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

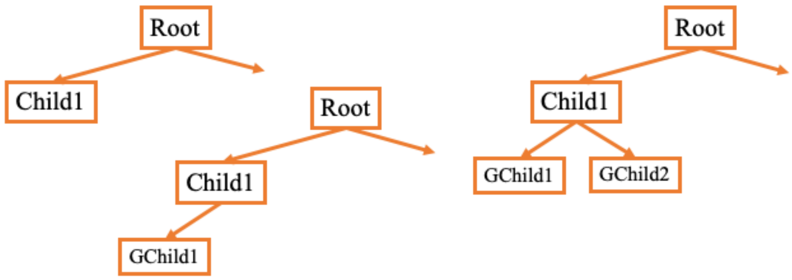
- Must **Search** the state space to move NPC to goal state
- Computational Issues:
  - Completeness
    - Will it find an answer if one exists?
  - Time complexity
  - Space complexity
  - Optimality
    - Will it find the best solution

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!

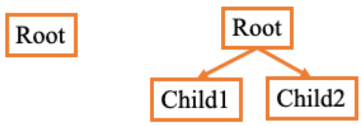
Depth First Search

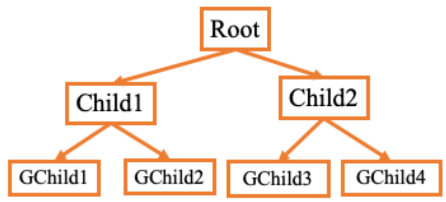
- Always expand the node that is deepest in the tree
- Not best solution (if you stop at first found)



Breadth-First Search

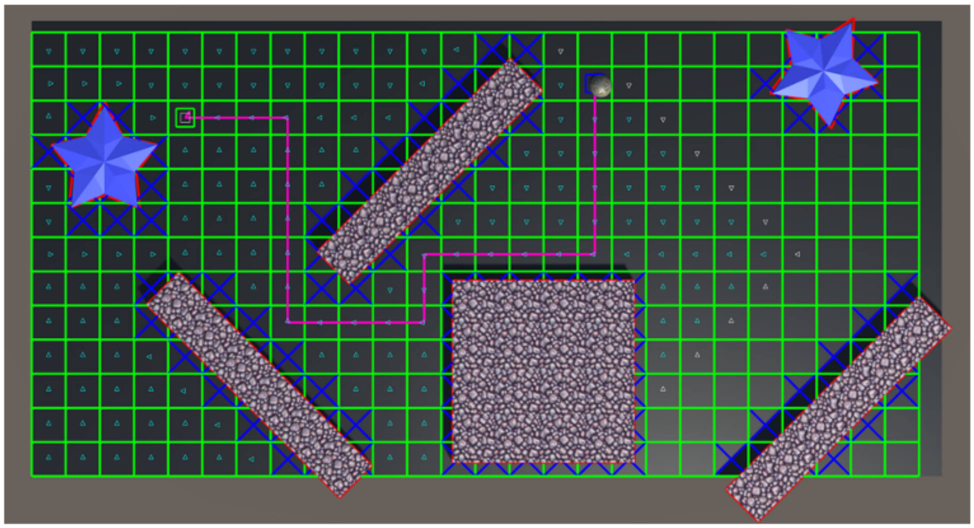
- Expand **Root node**
  - Expand all **Root node's children**
    - Expand all **Root node's grandchildren**





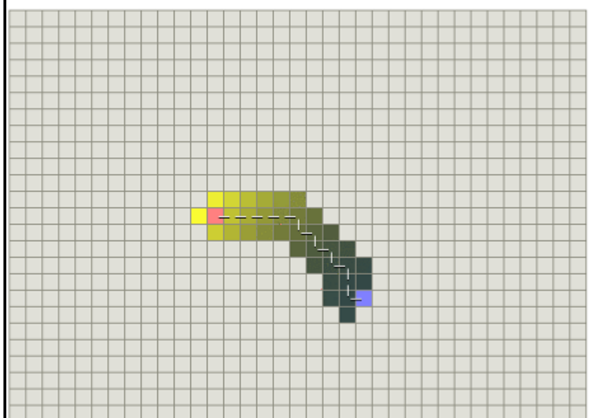
- **Problem: Memory size**

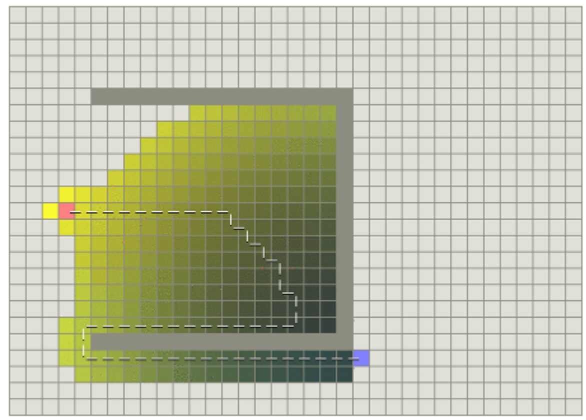
Breadth First



Greedy Best-First-Search

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

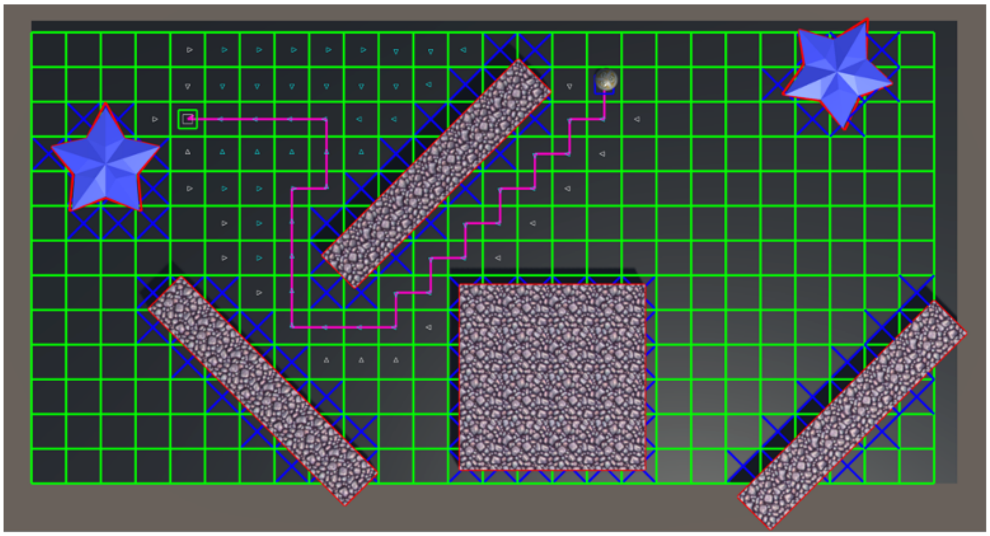




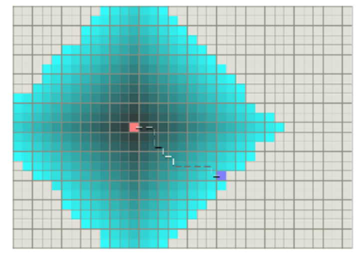
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

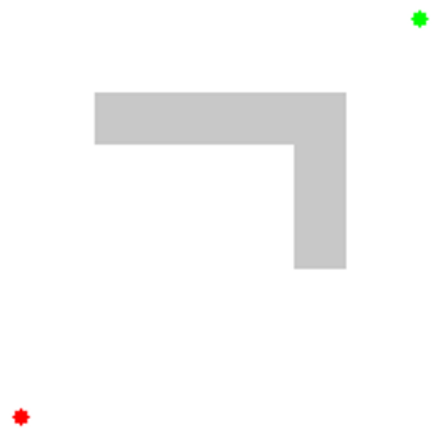
NOT guaranteed to find best solution

Greedy Best-First-Search

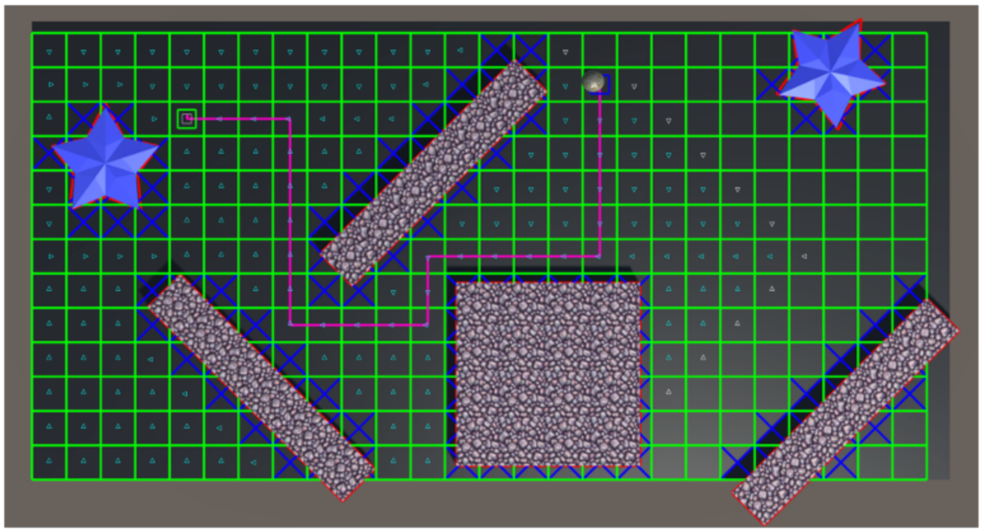


Dijkstra's Algorithm





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.

```
class Graph:
    # An array of connections outgoing from the given node.
    function getConnections(fromNode: Node) -> Connection[]

class Connection:
    # The node that this connection came from.
    fromNode: Node

    # The node that this connection leads to.
    toNode: Node

    # The non-negative cost of this connection.
    function getCost() -> float
```

```
function pathfindDijkstra(graph: Graph,
                          start: Node,
                          end: Node) -> Connection[]:
    # This structure is used to keep track of the information we need
    # for each node.
    class NodeRecord:
        node: Node
        connection: Connection
        costSoFar: float

    # Initialize the record for the start node.
    startRecord = new NodeRecord()
    startRecord.node = start
    startRecord.connection = null
    startRecord.costSoFar = 0

    # Initialize the open and closed lists.
    open = new PathfindingList()
    open += startRecord
    closed = new PathfindingList()

    # Iterate through processing each node.
    while length(open) > 0:
        # Find the smallest element in the open list.
        current: NodeRecord = open.smallestElement()

        # If it is the goal node, then terminate.
        if current.node == goal:
            break

        # Otherwise get its outgoing connections.
        connections = graph.getConnections(current)

        # Loop through each connection in turn.
        for connection in connections:
            # Get the cost estimate for the end node.
            endNode = connection.getToNode()
            endNodeCost = current.costSoFar + connection.getCost()

            # Skip if the node is closed.
            if closed.contains(endNode):
                continue

            # .. or if it is open and we've found a worse route.
            else if open.contains(endNode):
                # Here we find the record in the open list
                # corresponding to the endNode.
                endNodeRecord = open.find(endNode)
                if endNodeRecord.cost <= endNodeCost:
                    continue

            # Otherwise we know we've got an unvisited node, so make a
            # record for it.
            else:
                endNodeRecord = new NodeRecord()
                endNodeRecord.node = endNode

            # We're here if we need to update the node. Update the
            # cost and connection.
            endNodeRecord.cost = endNodeCost
            endNodeRecord.connection = connection

            # And add it to the open list.
            if not open.contains(endNode):
                open += endNodeRecord

        # We've finished looking at the connections for  the current
        # node, so add it to the closed list and remove it from the
        # open list.
        open -= current
        closed += current

    # We're here if we've either found the goal, or if we've no more
    # nodes to search, find which.
    if current.node != goal:
        # We've run out of nodes without finding the goal, so there's
        # no solution.
        return null

    else:
        # Compile the list of connections in the path.
        path = []

        # Work back along the path, accumulating connections.
        while current.node != start:
            path += current.connection
            current = current.connection.getFromNode()

        # Reverse the path, and return it.
        return reverse(path)
```

Dijkstra

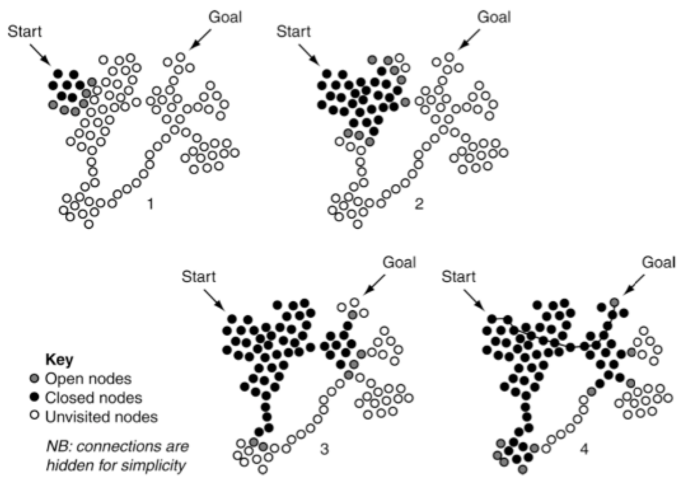
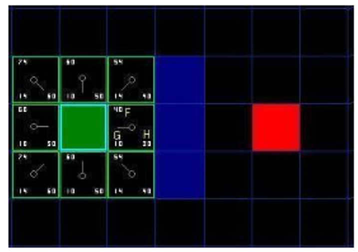


Figure 4.12: Dijkstra in steps

A* Search

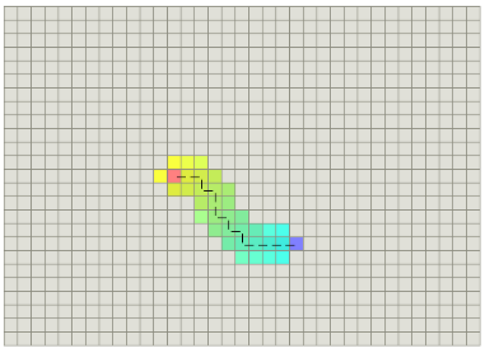
- Minimize sum of costs
- *g(n) + h(n)*
  - Cost so far **+** heuristic to goal
- Guaranteed to find best solution
  - So long as *h(n)* does **not overestimate cost** (and solution exists) – will revisit
- Example *h(n)*
  - Euclidean distance

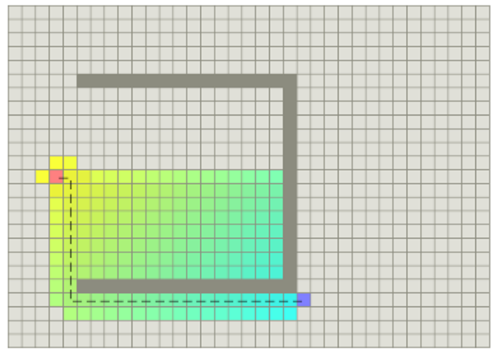


15

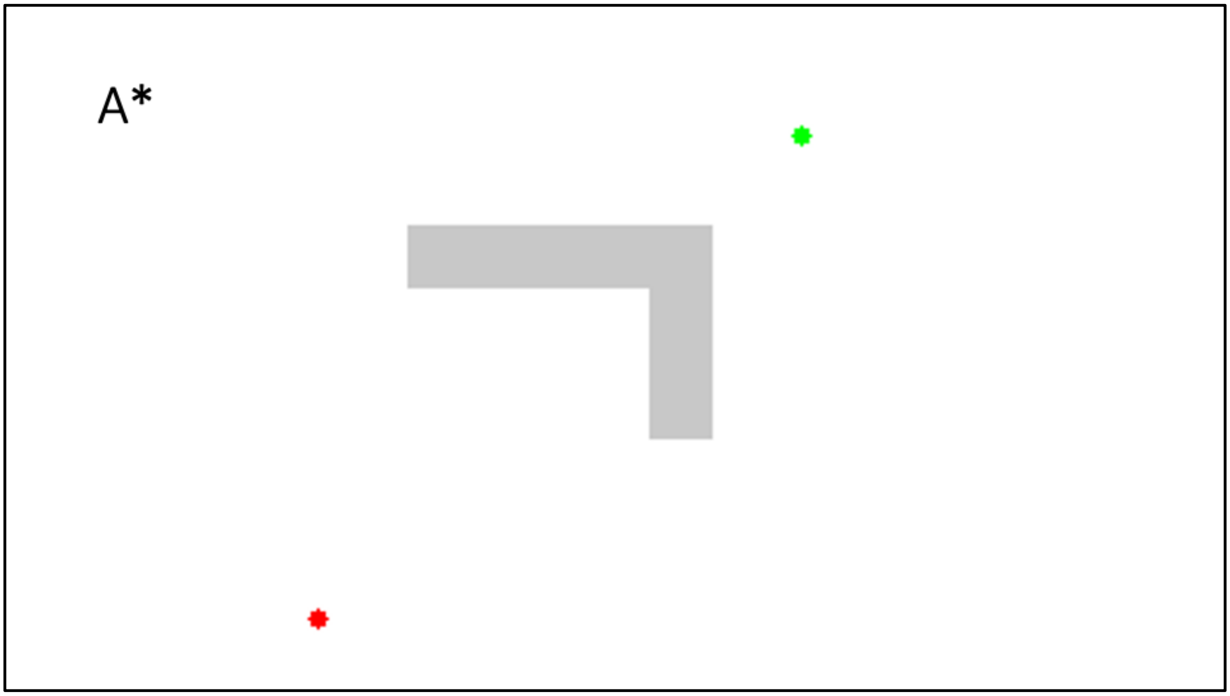
A* Search

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

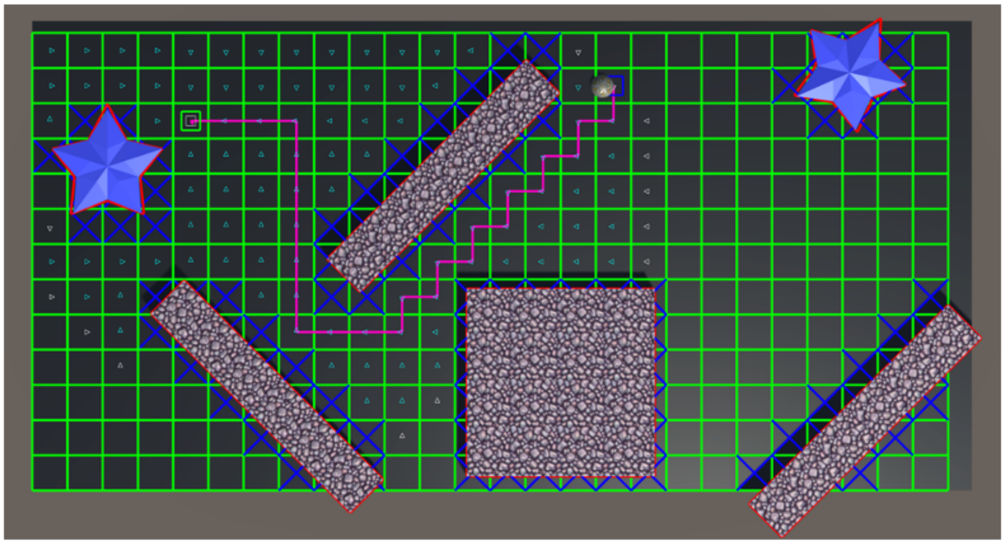




16

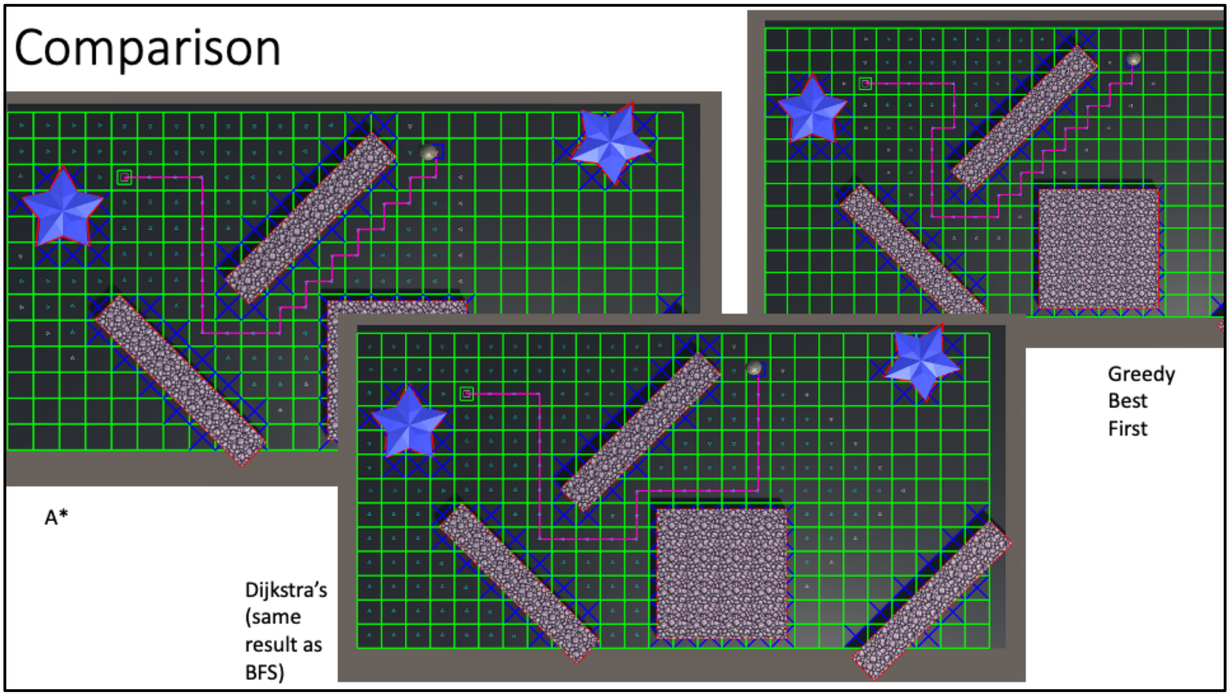

A*

A*



More nodes visited than greedy-best-first search but better path

Comparison



A*

Dijkstra's (same result as BFS)

Greedy Best First

```
function pathfindAStar(graph: Graph,
                       start: Node,
                       end: Node,
                       heuristic: Heuristic
                       ) -> Connection[]:
    # This structure is used to keep track of the
    # information we need for each node.
    class NodeRecord:
        node: Node
        connection: Connection
        costSoFar: float
        estimatedTotalCost: float

    # Initialize the record for the start node.
    startRecord = new NodeRecord()
    startRecord.node = start
    startRecord.connection = null
    startRecord.costSoFar = 0
    startRecord.estimatedTotalCost = heuristic.estimate(start)

    # Initialize the open and closed lists.
    open = new PathfindingList()
    open += startRecord
    closed = new PathfindingList()

    # Iterate through processing each node.
    while length(open) > 0:
        # Find the smallest element in the open list (using the
        # estimatedTotalCost).
        current = open.smallestElement()

        # If it is the goal node, then terminate.
        if current.node == goal:
            break

        # Otherwise get its outgoing connections.
        connections = graph.getConnections(current)

        # Loop through each connection in turn.
        for connection in connections:
            # Get the cost estimate for the end node.
            endNode = connection.getToNode()
            endNodeCost = current.costSoFar + connection.getCost()

            # If the node is closed we may have to skip, or remove it
            # from the closed list.
            if closed.contains(endNode):
                # Here we find the record in the closed list
                # corresponding to the endNode.
                endNodeRecord = closed.find(endNode)

                # If we didn't find a shorter route, skip.
                if endNodeRecord.costSoFar <= endNodeCost:
                    continue

                # Otherwise remove it from the closed list.
                closed -= endNodeRecord

                # We can use the node's old cost values to calculate
                # its heuristic without calling the possibly expensive
                # heuristic function.
                endNodeHeuristic = endNodeRecord.estimatedTotalCost -
                                   endNodeRecord.costSoFar

            # Skip if the node is open and we've not found a better
            # route.
            else if open.contains(endNode):
                # Here we find the record in the open list
                # corresponding to the endNode.
                endNodeRecord = open.find(endNode)

                # If our route is no better, then skip.
                if endNodeRecord.costSoFar <= endNodeCost:
                    continue

                # Again, we can calculate its heuristic.
                endNodeHeuristic = endNodeRecord.cost -
                                   endNodeRecord.costSoFar

            # Otherwise we know we've got an unvisited node, so make a
            # record for it.
            else:
                endNodeRecord = new NodeRecord()
                endNodeRecord.node = endNode

                # We'll need to calculate the heuristic value using
                # the function, since we don't have an existing record
                # to use.
                endNodeHeuristic = heuristic.estimate(endNode)

            # We're here if we need to update the node. Update the
            # cost, estimate and connection.
            endNodeRecord.cost = endNodeCost
            endNodeRecord.connection = connection
            endNodeRecord.estimatedTotalCost = endNodeCost +
                endNodeHeuristic

            # And add it to the open list.
            if not open.contains(endNode):
                open += endNodeRecord

        # We've finished looking at the connections for the current
        # node, so add it to the closed list and remove it from the
        # open list.
        open -= current
        closed += current

    # We're here if we've either found the goal, or if we've no more
    # nodes to search, find which.
    if current.node != goal:
        # We've run out of nodes without finding the goal, so there's
        # no solution.
        return null

    else:
        # Compile the list of connections in the path.
        path = []

        # Work back along the path, accumulating connections.
        while current.node != start:
            path += current.connection
            current = current.connection.getFromNode()

        # Reverse the path, and return it.
        return reverse(path)
```

A*

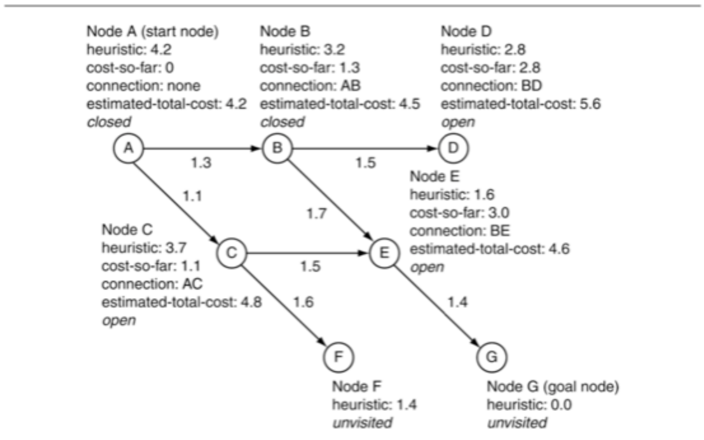


Figure 4.13: A* estimated-total-costs

A*

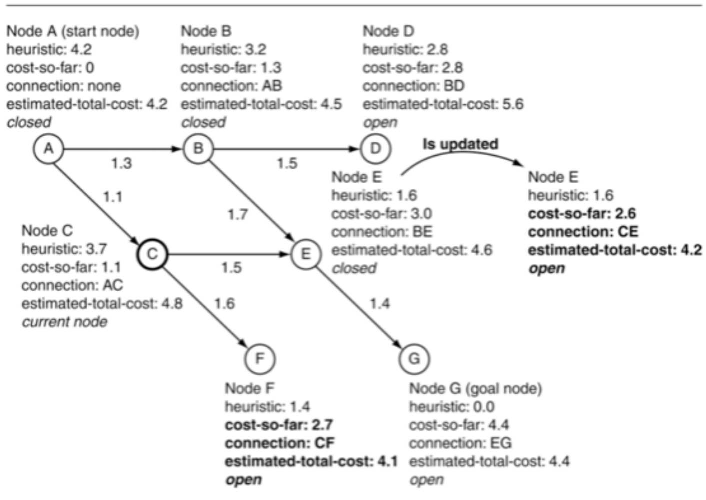


Figure 4.14: Closed node update

Pathfinding List (Open and Closed Sets)

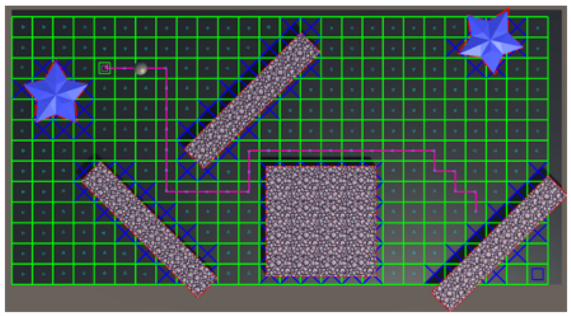
- Critical Operations:
  - Adding an entry to the list
  - Removing an entry from the list
  - Finding the smallest element
  - Finding an entry in the list corresponding to a particular node (find() or contains())
- Must find a balance between these four operations for best performance

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)

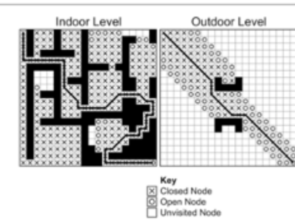
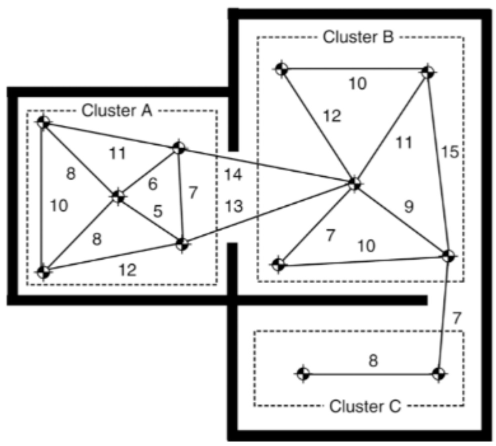


Figure 4.18: Euclidean distance fill characteristics

Cluster Heuristic



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

Lookup table

Figure 4.19: The cluster heuristic

Heuristic Outdoors

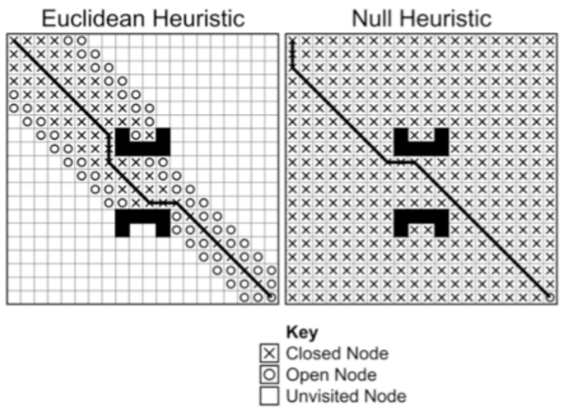


Figure 4.21: Fill patterns outdoors

Heuristic Indoors

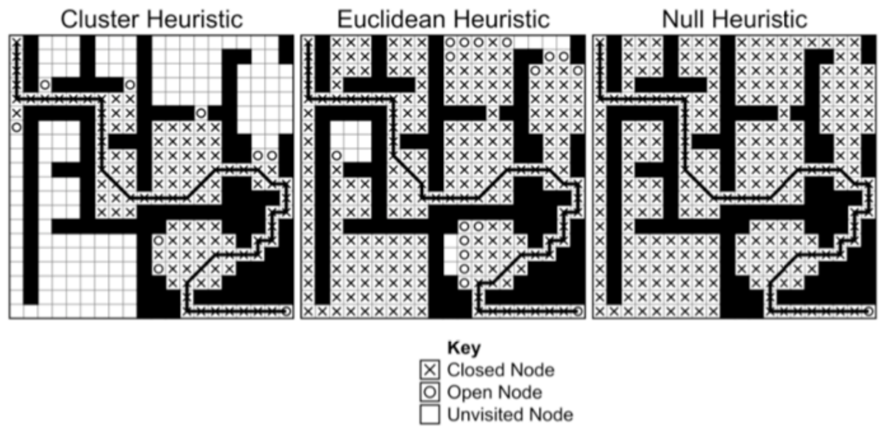


Figure 4.20: Fill patterns indoors