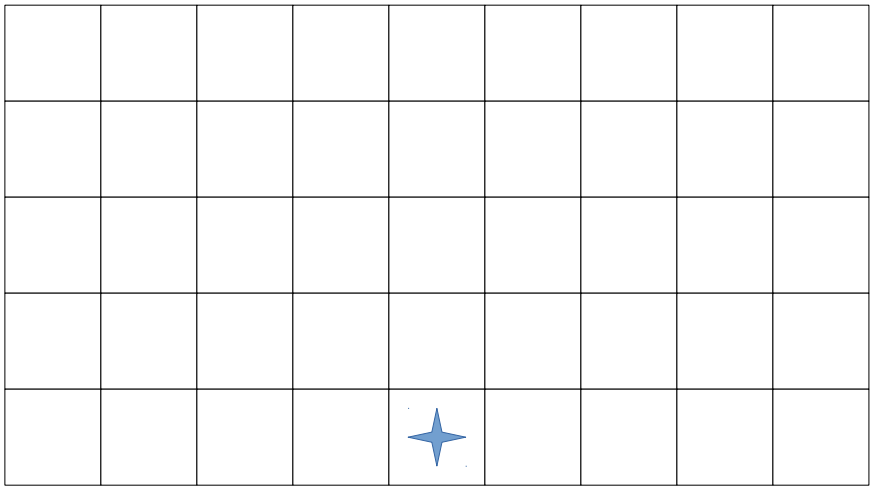
Pathfinding

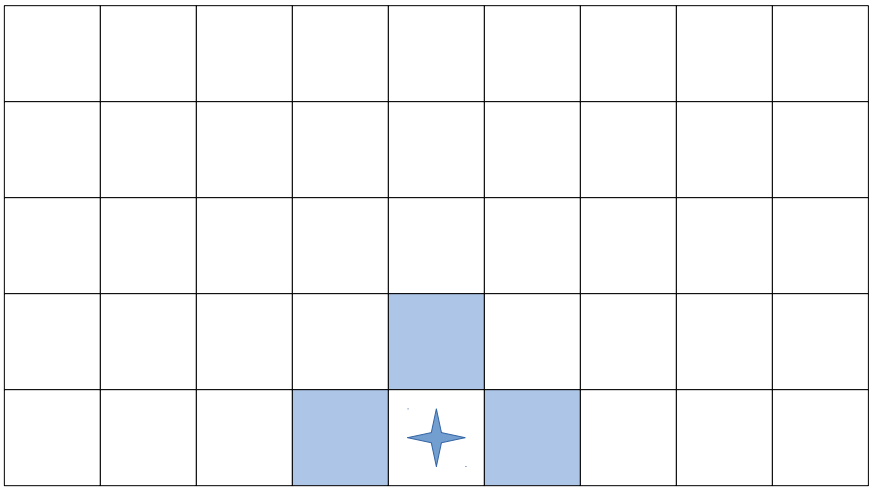
Breadth First Search

```
graph : Set of all nodes
frontier : Queue

frontier.enqueue(graph.start)
while frontier not empty
      current = frontier.dequeue()
      for neighbor in graph.neighbors(current)
            if neighbor not visited
                  frontier.enqueue(neighbor)
                  neighbor.visited = true
                  neighbor.predecessor = current

path : stack
current = goal
while current not start
      path.push(current)
      current = current.predecessor
path.push(start)  // May not be necessary
```

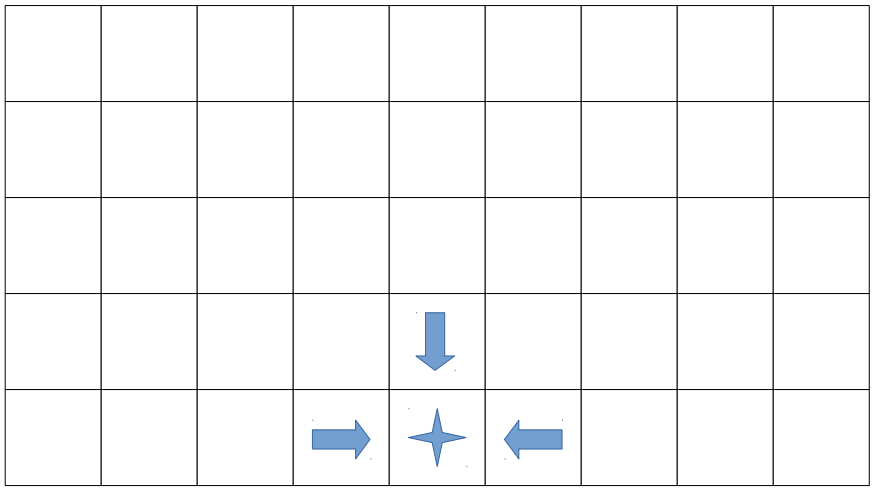
Step 1 : Define the start



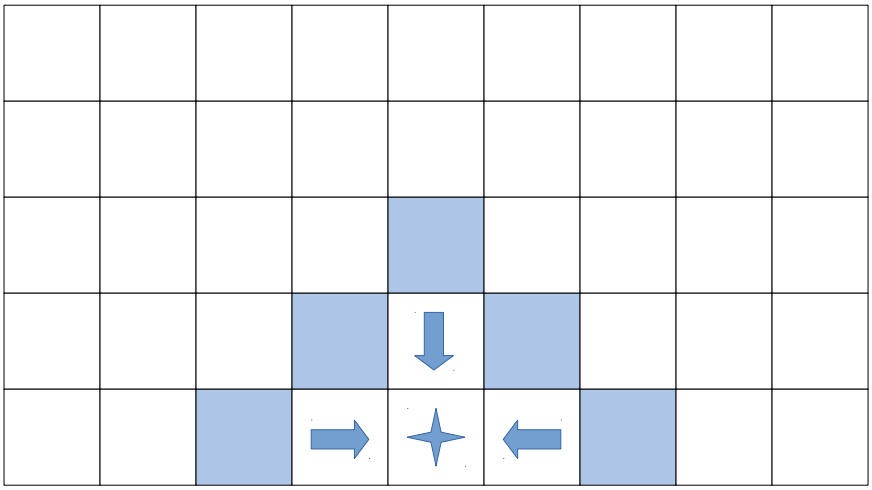
Step 2 : Add neighbors

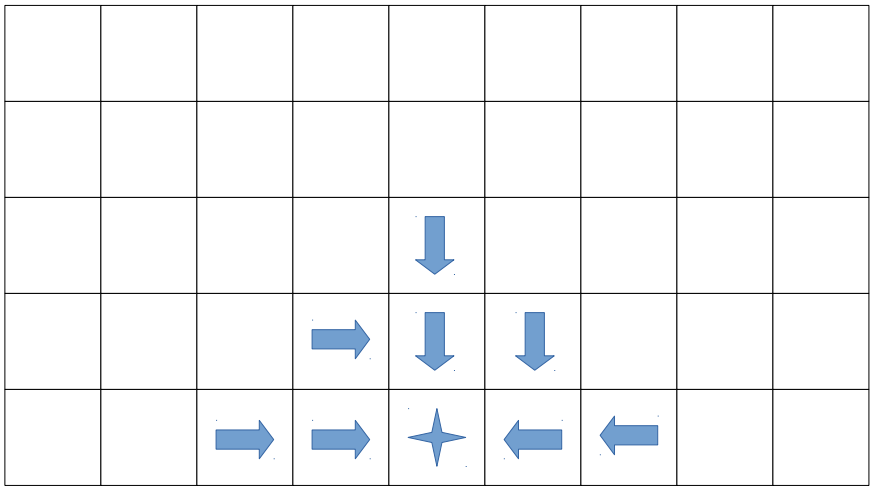


Step 3 : Process neighbors with predecessors



Step 4 : Continue with next set of neighbors





Greedy Search

Want to find the shortest path to one location. Therefore, have the frontier expand towards the goal more than it expands in other directions. To do this, need to define a heuristic that estimates (computes) how far a node is to the goal.

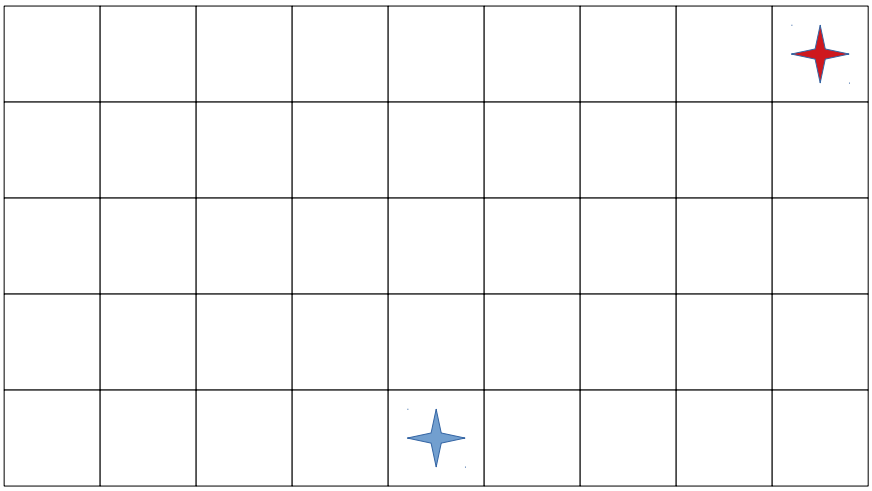
Heuristic

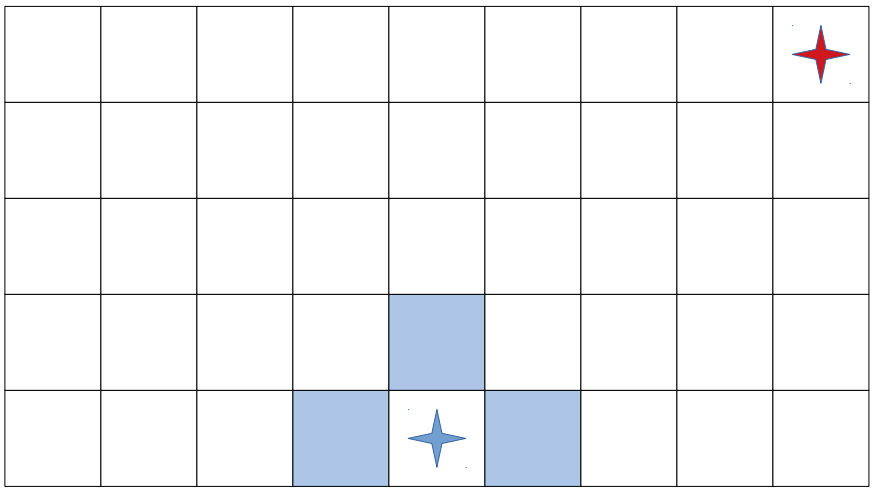
For a grid-based Tower Definse game, the Manhattan distance is a good approach:

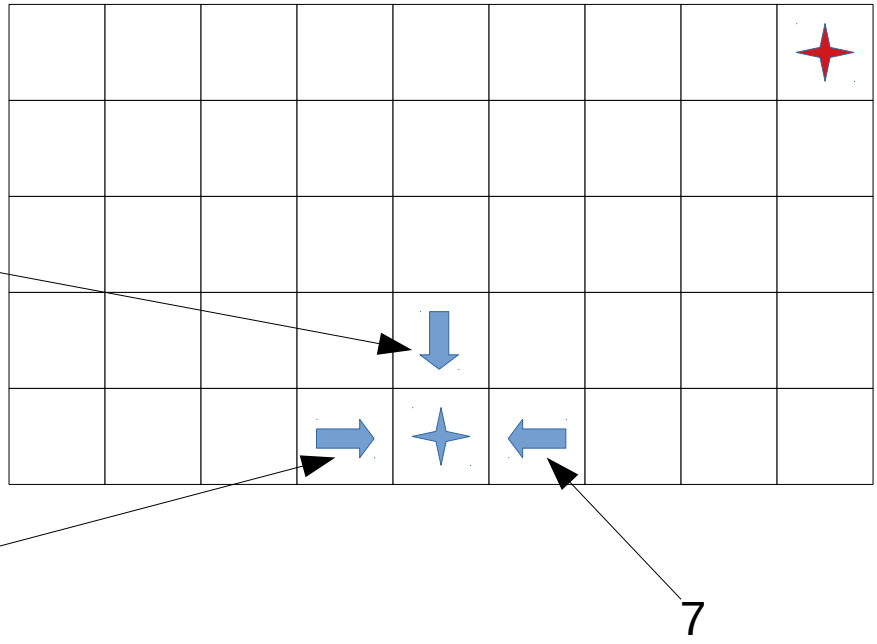
```
ManhattanDistance(a, b)
      return abs(a.x - b.x) + abs(a.y - b.y)
```

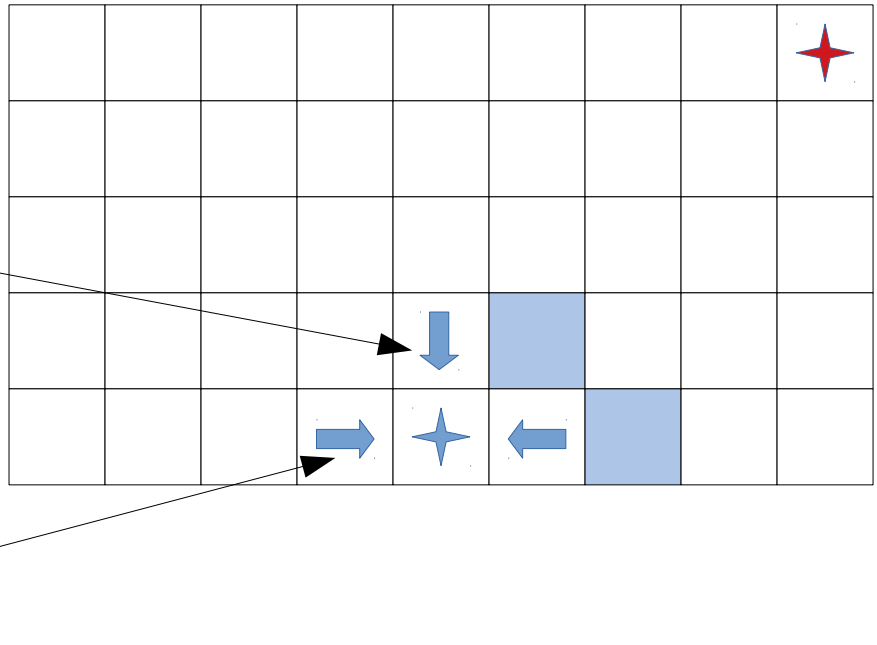
Replace the Queue with a PriorityQueue; min value at the top

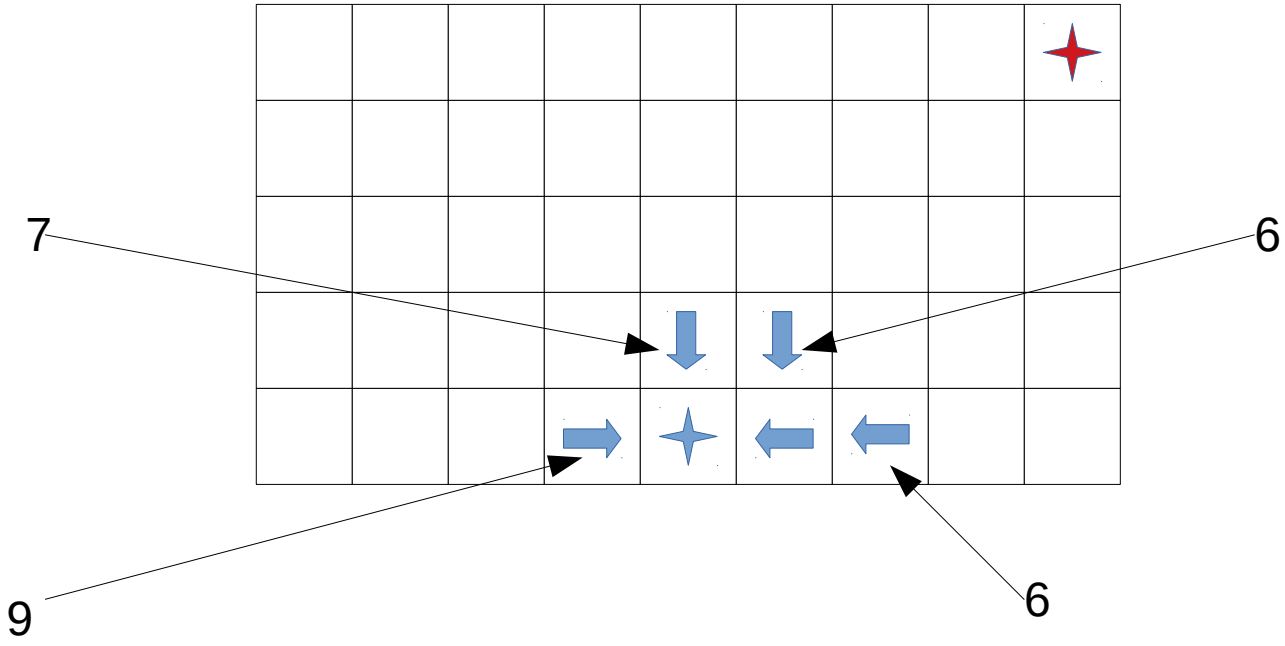
```
graph : Set of all nodes
frontier : PriorityQueue

frontier.enqueue(graph.start)
while frontier not empty
      current = frontier.dequeue()
      if current is goal then stop
      for neighbor in graph.neighbors(current)
            if neighbor not visited
                  priority = ManhattanDistance(goal, neighbor)
                  frontier.enqueue(neighbor, priority)
                  neighbor.visited = true
                  neighbor.predecessor = current
```

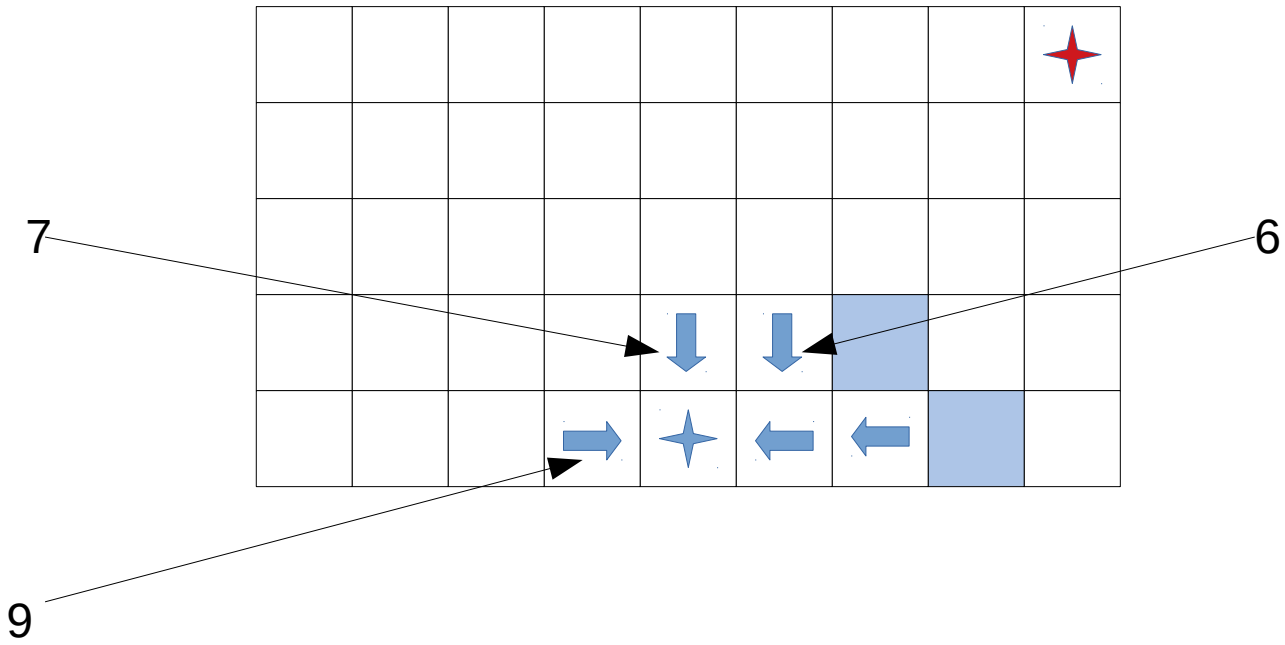


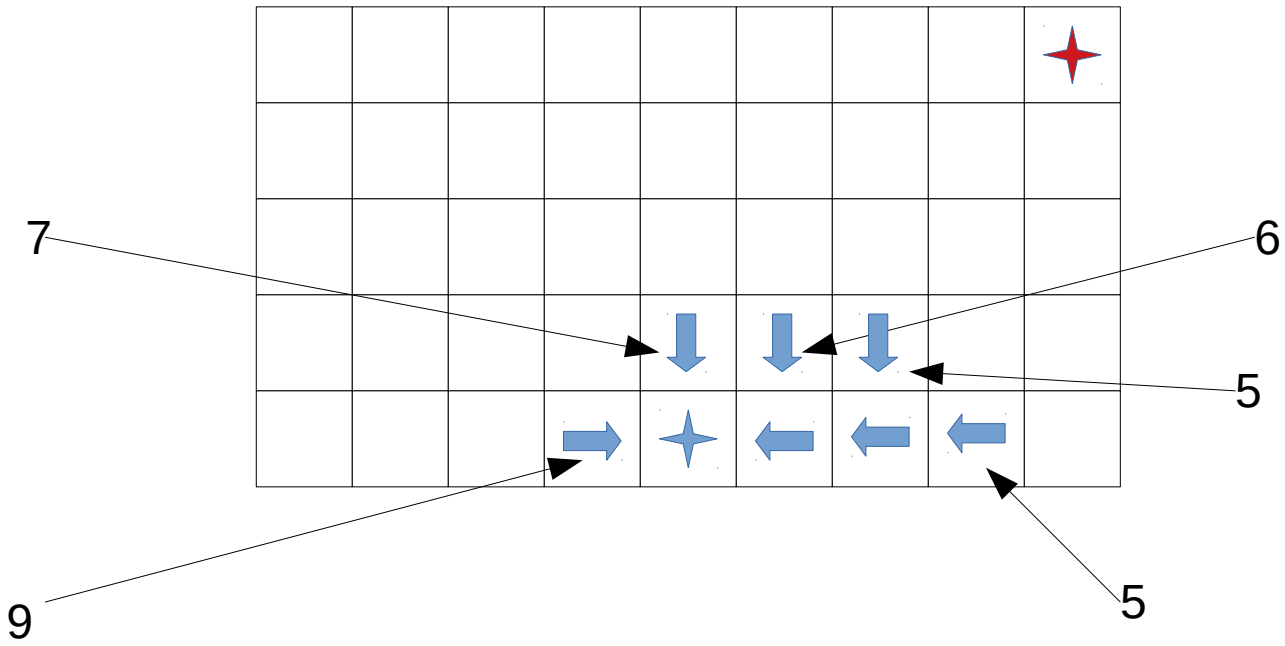



7
9
7


7
9


7
6
9
6


7
6
9


7
6
5
9
5

A Algorithm*

Problem with the Greedy Search is that it doesn't always create the shortest path when there are obstacles along the way, because the heuristic can't "see" the obstacles. The algorithm could get very near the goal, then run into a big wall and have to crawl along the wall. When instead, there exists a shorter path that starts to avoid the wall sooner.

The A* algorithm uses two distance calculations:

1. The estimated distance to the goal
2. The actual distance so far

```
graph : Set of all nodes
frontier : PriorityQueue
graph[start].actualCost = 0

frontier.enqueue(graph.start)
while frontier not empty
      current = frontier.dequeue()
      if current is goal then stop
      for neighbor in graph.neighbors(current)
            // The actualCost(current, neighbor) is just 1 for our TD game
            cost = current.actualCost + actualCost(current, neighbor)
            if neighbor not visited or cost < neighbor.actualCost
                  priority = cost + ManhattanDistance(goal, neighbor)
                  frontier.enqueue(neighbor, priority)
                  neighbor.visited = true
                  neighbor.predecessor = current
```