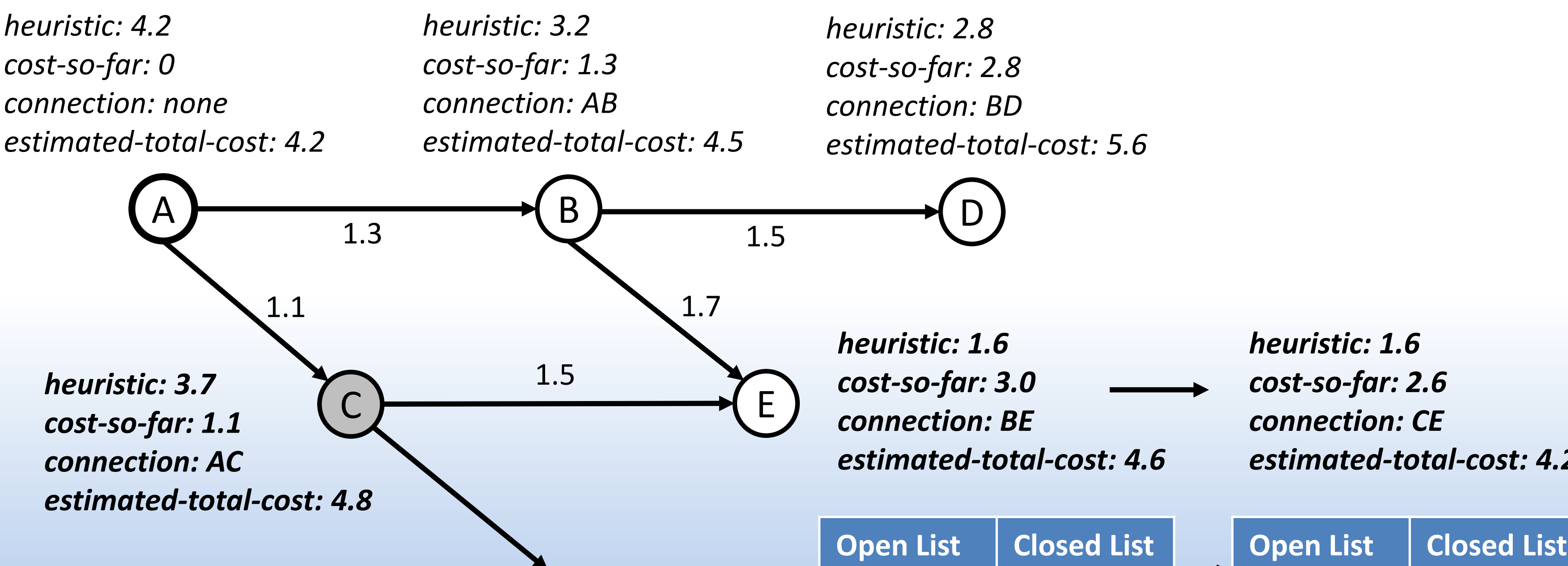
- Difference with Dijkstra
  - For the current node, select the node on the open list with **smallest estimated-total-cost**
  - **NOT** smallest cost-so-far



Open List	Closed List
C, D, E	A, **B**

A* Pathfinding

- Complication and solution
  - Compare the cost-so-far (the only reliable value)
    - » NOT the estimated-total-cost (only and estimated cost)
  - If node was on the closed list, put it back on the open list



Open List	Closed List
C, D	A, B, E

→

Open List	Closed List
D, **E**	A, B, **C**

A* Pathfinding

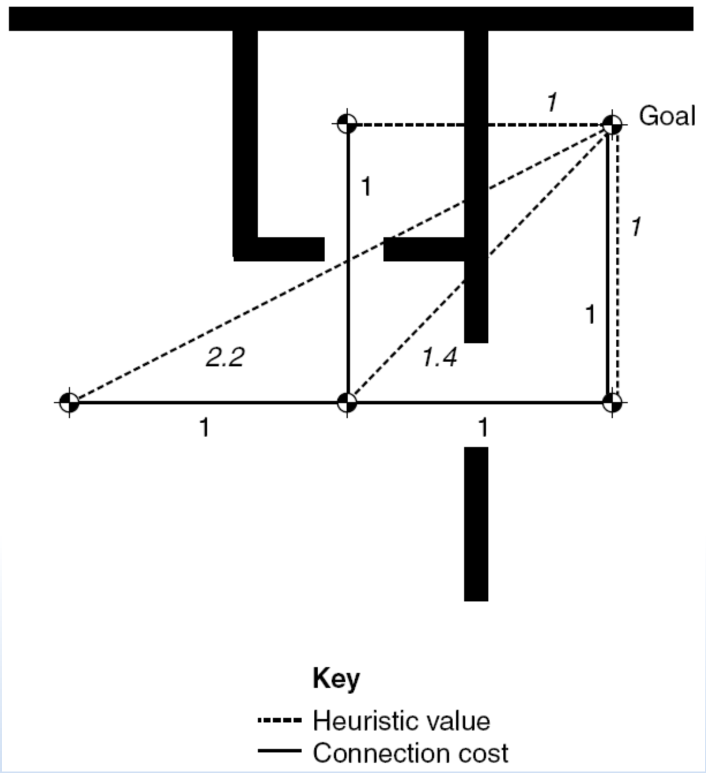
- Terminating the algorithm
  - When the goal node has the smallest value on the open list
  - A* completely relies on the fact that can theoretically produce non-optimal results
    - Depends on the choice of heuristic function
    - Many implementations just terminate upon finding the goal
- Retrieving the path
  - Start a goal node, work back until the start node (same as Dijkstra)

A* Pathfinding

- Heuristic function
  - ➢ Inconvenient to produce a different heuristic function for each possible goal
    - Often parameterised by the goal node
  - ➢ Might involve some algorithmic process
    - If the process is complex, time spent evaluating heuristics might dominate the pathfinding algorithm
  - ➢ For non-perfect heuristics
    - A* behaves slightly differently depending on whether underestimating or overestimating

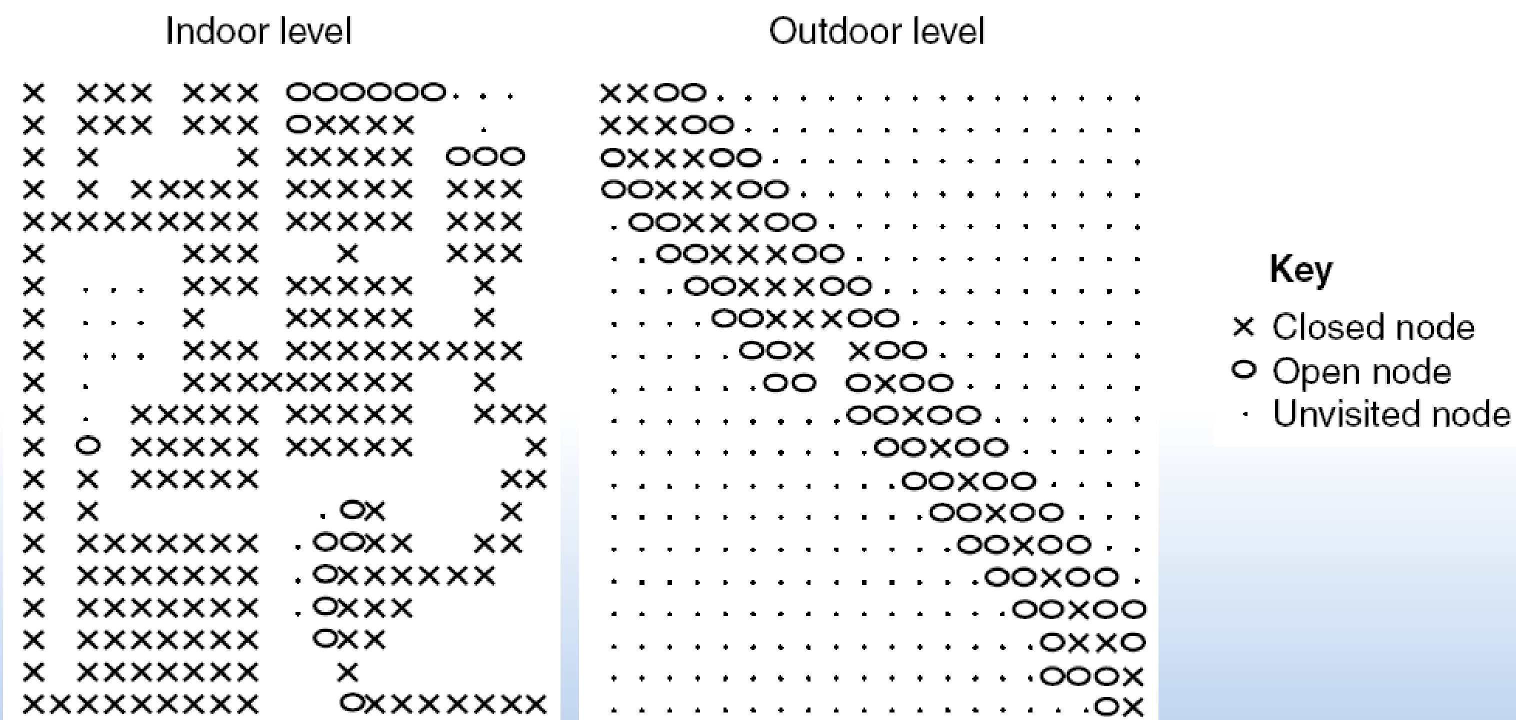
A* Pathfinding

- Common heuristics in games
  - ➢ Euclidean distance
    - If connection cost represents distance
    - Distance "as the crow flies", measured between two points through obstructions
    - Either accurate or guaranteed to underestimate
    - In outdoor settings with few constraints, can be very fast and accurate



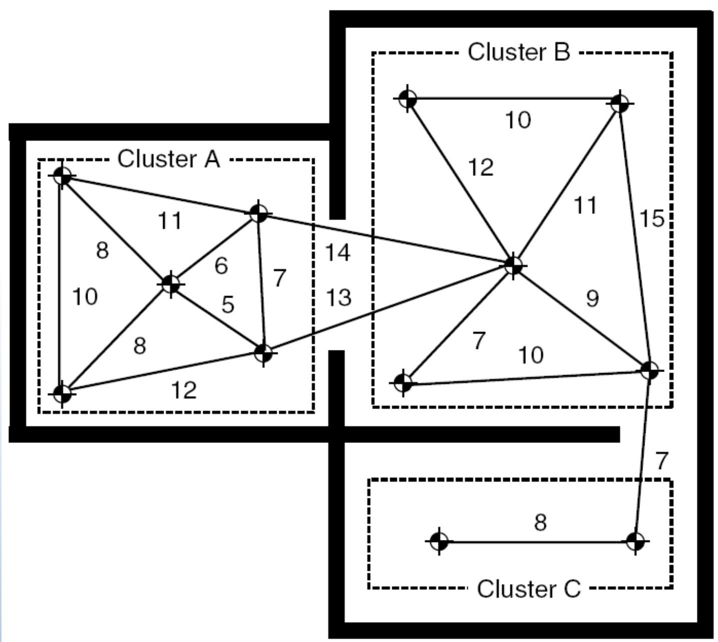
A* Pathfinding

- Euclidean distance
  - Dramatically underestimates in indoor settings
    - Might cause less than optimal pathfinding



A* Pathfinding

- Cluster heuristic
  - Group nodes together in clusters
  - Nodes in clusters represent highly interconnected region
  - Often done manually
  - Lookup table gives smallest path length



	A	B	C
A	x	13	29
B	13	x	7
C	29	7	x

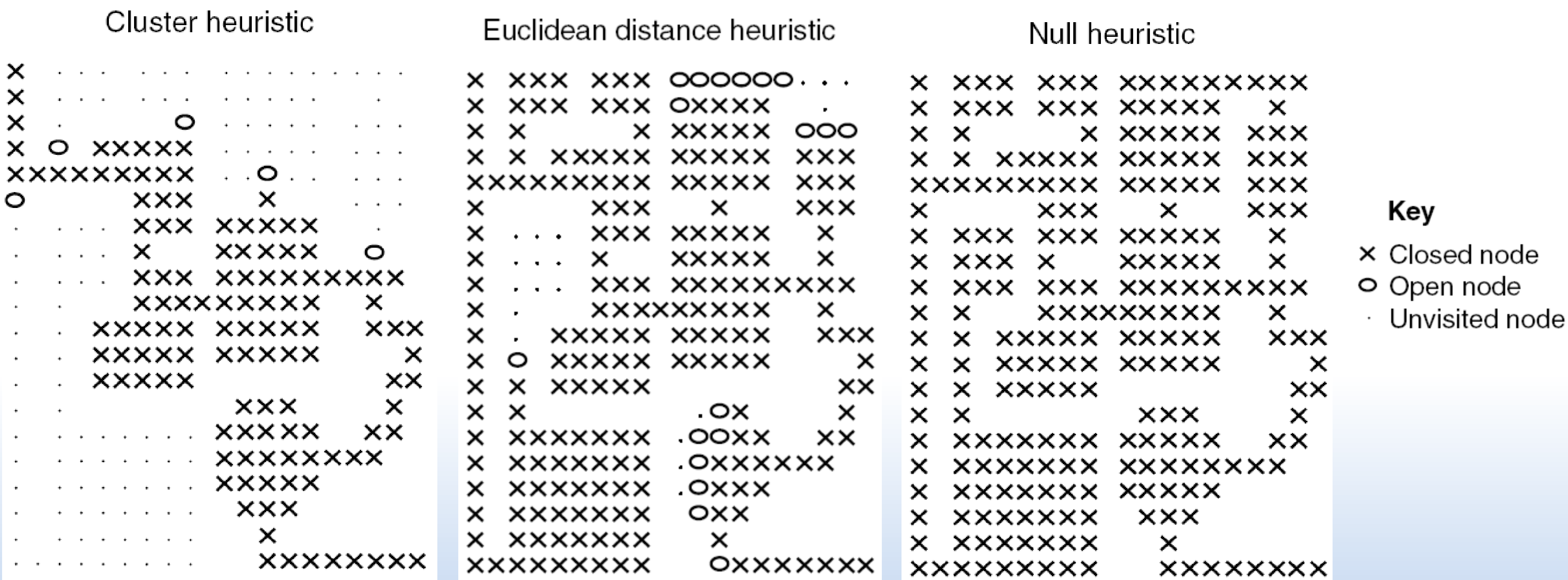
Lookup table

A* Pathfinding

- Often dramatically improves performance in indoor areas
  - As it takes convoluted routes into account
- However, all nodes in a cluster give the same heuristic value
  - A* cannot easily find the best route through a cluster
- A cluster tends to be almost completely filled before the algorithm moves to the next cluster
- Tradeoff
  - If a cluster is small, the accuracy can be excellent
    - » But the lookup table can be large
  - If too large, marginal performance gain
    - » Simpler heuristic better

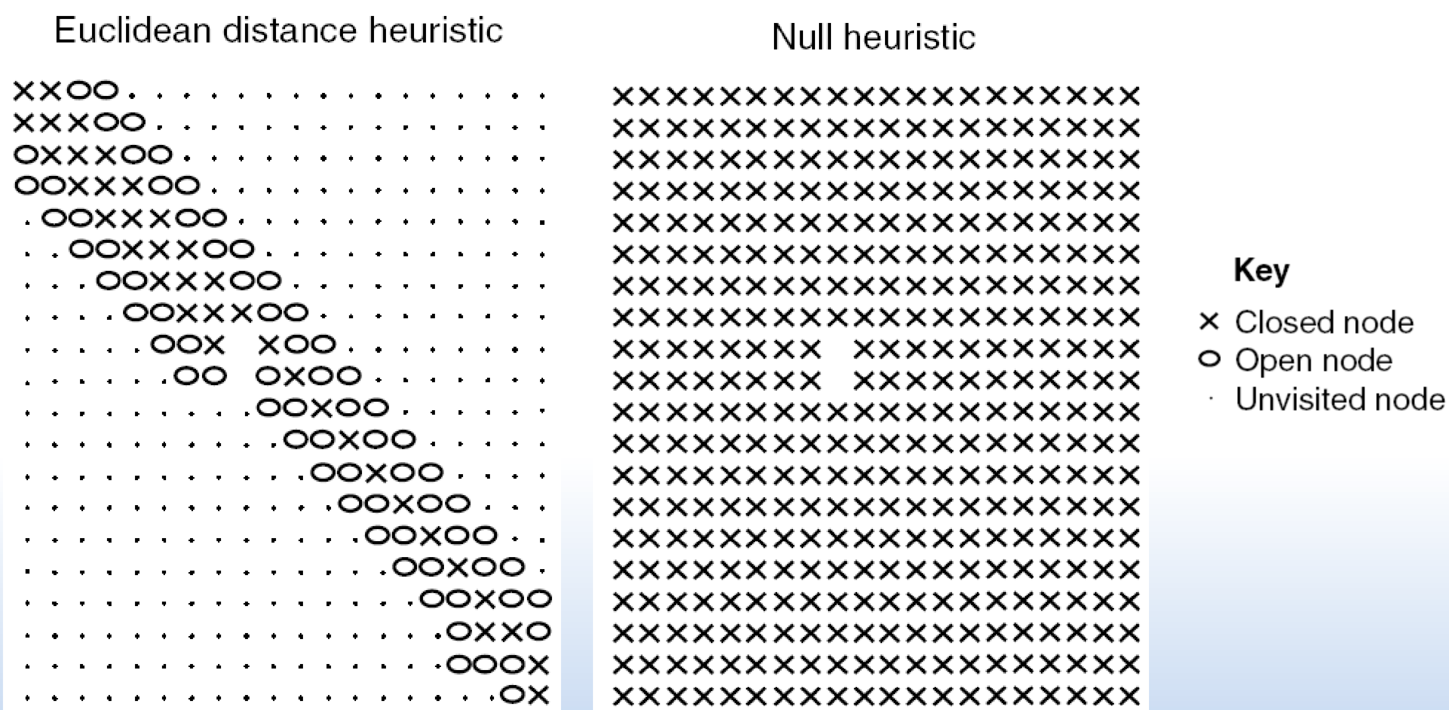
A* Pathfinding

- Cluster heuristic
  - Fill visualisation for an indoor setting



A* Pathfinding

- Cluster heuristic
  - Fill visualisation for an outdoor setting

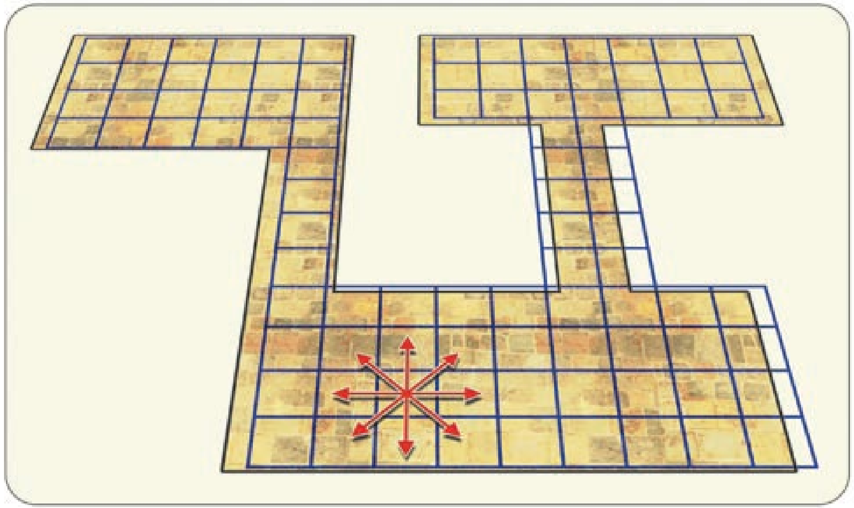


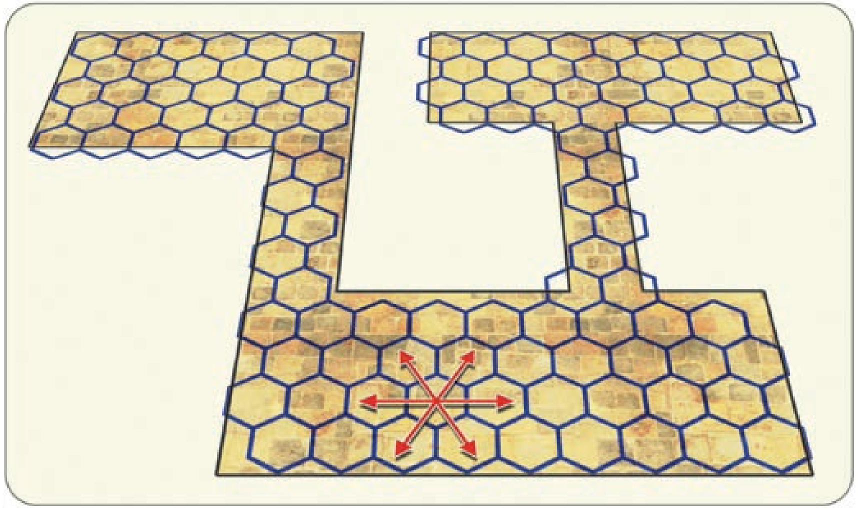
Pathfinding Networks

- Pathfinding network
  - Search a network of connected nodes
  - Network creation can be time-consuming
    - Generated before path-finding operations
  - Can be done manually, or generated automatically, or a mixture of both
  - Most games only use 2D network and most of these networks do not change during the game

Pathfinding Networks

- Rectangular grids (also known as tiles)
  - Can be square or hexagonal
  - Can be used to automatically generate path-finding networks
  - Easy to generate algorithmically and determine connectivity
  - Easy to implement
    - Can use a simple 2D array



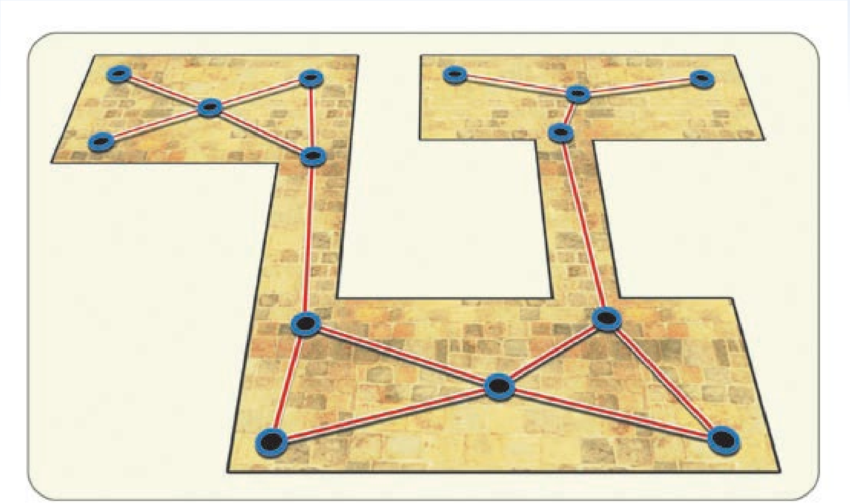


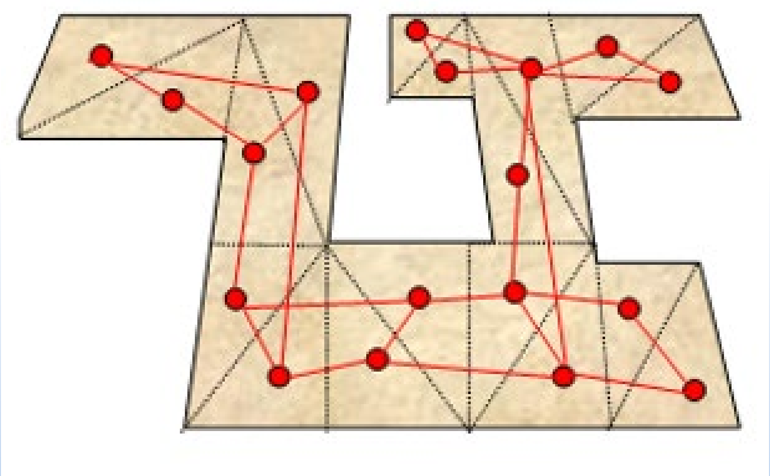
Pathfinding Networks

- Hex grids
  - Slightly harder to implement
  - Advantage
    - » Six adjacent nodes all equidistant from the current node centre
    - » Movement tends to be more regular

Pathfinding Networks

- Arbitrary grids
  - Nodes only placed at important locations
  - Usually done manually
  - Advantages
    - Nodes can be placed exactly along the desired movement for path following
    - Needs to store less data
  - Disadvantage
    - Manual work required
- Navigation mesh
  - Divides the environment into a mesh, assign nodes to the mesh



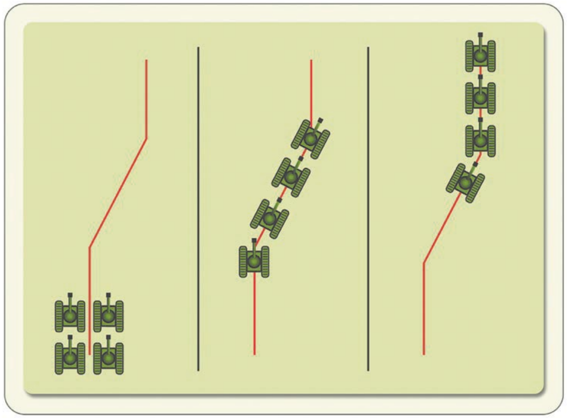


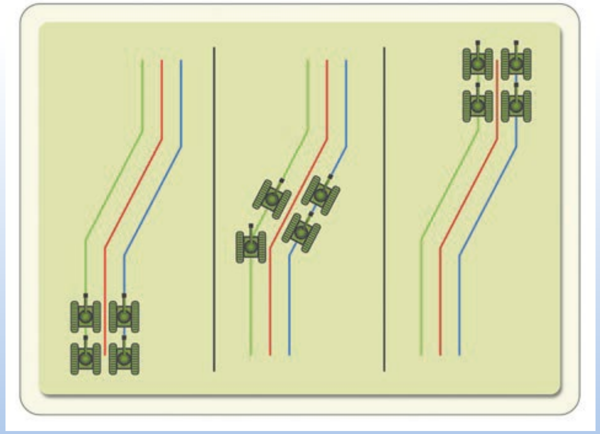
Advance Techniques

- Group pathfinding
  - ➢ Large number of pathfinding operations if done for each character
  - ➢ If all characters directed to move to the same location
    - Can use single path-finding operation
    - Then, share and/or adjust results
  - ➢ The simplest way is to find a path from the centre of group to the destination

Advance Techniques

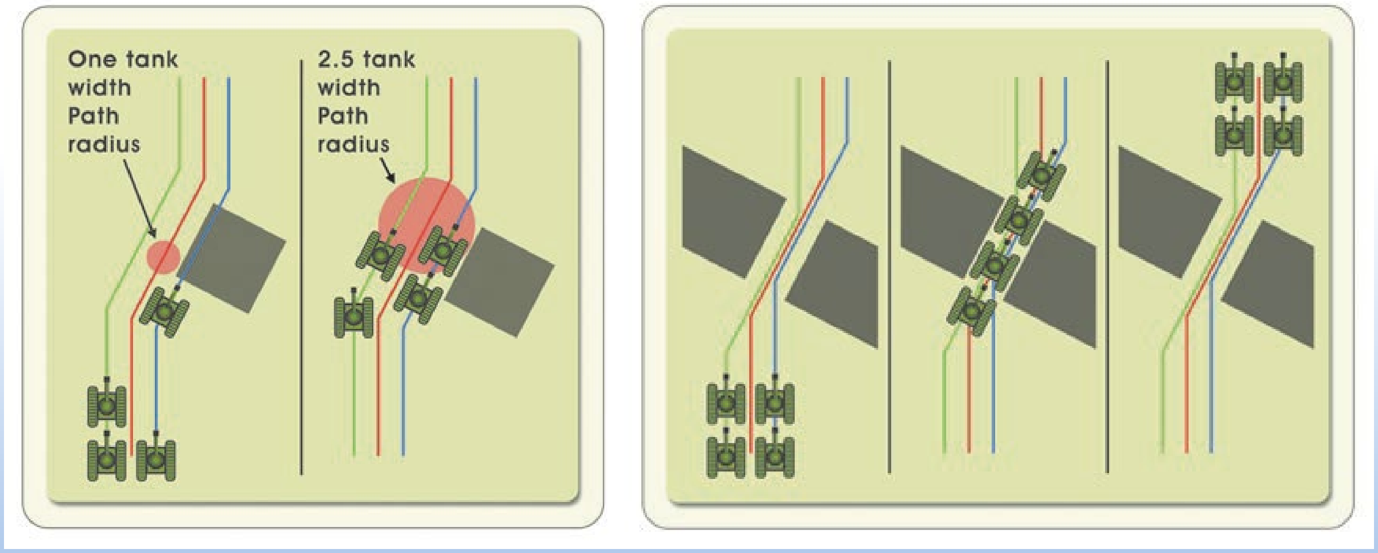
- Problem
  - Units will all line up in a row (called ‘antlining’)
    - Takes long time for last unit to arrive
    - If enemy near destination, group easily destroyed
  - To address antlining
    - Offset the individual unit paths
    - Formation much tighter while moving and at destination





Advance Techniques

- Must make special allowance for width of group
  - Single unit pathfinding locates a path that is as wide as a unit
  - For a group, need to locate a wider path
  - Care must be taken in width checking
    - Might be better to take a narrow opening rather than go around the long way
    - In this case, antlining is perfectly acceptable

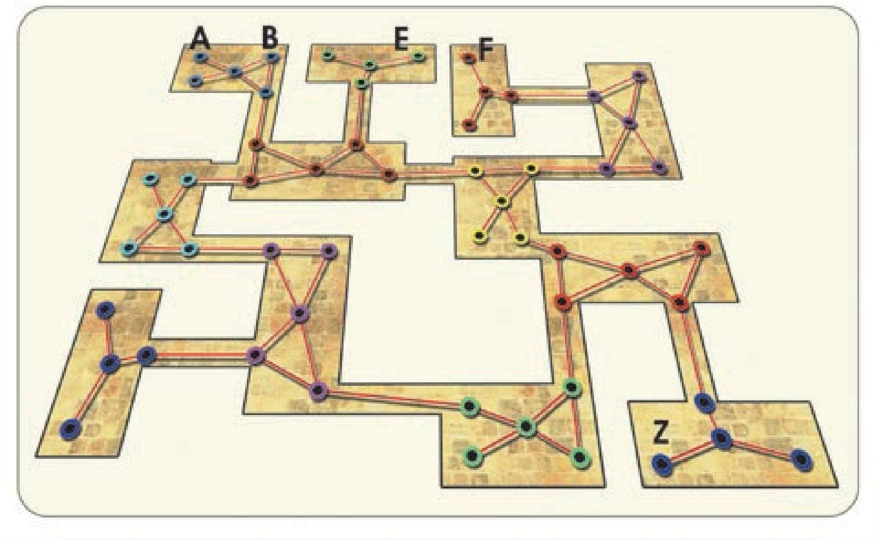


Advance Techniques

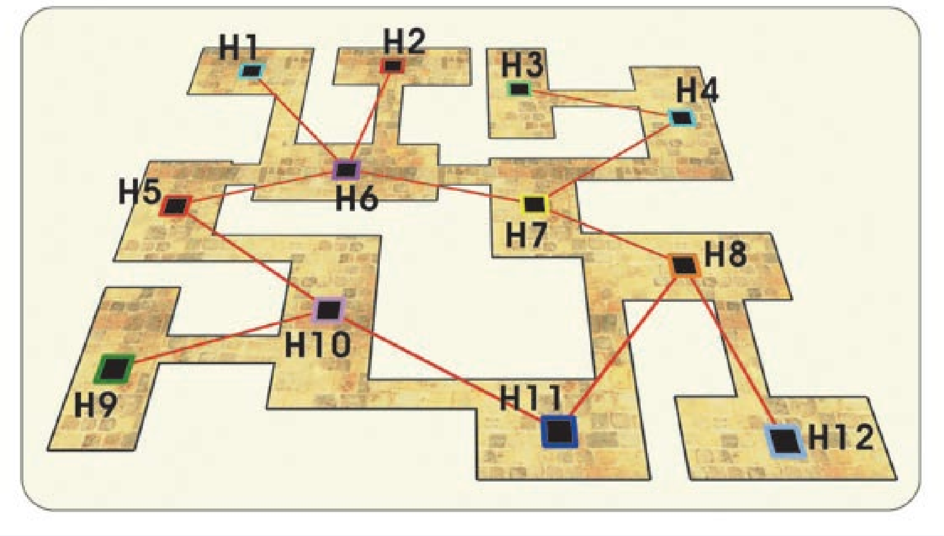
- Antlining and regrouping

Advance Techniques

- Hierarchical path-finding
  - Plan overview of route first then refine as needed



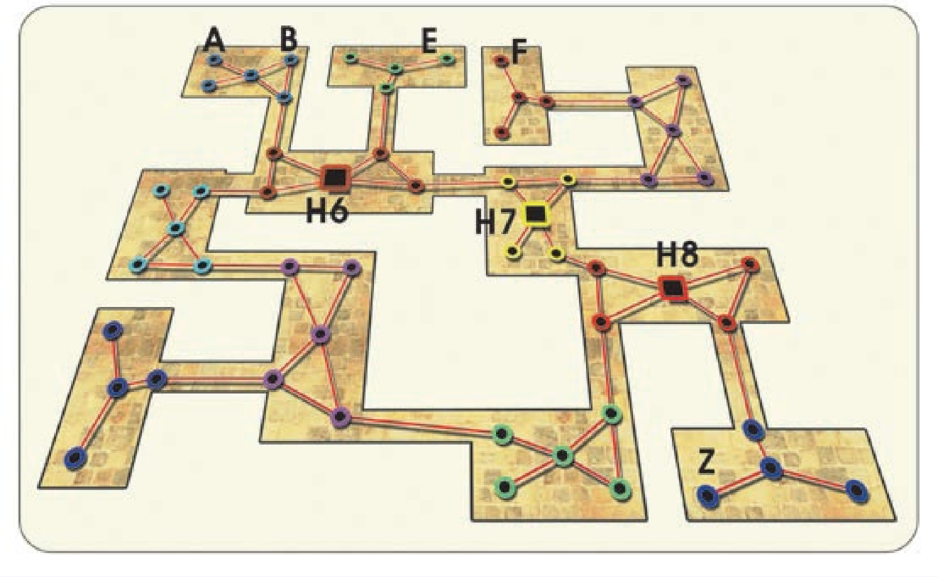
Original network contains 56 nodes and high level of detail



Hierarchical network contains less detailed representation and fewer nodes (12 nodes)

Advance Techniques

- Intermediate destinations
  - Creating shorter paths

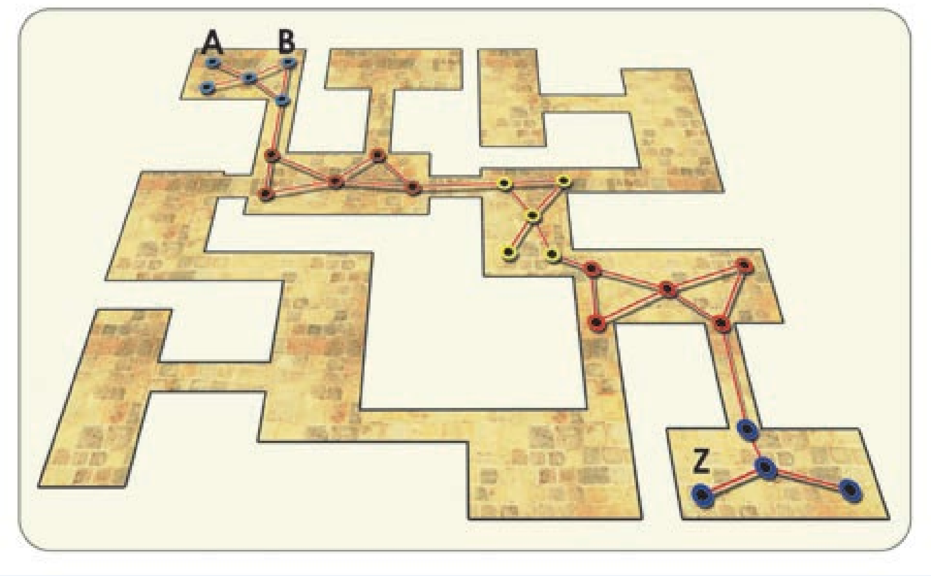


Path from A to Z

- Search on hierarchical network H1→H6→H7→H8→H12
- Shorter path from A to Z: A→H6 →H7→H8→Z

Advance Techniques

- Pruning the network
  - Remove unrelated nodes from consideration



Path from A to Z

- Search on hierarchical network H1→H6→H7→H8→H12
- After pruning only need to search 24 rather than full 56 nodes

References

- Among others, material sourced from
  - https://docs.unity3d.com
  - Jason Gregory, Game Engine Architecture, A.K. Peters
  - Ian Millington, Artificial Intelligence for Games, Morgan Kaufmann
  - Mat Buckland, Programming Game AI by Example, Wordware Publishing
  - John Alquist and Jeannie Novak, Game Development Essentials: Game Artificial Intelligence, Delmar Publishing
  - Screenshots are from various games, mainly sourced from www.ign.com and www.mobygames.com