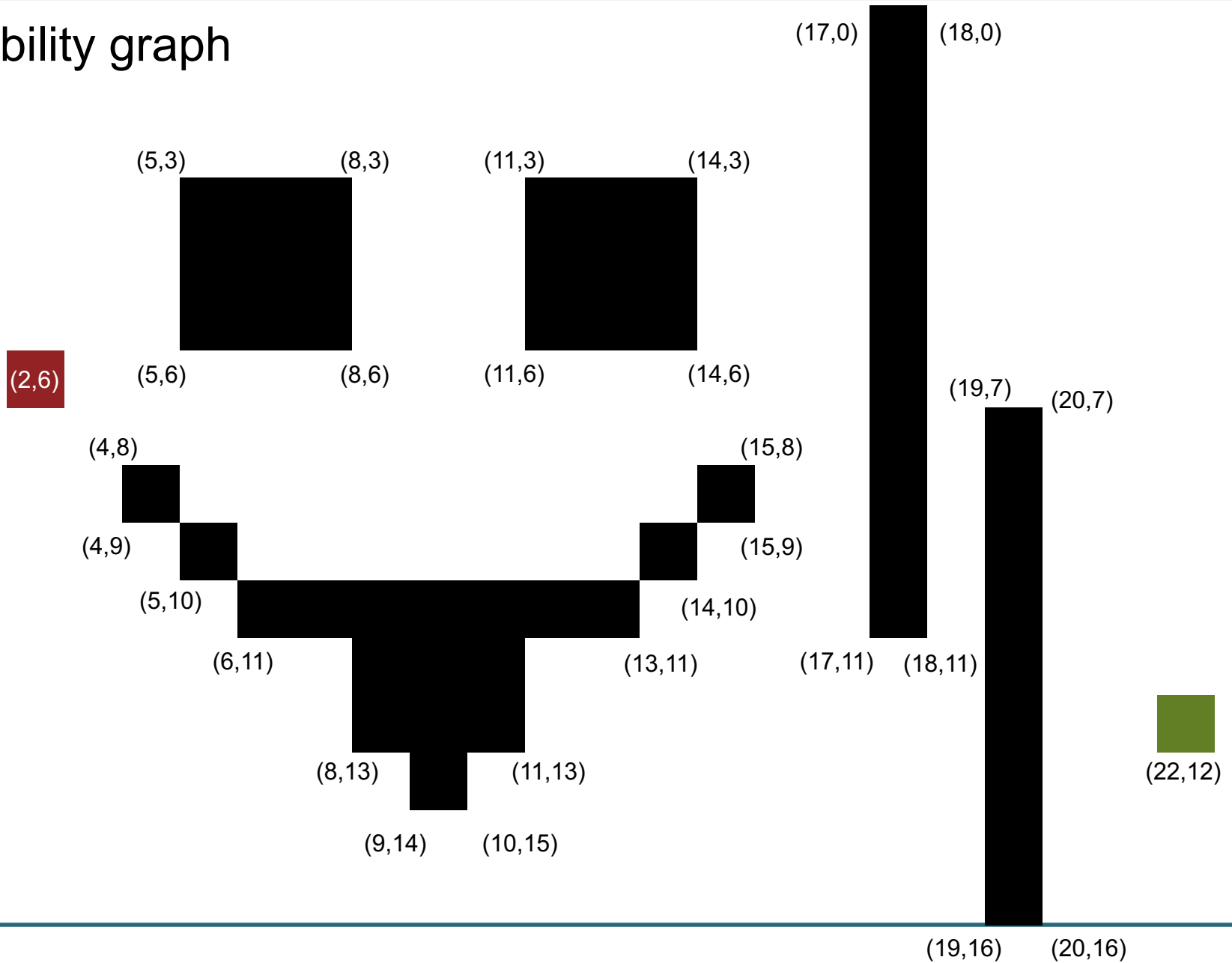


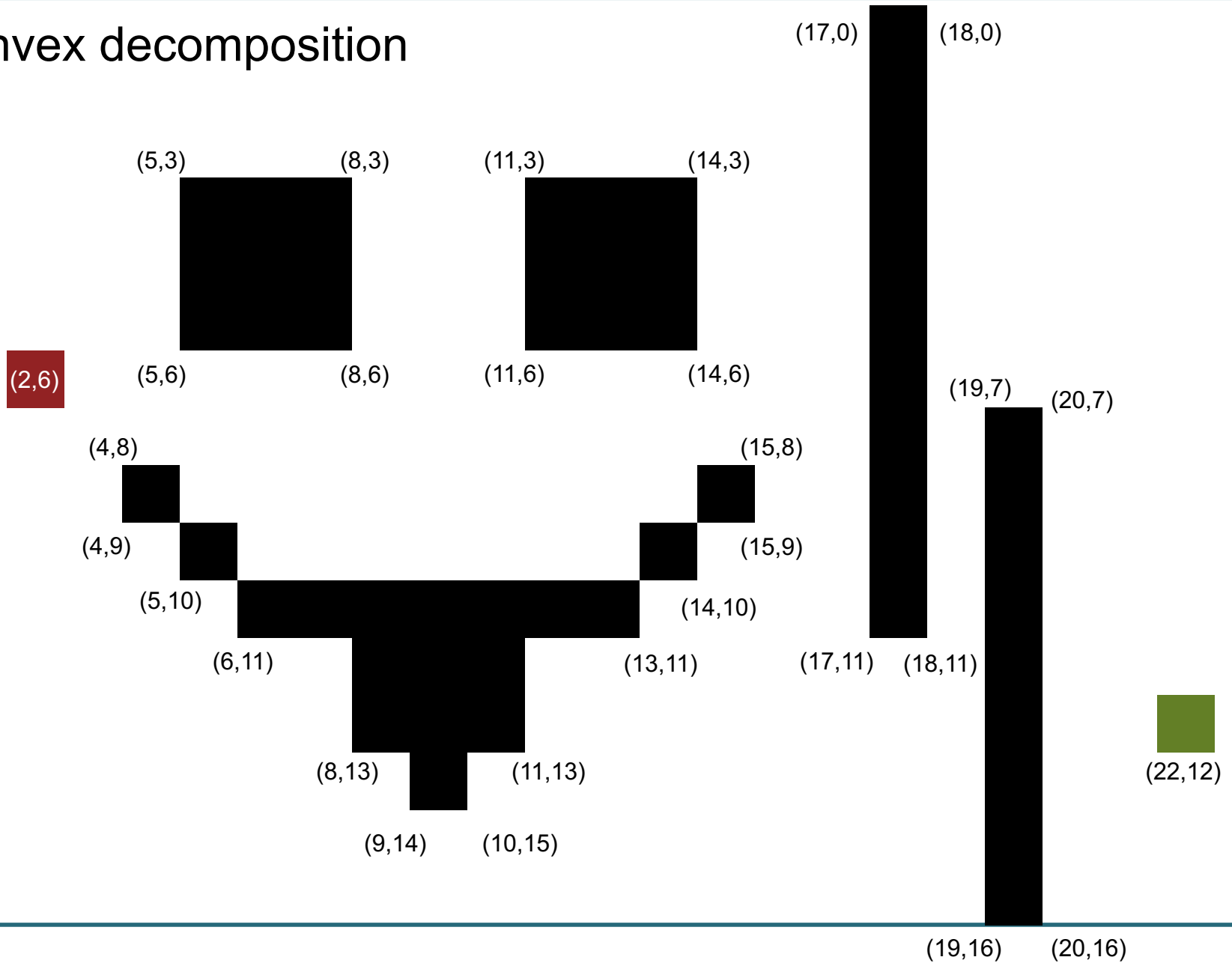
CS425 GAME PROGRAMMING

Path Finding

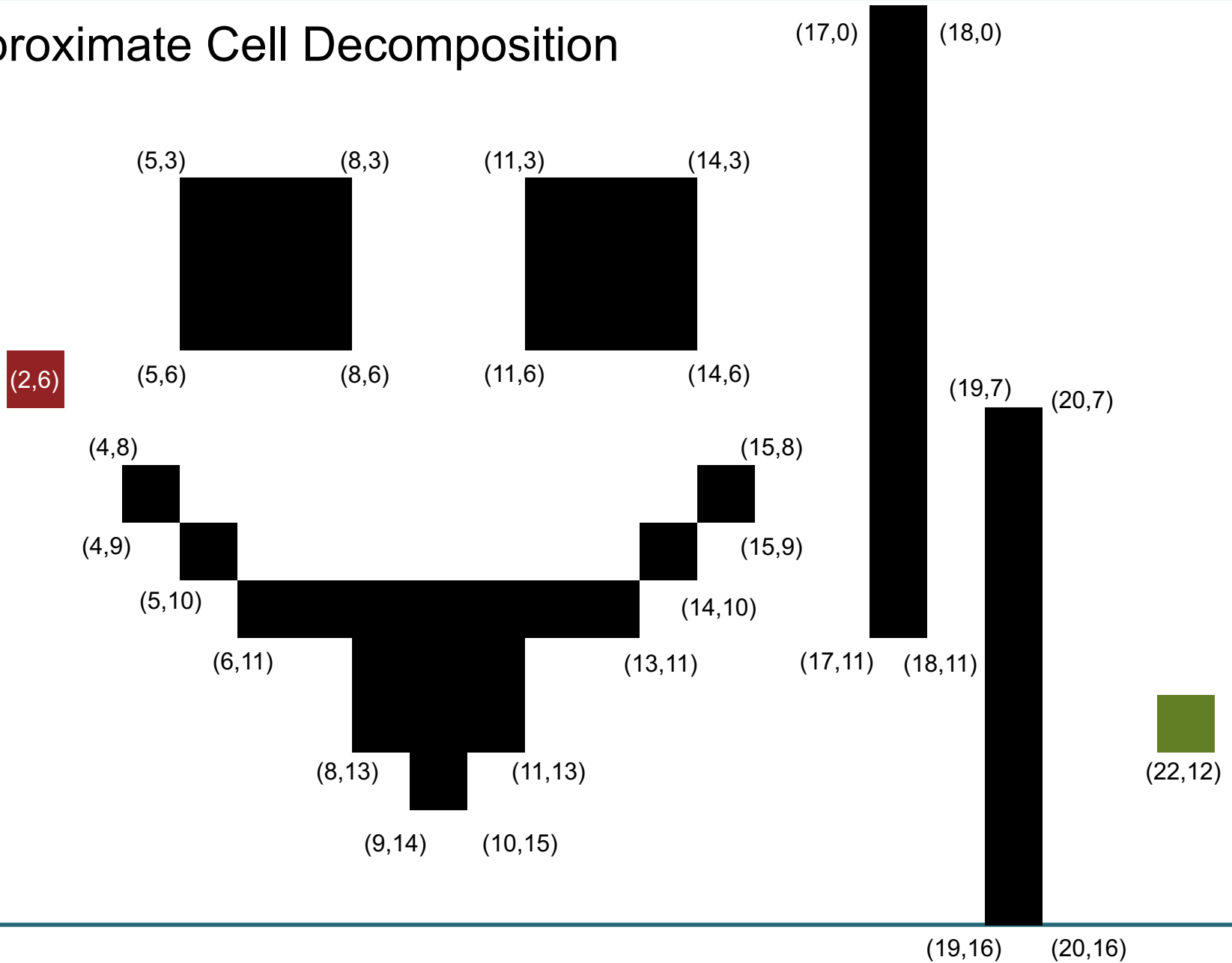
Visibility graph



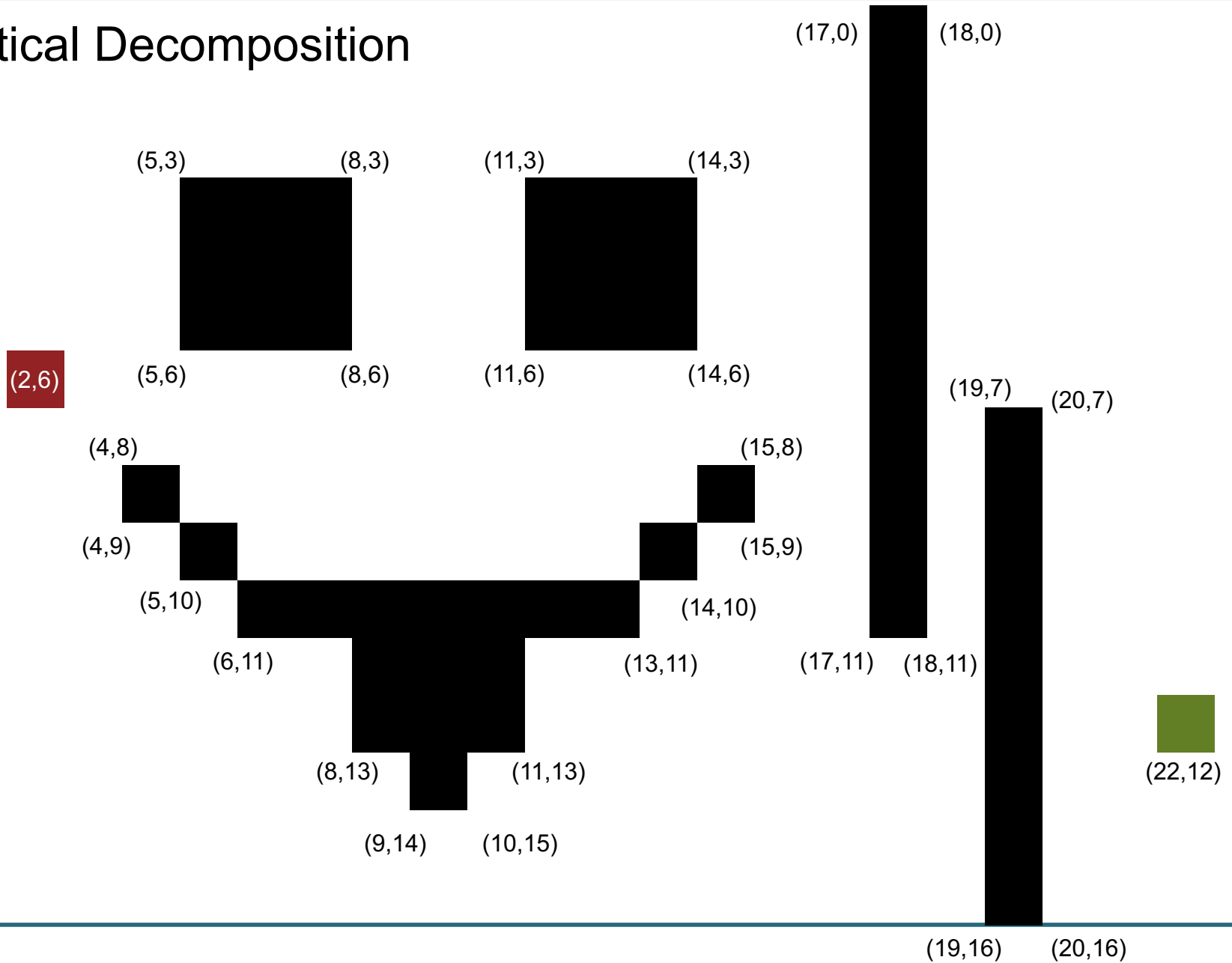
Convex decomposition



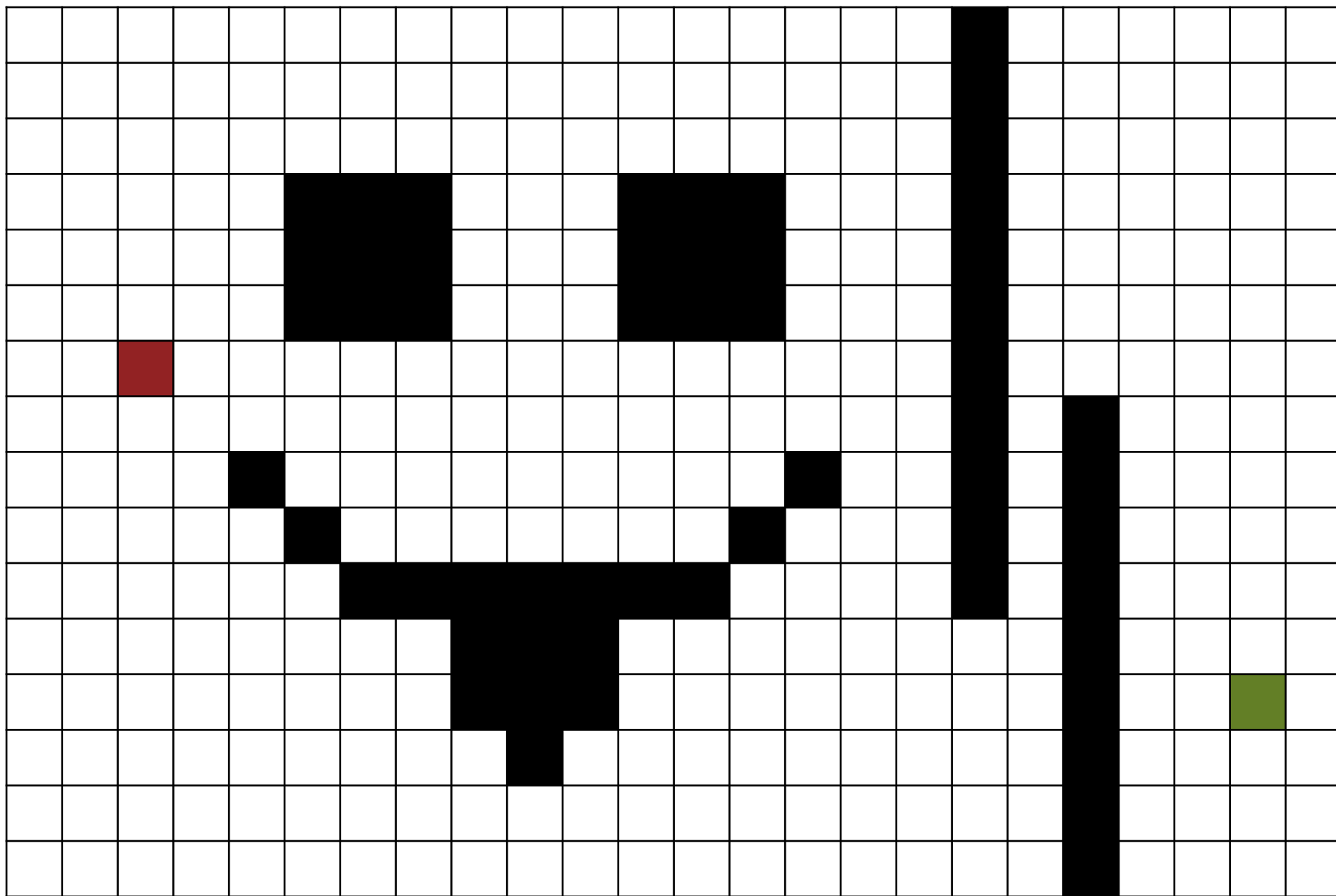
Approximate Cell Decomposition



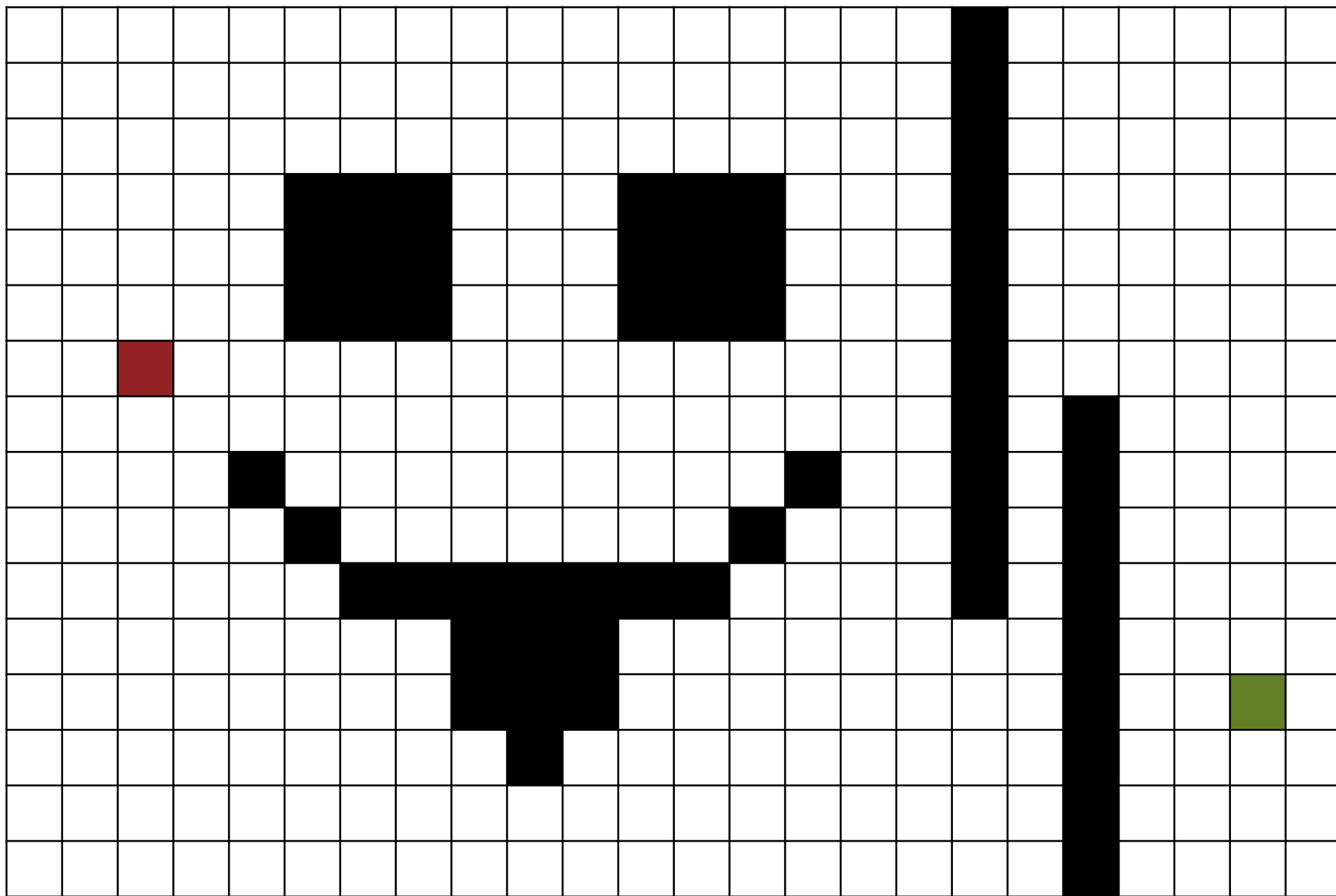
Vertical Decomposition



Dijkstra's



Best First



[illegible]

Pathfinding

In many games, computer controlled opponents need to move from one place to another in the game world

- If the start and end points are known in advance, and never change,
 - Can simply define a **fixed path**
 - Computed before the game begins
 - Use path following techniques
 - **Most platformers do this**
- If the start and end points vary, and are not known in advance (or may vary due to changes in game state),
 - Have to compute the path to follow while the game is running
 - Need to use a **pathfinding** algorithm



Searching for a Path

- A path is a list of cells, points, or nodes that an agent must traverse
- A pathfinding algorithm finds a path
 - From a start position to a goal position
- The following pathfinding algorithms can be used on
 - **Grids**
 - Waypoint graphs
 - Navigation meshes

Criteria for Evaluating Pathfinding Algorithms

- Quality of final path (Criteria?)
- Resource consumption during search
 - CPU and memory
- Whether it is a **complete** algorithm
 - A **complete** algorithm guarantees to find a path if one exists



What's the absolute simplest way to find a path?

Basic Path Finding Algorithm

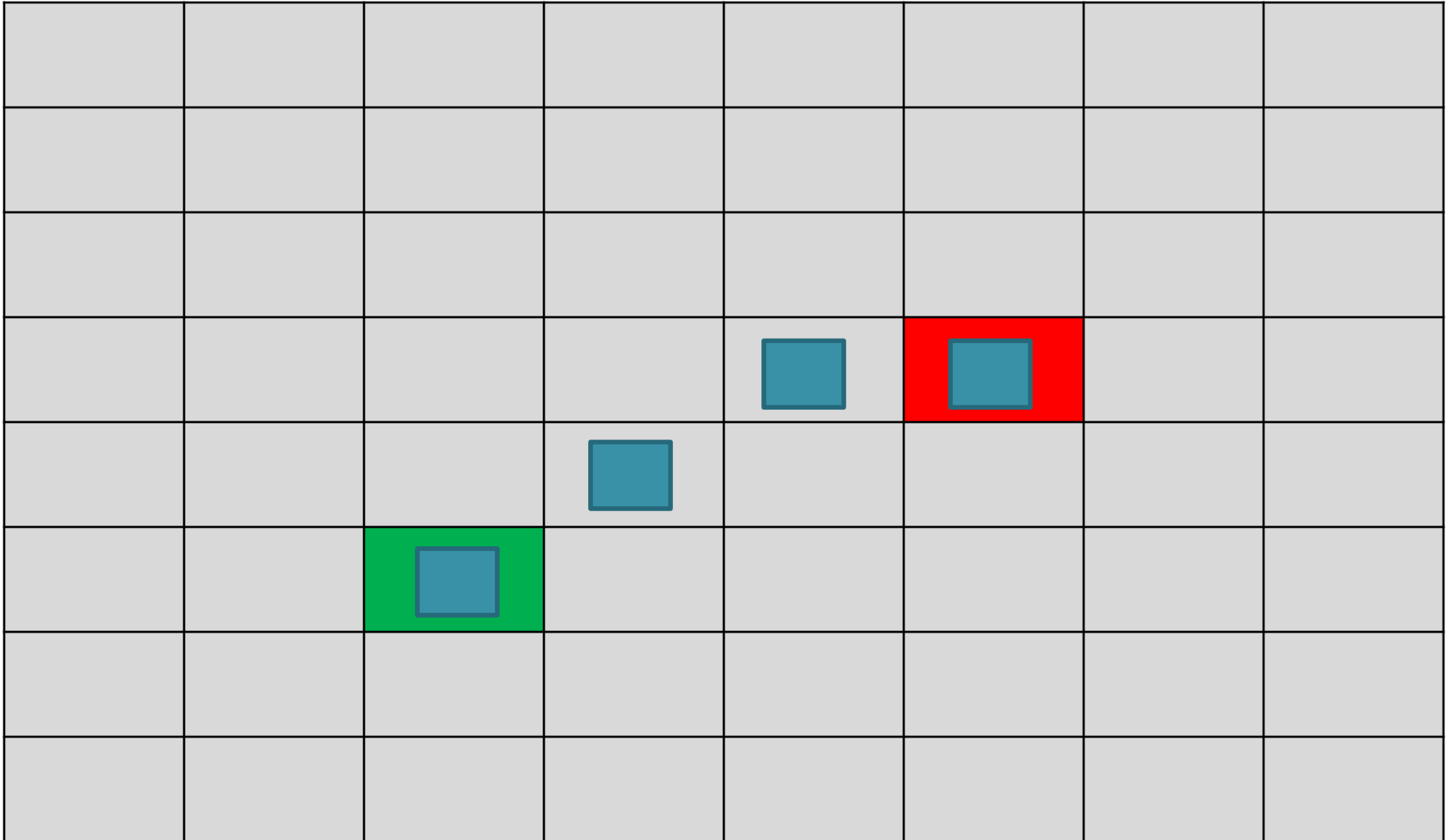
```
if (positionX > destinationX)  
    positionX--;
```

```
else if (positionX < destinationX)  
    positionX++;
```

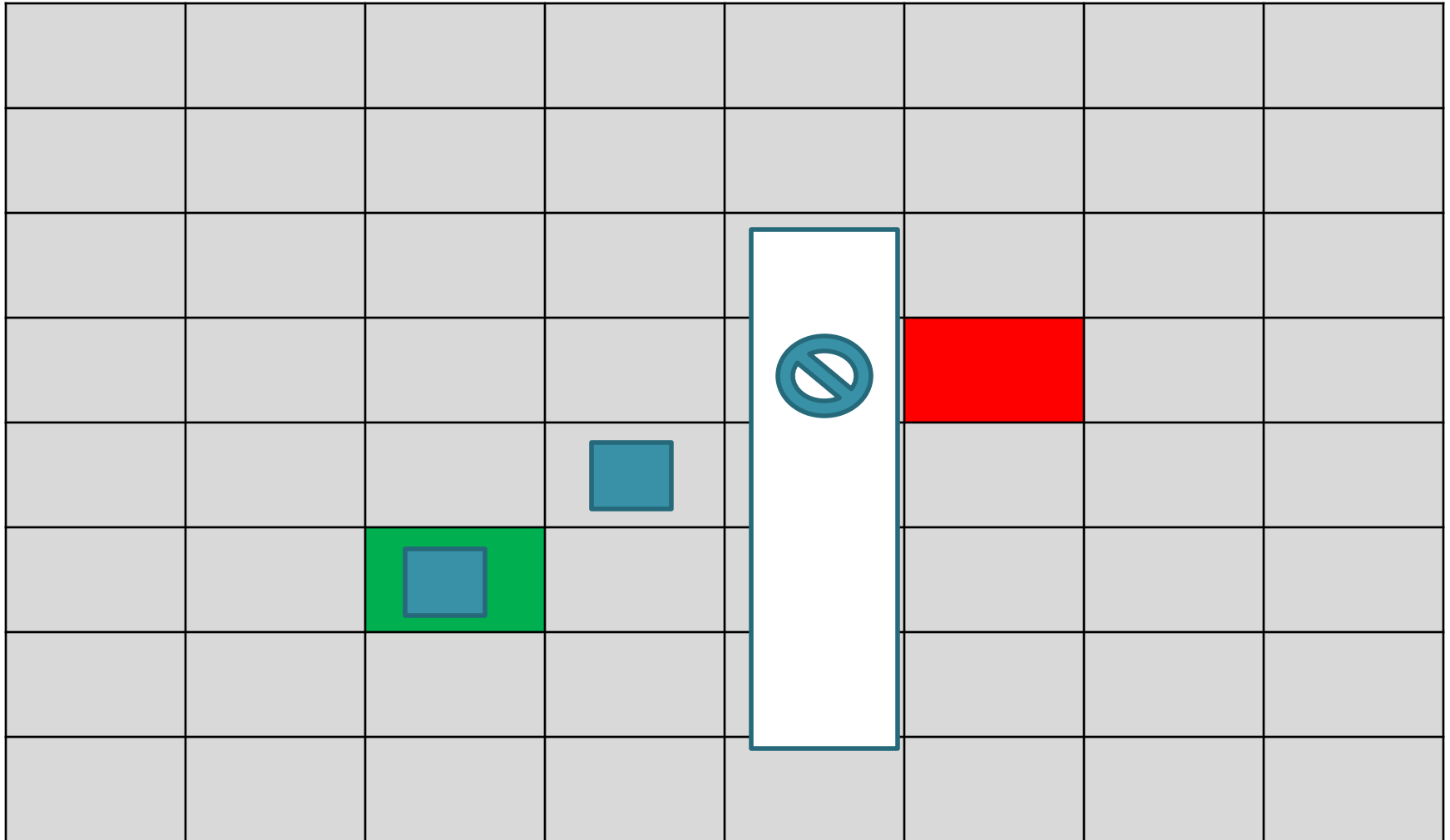
```
if (positionZ > destinationZ)  
    positionZ--;
```

```
else if (positionZ < destinationZ)  
    positionZ++;
```

Basic Path Finding Algorithm



Basic Path Finding Algorithm Limitations



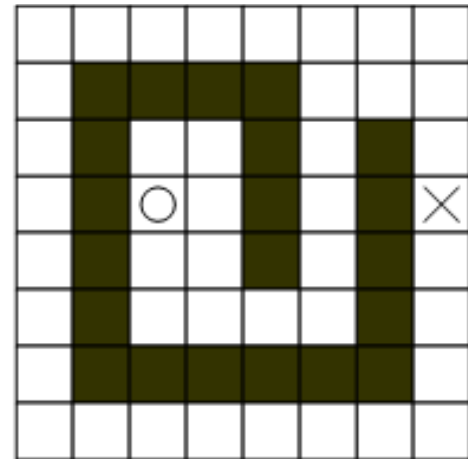
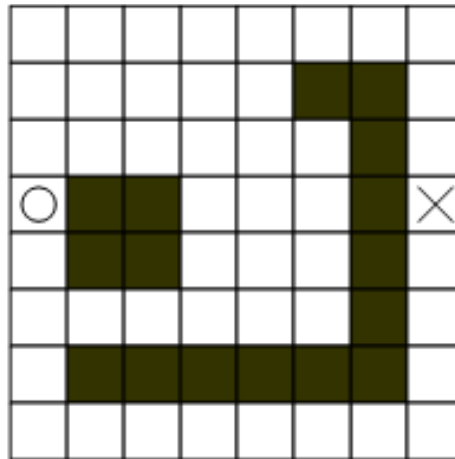
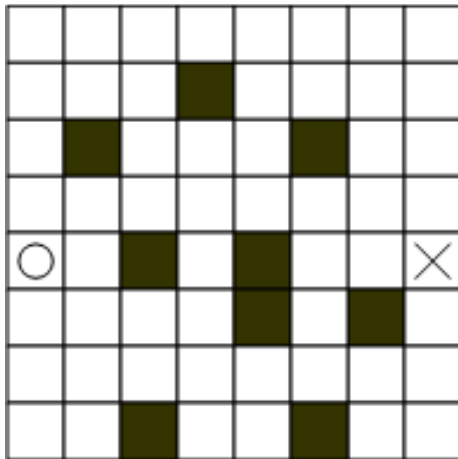
Random Trace Obstacle Avoidance

- For simple obstacles that aren't large
- Algorithm:
 1. Go towards target destination
 2. When you hit an obstacle:
 - Back up
 - Randomly turn right or left 45-90 degrees
 - Move forward again for some random amount in a range
 3. Go back to step 1
- Easy & fast



Random Trace

- How will Random Trace do on the following maps?



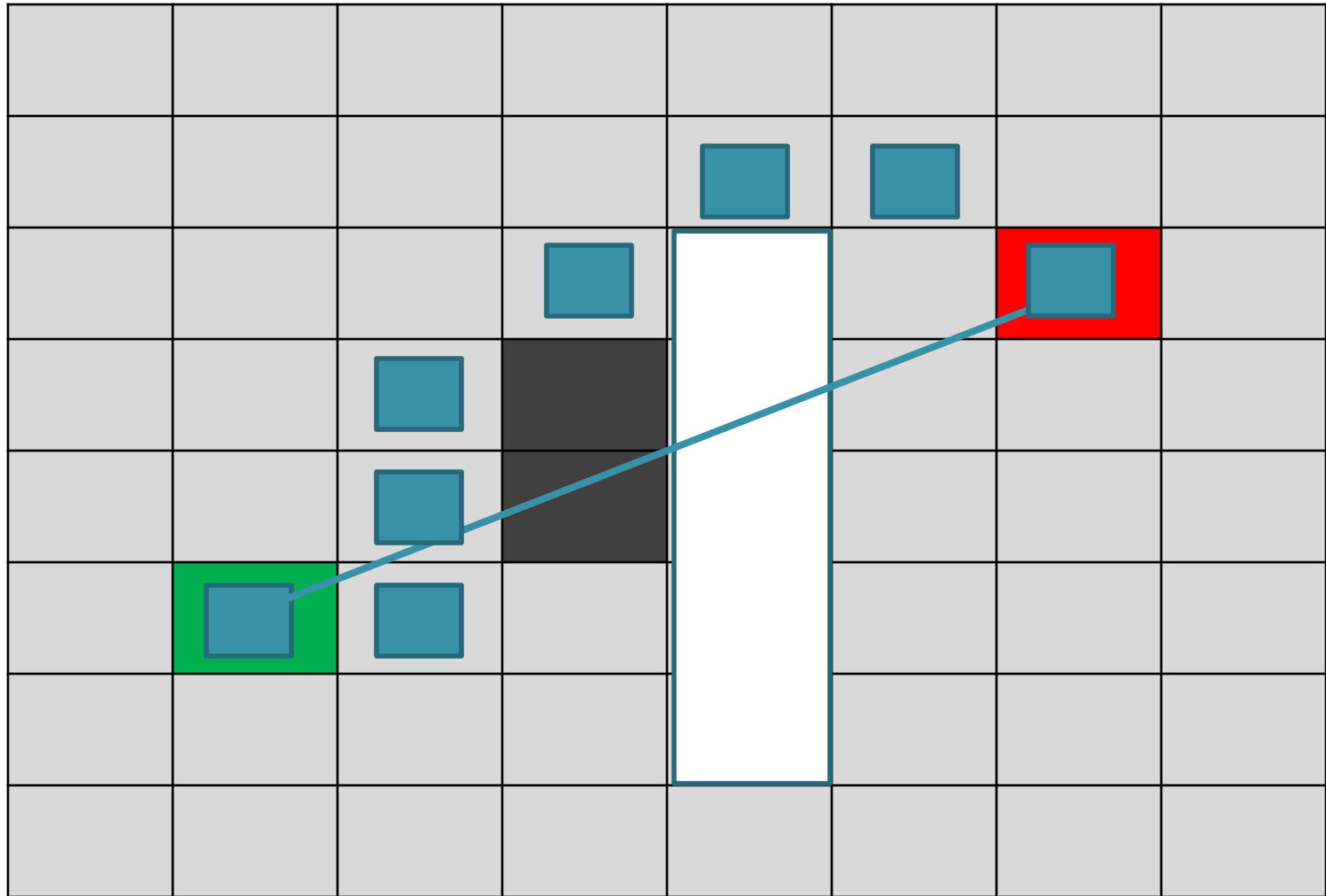
Random Trace Characteristics

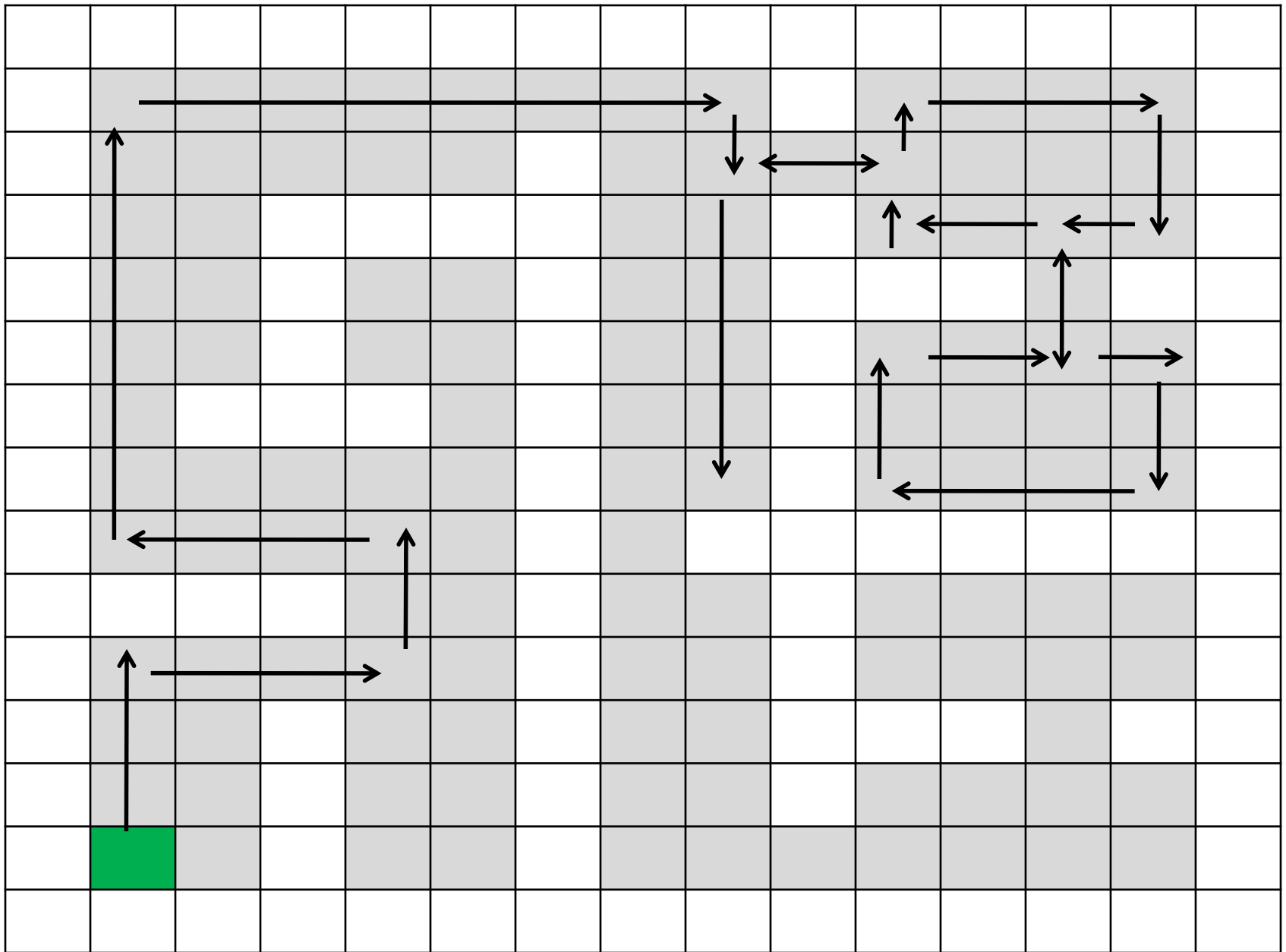
- Not a ***complete*** algorithm
- Found paths are unlikely to be optimal
- Consumes very little memory

Tracing Around Obstacles

1. Go towards target destination
2. If you hit an obstacle:
 - a) Trace the contour of the obstacle blocking the path
 - b) Periodically test if a line to your destination intersects the obstacle
 - i. If no, stop tracing and head towards destination
 - ii. If yes, go back to tracing

Tracing Around Obstacles





Breadcrumb Path Finding



Search

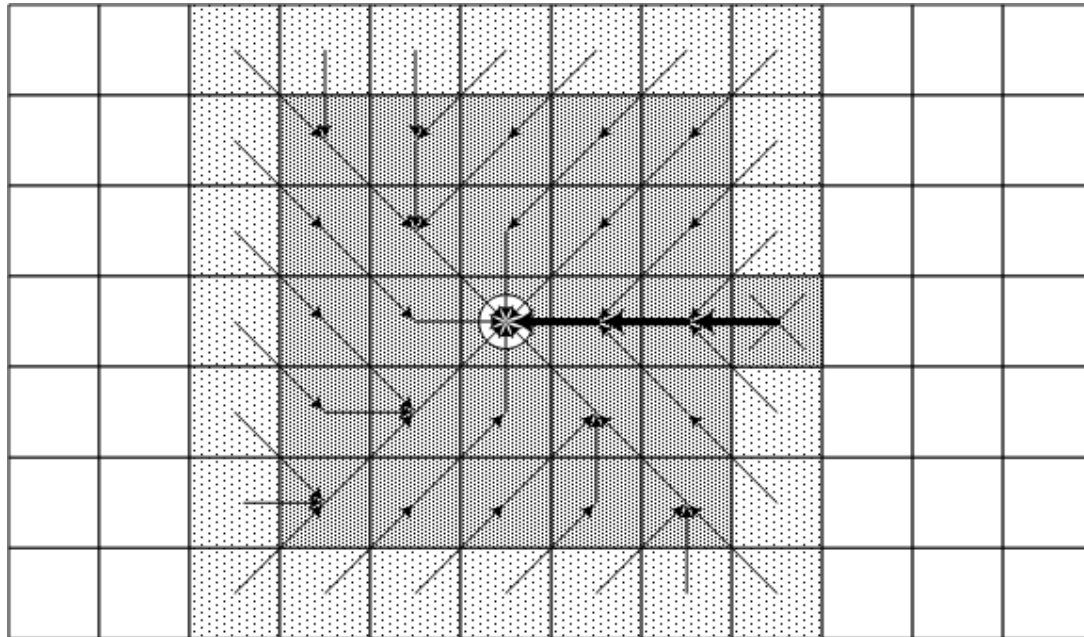
- Uninformed Search
 - Use no information obtained from the environment
 - Blind Search: BFS (Wavefront), DFS
- Informed Search
 - Use evaluation function
 - More efficient
 - Heuristic Search: A^* , D^* , etc.

Overall Structure of the Algorithms

1. Create start point node – push onto open list
2. While open list is not empty
 - A. Pop node from open list (call it currentNode)
 - B. If currentNode corresponds to goal, break from step 2
 - C. Create new nodes (successors nodes) for cells around currentNode and push them onto open list
 - D. Put currentNode onto closed list

Breadth-First

- Finds a path from the start to the goal by examining the search space node-by-node



Breadth-First Characteristics

- Exhaustive search
 - Systematic, but not clever
- Consumes substantial amount of CPU and memory
- Guarantees to find paths that have fewest number of nodes in them
- ***Complete*** algorithm

Dijkstra

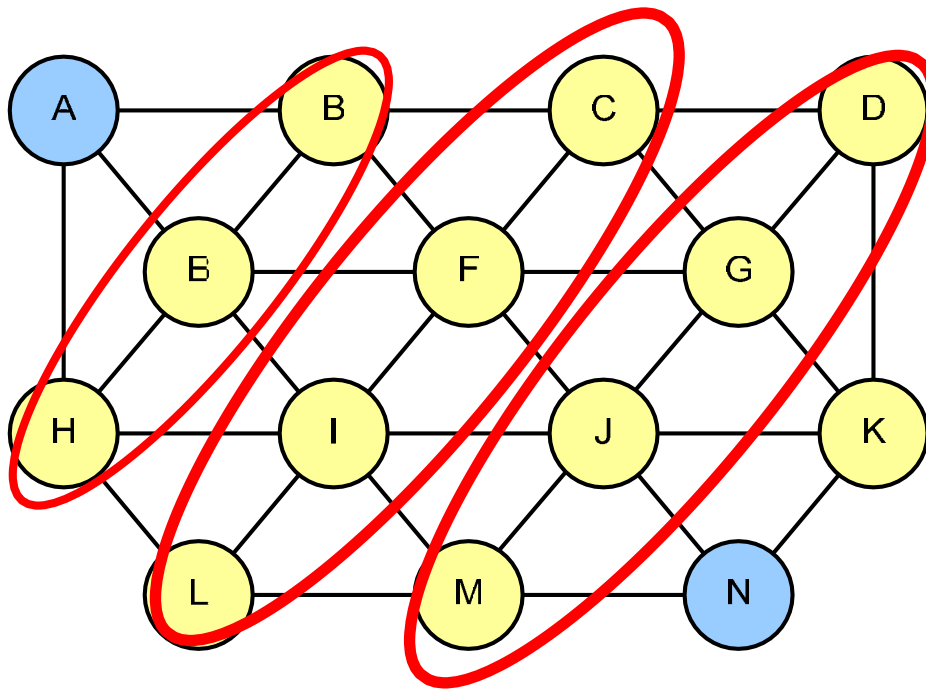
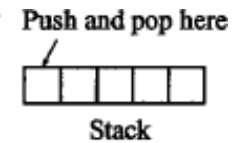
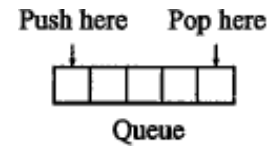
- Selects closest unprocessed node from the start, and updates its value, and that of its neighbors.
- Nodes are valued by the distance from the start.
- Disregards distance to goal
 - Keeps track of the cost of every path
 - No guessing
- Computes accumulated cost paid to reach a node from the start
 - Uses the cost (called the given cost) as a priority value to determine the next node that should be brought out of the open list

Dijkstra Characteristics

- Exhaustive search
- At least as resource intensive as Breadth-First
- Always finds the most optimal path
- ***Complete*** algorithm

Uninformed Search

Graph Search from A to N

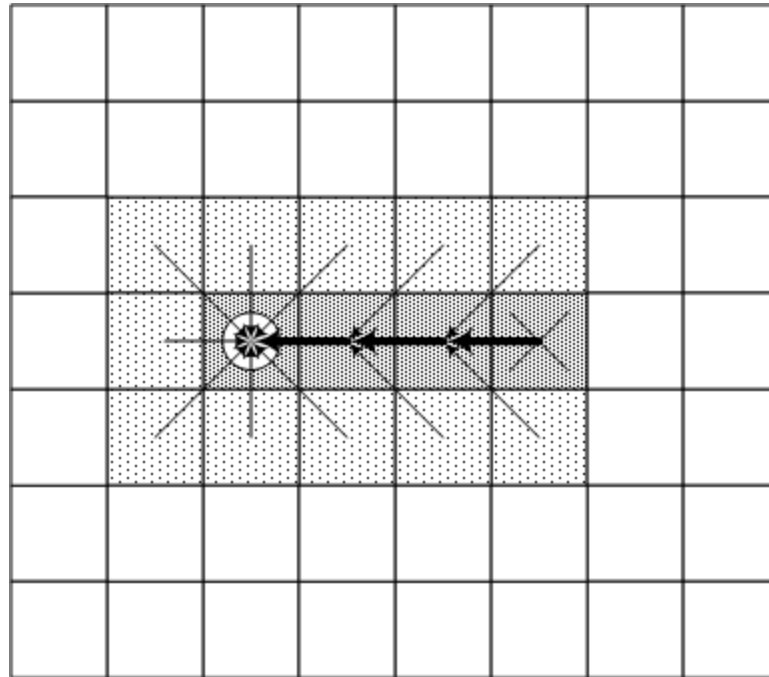


— BFS

Best-First

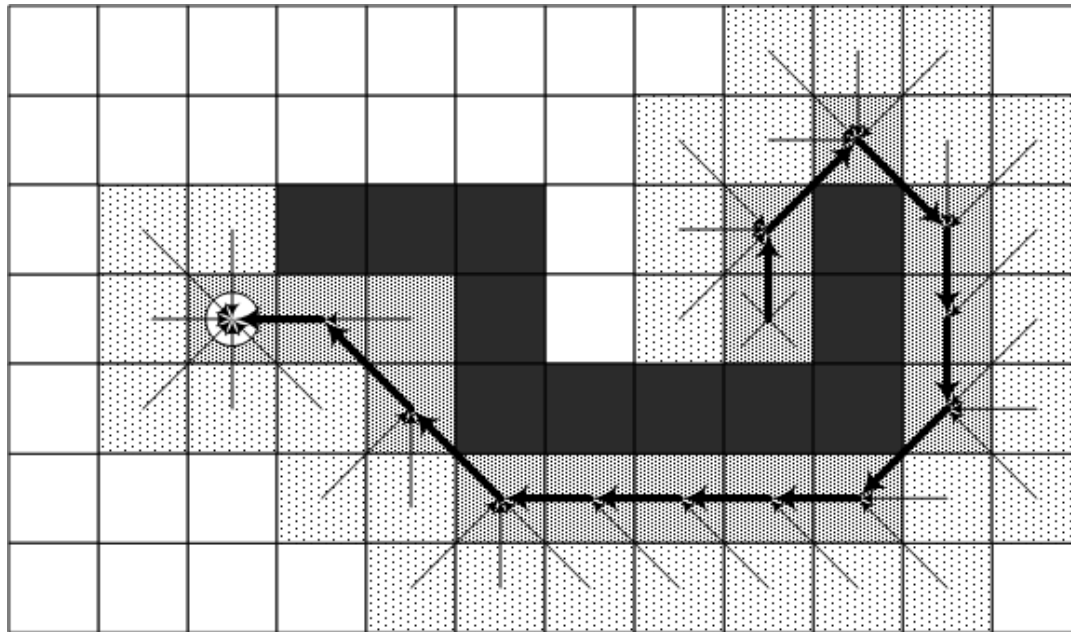
- Similar to Dijkstra's Algorithm, but uses a heuristic to determine the value of node.
- Uses problem specific knowledge to speed up the search process
- Head straight for the goal
- Computes the distance of every node to the goal
 - Uses the distance (or heuristic cost) as a priority value to determine the next node that should be brought out of the open list

Best-First (continued)



Best-First (continued)

- Situation where Best-First finds a suboptimal path



Best-First Characteristics

- Heuristic search
- Uses fewer resources than Breadth-First
- Tends to find good paths
 - No guarantee to find most optimal path
- ***Complete*** algorithm

Informed Search: A*

Notation

- n → node/state
- $c(n_1, n_2)$ → the length of an edge connecting between n_1 and n_2
- $b(n_1) = n_2$ → backpointer of a node n_1 to a node n_2 .

A* pathfinding

- A* algorithm is widely used as the conceptual core for pathfinding in computer games
 - Most commercial games use variants of the algorithm tailored to the specific game
 - Original paper:
 - A Formal Basis for the Heuristic Determination of Minimum Cost Paths. Hart, P.E.; Nilsson, N.J.; Raphael, B. *IEEE Trans. Systems Science and Cybernetics*, 4(2), July **1968**, pp. 100-107
- A* is a graph search algorithm
 - There is a LARGE research literature on graph search algorithms, and computer game pathfinding

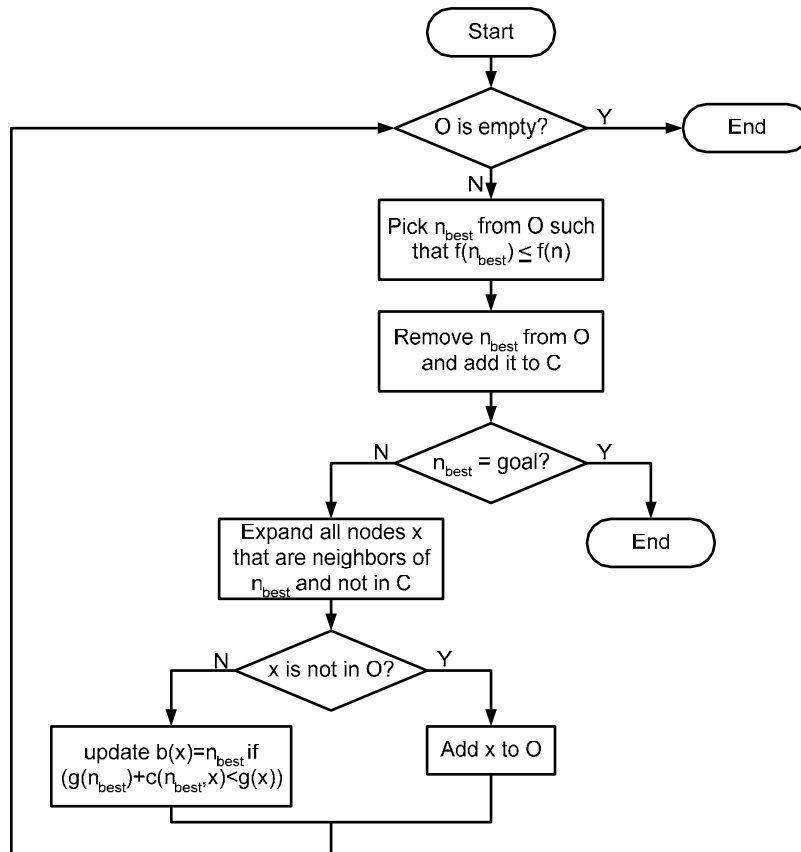
Understanding A*

- To understand A*
 - First understand Breadth-First, Best-First, and Dijkstra algorithms
- These algorithms use nodes to represent candidate positions on paths

Understanding A*

- All of the following algorithms use two lists
 - The ***open*** list
 - The ***closed*** list
- Open list keeps track of promising nodes
- When a node is examined from open list
 - Taken off open list and checked to see whether it has reached the goal
- If it has not reached the goal
 - Used to create additional nodes
 - Then placed on the closed list

A*: Algorithm

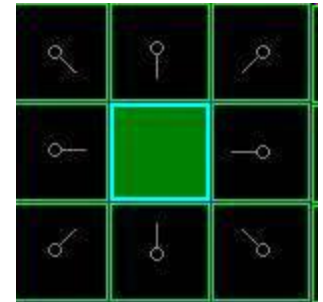


The search requires 2 lists to store information about nodes

- 1) **Open list (O)** stores nodes for expansions
- 2) **Closed list (C)** stores nodes which we have explored

Open and Closed Lists

- Starting the search
 - Add A to **open list** of nodes to be considered
 - Look at adjacent nodes, and add all walkable nodes to the **open list**
 - i.e., ignore walls, water, or other illegal terrain
 - For each of these squares, note that their **parent node** is the starting node, A
 - Remove A from open list, add to **closed list**
- Open list
 - Contains nodes that might fall along the path (or might not)
 - A list of nodes that need to be “checked out”
- Closed list
 - A list of nodes that no longer need to be considered



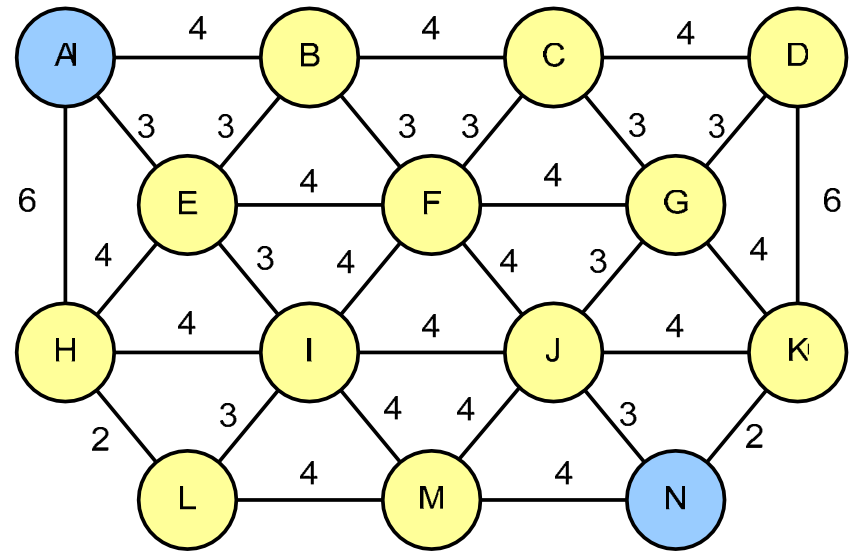
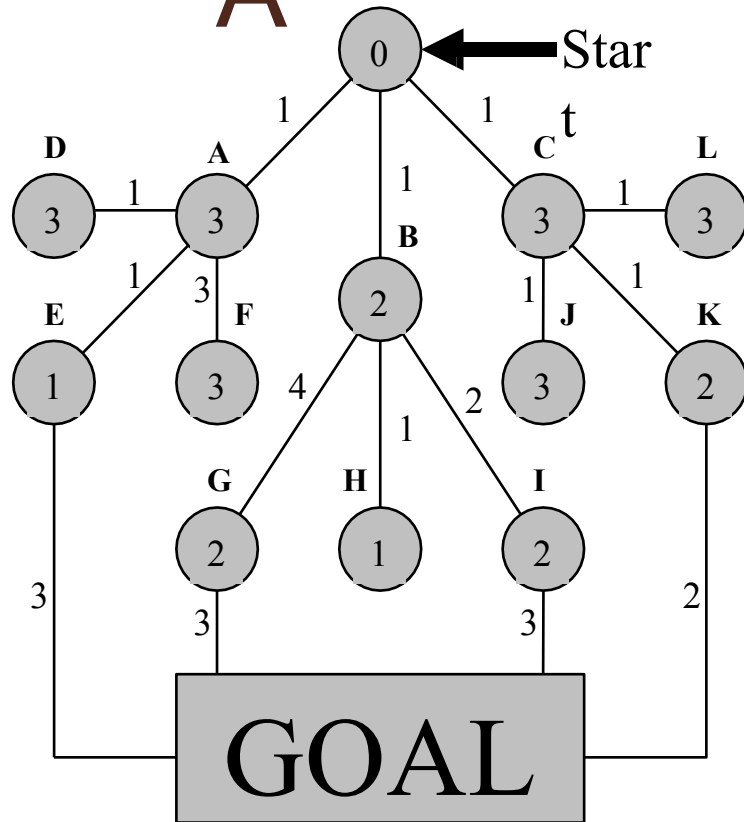
State of A* after the first step has completed. All adjacent nodes are part of the open list. The circle with line points to the parent node. The blue outline around the green start node indicates it is in the closed list.

A* Characteristics

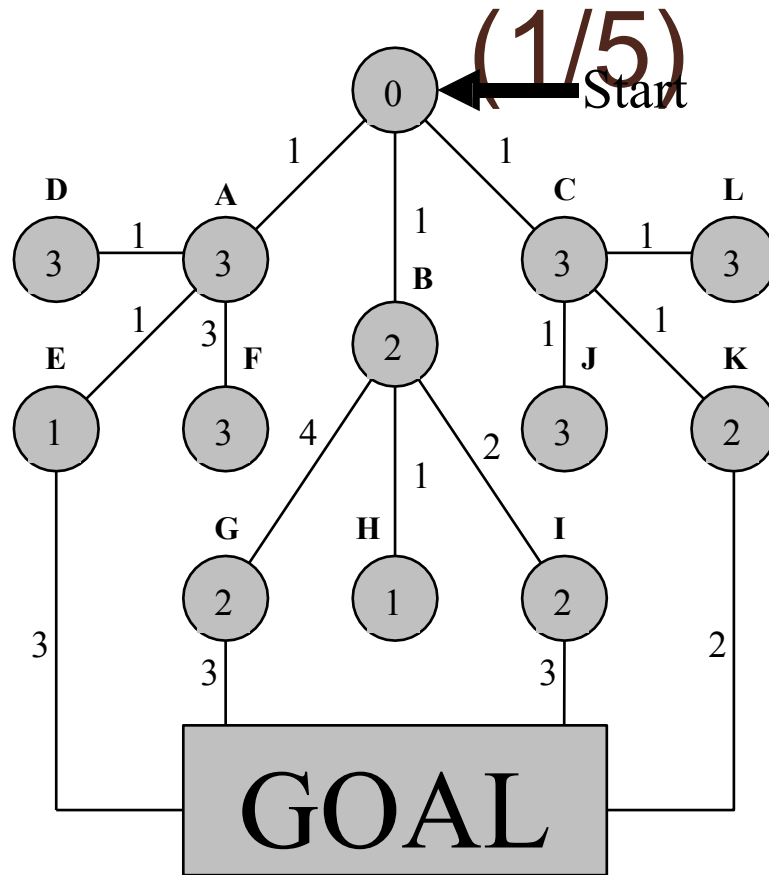
- Heuristic search
- On average, uses fewer resources than Dijkstra and Breadth-First
- ***Complete*** algorithm
- *Admissible* heuristic guarantees it will find the most optimal path
 - **Admissible** if it never overestimates the cost to reach the destination node
 - E.g. roadmap domain: Euclidean distance to destination
 - Does our formulation have an admissible heuristic?

Two Examples Running

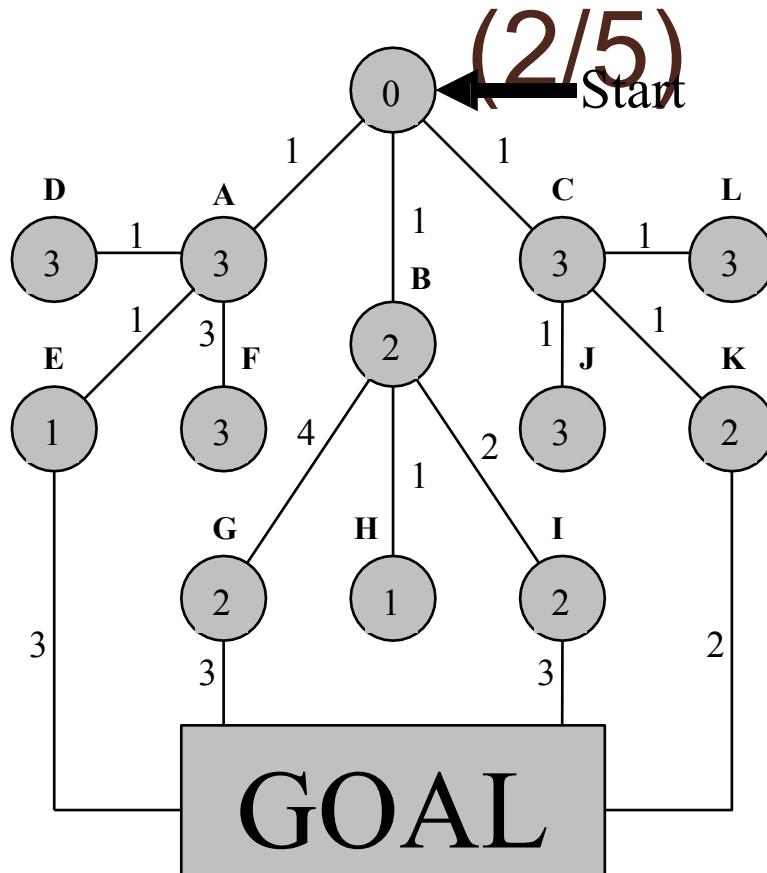
A^*



Example



Example



First expand the start node

B (3)
A (4)
C (4)

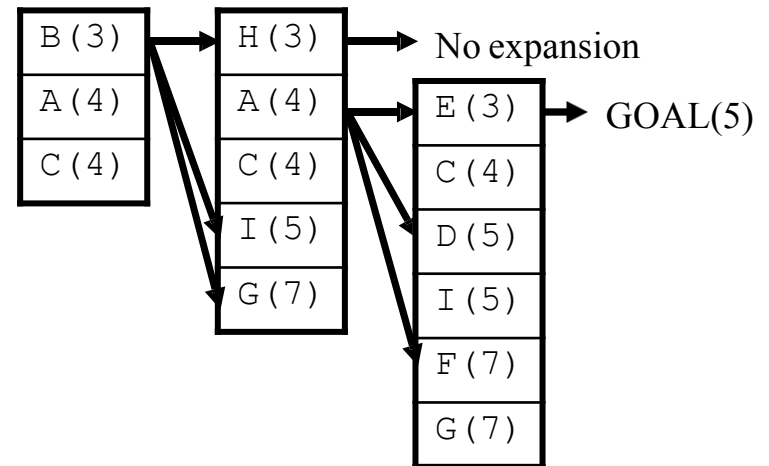
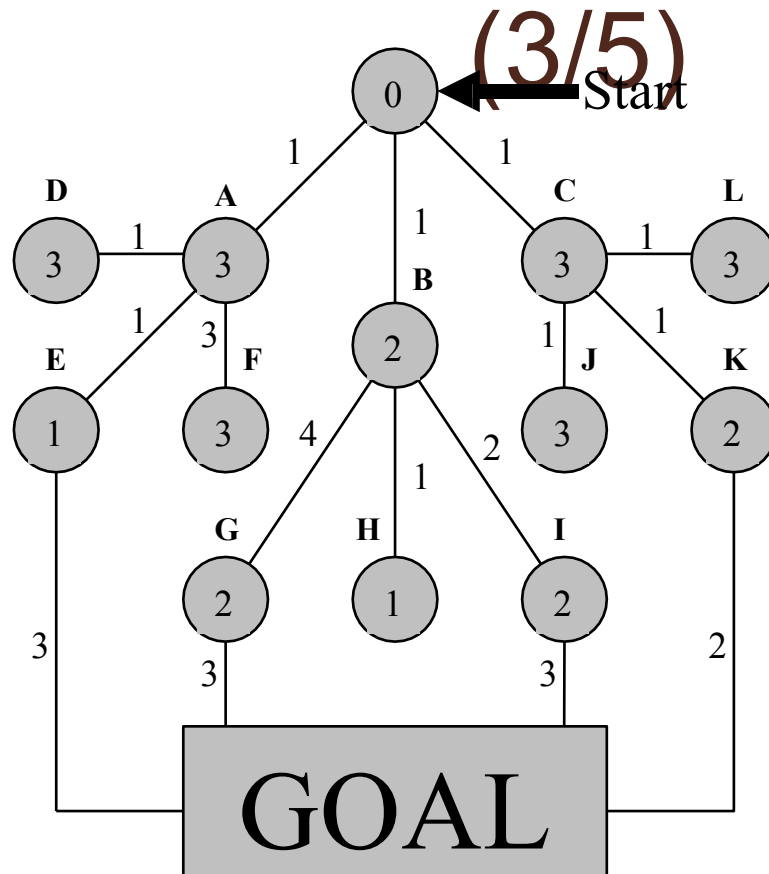
If goal not found,
expand the first node
in the priority queue
(in this case, B)

H (3)
A (4)
C (4)
I (5)
G (7)

Insert the newly expanded
nodes into the priority queue
and continue until the goal is
found, or the priority queue is
empty (in which case no path
exists)

Note: for each expanded node,
you also need a pointer to its respective
parent. For example, nodes A, B and C
point to Start

Example



We've found a path to the goal:

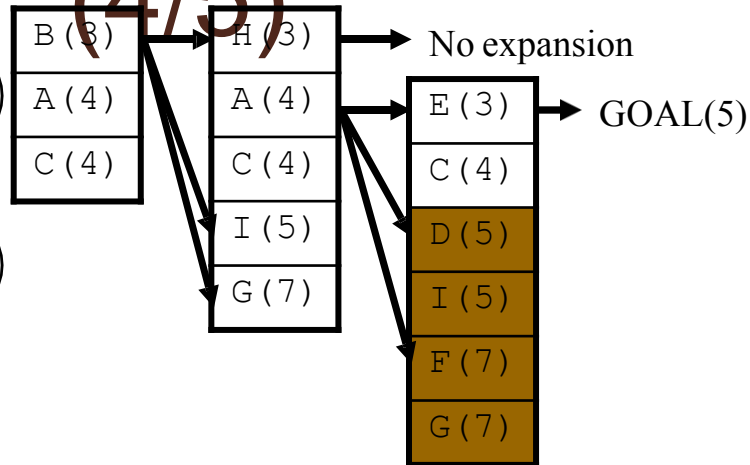
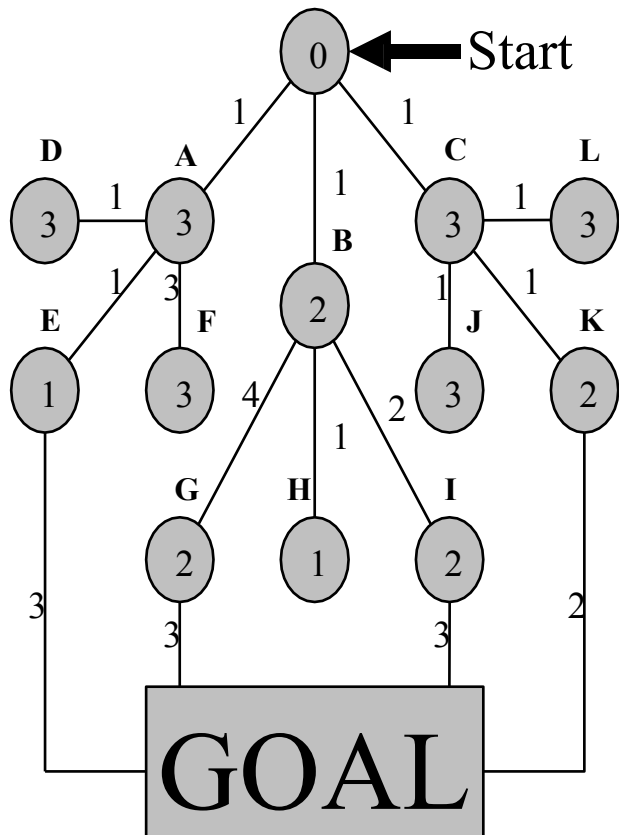
Start \Rightarrow A \Rightarrow E \Rightarrow Goal

(from the pointers)

Are we done?

Example

(4/5)

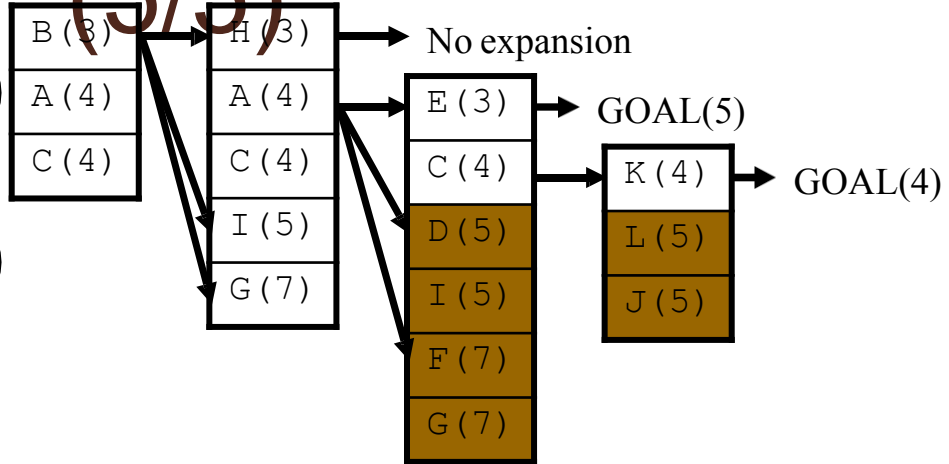


There might be a shorter path, but assuming non-negative arc costs, nodes with a lower priority than the goal cannot yield a better path.

In this example, nodes with a priority greater than or equal to 5 can be pruned.

Why don't we expand nodes with an equivalent priority? (why not expand nodes D and I?)

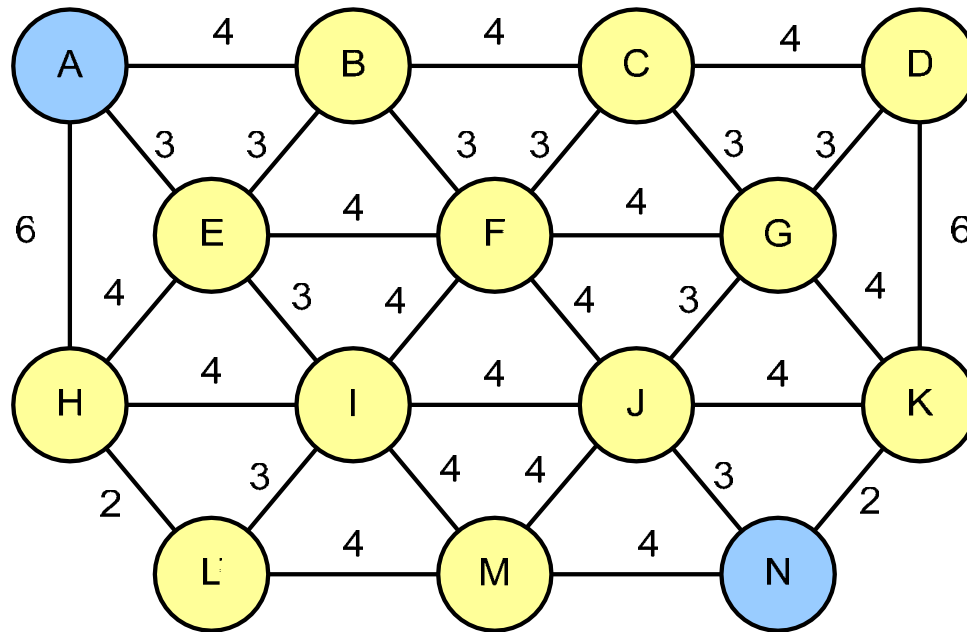
(5/5)




As we can see from this example, there was a shorter path through node K. To find the path, simply follow the back pointers.

If the priority queue still wasn't empty, we would continue expanding while throwing away nodes with priority lower than 4.
(remember, lower numbers = higher priority)

A*: Example (1/6)

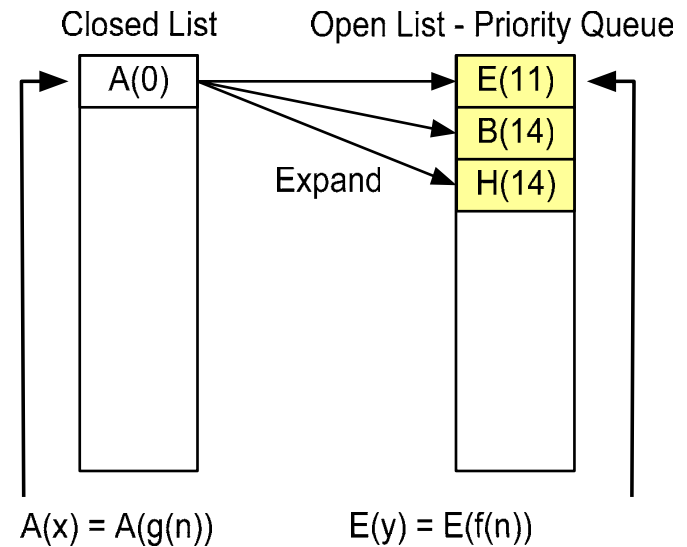
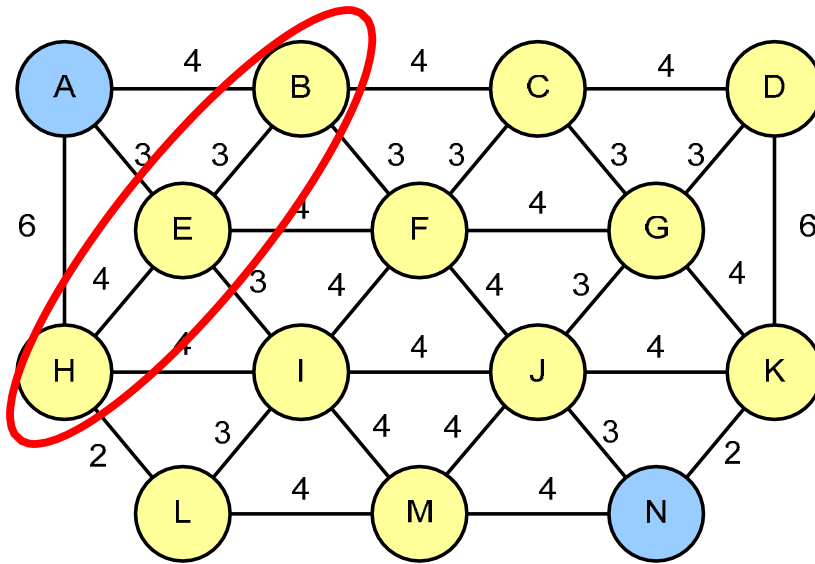


Legend  operating cost

Heuristics

A = 14	H = 8
B = 10	I = 5
C = 8	J = 2
D = 6	K = 2
E = 8	L = 6
F = 7	M = 2
G = 6	N = 0

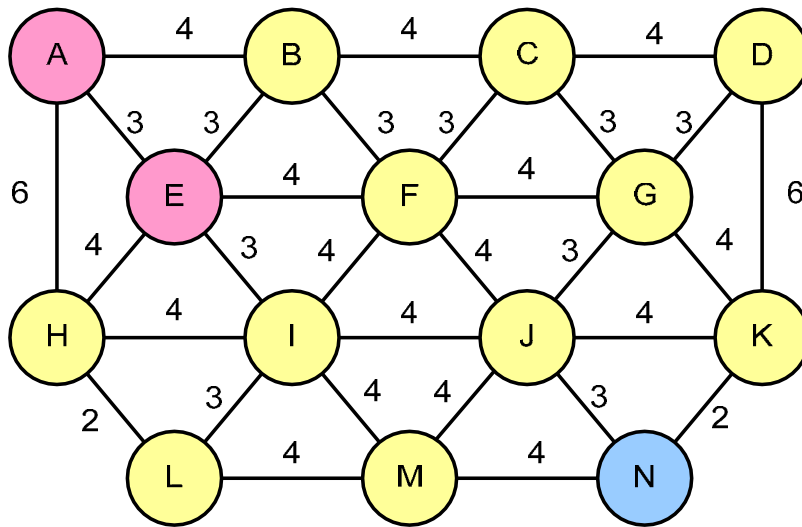
A*: Example (2/6)



Heuristics

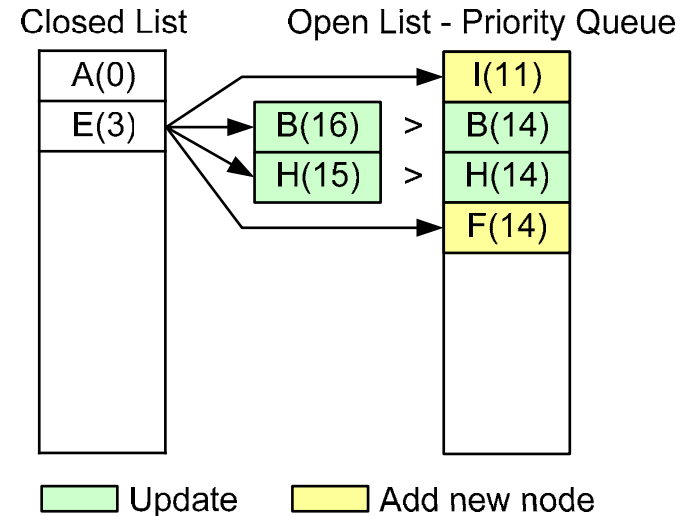
A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
 H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

A*: Example (3/6)



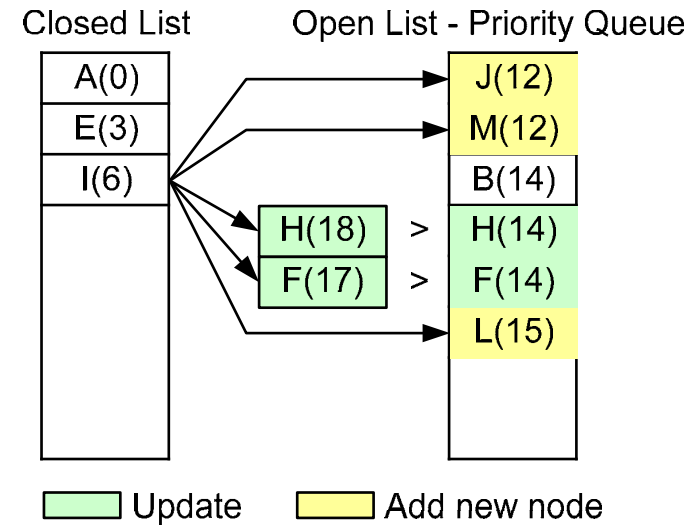
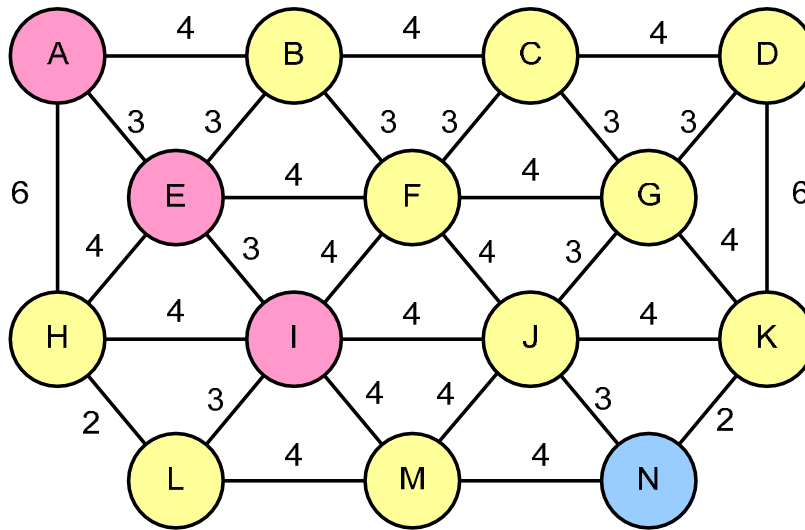
Heuristics

A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0



Since $A \rightarrow B$ is smaller than $A \rightarrow E \rightarrow B$, the f-cost value of B in an open list needs not be updated

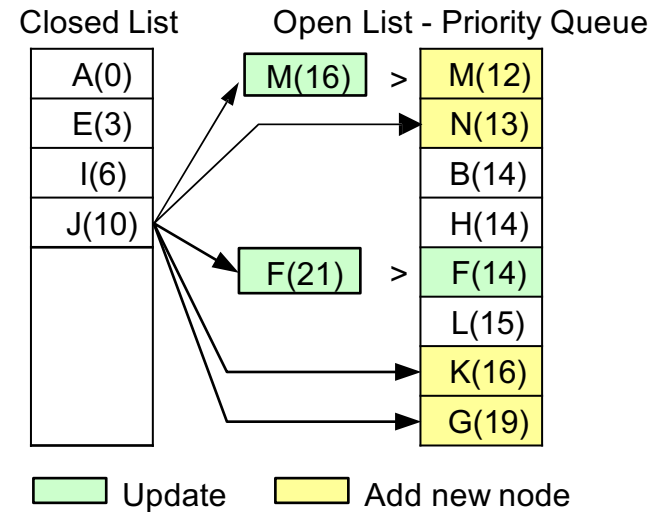
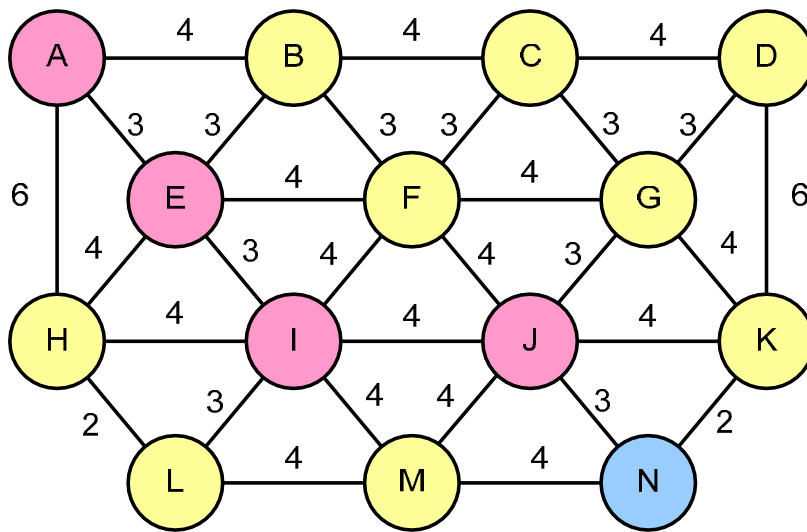
A*: Example (4/6)



Heuristics

A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

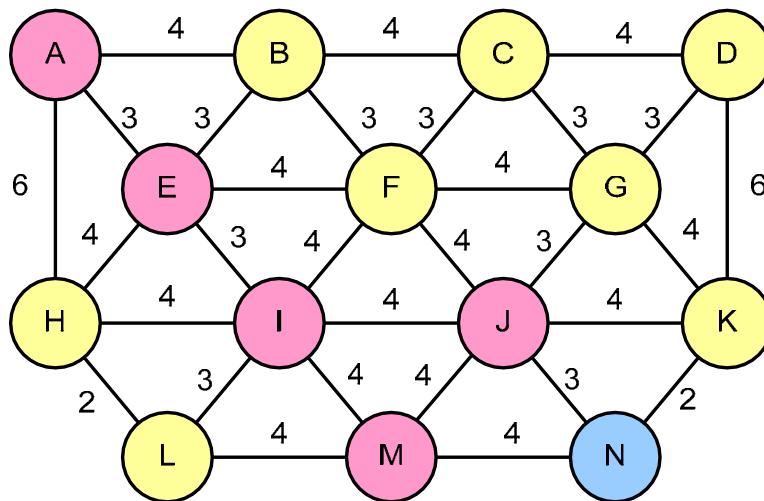
A*: Example (5/6)



Heuristics

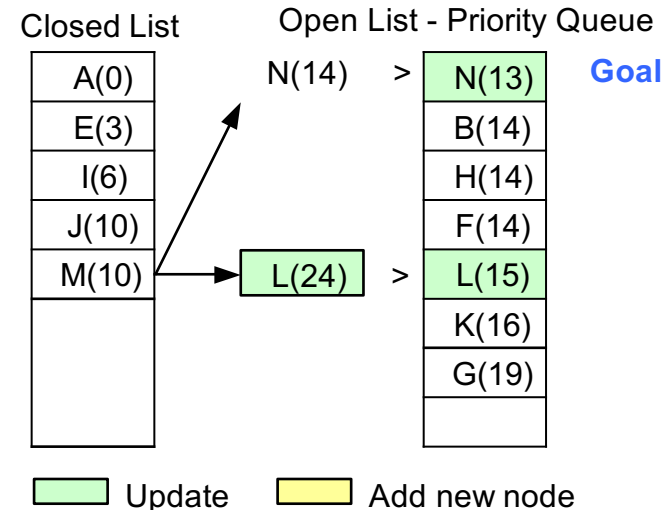
A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

A*: Example (6/6)



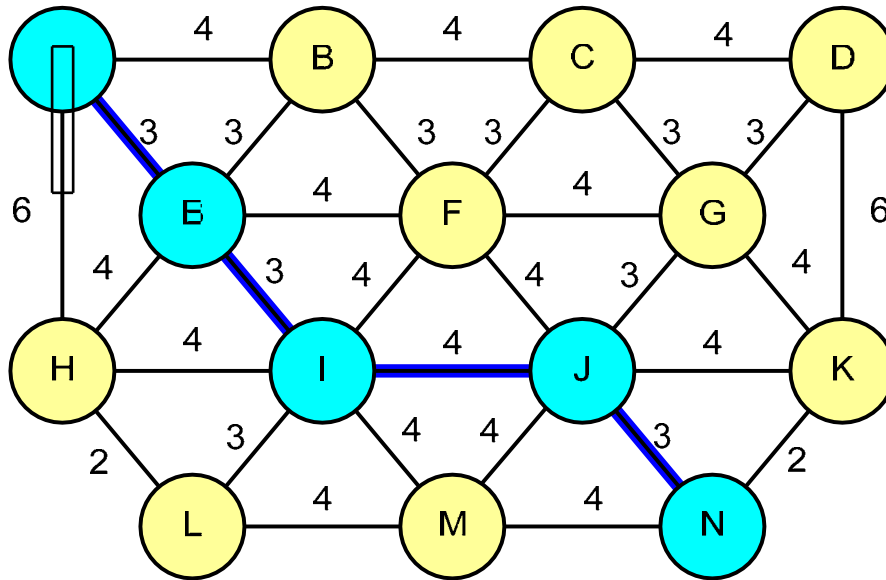
Heuristics

A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0



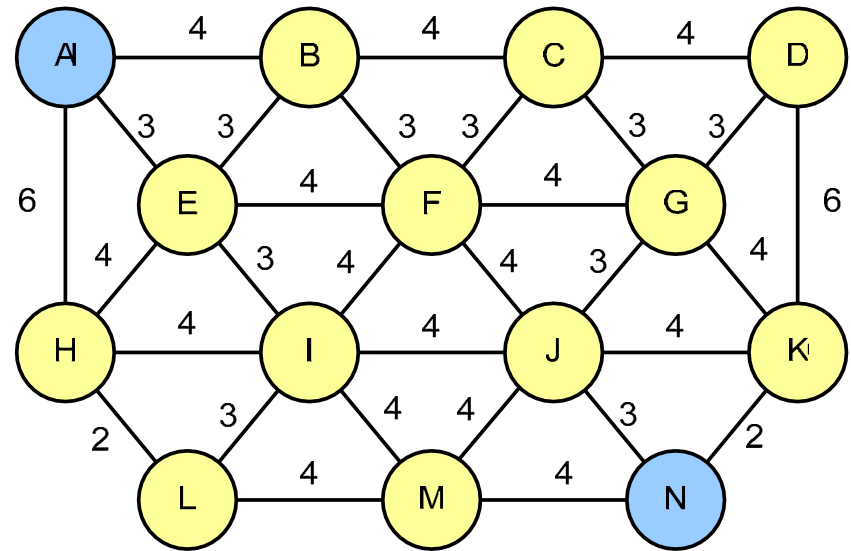
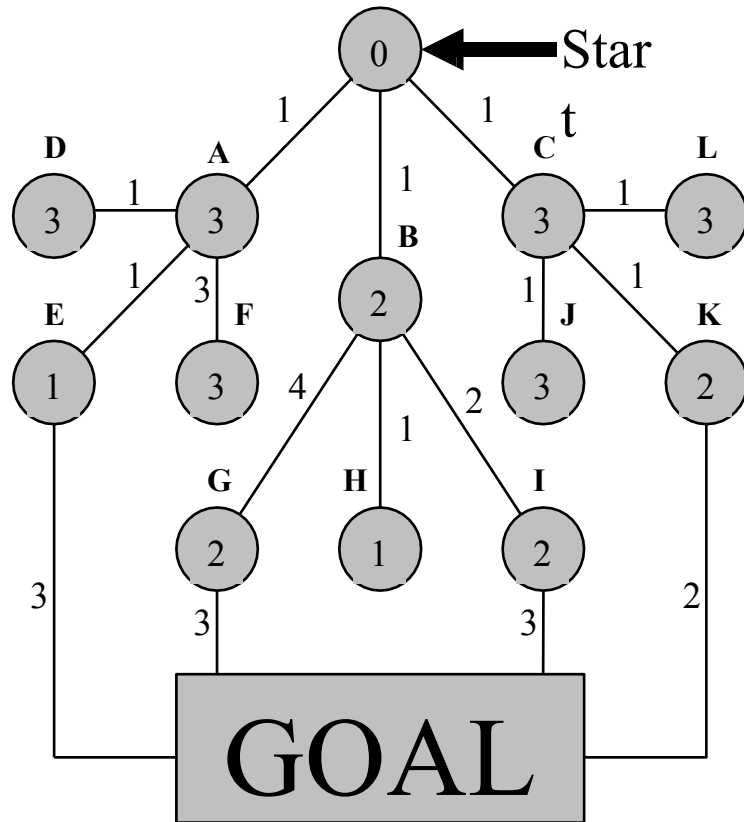
Since the path to N from M is greater than that from J, **the optimal path to N is the one traversed from J**

A*: Example Result

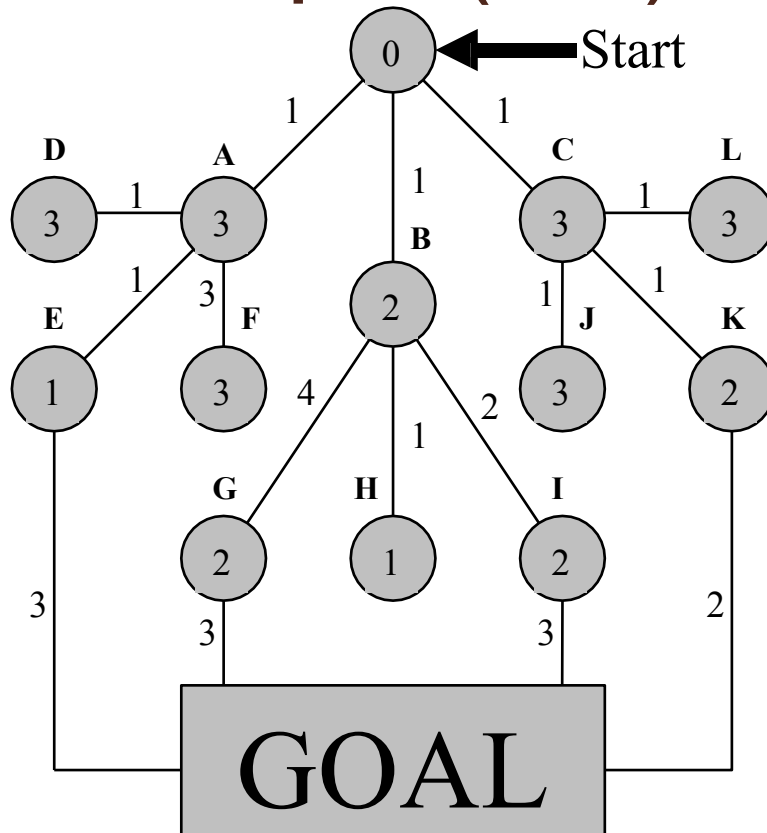


Generate the path
from the goal node
back to the start
node through the
back-pointer
attribute

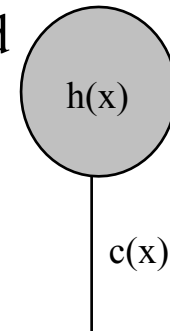
Two Examples Running A*



Example (1/5)



Legend



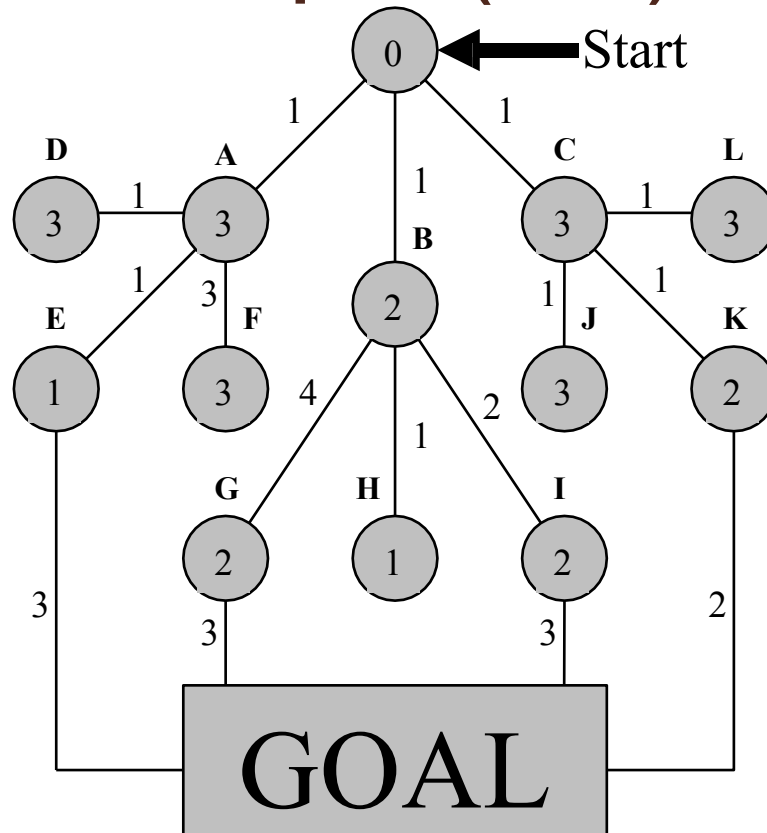
$$\text{Priority} = g(x) + h(x)$$

Note:

$g(x)$ = sum of all previous arc costs, $c(x)$,
from start to x

Example: $c(H) = 2$

Example (2/5)



First expand the start node

B (3)
A (4)
C (4)

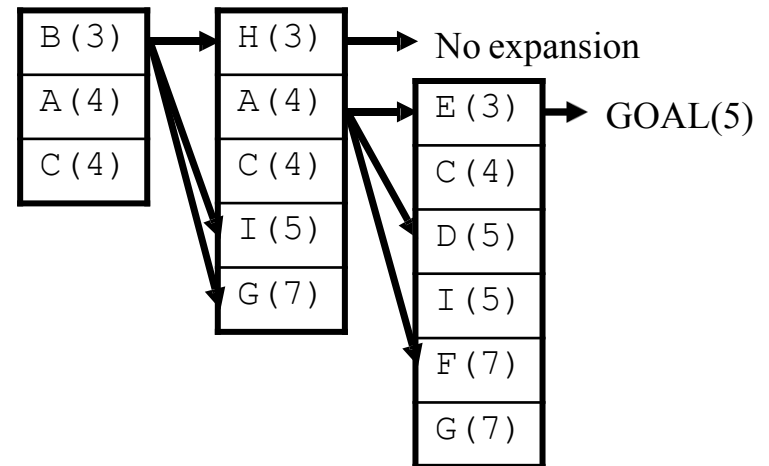
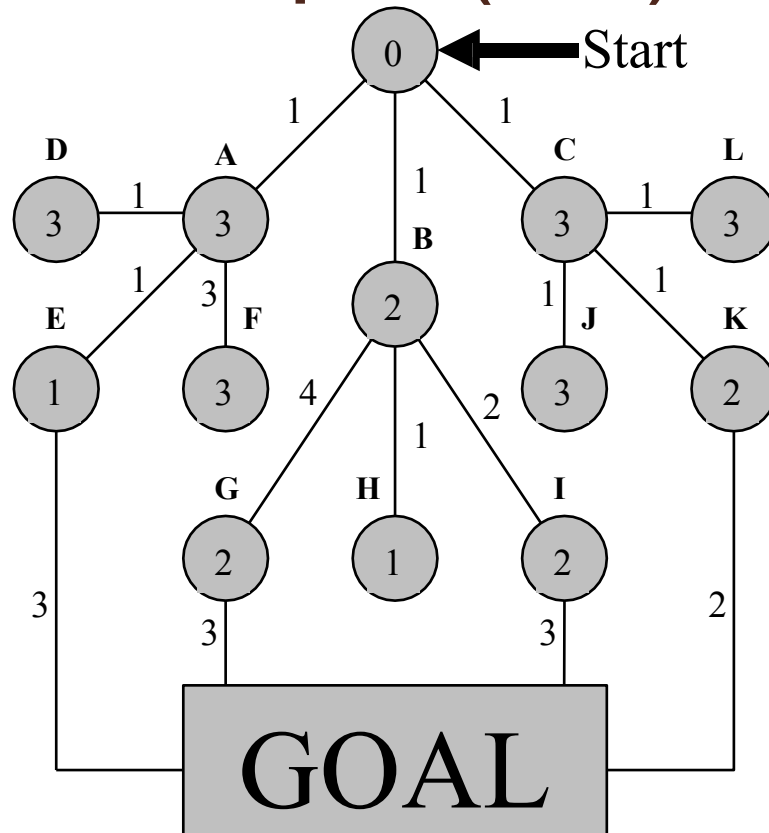
If goal not found,
expand the first node
in the priority queue
(in this case, B)

H (3)
A (4)
C (4)
I (5)
G (7)

Insert the newly expanded
nodes into the priority queue
and continue until the goal is
found, or the priority queue is
empty (in which case no path
exists)

Note: for each expanded node,
you also need a pointer to its respective
parent. For example, nodes A, B and C
point to Start

Example (3/5)

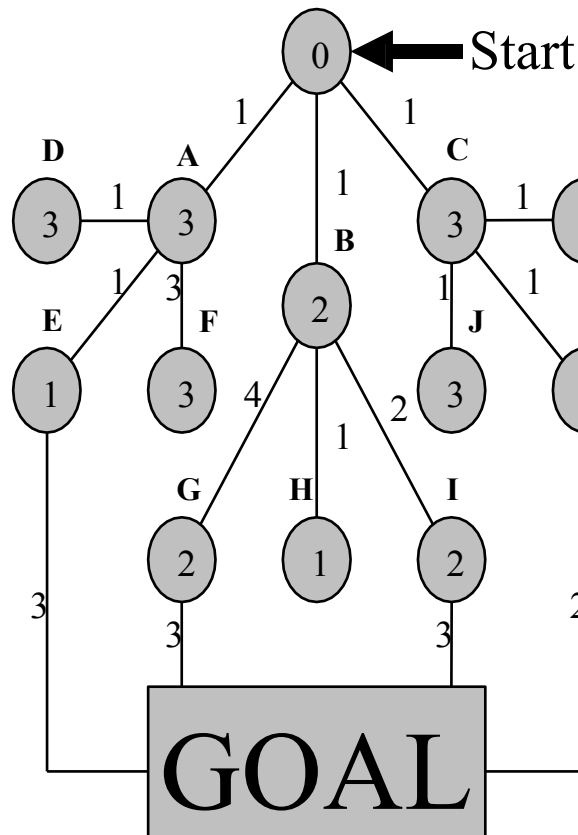


We've found a path to the goal:

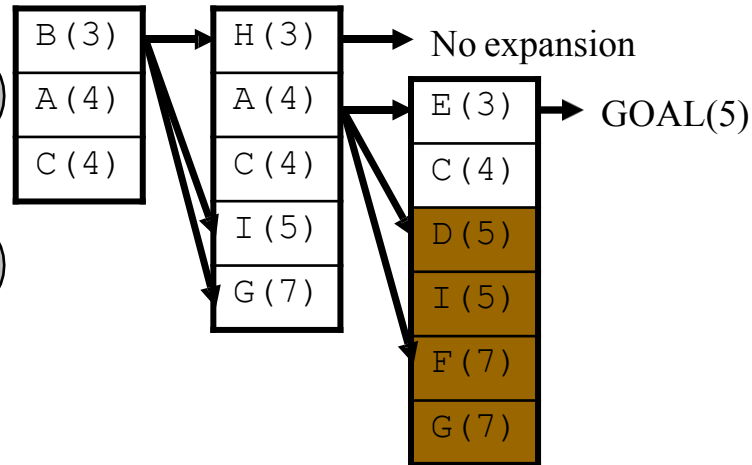
Start => A => E => Goal

(from the pointers)

Are we done?



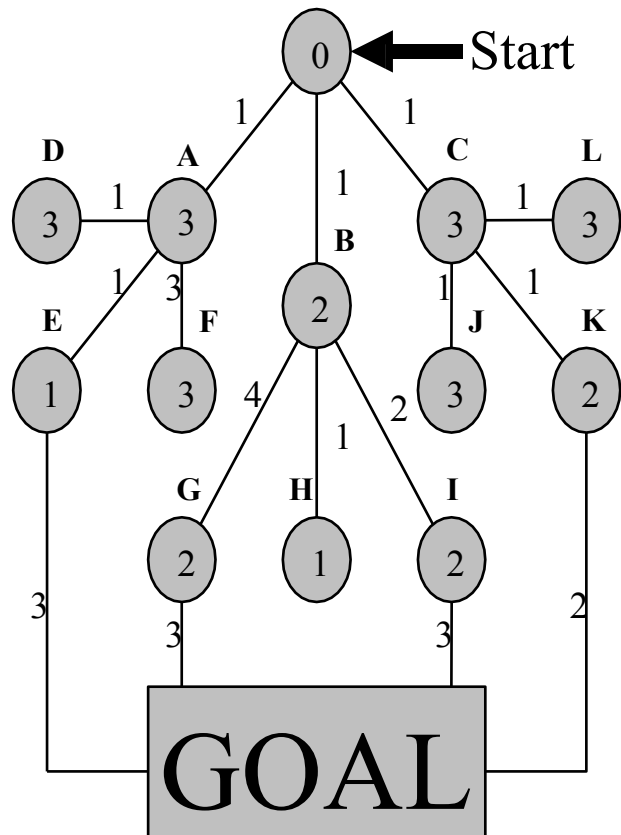
Example (4/5)



There might be a shorter path, but assuming non-negative arc costs, nodes with a lower priority than the goal cannot yield a better path.

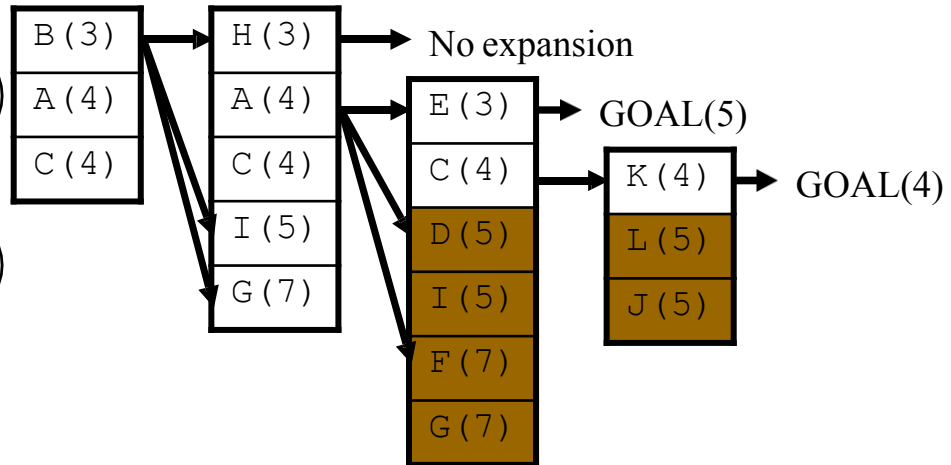
In this example, nodes with a priority greater than or equal to 5 can be pruned.

Why don't we expand nodes with an equivalent priority? (why not expand nodes D and I?)



If the priority queue still wasn't empty, we would continue expanding while throwing away nodes with priority lower than 4.
(remember, lower numbers = higher priority)

Example (5/5)

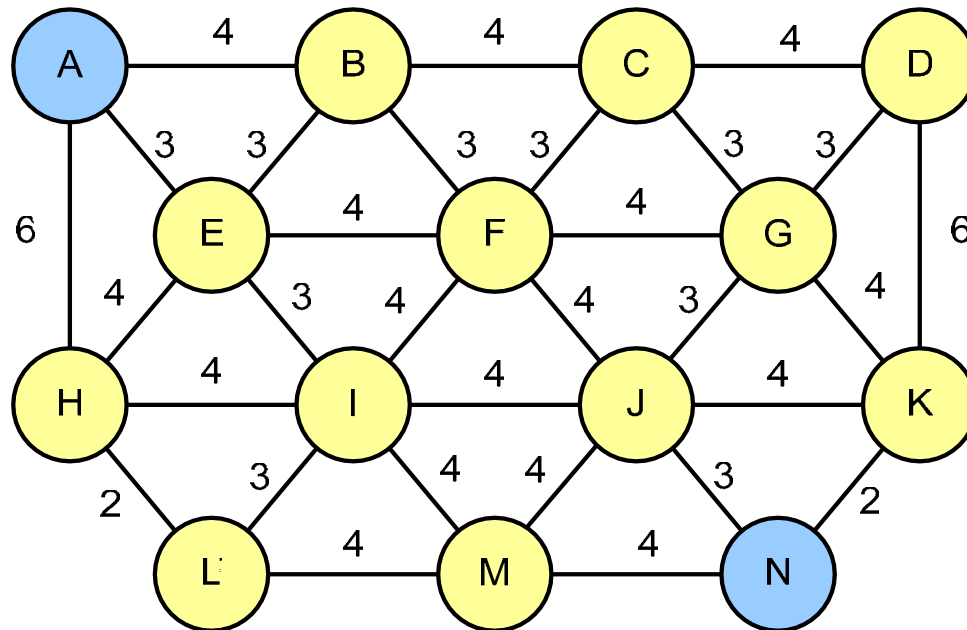



We can continue to throw away nodes with priority levels lower than the lowest goal found.

As we can see from this example, there was a shorter path through node K. To find the path, simply follow the back pointers.

Therefore the path would be:
Start => C => K => Goal

A*: Example (1/6)



Legend  operating cost

Heuristics

A = 14 H = 8

B = 10 I = 5

C = 8 J = 2

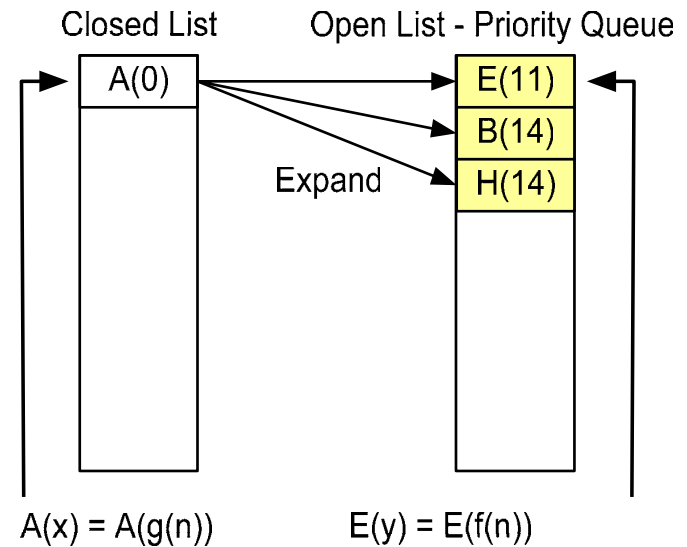
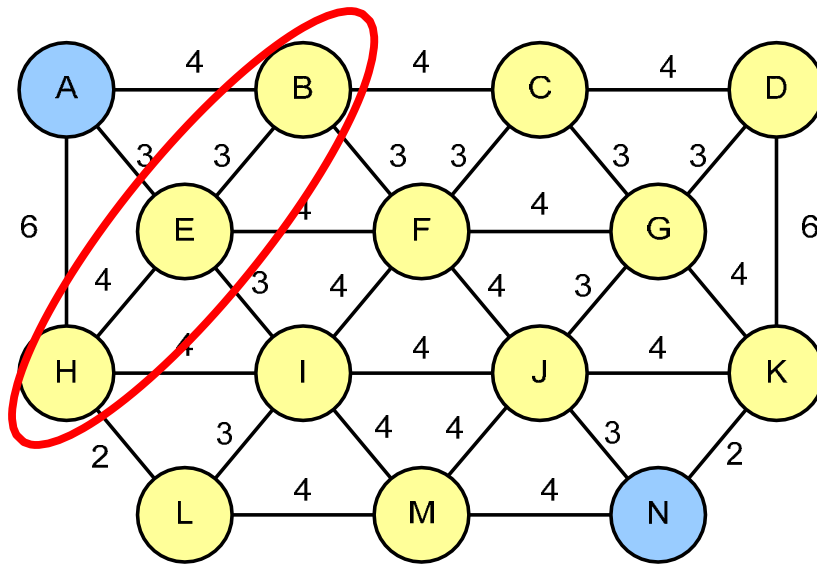
D = 6 K = 2

E = 8 L = 6

F = 7 M = 2

G = 6 N = 0

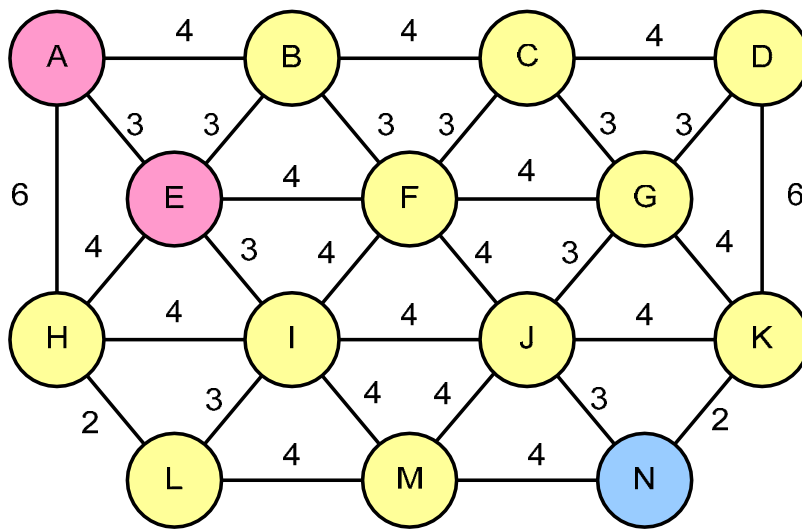
A*: Example (2/6)



Heuristics

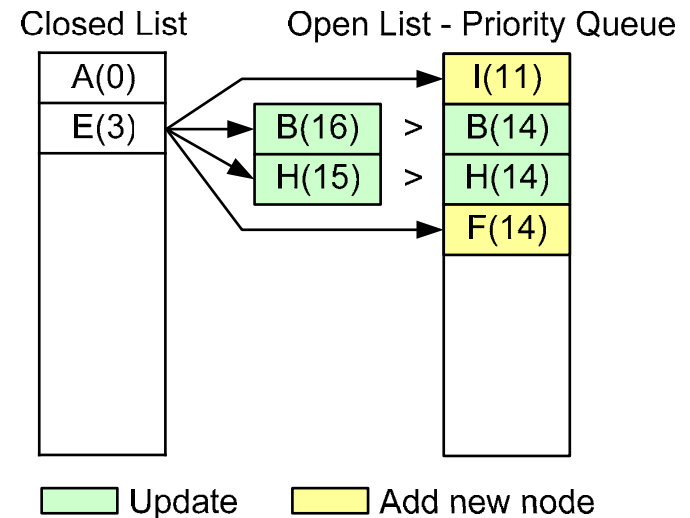
A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
 H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

A*: Example (3/6)



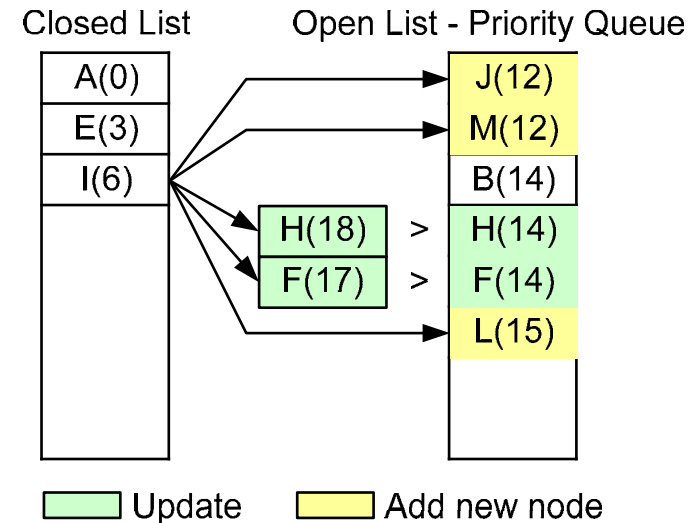
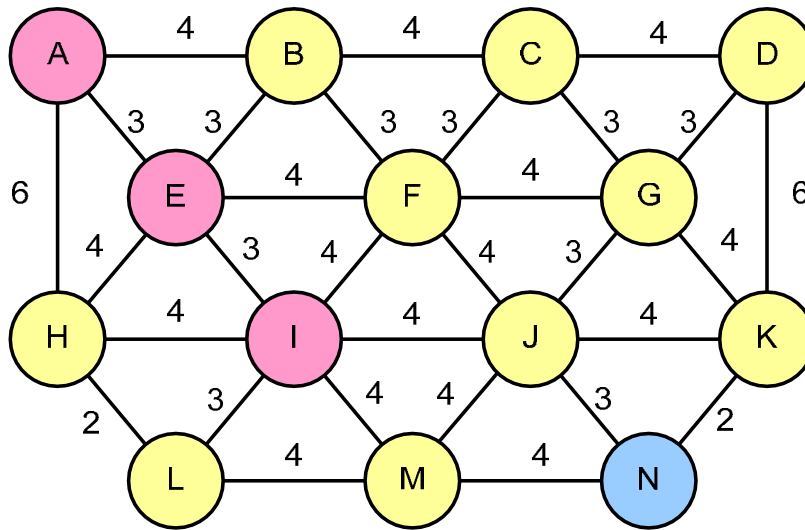
Heuristics

A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0



Since $A \rightarrow B$ is smaller than $A \rightarrow E \rightarrow B$, the f-cost value of B in an open list needs not be updated

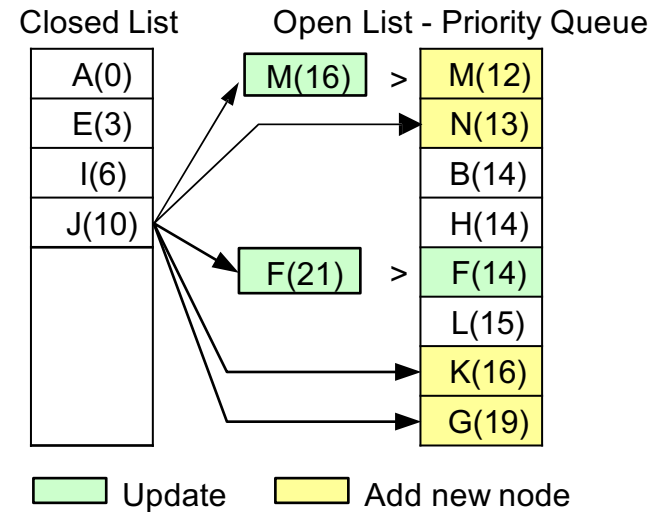
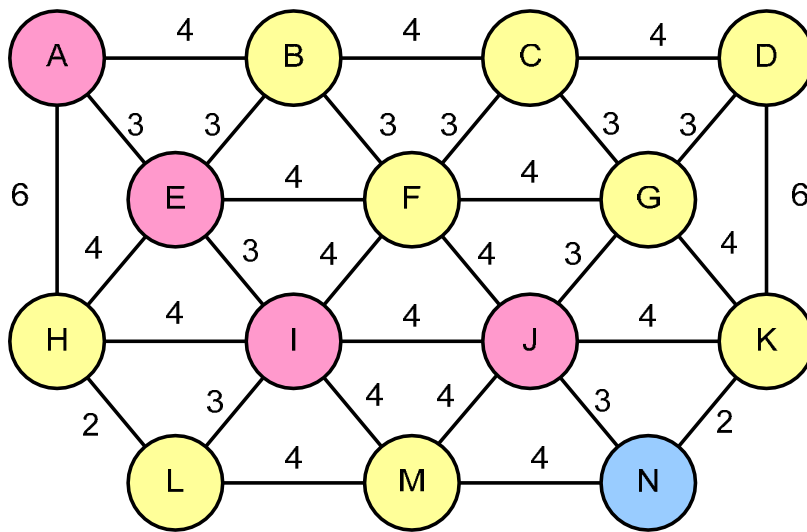
A*: Example (4/6)



Heuristics

A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
 H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

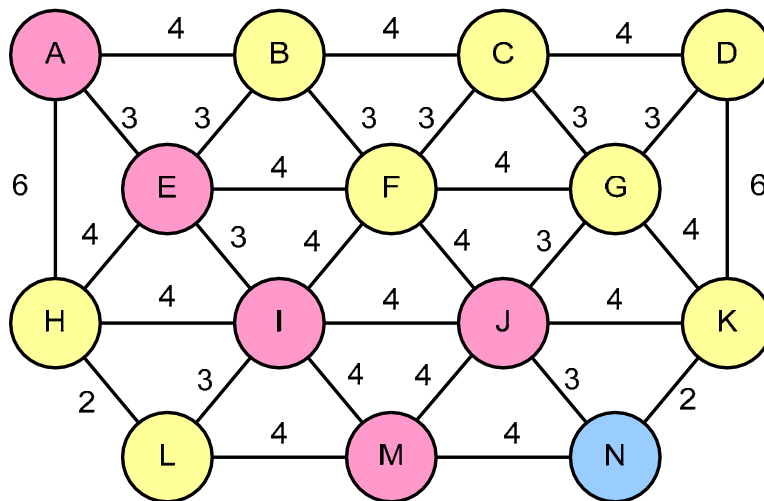
A*: Example (5/6)



Heuristics

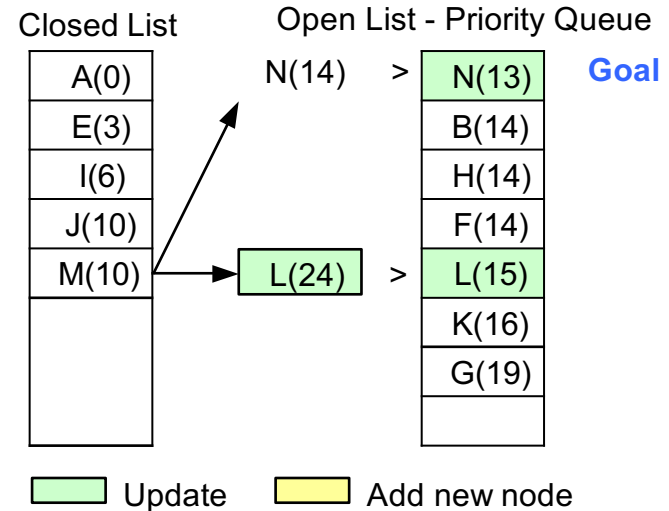
A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

A*: Example (6/6)



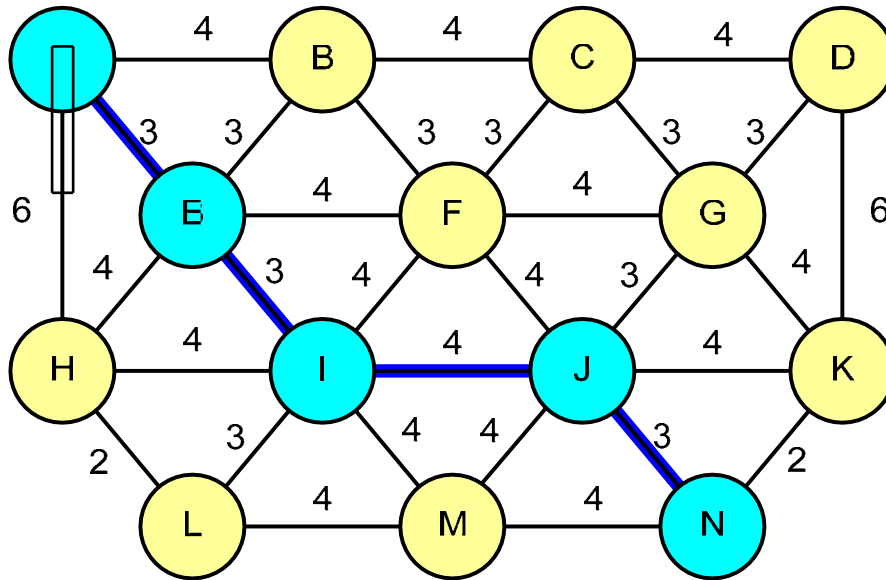
Heuristics

A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0



Since the path to N from M is greater than that from J, **the optimal path to N is the one traversed from J**

A*: Example Result



Generate the path
from the goal node
back to the start
node through the
back-pointer
attribute

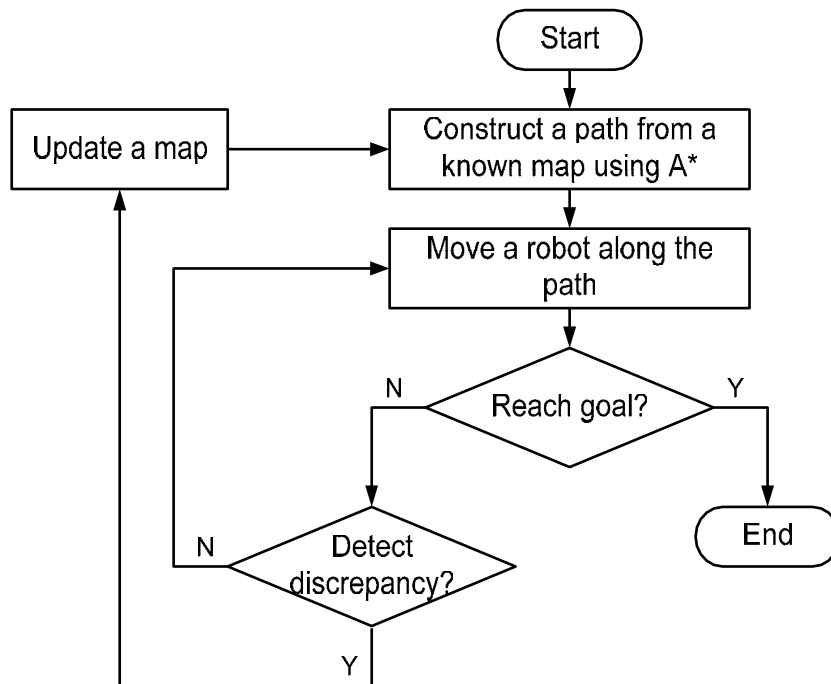
Optimizations (from *Game AI Pro*)

- Precompute all Paths (Roy-Floyd-Warshall)
 - Store in lookup table
 - Compress
- Use an Optimal Search Space Representation
 - Grid, waypoint, navmesh (number of nodes)
- Preallocate All Necessary Memory
- Overestimating the Heuristic (faster, but no optimal path)
- Better Heuristics
- Open List Sorting
- Don't Backtrack during the Search
- Cache Successors (Neighbors)

A* Variants

- $f = w_1 * g + w_2 * h$
 - $w_1 = 1$ and $w_2 = 0$
 - $w_1 = 0$ and $w_2 = 1$
 - $w_1 = 1$ and $w_2 = 1$
 - $w_1 = 1$ and $w_2 = 10$

A* Replanner – unknown map



- Optimal
- Inefficient and impractical in expansive environments – the goal is far away from the start and little map information exists (Stentz 1994)

How can we do better in a partially known and dynamic environment?

D* Search (Stentz 1994)

- Stands for “Dynamic A* Search”
- Dynamic: Arc cost parameters can change during the problem solving process—replanning online
- Functionally equivalent to the A* replanner
- Initially plans using the Dijkstra’s algorithm and allows intelligently caching intermediate data for speedy replanning
- Benefits
 - Optimal
 - Complete
 - More efficient than A* replanner in expansive and complex environments
 - Local changes in the world do not impact on the path much
 - Most costs to goal remain the same
 - It avoids high computational costs of backtracking

D* Search

- Node status

- OPEN and CLOSED, same as A*
- RAISE, indicating its cost is higher than the last time it was on the OPEN list
- LOWER, indicating its cost is lower than the last time it was on the OPEN list

- D* begins by searching backwards from the goal node

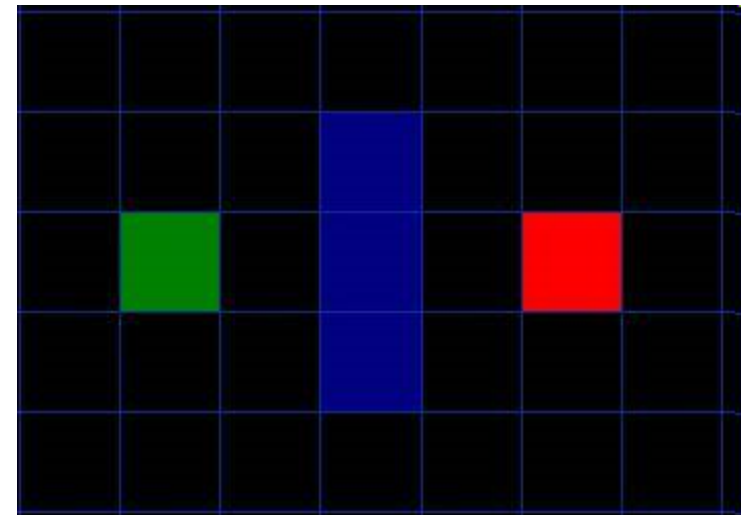
- Why does this help?

- When an unknown obstacle is encountered, propagate the changes

- Only worked on the points which are affected by change of cost, using either RAISE or LOWER labels
- Essentially, updating the costs until the start is reachable again

Running Example

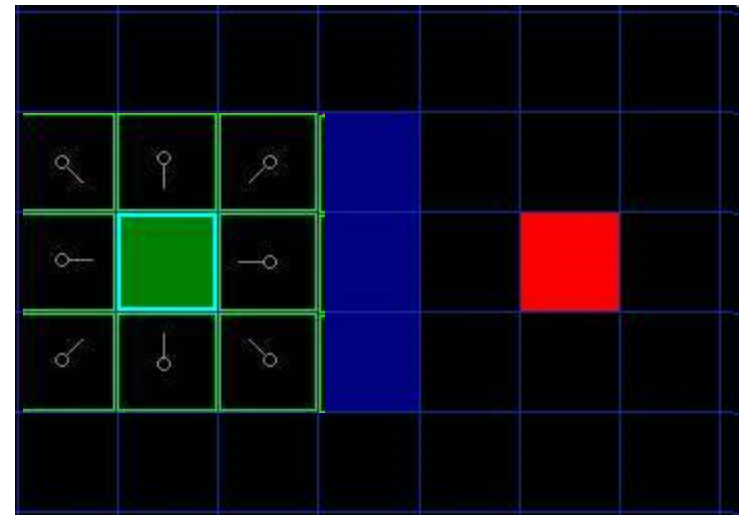
- A unit in the green square wants to move to the red square
 - From here on out, we'll call the squares "nodes" to be consistent with the research literature
- Moving
 - Horizontally or vertically requires 10 movement points
 - Diagonal movement requires 14 movement points
 - Cannot move through blue squares (wall, unwalkable)
- Observations
 - Can't just draw a line between A and B and follow that line
 - Wall in-between
 - It's not ideal to just follow the minimal line between A and B until you hit the wall, then walk along wall
 - Not an optimal path



Pathfinding example.
Green square is starting location, red square is desired goal. Blue is a wall. Black is walkable, blue is unwalkable.

Where to go next...

- Now need to determine which node to consider next
 - Do not want to consider all nodes, since the number of nodes expands exponentially with each square moved (bad)
- Want to pick the node that is **closest** to the destination, and explore that one first
- Intuitive notion of “closest”
 - Add together:
 - The cost we have paid so far to move to a node (“G”)
 - An estimate of the distance to the end point (“H”)



Computing the cost

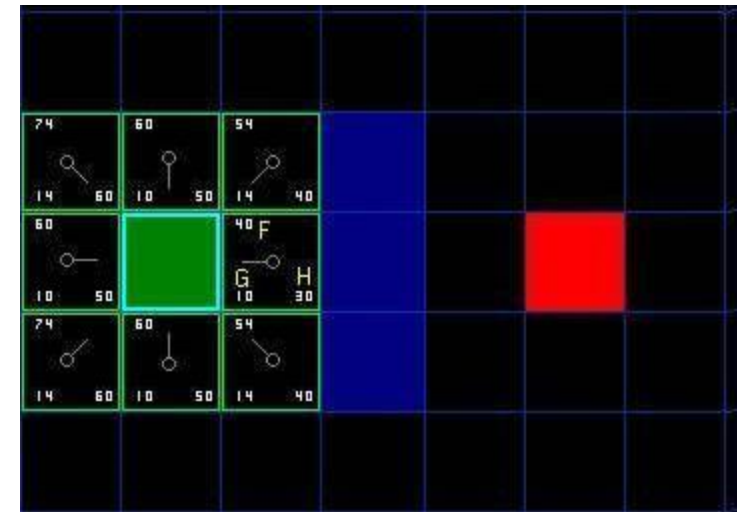
- $F=G+H$
 - Movement cost to get to destination
- G
 - Movement cost to move from start node A to a given node on the grid
 - Following the path generated to get there
 - Add 10 for horizontal/vertical move, 14 for diagonal move
- H
 - Estimated movement cost to move from that given node on the grid to the final destination, node B
 - Known as the **heuristic**
 - A **guess** as to the remaining distance
 - There are many ways to make this guess—this lecture shows just one way

A* Heuristic

- The heuristic is an estimate for the remaining cost to the end point
- This estimate should be as close as possible to the actual cost, but never higher (under estimate).
- For a grid of squares, the heuristic is usually based on the number of squares in between the current node and the end node.
(note this is *different* from the Euclidean distance)

Computing H using Manhattan method

- Manhattan method
 - Calculate the total number of nodes moved horizontally/vertically to reach the target node (B) from the current square
 - Ignore diagonal movement, and ignore obstacles that may be in the way.
 - Multiply total # of nodes by 10, the cost for moving one node horizontally/vertically.
 - Called Manhattan method because it is like calculating the number of city blocks from one place to another, where you can't cut across the block diagonally.



After first step, showing costs for each node.

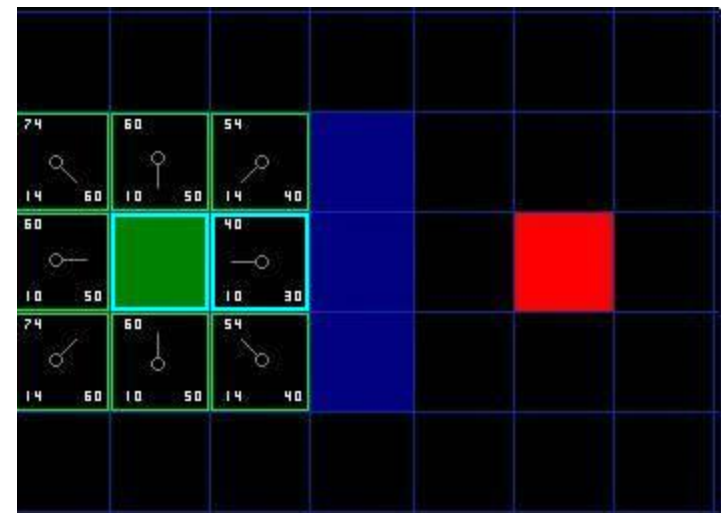
G = movement cost to reach the current node (lower left)

H = Manhattan estimate of cost to reach the destination from node (lower right)

F = G + H (total cost)

Continuing the search

- Pick the node with lowest F value
 - Drop it from open list, add to closed list
 - Check adjacent nodes
 - Add those that are walkable, and not already on open list
 - The parent of the added nodes is the current node
- If an adjacent nodes is already on the open list, check to see if this path to that node is a better one.
 - Check if G score for that node is lower if we use the current node to get there. If not, don't do anything.
 - On the other hand, if G cost of new path is lower change the parent of the adjacent square to the selected square
 - (in the diagram, change the direction of the pointer to point at the selected square)
 - Finally, recalculate both the F and G scores of that square.

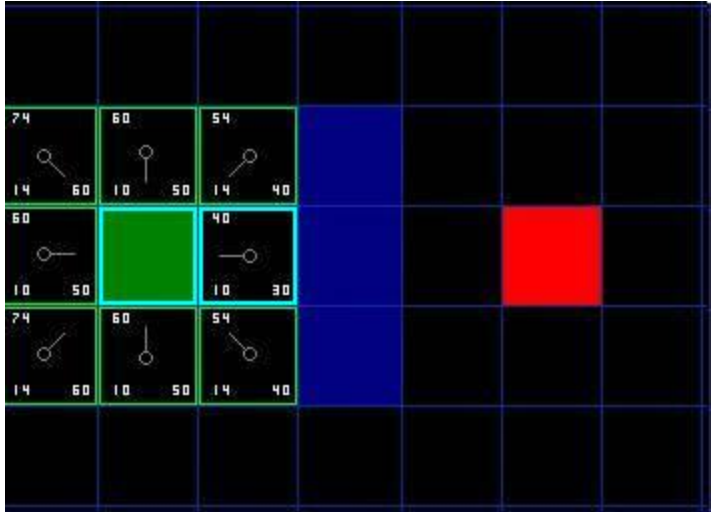


After picking node with lowest F value. Node is in blue, indicating it is now in closed list.

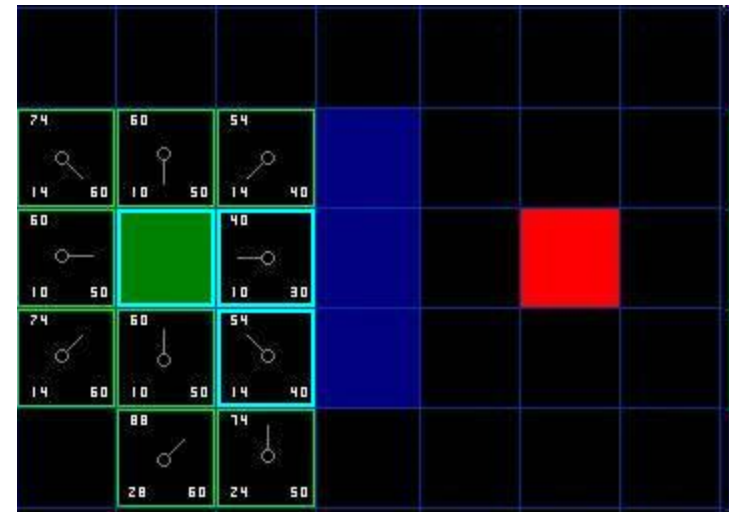
Example of needing to update parent and G and F values...

		14+70					
		10+60	28+50				
		14+50	24+40				
		24+40	28+30				

Example

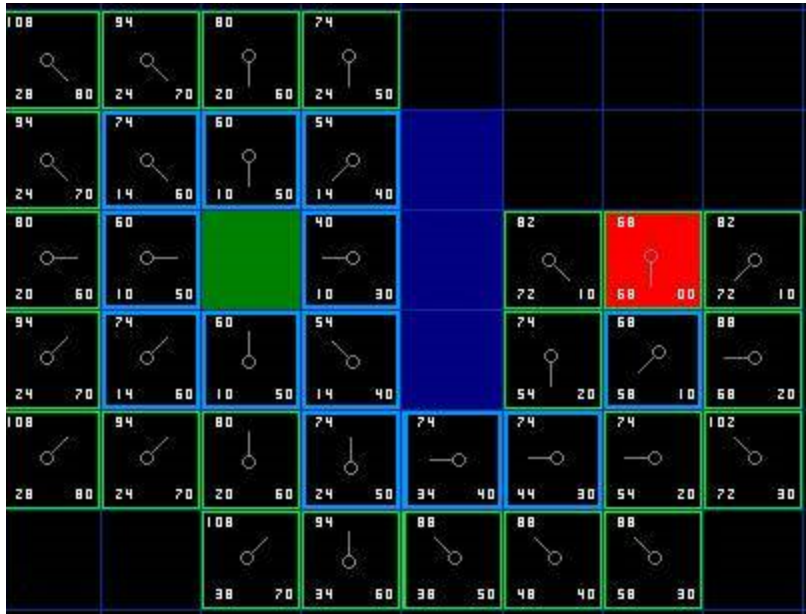


Consider node to right. Check adjacent squares. Ignore starting point (on closed list). Ignore walls (unwalkable). Remaining nodes already on open list, so check to see if it is better to reach those nodes via the current node (compare G values). G score is always higher going through current node. Need to pick another node.

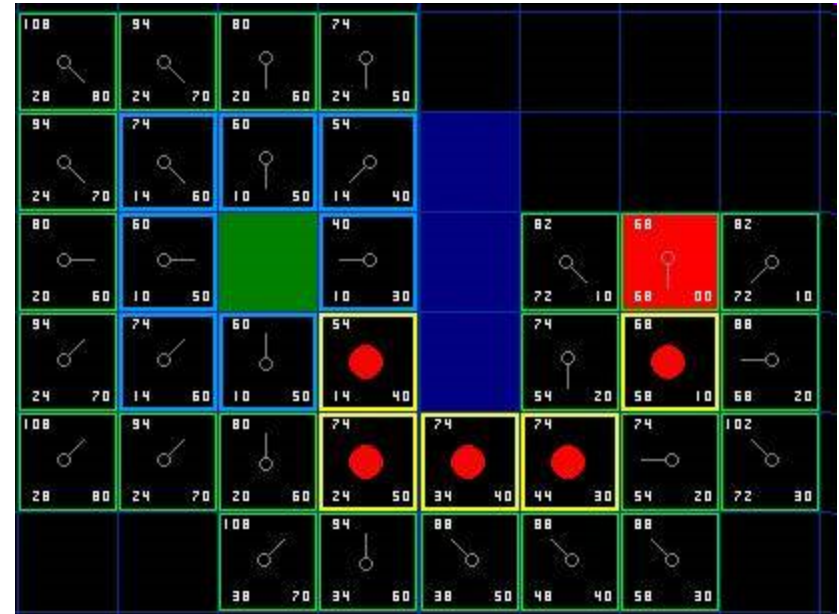


Go through list of open nodes, and pick one with lowest F cost. There are two choices with equal F cost. Pick the lower one (could have picked upper one). Check adjacent nodes. Ignore walls and closed list. Ignore diagonal under wall (game-specific corner rule). Add two new squares to open list (below and below diagonal). Check G value of path via remaining square (to left) – not a more efficient path.

Example continued



Repeat process until target node is added to the closed list. Results in situation above.



To determine final path, start at final node, and move backwards from one square to the next, following parent/child relationships (follow arrows).

A*

- What are the steps of the A* algorithm?

Some issues

- Collision avoidance of other agents
 - Need to avoid other units in the game
 - One approach: make squares of non-moving units unwalkable
 - Can penalize nodes on movement paths of other units
- Variable terrain cost
 - In many games, some terrain costs more to move over than others (mountains, hills, swamps)
 - Change movement costs for these terrains
 - But, can cause units to preferentially choose low-cost movement paths
 - Example: units always go through mountain pass, and then get whacked by big unit camping at exit
 - Add influence cost to nodes where many units have died

More issues

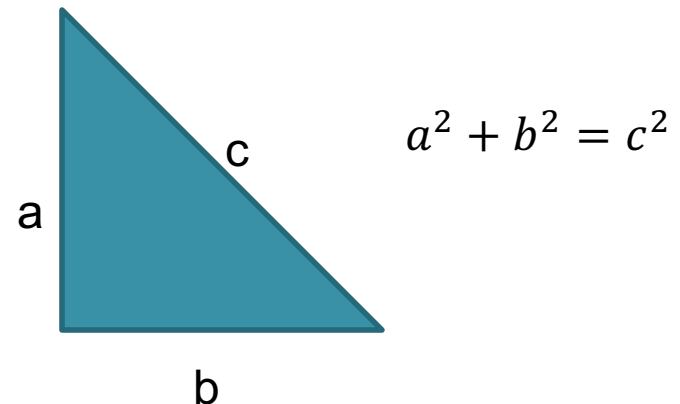
- Unexplored areas
 - Pathfinding can make units “too smart” since their movement implies knowledge of unmapped areas of the map
 - Have a knownWalkable array for explored areas
 - Mark rest of the map as walkable, until proven otherwise
- Memory/speed
 - Algorithm can use a lot of memory
 - Can also slow things down
 - Spread A* computations over many game ticks

Possible Heuristics

- Straight-line (Euclidean) distance
 - Underestimates distances (assumes paths can take any angle)
- Manhattan distance
 - Overestimates distances
- Octile distance (45 and 90 degree angles allowed)
 - Corresponds to movement in the world
 - $\text{Max}(\text{deltaX}, \text{deltaY}) + 0.41 * \text{min}(\text{deltaX}, \text{deltaY})$, assuming diagonal movement has cost of 1.41

A* Considerations

- You must define how diagonal squares work with map grids. If movement is continuous, consider multiplying the cost of diagonal movement by $\sqrt{2}$
- Defining an upper bound cutoff can prevent the search from taking too long.



Examples

- <http://movingai.com/astar-var.html>
- <http://www.cs.du.edu/~sturtevant/papers.html>

Start simple

- What do you need to represent?
 - Data structures?
- What are the simplest methods we'll need?
 - Accessor methods
 - Adjacency?
 - Occupancy?
- What are the classes and how do they interact?