

Game AI: Path Planning

Jeff Wilson

Planning

- Part of intelligence is the ability to plan
- Move to a goal
 - A Goal State
- Represent the world as a set of **States**
 - Each configuration is a separate state
- Change state by applying **Operators**
 - An Operator changes configuration from one **state** to another **state**

Path Planning Algorithms

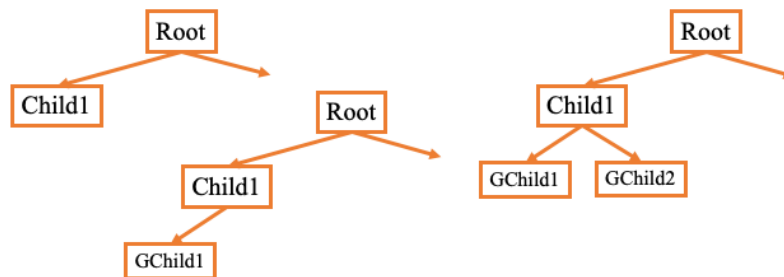
- Must **Search** the state space to move NPC to goal state
- Computational Issues:
 - Completeness
 - Will it find an answer if one exists?
 - Time complexity
 - Space complexity
 - Optimality
 - Will it find the best solution

Search Strategies

- **Blind search**
 - No domain knowledge
 - Only goal state is known
- **Heuristic search**
 - Domain knowledge represented by **heuristic rules**
 - Heuristics drive low-level decisions
 - Video games provide domain knowledge that can be leveraged!

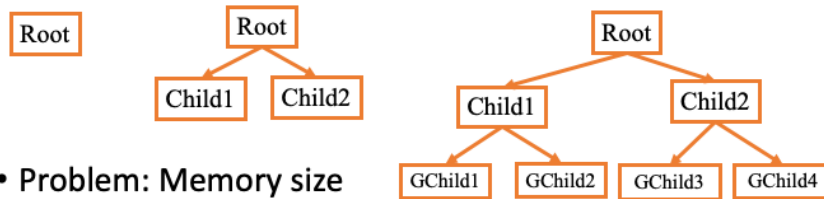
Depth First Search

- Always expand the node that is deepest in the tree
- Not best solution (if you stop at first found)



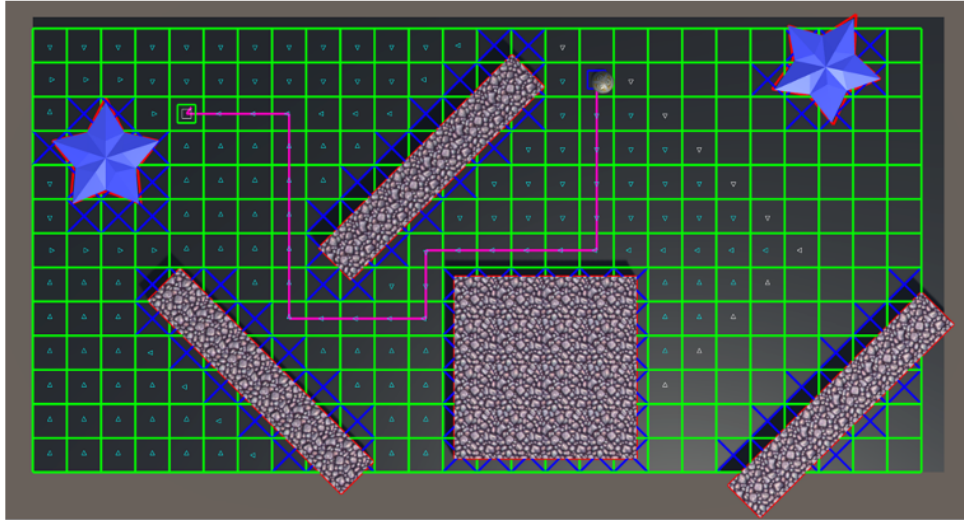
Breadth-First Search

- Expand **Root node**
 - Expand all **Root node's children**
 - Expand all **Root node's grandchildren**



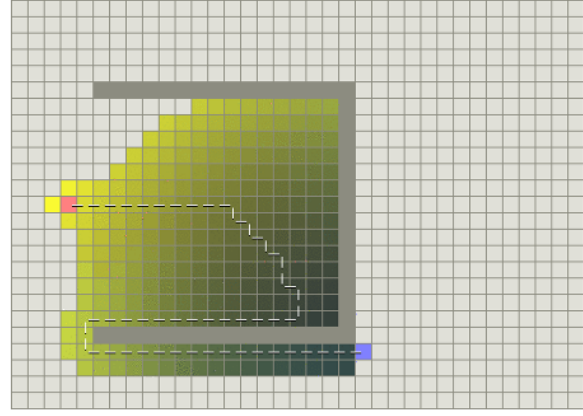
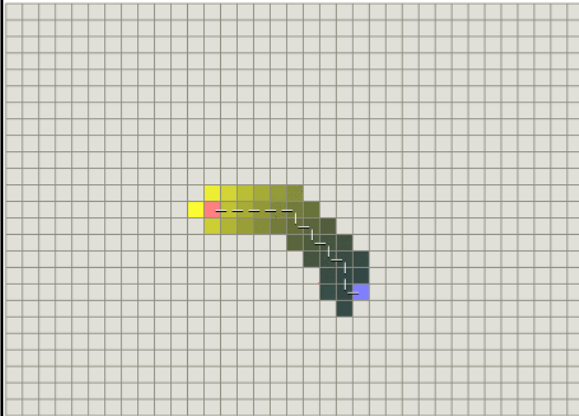
- Problem: Memory size

Breadth First



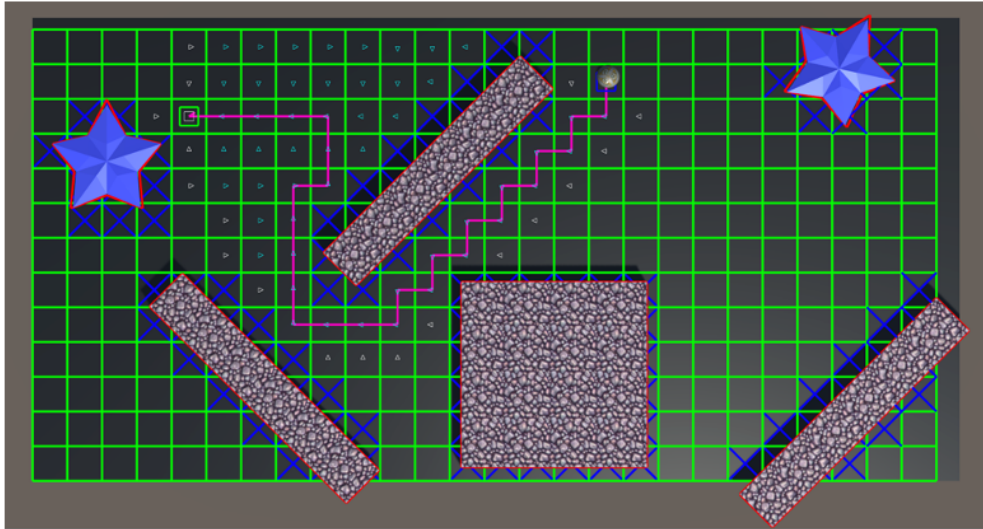
Greedy Best-First-Search

- Expand nodes *closest* to goal <- **Heuristic!**

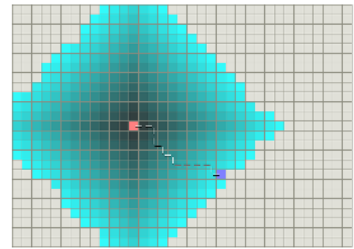


<http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>

NOT guaranteed to find best solution

[illegible]

Dijkstra's Algorithm



Dijkstra's Algorithm

Note that Dijkstra's Algorithm gives no different a result than BFS. This is because on the grid, all edge distances are equal.

```

1 class Graph:
2     # An array of connections outgoing from the given node.
3     function getConnections(fromNode: Node) -> Connection[]
4
5 class Connection:
6     # The node that this connection came from.
7     fromNode: Node
8
9     # The node that this connection leads to.
10    toNode: Node
11
12    # The non-negative cost of this connection.
13    function getCost() -> float
14
15
16 function pathfindDijkstra(graph: Graph,
17     start: Node,
18     end: Node) -> Connection[]:
19     # This structure is used to keep track of the information we need
20     # for each node.
21     class NodeRecord:
22         node: Node
23         connection: Connection
24         costSoFar: float
25
26     # Initialize the record for the start node.
27     startRecord = new NodeRecord()
28     startRecord.node = start
29     startRecord.connection = null
30     startRecord.costSoFar = 0
31
32     # Initialize the open and closed lists.
33     open = new PathfindingList()
34     open += startRecord
35     closed = new PathfindingList()
36
37     # Iterate through processing each node.
38     while length(open) > 0:
39         # Find the smallest element in the open list.
40         current = NodeRecord = open.smallestElement()
41
42         # If it is the goal node, then terminate.
43         if current.node == goal:
44             break
45
46         # Otherwise get its outgoing connections.
47         connections = graph.getConnections(current)
48
49         # Loop through each connection in turn.
50         for connection in connections:
51             # Get the cost estimate for the end node.
52             endNode = connection.getToNode()
53
54             endNodeCost = current.costSoFar + connection.getCost()
55
56             # Skip if the node is closed.
57             if closed.contains(endNode):
58                 continue
59
60             # .. or if it is open and we've found a worse route.
61             else if open.contains(endNode):
62                 # Here we find the record in the open list
63                 # corresponding to the endNode.
64                 endNodeRecord = open.find(endNode)
65                 if endNodeRecord.cost <= endNodeCost:
66                     continue
67
68             # Otherwise we know we've got an unvisited node, so make a
69             # record for it.
70             else:
71                 endNodeRecord = new NodeRecord()
72                 endNodeRecord.node = endNode
73
74                 # We're here if we need to update the node. Update the
75                 # cost and connection.
76                 endNodeRecord.cost = endNodeCost
77                 endNodeRecord.connection = connection
78
79                 # And add it to the open list.
80                 if not open.contains(endNode):
81                     open += endNodeRecord
82
83             # We've finished looking at the connections for the current
84             # node, so add it to the closed list and remove it from the
85             # open list.
86             open -= current
87             closed += current
88
89             # We're here if we've either found the goal, or if we've no more
90             # nodes to search, find which.
91             if current.node != goal:
92                 # We've run out of nodes without finding the goal, so there's
93                 # no solution.
94                 return null
95
96             else:
97                 # Compile the list of connections in the path.
98                 path = []
99
100                # Work back along the path, accumulating connections.
101                while current.node != start:
102                    path += current.connection.getFromNode()
103                    current = current.connection.getFromNode()
104
105                # Reverse the path, and return it.
106                return reverse(path)

```

Dijkstra

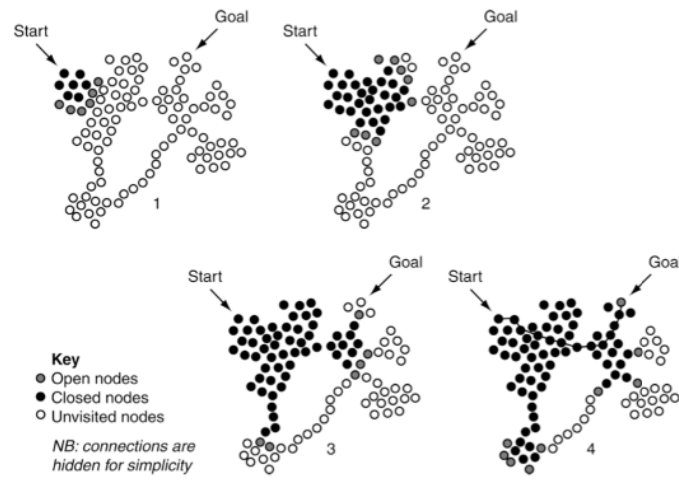
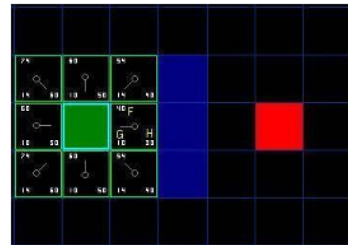


Figure 4.12: Dijkstra in steps

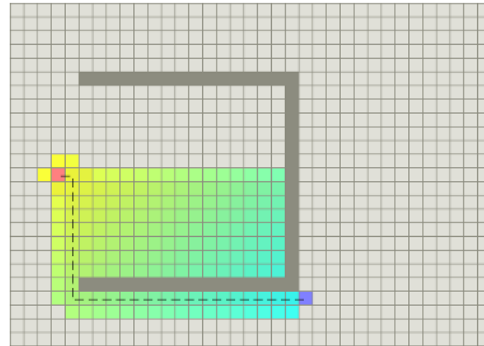
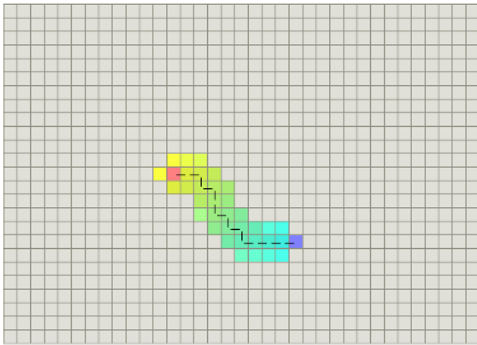
A* Search

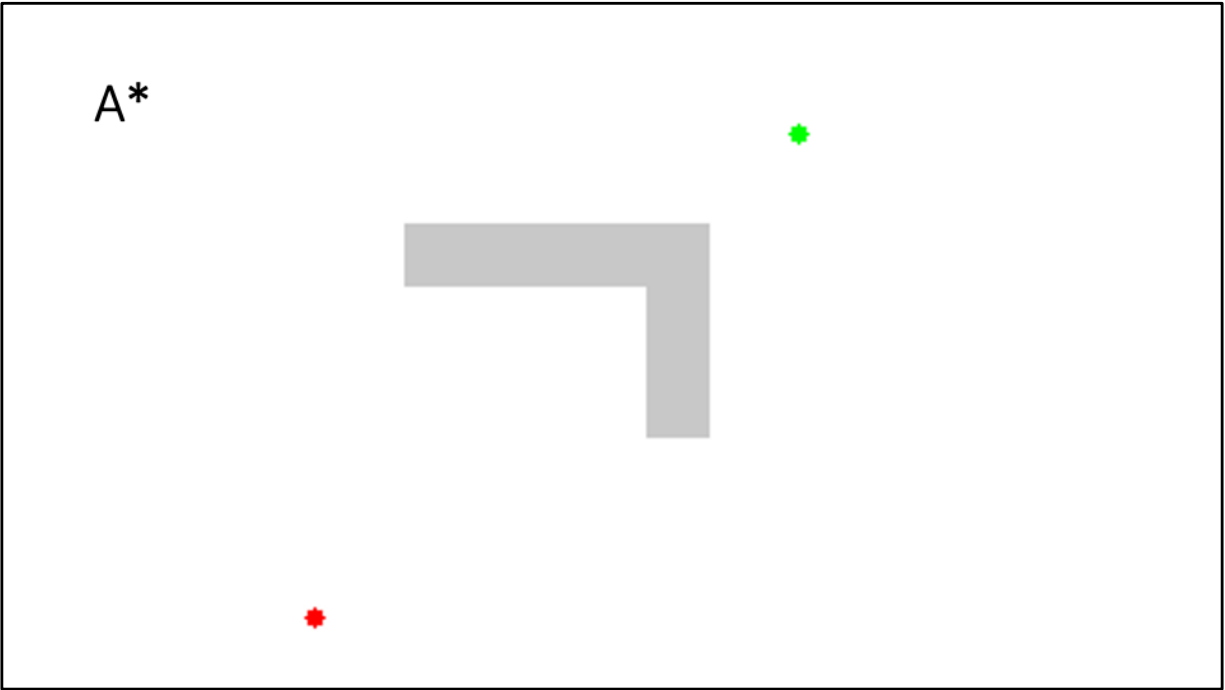
- Minimize sum of costs
- $g(n) + h(n)$
 - Cost so far + heuristic to goal
- Guaranteed to find best solution
 - So long as $h(n)$ does **not overestimate cost** (and solution exists) – will revisit
- Example $h(n)$
 - Euclidean distance



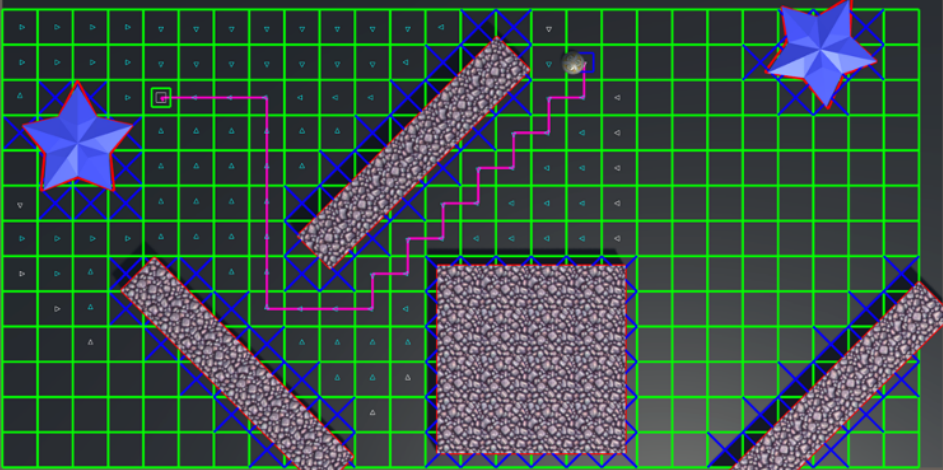
A* Search

- Fails only when there is no solution
 - Avoid searching the whole space





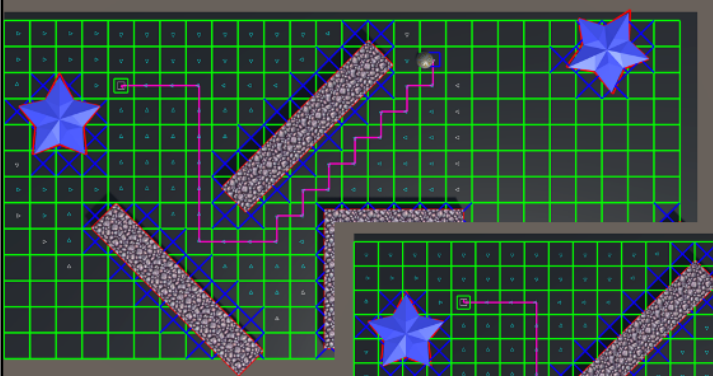
A*



The diagram illustrates the A* search algorithm on a 2D grid. The grid is composed of green squares, with obstacles represented by grey, textured rectangles. A red star marks the start position, and a blue star marks the goal position. A red path shows the sequence of nodes visited by the algorithm, starting from the red star and ending at the blue star. The path is marked with red 'X' symbols at each step. The grid also features blue 'X' symbols and small blue triangles, likely representing the heuristic and priority values for each node.

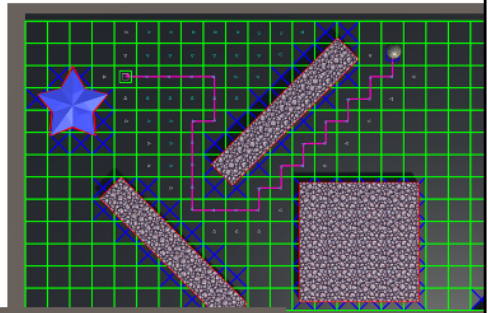
More people visited than people left, first search, but better path

Comparison

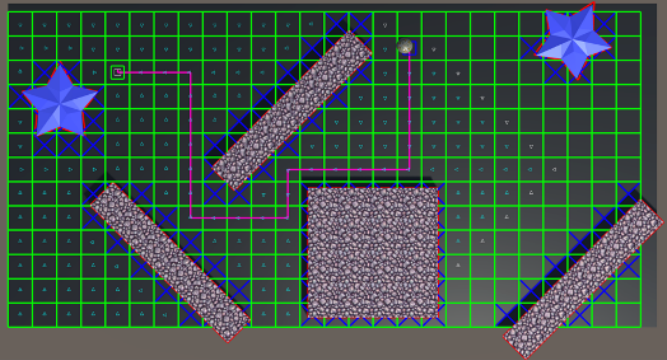


A*

Dijkstra's
(same
result as
BFS)



Greedy
Best
First



```

1 function pathfindAStar(graph: Graph,
2   start: Node,
3   end: Node,
4   heuristic: Heuristic
5   ) -> Connection[]:
6   # This structure is used to keep track of the
7   # information we need for each node.
8   class NodeRecord:
9     node: Node
10    connection: Connection
11    costSofar: float
12    estimatedTotalCost: float
13
14   # Initialize the record for the start node.
15   startRecord = new NodeRecord()
16   startRecord.node = start
17   startRecord.connection = null
18   startRecord.costSofar = 0
19   startRecord.estimatedTotalCost = heuristic.estimate(start)
20
21   # Initialize the open and closed lists.
22   open = new PathfindingList()
23   open += startRecord
24   closed = new PathfindingList()
25
26   # Iterate through processing each node.
27   while length(open) > 0:
28     # Find the smallest element in the open list (using the
29     # estimatedTotalCost).
30     current = open.smallestElement()
31
32     # If it is the goal node, then terminate.
33     if current.node == goal:
34       break
35
36     # Otherwise get its outgoing connections.
37     connections = graph.getConnections(current)
38
39     # Loop through each connection in turn.
40     for connection in connections:
41       # Get the cost estimate for the end node.
42       endNode = connection.getTargetNode()
43       endNodeCost = current.costSofar + connection.getCost()
44
45       # If the node is closed we may have to skip, or remove it
46       # from the closed list.
47       if closed.contains(endNode):
48         # Here we find the record in the closed list
49         # corresponding to the endNode.
50         endNodeRecord = closed.find(endNode)
51
52         # If we didn't find a shorter route, skip.
53         if endNodeRecord.costSofar <= endNodeCost:
54           continue
55
56         # Otherwise remove it from the closed list.
57         closed -= endNodeRecord
58
59         # We can use the node's old cost values to calculate
60         # its heuristic without calling the possibly expensive
61         # heuristic function.
62         endNodeHeuristic = endNodeRecord.estimatedTotalCost -
63           endNodeRecord.costSofar
64
65         # Skip if the node is open and we've not found a better
66         # route.
67         else if open.contains(endNode):
68           # Here we find the record in the open list
69           # corresponding to the endNode.
70           endNodeRecord = open.find(endNode)
71
72           # If our route is no better, then skip.
73           if endNodeRecord.costSofar <= endNodeCost:
74             continue
75
76           # Again, we can calculate its heuristic.
77           endNodeHeuristic = endNodeRecord.cost -
78             endNodeRecord.costSofar
79
80           # Otherwise we know we've got an unvisited node, so make a
81           # record for it.
82           else:
83             endNodeRecord = new NodeRecord()
84             endNodeRecord.node = endNode
85
86             # We'll need to calculate the heuristic value using
87             # the function, since we don't have an existing record
88             # to use.
89             endNodeHeuristic = heuristic.estimate(endNode)
90
91           # We're here if we need to update the node. Update the
92           # cost, estimate and connection.
93           endNodeRecord.cost = endNodeCost
94           endNodeRecord.connection = connection
95           endNodeRecord.estimatedTotalCost = endNodeCost +
96             endNodeHeuristic
97
98           # And add it to the open list.
99           if not open.contains(endNode):
100             open += endNodeRecord
101
102           # We've finished looking at the connections for the current
103
104           # node, so add it to the closed list and remove it from the
105           # open list.
106           open -= current
107           closed += current
108
109           # We're here if we've either found the goal, or if we've no more
110           # nodes to search, find which.
111           if current.node != goal:
112             # We've run out of nodes without finding the goal, so there's
113             # no solution.
114             return null
115           else:
116             # Compile the list of connections in the path.
117
118             path = []
119
120             # Work back along the path, accumulating connections.
121             while current.node != start:
122               path += current.connection
123               current = current.connection.getFromNode()
124
125             # Reverse the path, and return it.
126             return reverse(path)

```

A*

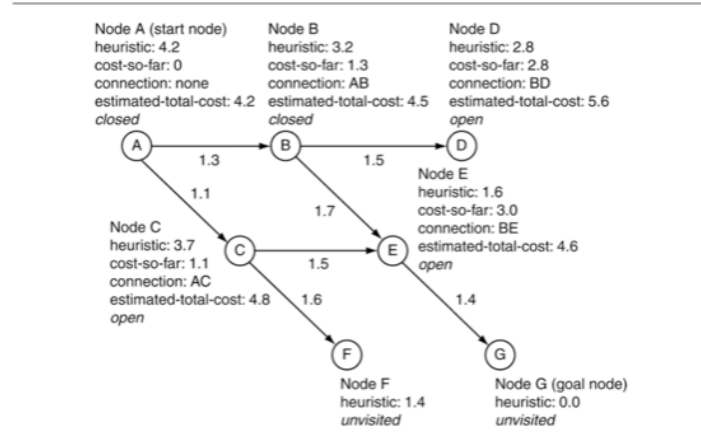


Figure 4.13: A* estimated-total-costs

A*

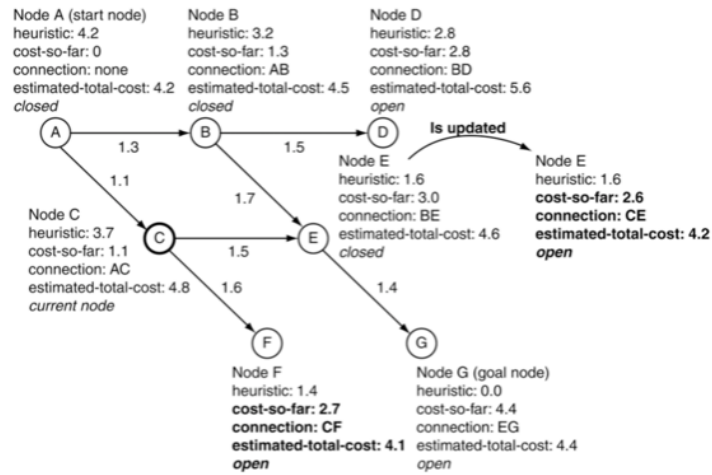


Figure 4.14: Closed node update

Pathfinding List (Open and Closed Sets)

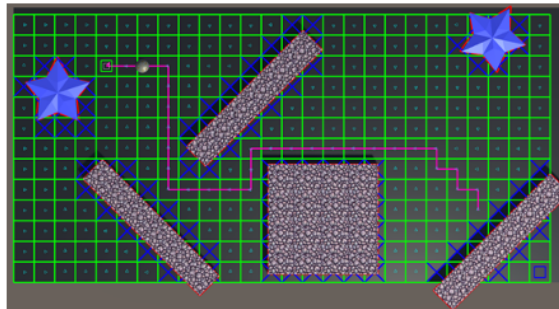
- Critical Operations:
 - Adding an entry to the list
 - Removing an entry from the list
 - Finding the smallest element
 - Finding an entry in the list corresponding to a particular node (find() or contains())
- Must find a balance between these four operations for best performance

Pathfinding List

- Naïve: Simple linked list (finding may visit all elements)
- Sorted List: Finding is efficient but cost to add increases. A* adds a lot with relatively fewer find-smallest-element calls
- Priority Queue/Heap: array-based tree structure. Smallest element head of tree. Remove smallest element and adding new element take $O(\log n)$
- Bucketed Priority Queue: Sorted buckets of unsorted lists. Can be tuned for application. Often not worth the trouble

What if goal node cannot be reached?

- A* will search nodes connected to start node
- A* algorithm can be easily modified to return the closest node by searching the closed set for the node with the lowest heuristic score. This is often reasonable behavior in a video game.



Heuristic Function for A*

- Computational performance is important
- Underestimate completely (0 heuristic) is Dijkstra!
- Perfect Heuristic: A* would go straight to correct node in $O(p)$ (but such a heuristic solves for exactly what we are looking for in the first place!)
- Overestimate: May not return the best path

Admissible Heuristic

- An *admissible heuristic* is one that guarantees that the shortest path can be found with the search because it *never overestimates the cost of reaching the goal*
- A heuristic that does not overestimate is *admissible*
- Otherwise we say a heuristic is *inadmissible*
- Euclidean Distance is *admissible*
- In games, it **perfectly acceptable** to use either *admissible* or *inadmissible*
- **“Overestimates can make A* faster if they are almost perfect but home in on the goal more quickly” – M&F**

Heuristics

- Euclidian Distance – As the crow flies/line of sight distance
 - Admissible
- Cluster Heuristic – Distance between cluster (pre-computed)
 - May or may not be admissible
 - Poor search through clusters due to equal estimates for all nodes within cluster (could use hierarchical A* instead)

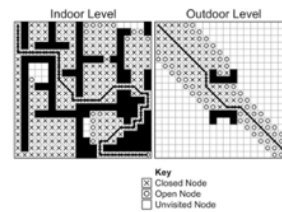


Figure 4.18: Euclidean distance fill characteristics

Cluster Heuristic

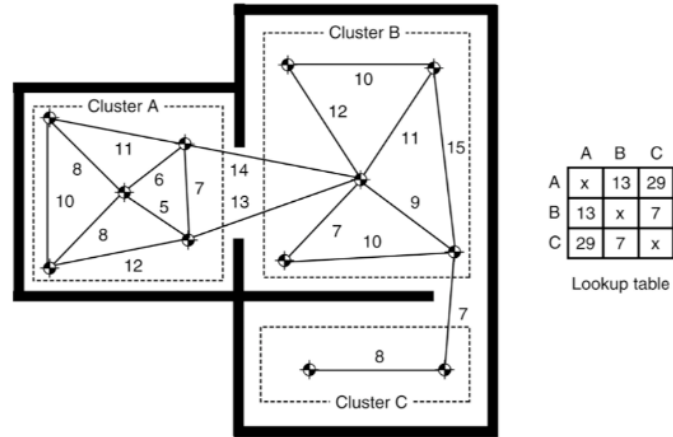


Figure 4.19: The cluster heuristic

Heuristic Outdoors

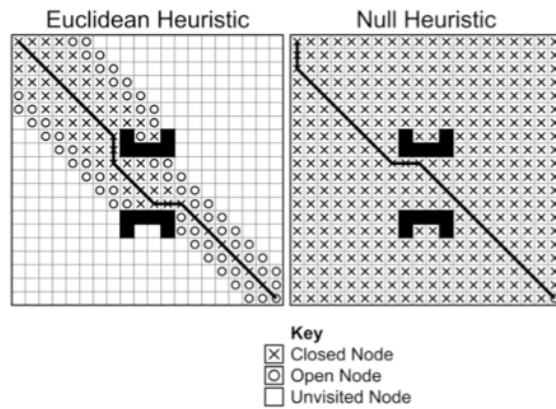


Figure 4.21: Fill patterns outdoors

Heuristic Indoors

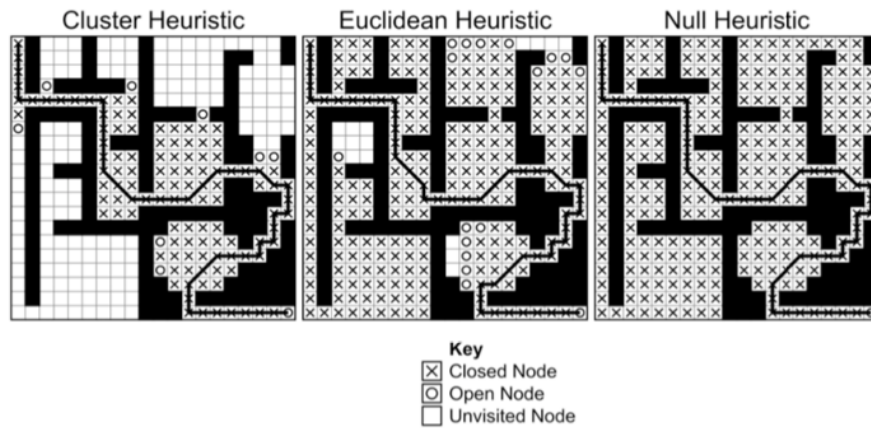


Figure 4.20: Fill patterns indoors