Informed Search

Dongchen HE

Office: SHB122, CUHK

The Chinese University of Hong Kong

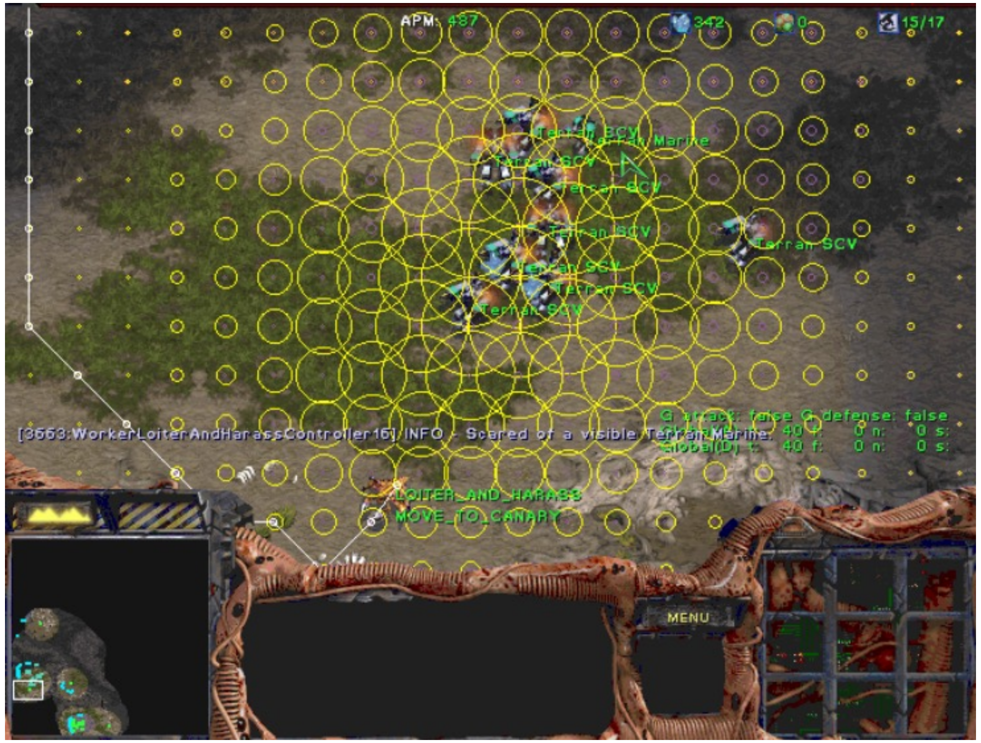
November 8, 2023

Outline

- Search Heuristic
- Greedy search
- A* search
- Solve problem by A*

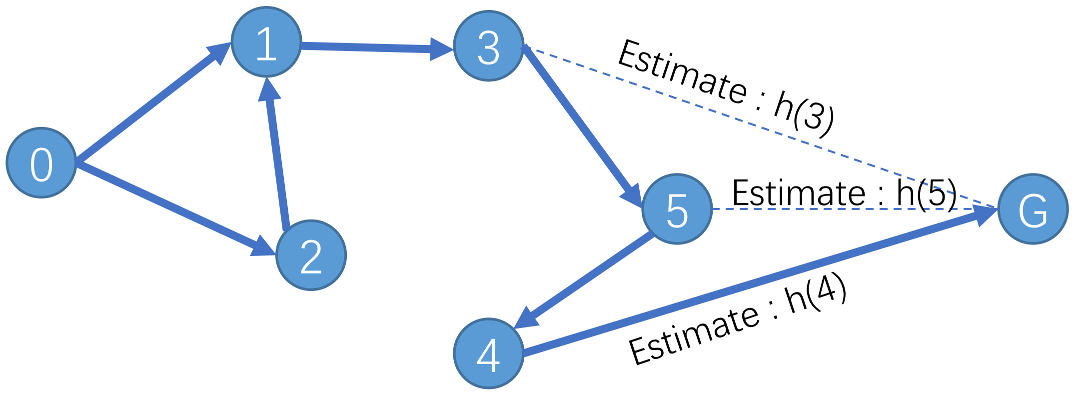
A* search application

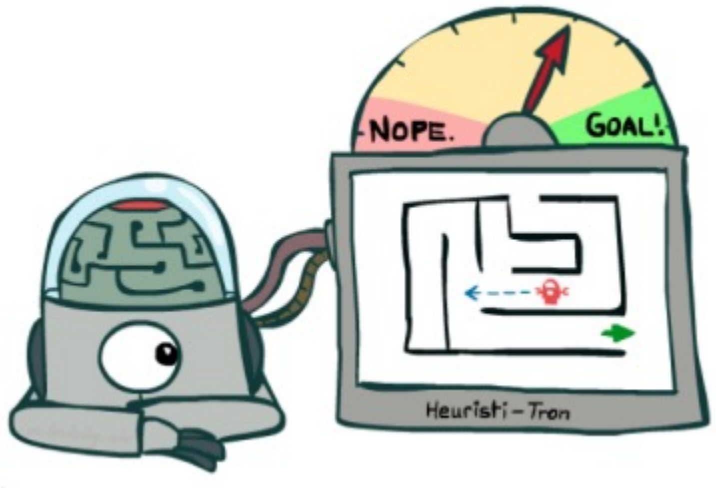
- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis



Search Heuristic

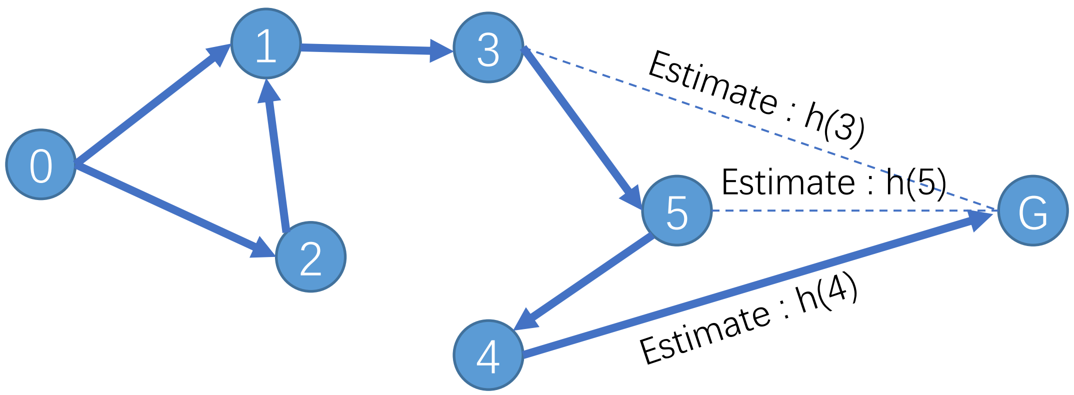
- Heuristic function h(x)
- h(x) = estimated cost of the cheapest path from the state at node x to a goal state





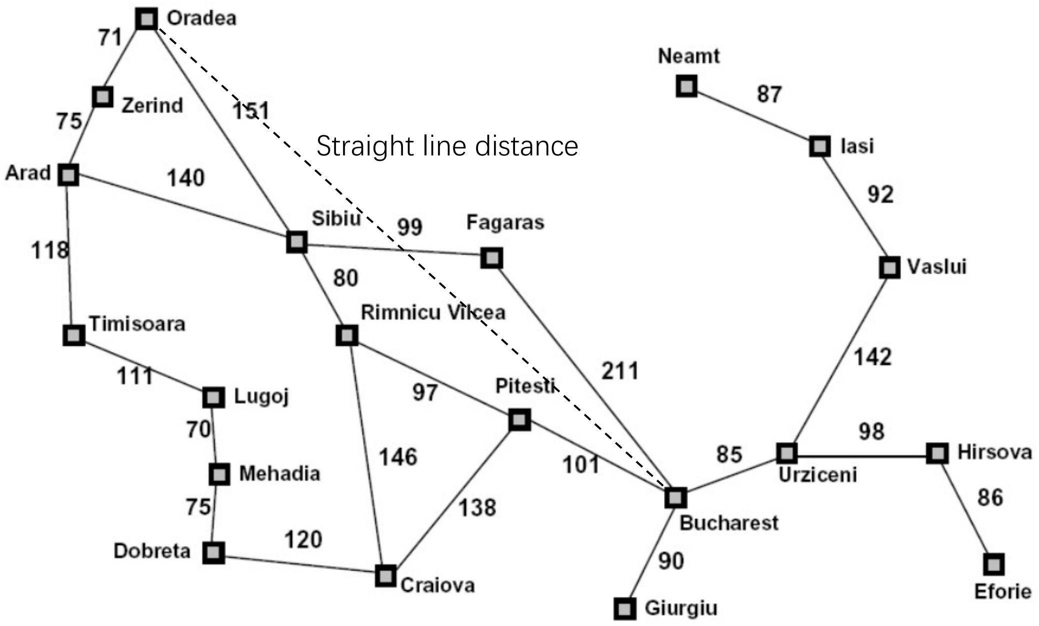
Search Heuristic

- Admissibility: heuristic cost ⩽ actual cost to goal
  - h(A) ⩽ actual cost from A to G
  - The heuristic values must be lower bounds on the actual cost (Underestimate)
- Consistency: heuristic "arc" cost ⩽ actual cost for each arc
  - h(A) – h(C) ⩽ cost(A to C)
  - If an action has cost c, then taking that action can only cause a drop in heuristic of at most c



Search Heuristic

- Driver: Find a path to Bucharest
  - Action: Only can move to adjacent node
  - Cost: length of path
- h(x) = Straight line distance from source to goal

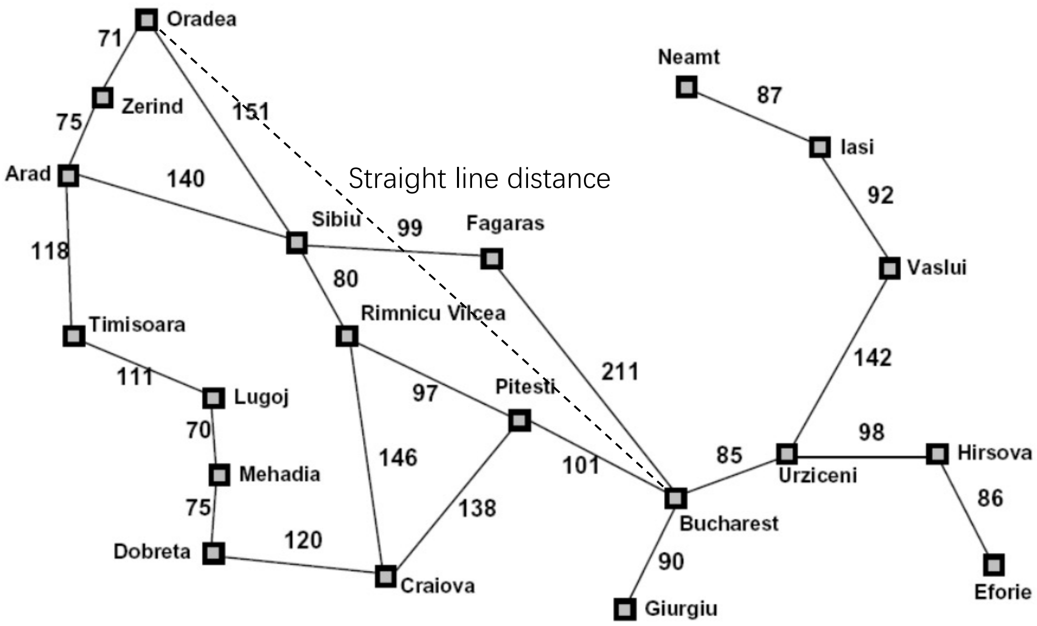


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Search Heuristic

- Admissibility: heuristic cost ⩽ actual cost to goal
  - h(A) ⩽ actual cost from A to G
- Admissible?
  - h(x) = Straight line distance from source to goal



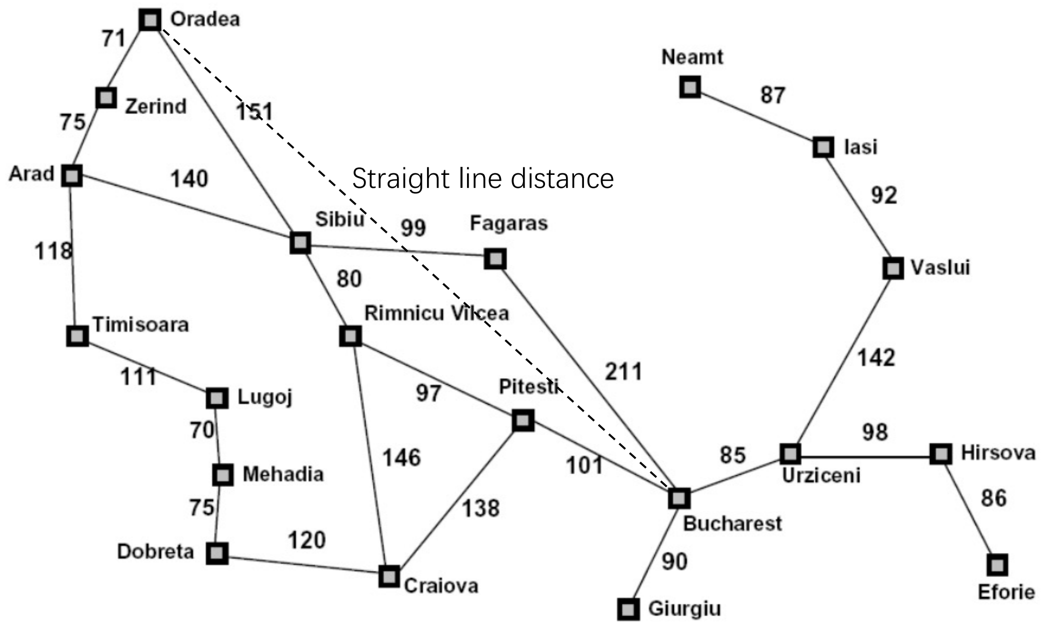
Straight−line distance to Bucharest

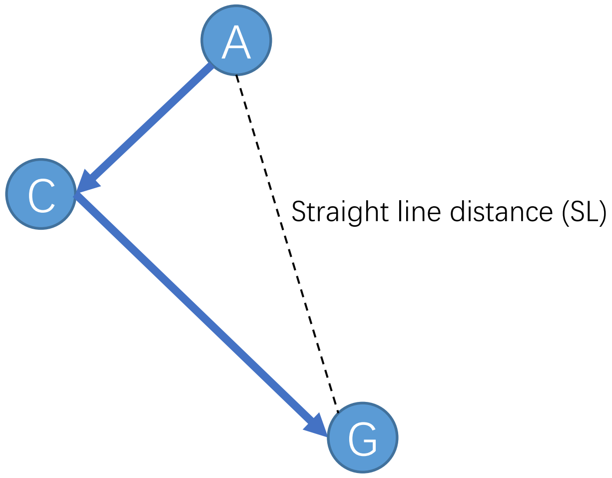
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Search Heuristic

- Admissibility: heuristic cost ⩽ actual cost to goal
  - h(A) ⩽ actual cost from A to G
- Admissible? Yes! The shortest distance between two points is a line

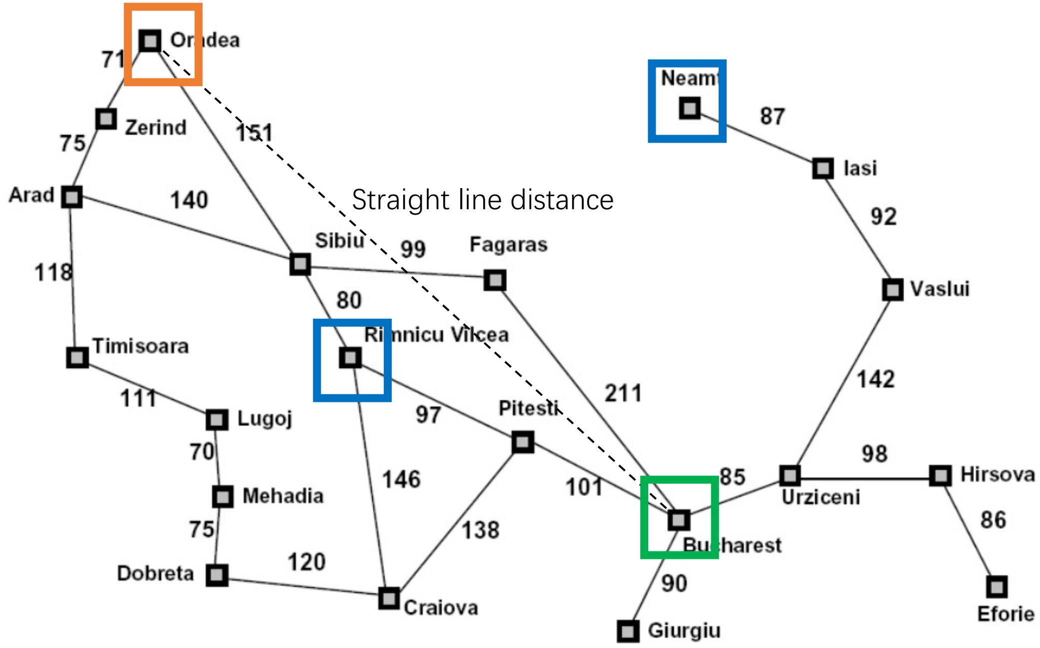




SL(AG) < AC + CG

Search Heuristic

- Consistency: heuristic “arc” cost ⩽ actual cost for each arc
  - h(A) – h(C) ⩽ cost(A to C)
- Consistent?
  - h(x) = Straight line distance from source to goal



Straight−line distance to Bucharest

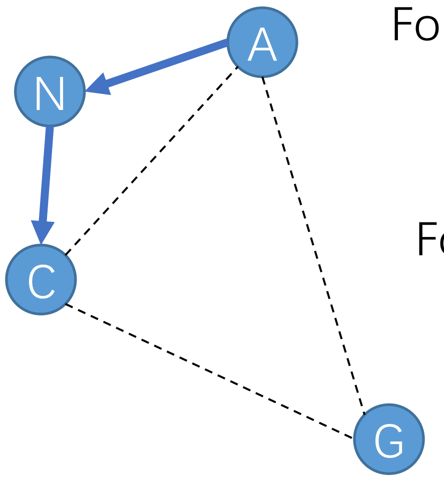
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Search Heuristic

- Consistency: heuristic "arc" cost ⩽ actual cost for each arc
  - h(A) – h(C) ⩽ cost(A to C)
- Consistent? Yes! Consider two cases:

Case1: G not in path AC



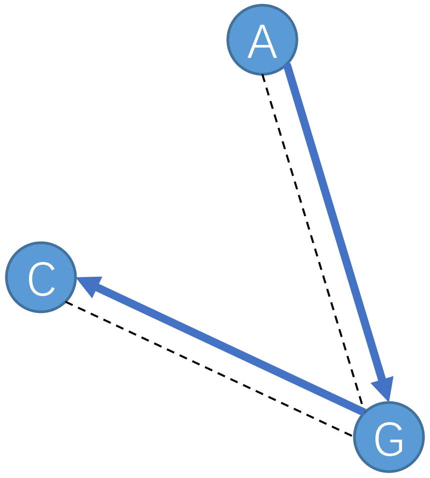
For ACG: SL(AC) + SL(CG) > SL(AG)

SL(AC) > SL(AG) - SL(CG)

For ANC: SL(AC) < AN + NC

$$\underbrace{AN + NC}_{\text{cost(A to C)}} \geq SL(AC) \geq \underbrace{SL(AG) - SL(CG)}_{h(A) - h(C)}$$

Case2: G in path AC



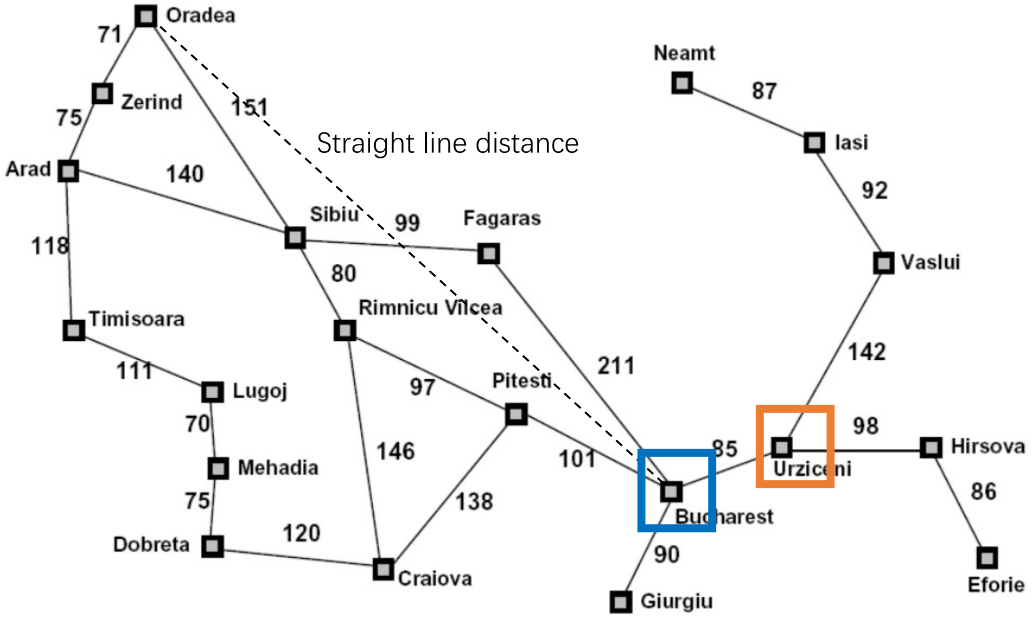
$$\underbrace{AG + GC}_{\text{cost(A to C)}} \geq SL(AG) + SL(CG) \geq \underbrace{SL(AG) - SL(CG)}_{h(A) - h(C)}$$

Search Heuristic

- If our goal is Urziceni, but all we know is the straight-line distances to Bucharest
- h(x) = SL(node, Bucharest) + cost(Bucharest, Urziceni)

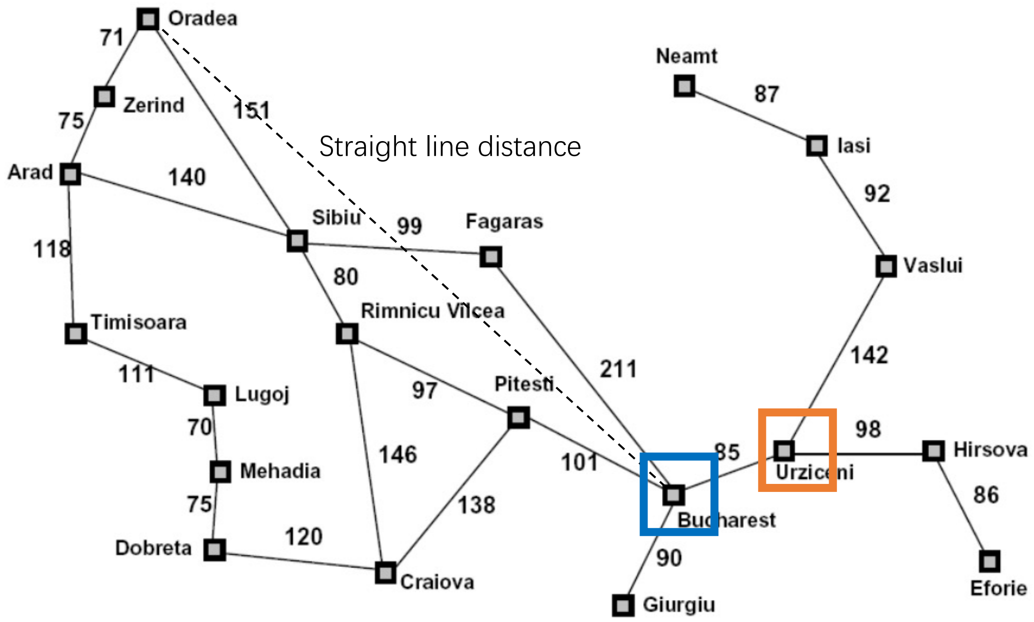
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)



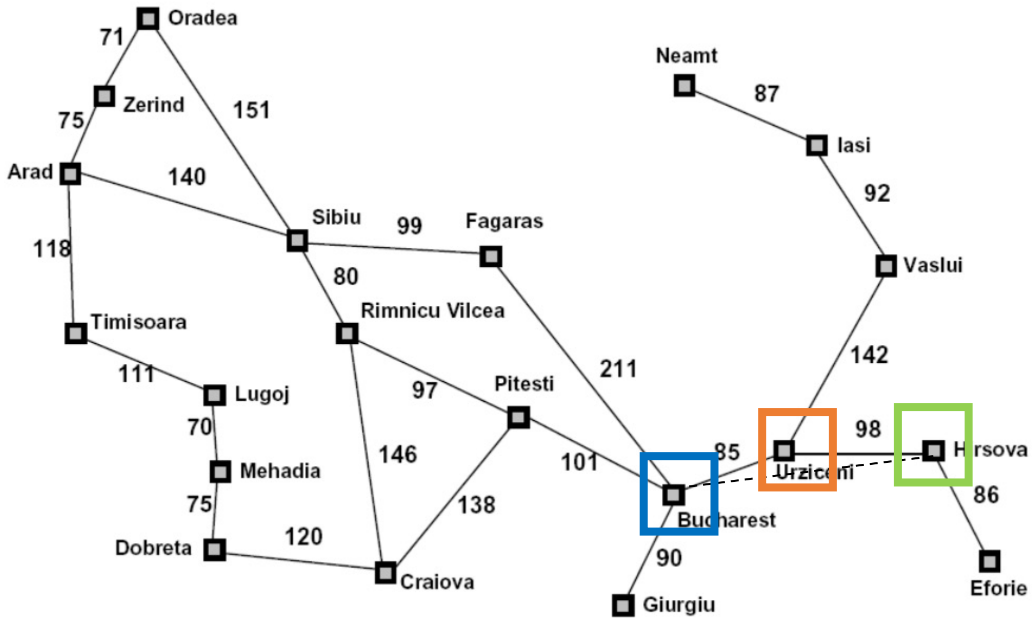
Search Heuristic

- Admissibility: heuristic cost ⩽ actual cost to goal
  - h(A) ⩽ actual cost from A to G
- Admissible?
  - h(x) = h(x) = SL(node, Bucharest) + cost(Bucharest, Urzizeni)



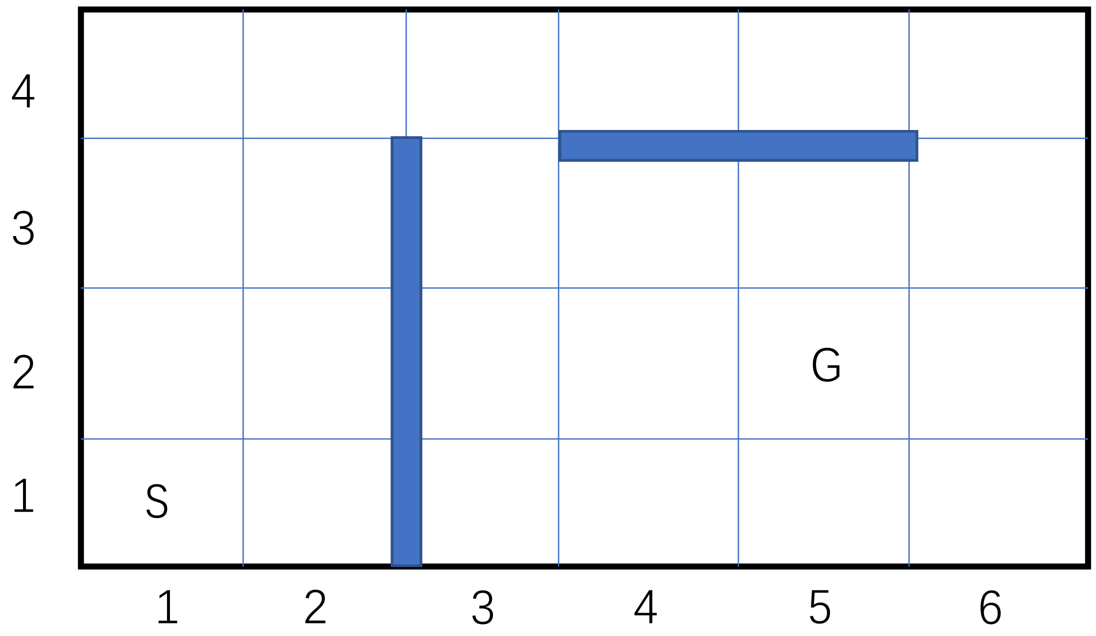
Search Heuristic

- Admissible?
  - No, h(Hirsova) > actual cost from Hirsova to Urziceni
- If Admissibility can not be hold, consistency can not be hold
  - Admissibility: h(A) ⩽ cost(A to G)
  - Consistency: h(A) – h(C) ⩽ cost(A to C)



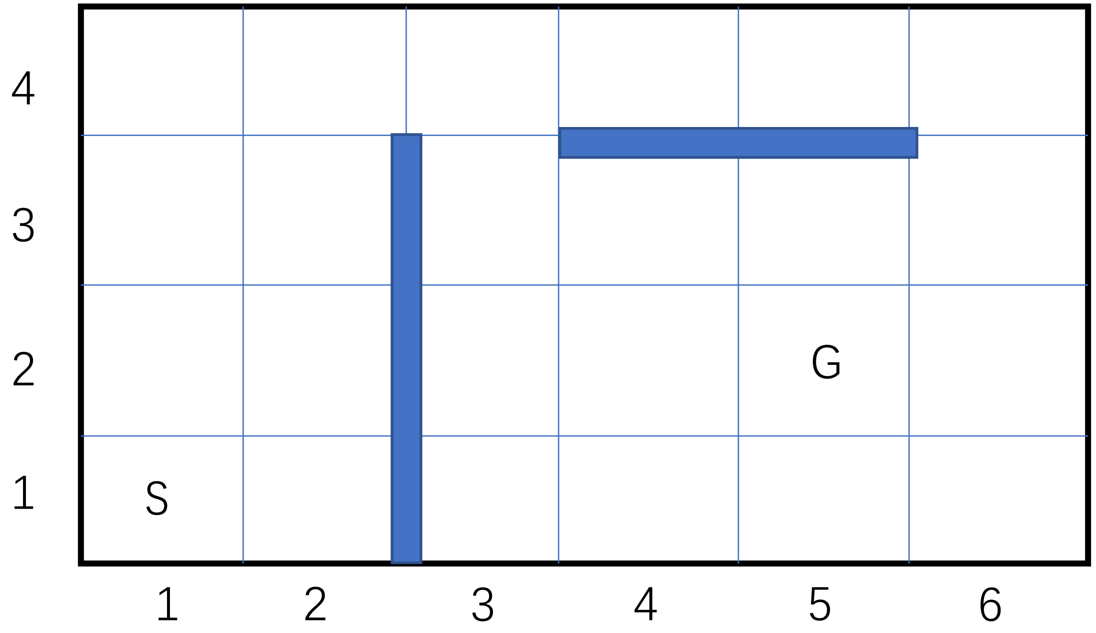
Search Heuristic

- Grid world search: Find a path to Goal
  - Action: up, down, left, right from tile to tile
    - Can not cross wall.
  - Cost: number of moves
- Possible h(n)?



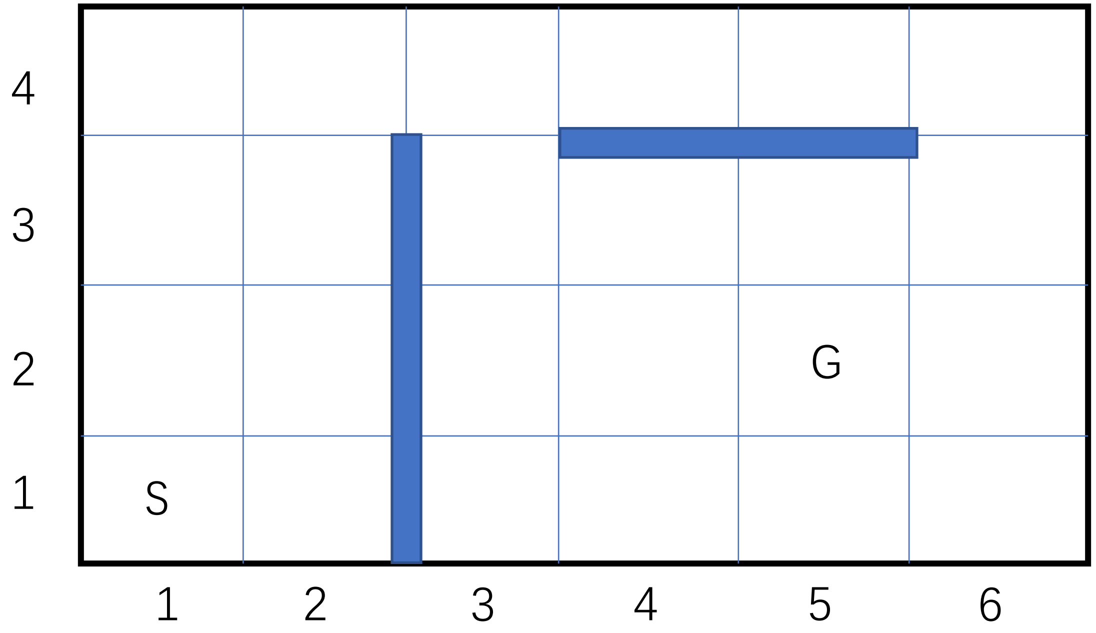
Search Heuristic

- Possible h(n)?
  - Finding a good way to estimate the cost without solving the problem
  - Should be quick and cheap to compute
  - Admissible and consistent



Search Heuristic

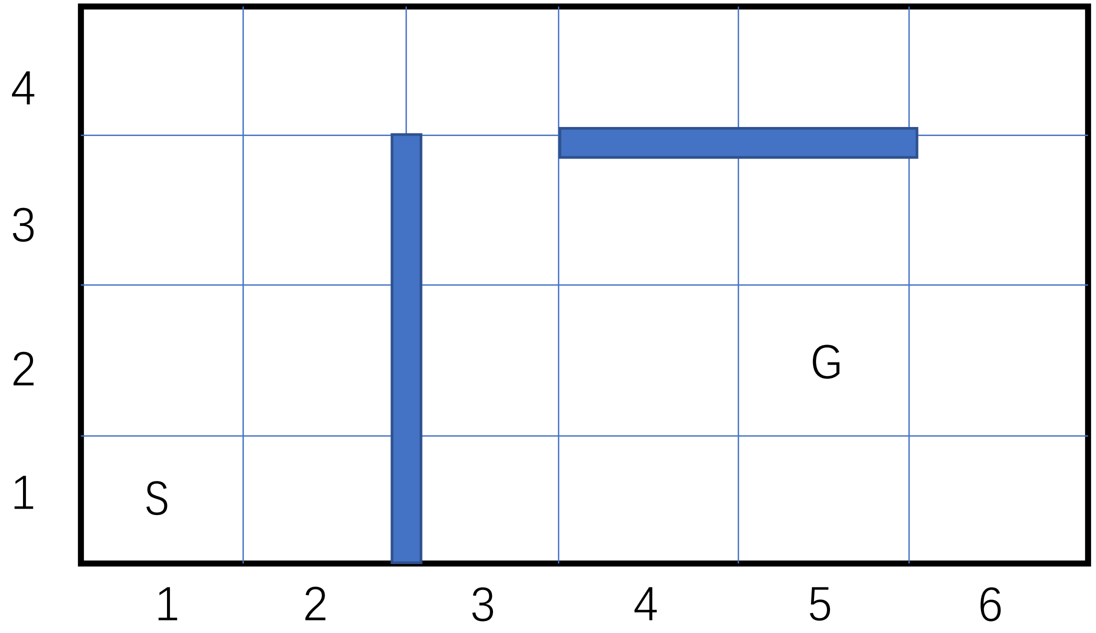
- Possible h(n): imagine the wall is not there
  - Euclidean distance: like straight line distance
  - Manhattan distance (L1 distance) to the goal G: Sum of the absolute difference of their coordinates



Search Heuristic

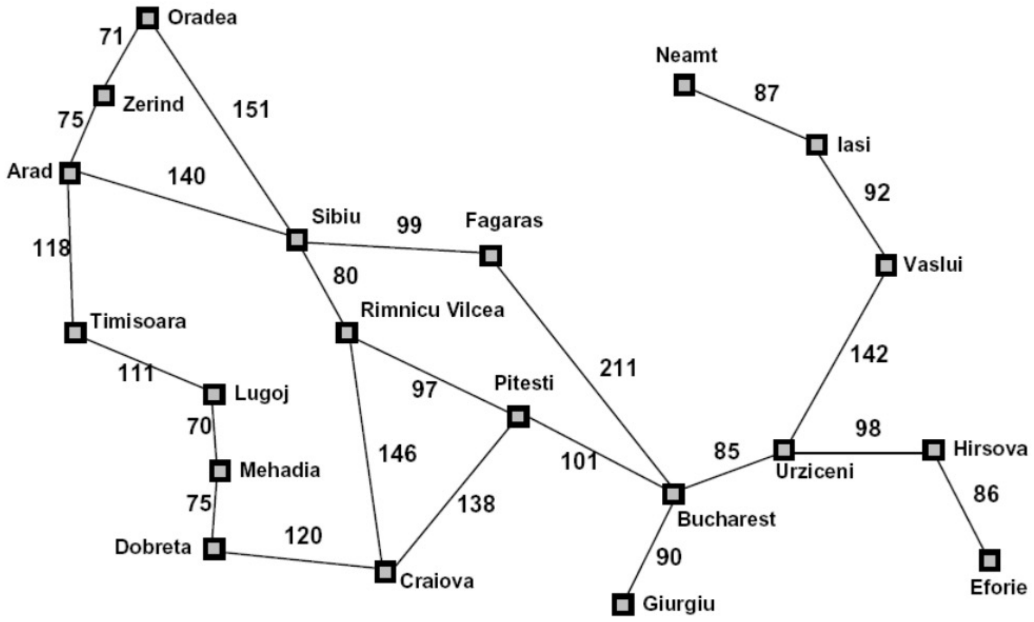
- Admissible? h(A) ⩽ actual cost from A to G
  - Euclidean distance: Yes
  - Manhattan distance (L1 distance) to the goal G: Yes, if no walls, it is equal to the solution. Now the actual path get longer.

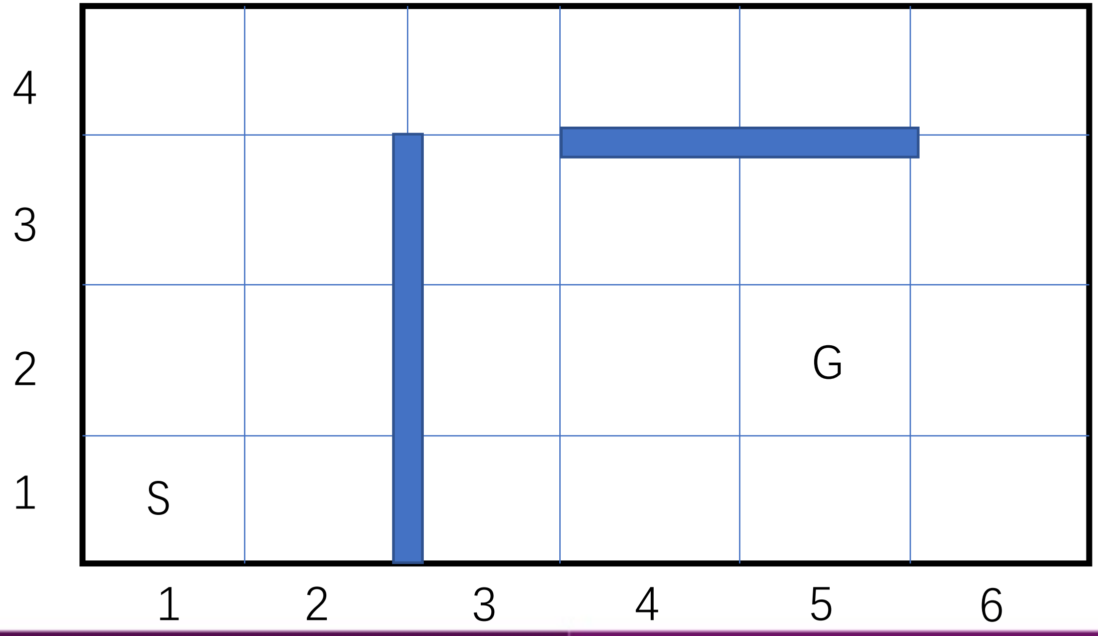
Consistent? Yes



Search Heuristic

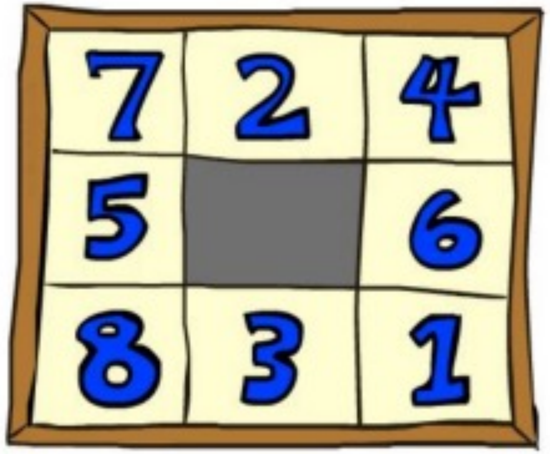
- Construct a heuristic function
  - Identify a relaxed version of the problem
    - Where one or more constraints have been dropped
    - Problem with fewer restrictions on the actions
  - Grid world: We assume the agent can move through walls
  - Driver: Agent can move straight



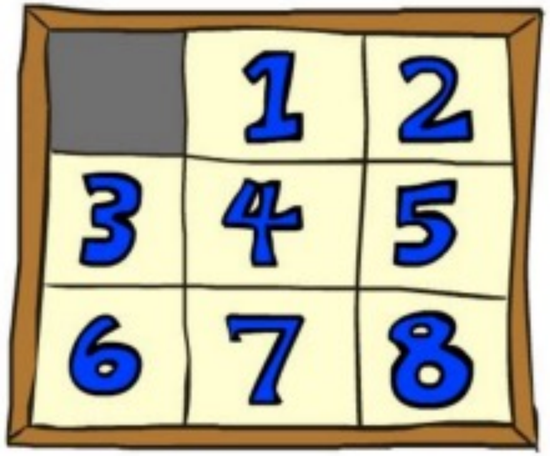


Search Heuristic

- Eight puzzle: Display the numbers in the “goal state”
  - Action: Move one object to the empty square
- Constraint in action:
  - Tiles can only move to the empty square
  - Tiles can only move to adjacent position
  - Two tiles can not occupy same spot



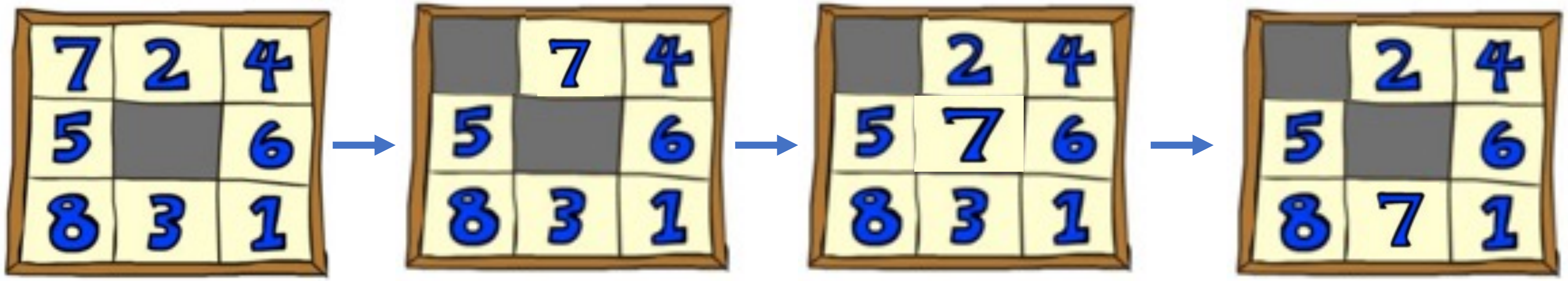
Start State



Goal State

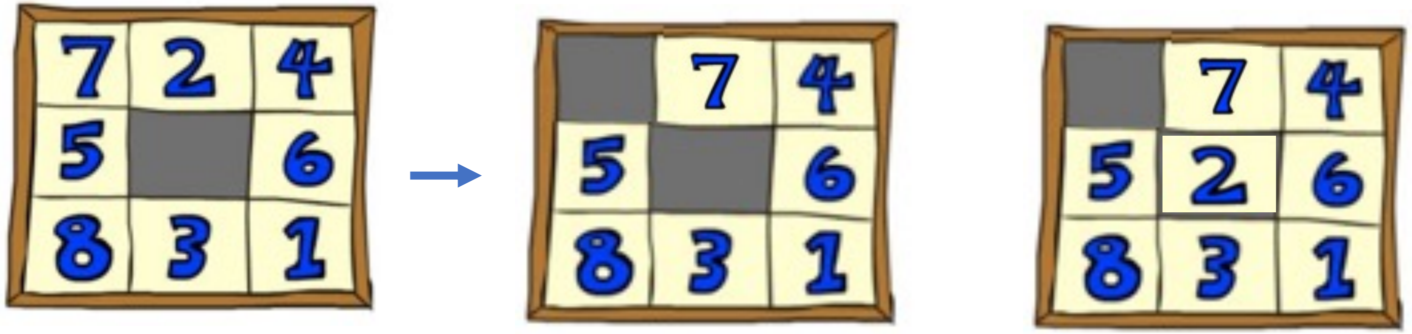
Search Heuristic

- What if we remove “Two tiles can not occupy same spot” and “Tiles can only move to the empty square” ?
  - Then every tiles can directly move toward goal position
- h(n): Sum of number of moves between each tile’s current position and its goal position
- Admissible? Yes. Consistency?



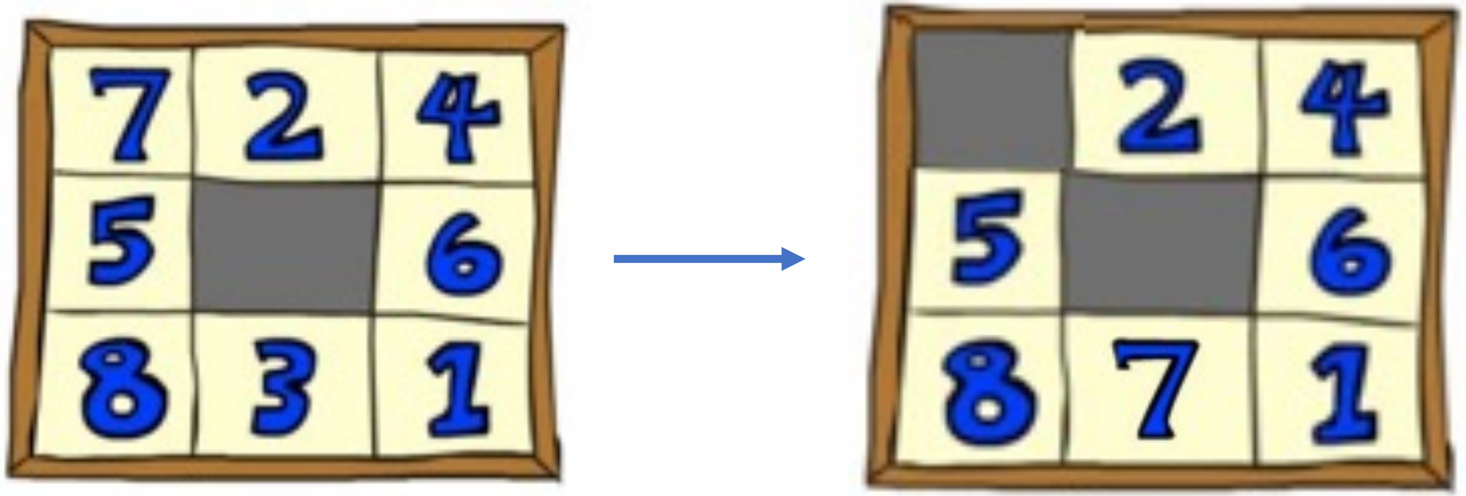
Search Heuristic

- h(n): Sum of number of moves between each tile’ s current position and its goal position
- Consistency: If an action has cost c, then taking that action can only cause a drop in heuristic of at most c
- Move 7 to the right spot
  - Drop in heuristic is 1
  - Actual Cost is at least 1



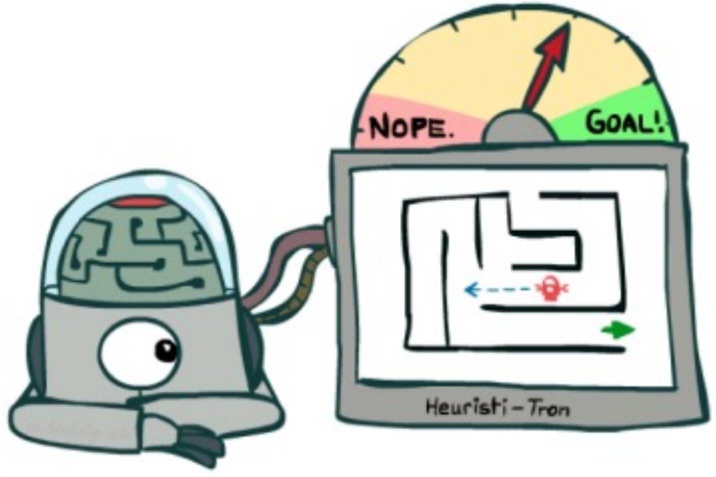
Search Heuristic

- What if we remove all the three constraints?
  - Then every tiles can jump to goal position
- h(n): Number of Misplaced Tiles
- Admissible? Yes. Consistent? Yes



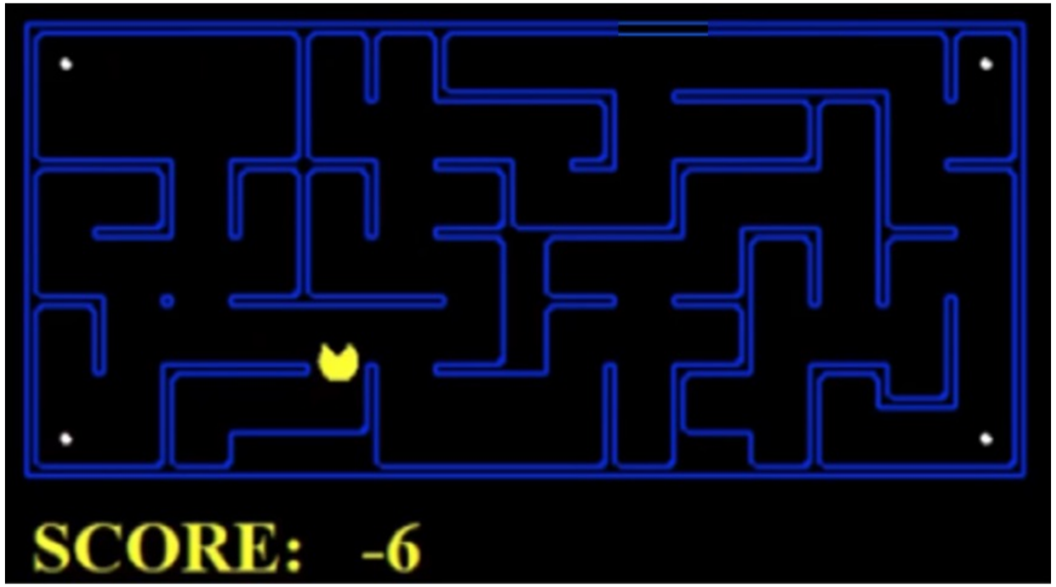
Search Heuristic

- More biased estimate
  - Remove more constraint
  - Advantage:
    - Easy and cheap to solve
  - Disadvantage:
    - More node expanded in search algorithm
- More accurate estimate
  - Close to actual cost
  - Advantage:
    - Less node expanded
  - Disadvantage:
    - more calculation for the heuristic function



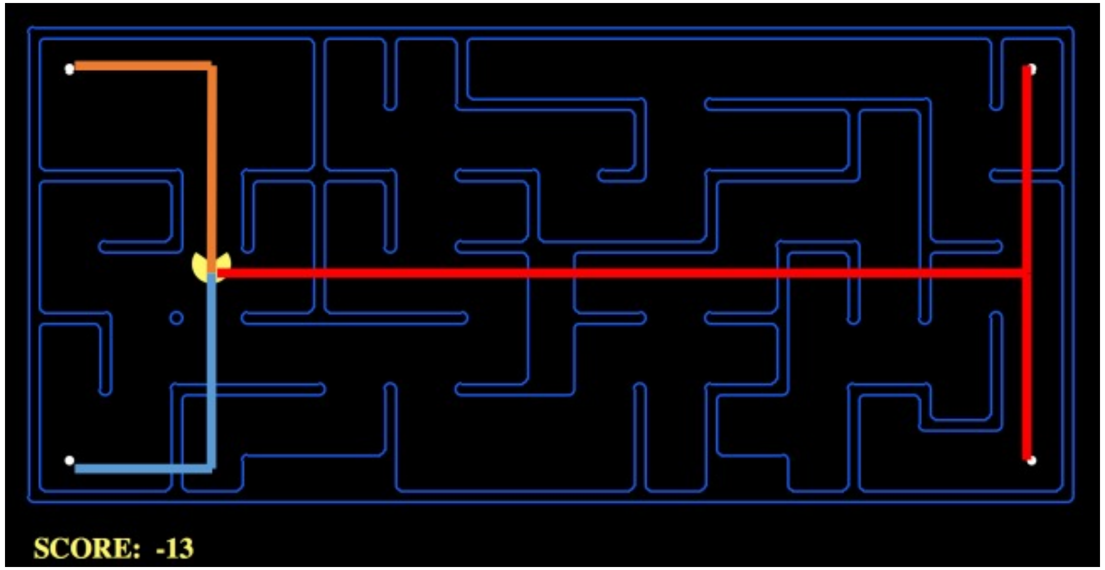
Search Heuristic

- Corner Problem: Find four dots in four corners
  - Action: up, down, left, right from one position to another
    - Can not cross wall
  - Noted:
    - Graph search: Never expand a state twice
    - Encoded state as (position, unvisited corner)



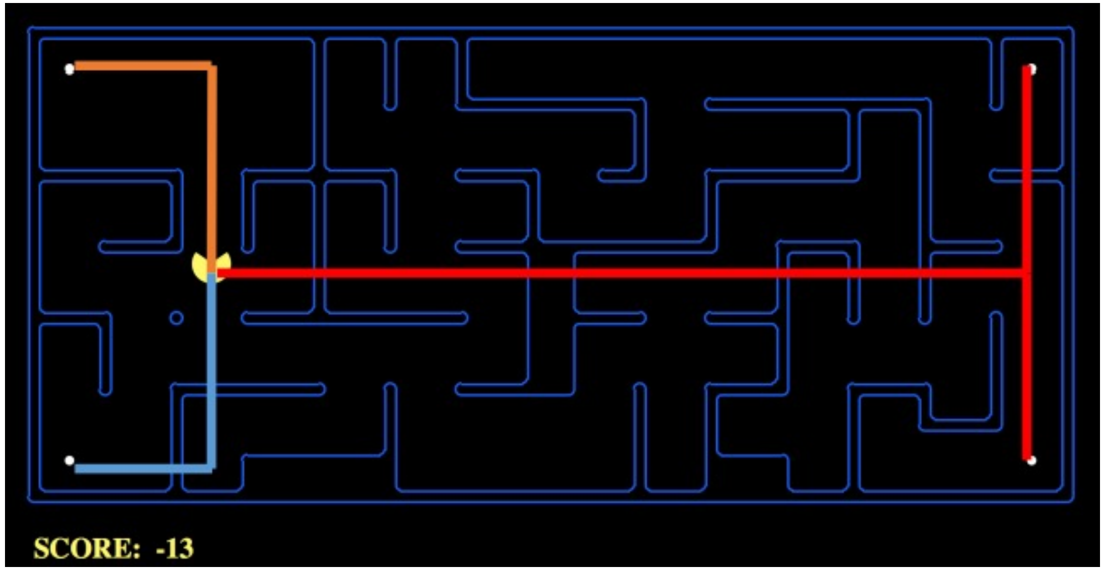
Search Heuristic

- Constraint:
  - Can not cross wall
  - …
- Intuition: Use Manhattan distance
  - Sum of Manhattan distance to unvisited corners.



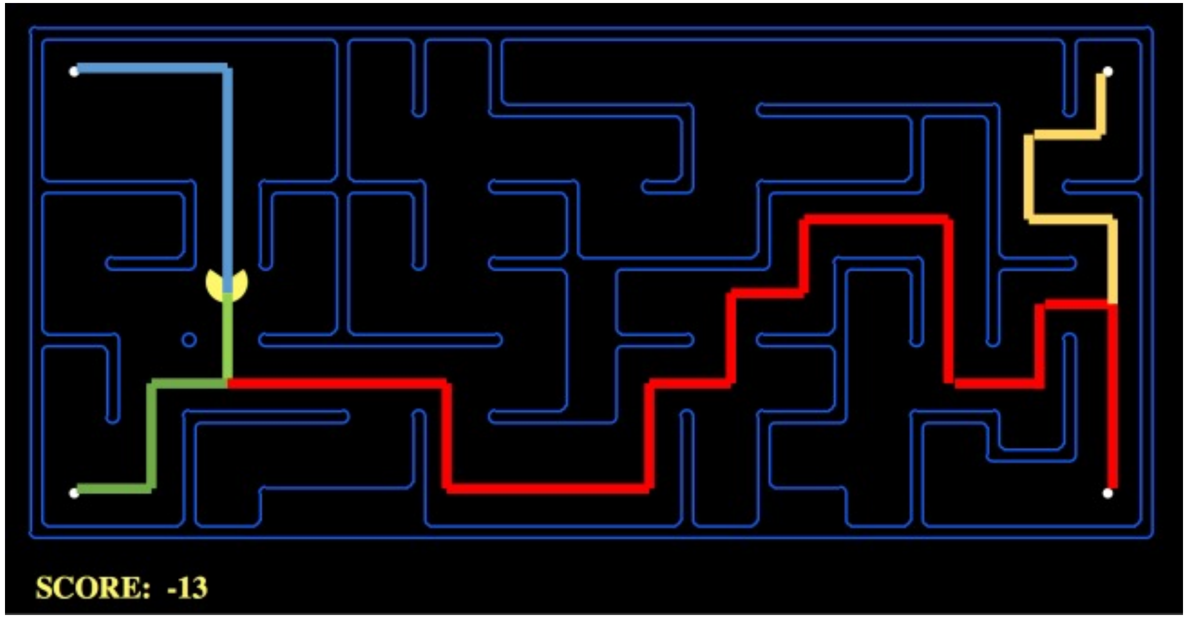
Search Heuristic

- h(n): Sum of Manhattan distance to unvisited corners.
  - Is it a good heuristic function?
- No! Heuristic value might not change.



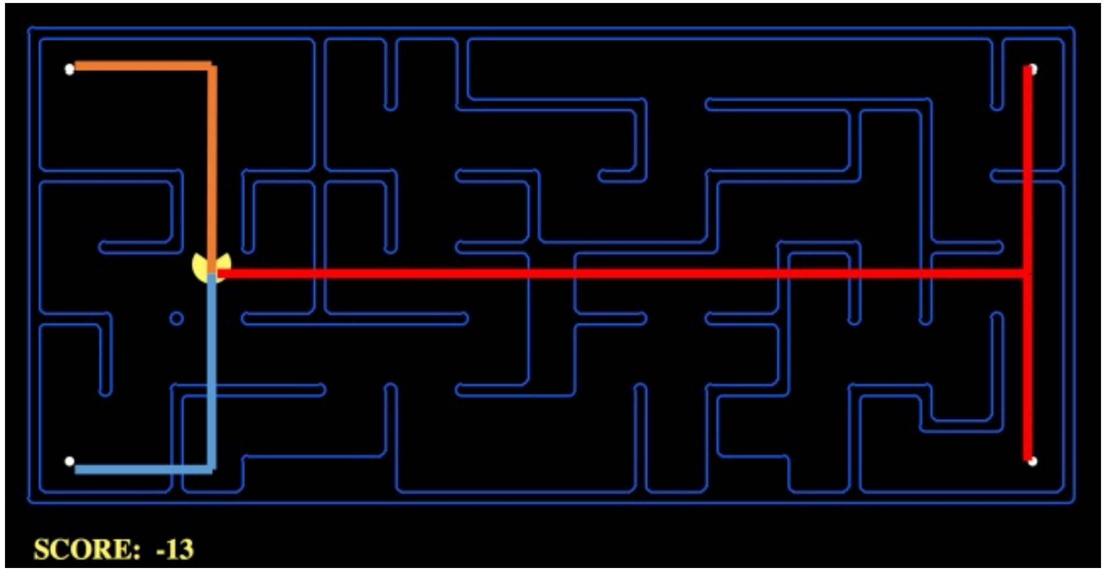
Search Heuristic

- Observe:
  - Corner with largest distance is the last one to be visited
  - Always visit the closest unvisited corner
  - After reach one corner, agent always follow the border to reach another closest corner



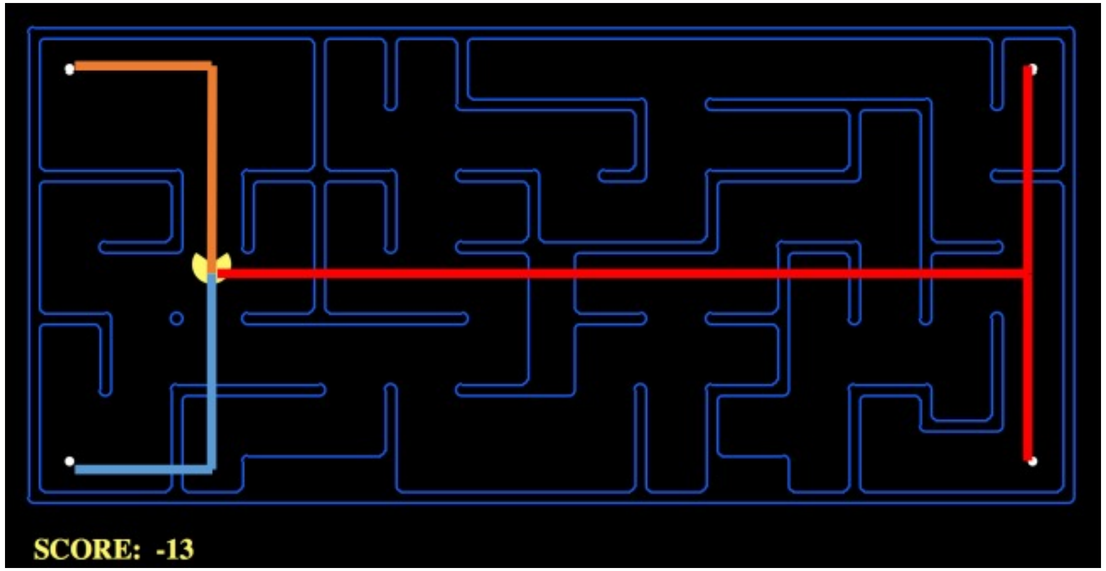
Search Heuristic

- Heuristic 1:
  - Remove walls
  - Corner with largest distance is the last one to be visited
- h(n): The largest Manhattan distance to the corner
  - After visiting farthest corner, the game reaches a goal state.



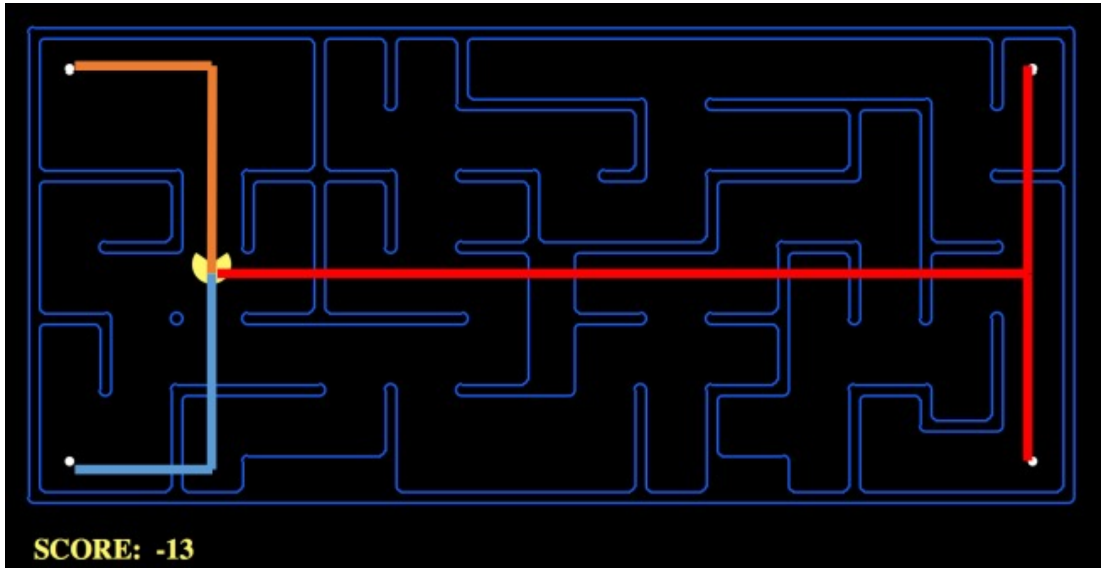
Search Heuristic

- h(n): The largest Manhattan distance to the corner
- Admissible? Obviously.
- Consistent?
  - Recall definition: If an action has cost c, then taking that action can only cause a drop in heuristic of at most c
  - Yes, its heuristic is calculated by the Manhattan distance



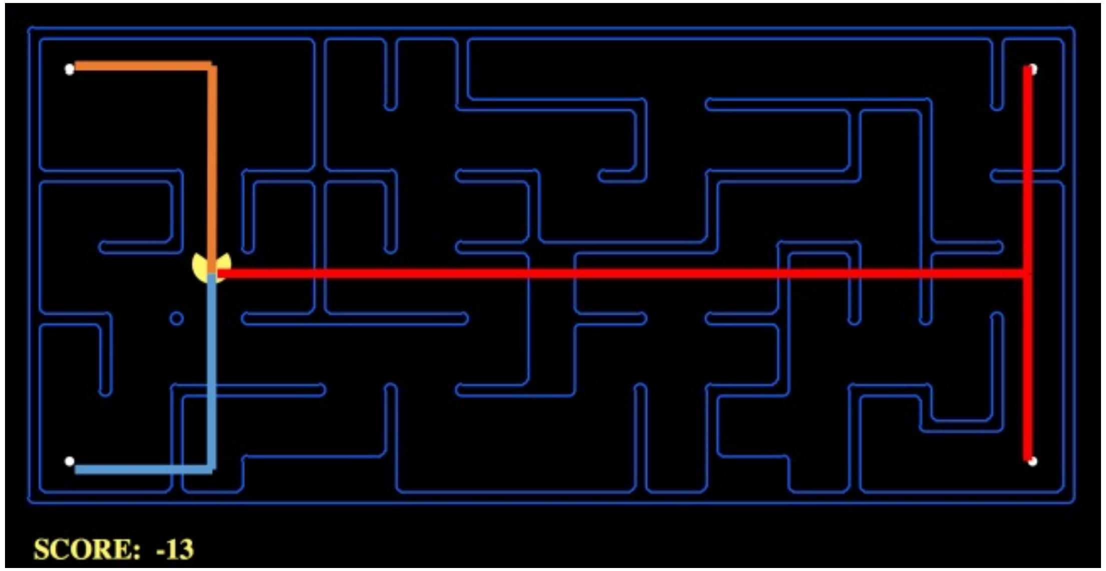
Search Heuristic

- Heuristic 2:
  - Remove walls
  - Always visit the closest unvisited corner
- h(n): The Manhattan distance to the closest corner
  - Actually, we can not use this heuristic
  - Anything wrong?



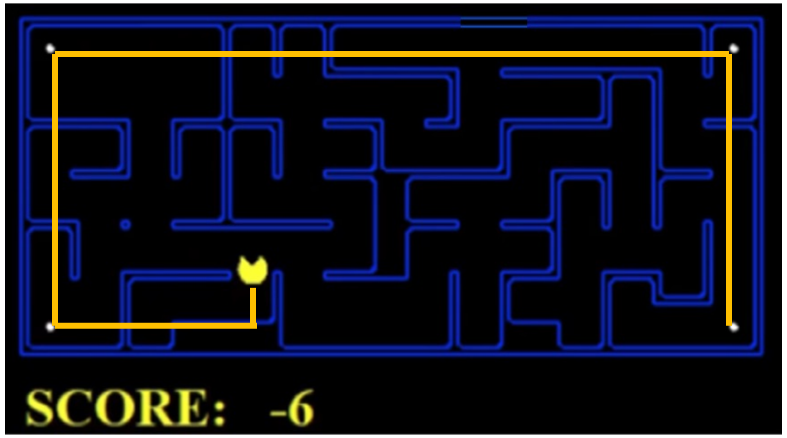
Search Heuristic

- Heuristic 2:
  - Remove walls
  - Always visit the closest unvisited corner
- h(n): The Manhattan distance to the closest corner
  - h(corner node) = 0? Only h(goal) can equal to 0!
  - Add something to h(corner node)



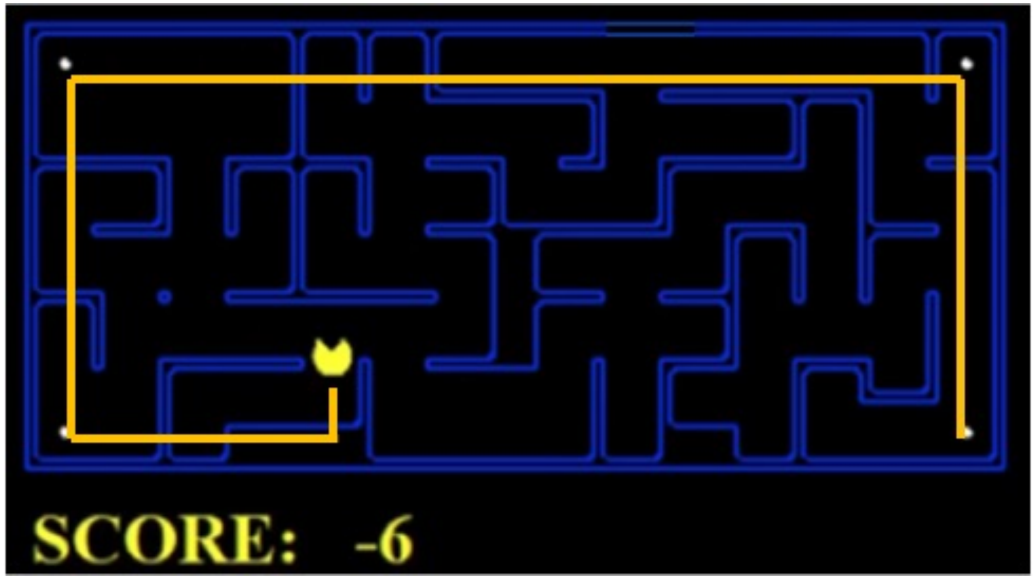
Search Heuristic

- Heuristic 2:
  - Remove walls
  - Always visit the closest unvisited corner
  - After reach one corner, agent always follow the border to reach another closest corner
- h(n): Manhattan distance to the closest corner + Distance between all the remaining corner



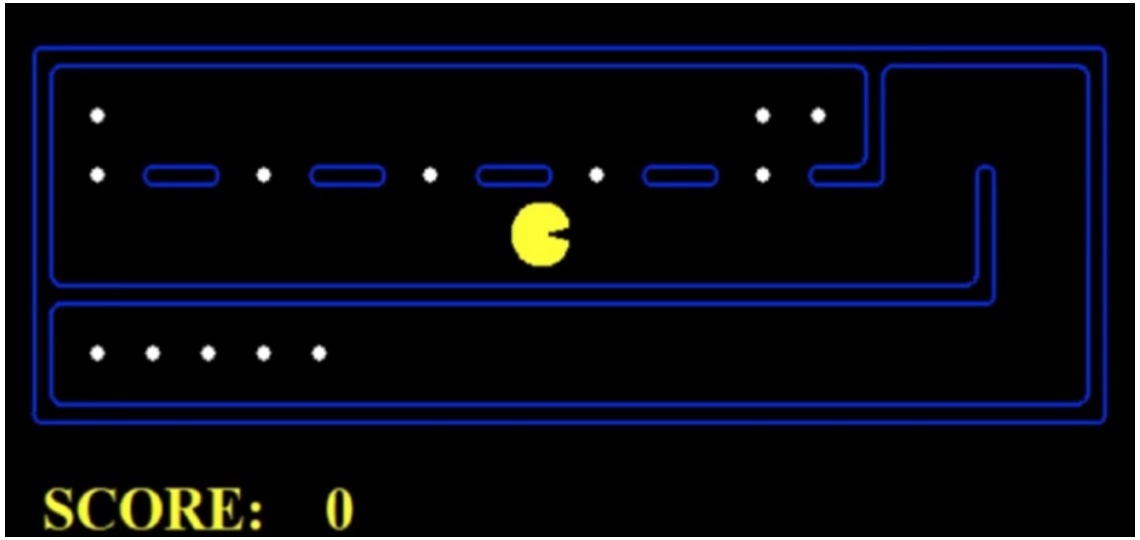
Search Heuristic

- h(n): Manhattan distance to the closest corner + distance between all the remaining dots
- Admissible? Obviously.
- Consistent?
  - Yes, similar to heuristic 1, its heuristic is calculated by the Manhattan distance



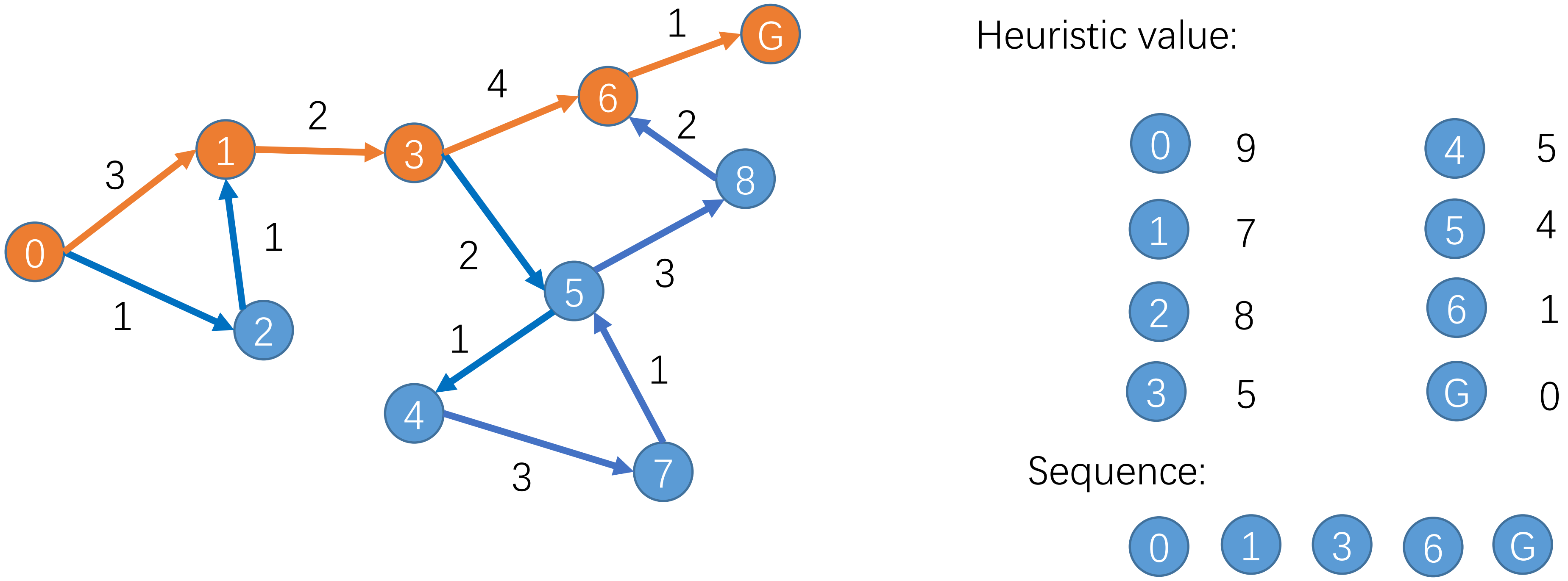
Search Heuristic

- Eat all dots: Find all dots
  - Action: up, down, left, right from one position to another
    - Can not cross wall
  - Possible h(n):
    - Largest distance between current position to each dot
    - ...Think about it!



Greedy search

- Greedy search expand a node that is closed to a goal state
- It modifies the breadth-first strategy by always expanding the lowest-heuristic node on the fringe



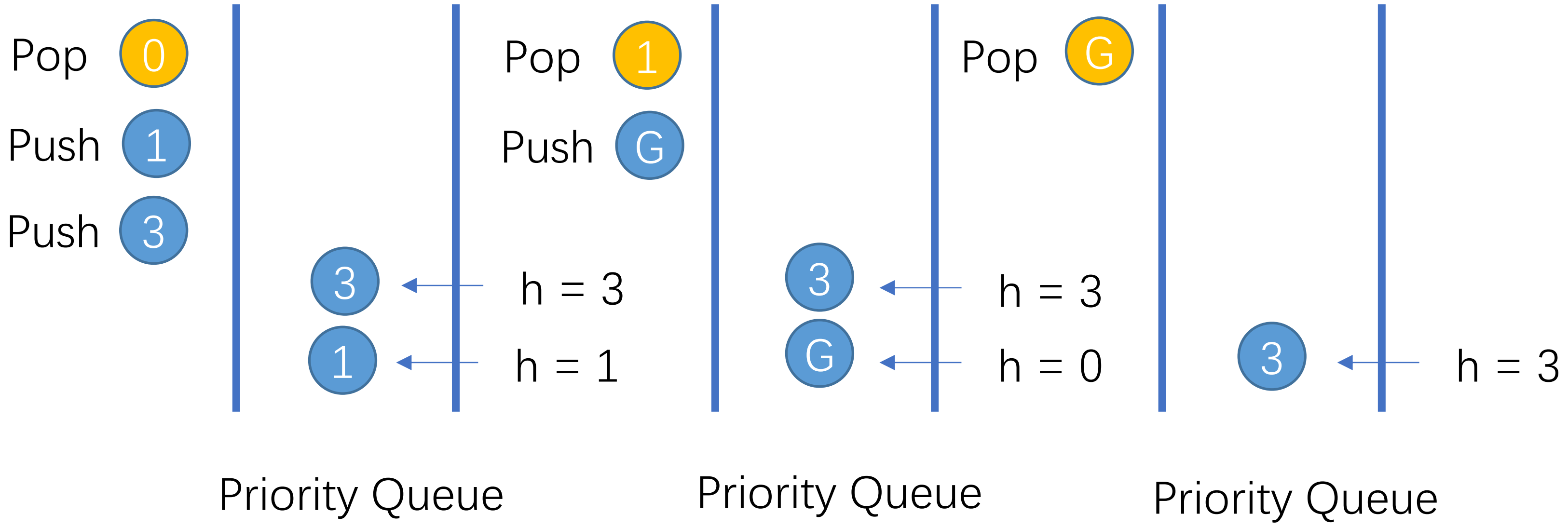
Greedy search

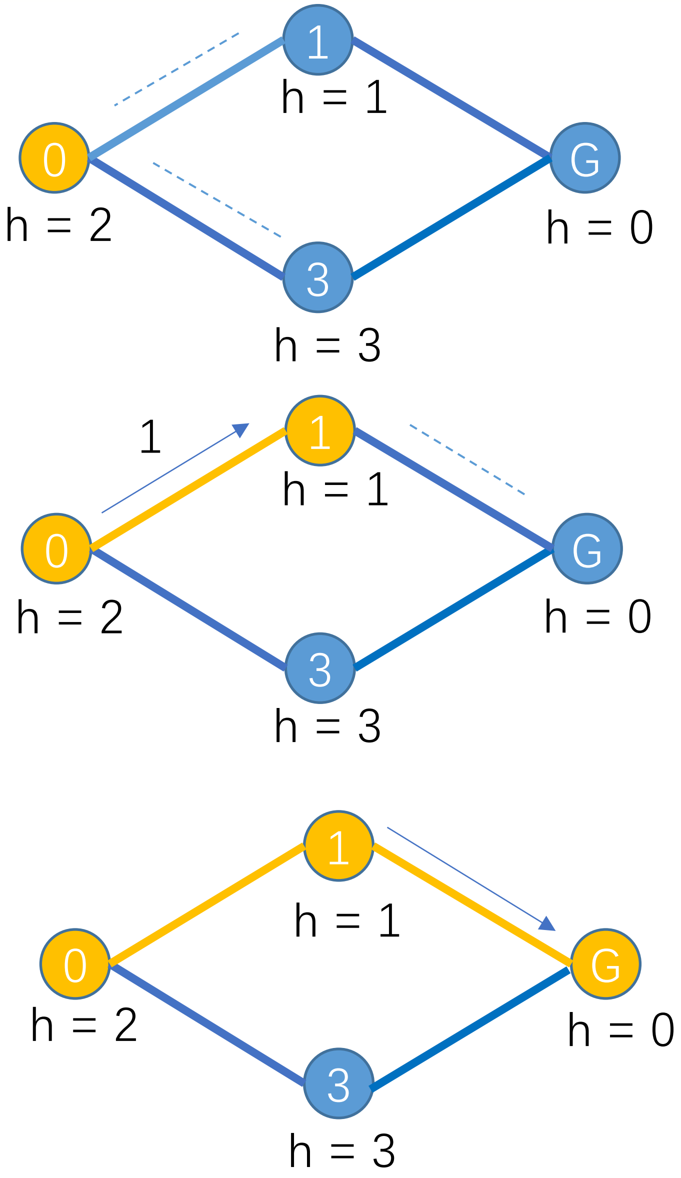
Priority Queue matches the process of Greedy

Push: Store neighbors of current node

Pop: Choose nodes with lowest heuristic

Priority: Sorting nodes with heuristic





Greedy search

UCS:

```
visited_list = [ ]
Def UCS(node):
    queue = priority_queue.push(node)
    while queue is not empty:
        node = queue.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    v.cost = node.cost + Cost(node, v)
                    queue.push(v)
                    queue.sort(by_cost)
```

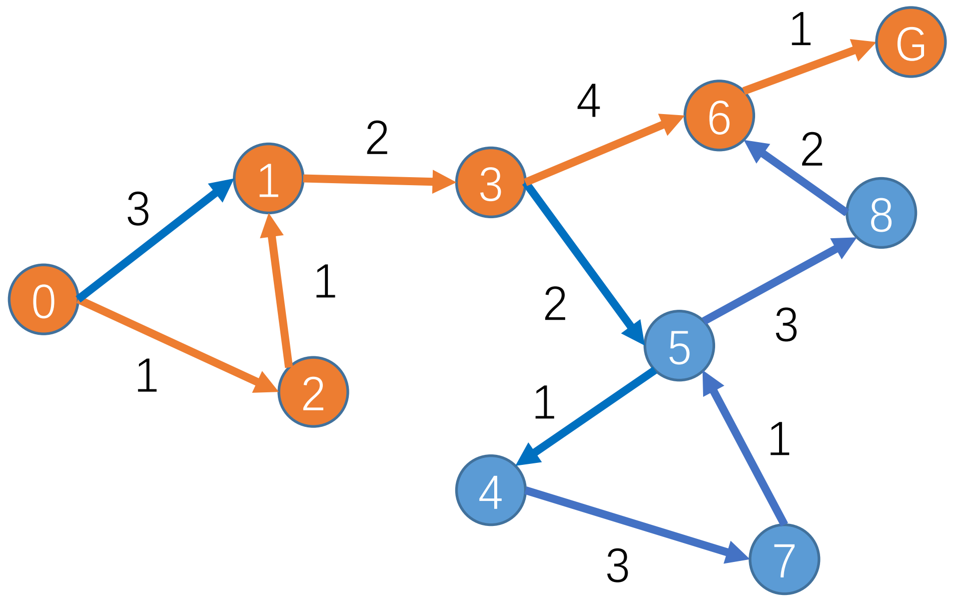
Greedy Search:

```
visited_list = [ ]
Def GreedySearch(node):
    queue = priority_queue.push(node)
    while queue is not empty:
        node = queue.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    v.heuristic = Heuristic(v)
                    queue.push(v)
                    queue. sort(by_ heuristic)

Def Heuristic(node):
    return abs(node.x – goal.x) + abs(node.y – goal.y)
```

A* search

- A* search expand a node that is closed to a goal state
- It modifies the breadth-first strategy by always expanding the node with lowest cost and heuristic on the fringe



	Cost	Heuristic	f
0	0	9	9
1	2	7	9
2	1	8	9
3	4	5	9

	Cost	Heuristic	f
4	7	5	12
5	6	4	10
6	8	1	9
G	9	0	9

Sequence:

0 2 1 3 6 G

A* search

Greedy Search:

```
For v in Neighbors(node):
    if v not in visited_list:
        v.heuristic = Heuristic(v)
        queue.push(v)
        queue. sort(by_ heuristic)
```

UCS:

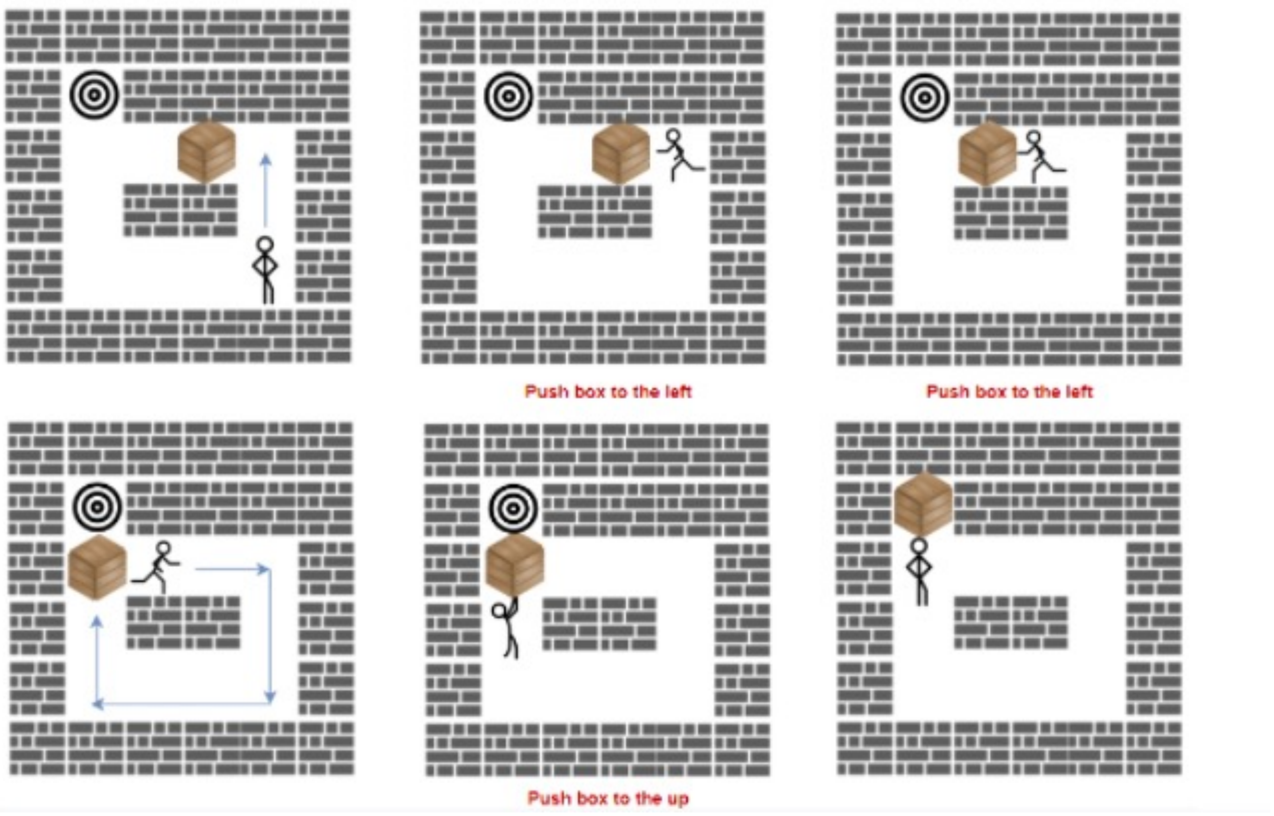
```
For v in Neighbors(node):
    if v not in visited_list:
        v.cost = node.cost + Cost(node, v)
        queue.push(v)
        queue. sort(by_cost)
```

A* Search:

```
visited_list = [ ]
Def AStar(node):
    queue = priority_queue.push(node)
    while queue is not empty:
        node = queue.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    v.cost = node.cost + Cost(node, v)
                    v.heuristic = Heuristic(v)
                    v.f =  v.cost + v.heuristic
                    queue.push(v)
                    queue. sort(by_ f)
```

A* practice

- A storekeeper is a game in which the player pushes boxes around in a warehouse trying to get them to target locations.
- Find Minimum Moves to Move a Box to Their Target Location



A* practice

- The game is represented by an m x n grid of characters grid where each element is a wall, floor, or box.
- Your task is to move the box 'B' to the target position 'T' under the following rules:
  - The character 'S' represents the player. The player can move up, down, left, right in grid if it is a floor (empty cell).
  - The character '.' represents the floor which means a free cell to walk.
  - The character '#' represents the wall which means an obstacle (impossible to walk there).
  - There is only one box 'B' and one target cell 'T' in the grid.
  - The box can be moved to an adjacent free cell by standing next to the box and then moving in the direction of the box. This is a push.
  - The player cannot walk through the box.
  - Return the minimum number of pushes to move the box to the target. If there is no way to reach the target, return -1.

A* practice

- The game is represented by an m x n grid of characters grid where each element is a wall, floor, or box.
  - # wall
  - B box
  - T target position
  - S start point
  - . free cell to walk

```
Input: grid = [["#","#","#","#","#","#"],
               ["#","T",".",".","#","#"],
               ["#",".","#","B",".","#"],
               ["#",".",".",".",".","#"],
               ["#",".",".",".","S","#"],
               ["#","#","#","#","#","#"]]
Output: 5
Explanation:  push the box down, left, left, up and up.
```

```
Input: grid = [["#","#","#","#","#","#","#"],
               ["#","S","#",".","B","T","#"],
               ["#","#","#","#","#","#","#"]]
Output: -1
```

A* practice

- Use search algorithm to find minimum moves to move a box to the target location
- Formulate to a search problem
  - Agent traverse the grid world
    - Encode state as (person, box)
    - Find all possible state
    - If box == target, end the algorithm

A* practice

- Build a BFS first:
- Step 1 process input:

```
rows, cols = len(grid), len(grid[0])
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == "T":
                target = (r, c)
            if grid[r][c] == "B":
                start_box = (r, c)
            if grid[r][c] == "S":
                start_person = (r, c)
```

```
Input: grid = [["#","#","#","#","#","#"],
               ["#","T",".",".","#","#"],
               ["#",".","#","B",".","#"],
               ["#",".",".",".",".","#"],
               ["#",".",".",".","S","#"],
               ["#","#","#","#","#","#"]]
Output: 5
Explanation:  push the box down, left, left, up and up.
```

A* practice

- Build a BFS first:
- Step 2 build environment:

```
def out_bounds(location):
# return whether the location is in the grid and not a wall
    r, c = location
    if r < 0 or r >= rows:
        return True
    if c < 0 or c >= cols:
        return True
    return grid[r][c] == "#"
```

```
Input: grid = [["#","#","#","#","#","#"],
               ["#","T",".",".","#","#"],
               ["#",".","#","B",".","#"],
               ["#",".",".",".",".","#"],
               ["#",".",".",".","S","#"],
               ["#","#","#","#","#","#"]]
Output: 5
Explanation:  push the box down, left, left, up and up.
```

A* practice

- Build a BFS first:
- Step 3 find data structure:
  - For BFS, we use Queue (First in first out)
  - Also build a visited list
- What should be encoded in a state in Queue?
  - Target: Moves of box
  - Position state: (person, box)
  - Then every state is (Moves, person, box)
- What should be encoded in a state in visited list?
  - Position state: (person, box)

```
queue = [[0, start_person, start_box]]
visited = set()
```

A* practice

- Build a BFS first:
- Step 4 build BFS:
  - In last tutorial

```
visited_list = [ ]
Def BFS_iterative(node):
    queue = queue.push(node)
    while queue is not empty:            Stop condition
        node = queue.pop()
        if node not in visited_list:     Visit control
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):    Store newly found successors
                if v not in visited_list:
                    queue.push(v)
```

A* practice

- BFS:

Stop condition

```
while queue:
    moves, person, box = queue.pop(0)
    if box == target:
        return moves
```

Visit control

```
    if (person, box) in visited: # do not visit same state again
        continue
    visited.add((person, box))
```

Store newly found successors

Environment

```
    for dr, dc in [[0, 1], [1, 0], [-1, 0], [0, -1]]:
        new_person = (person[0] + dr, person[1] + dc)
        if out_bounds(new_person):
            continue
        if new_person == box:
            new_box = (box[0] + dr, box[1] + dc)
            if out_bounds(new_box):
                continue
            queue.append([moves + 1, new_person, new_box])
        else:
            queue.append([moves, new_person, box])
```

A* practice

- Build A∗ from BFS:
- Step 5 Heuristic:
  - Target:
    - Minimum Moves to Move a Box to Their Target Location
  - Constraint on box:
    - Can not cross wall
    - …
  - Possible heuristic function:
    - Manhattan distance between box and target location

```
def heuristic(box):
        return abs(target[0] - box[0]) + abs(target[1] - box[1])
```

A* practice

- Build A∗ from BFS:
- Step 6 Build A∗:
  - Replace queue with priority queue
    - Sorted by (heuristic + cost), cost is number of moves currently
    - In python, you can use module heapq
  - What should be encoded in a state in Priority Queue?
    - F value: (heuristic + cost)
    - (Moves, person, box)

A* practice

- A*:

```
while priority_queue:
    f, moves, person, box = heapq.heappop(priority_queue)
    if box == target:
        return moves
    if (person, box) in visited: # do not visit same state again
        continue
    visited.add((person, box))

    for dr, dc in [[0, 1], [1, 0], [-1, 0], [0, -1]]:
        new_person = (person[0] + dr, person[1] + dc)
        if out_bounds(new_person):
            continue

        if new_person == box:
            new_box = (box[0] + dr, box[1] + dc)
            if out_bounds(new_box):
                continue
            heapq.heappush(priority_queue, [heuristic(new_box) + moves + 1, moves + 1, new_person, new_box])
        else:
            heapq.heappush(priority_queue, [heuristic(box) + moves, moves, new_person, box]) # box remains same
```

Thank you!