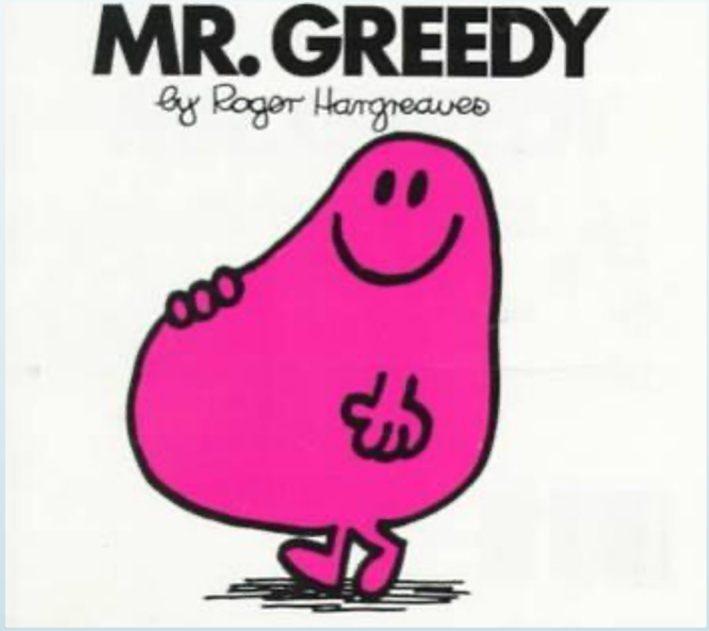
Introduction to Artificial Intelligence

Week 5

Learning by Searching

Greedy Approaches

- Makes the locally best choice at each step in the hope that the final solution will be optimal
- Does not always find the optimal solution



Greedy Algorithm Basics

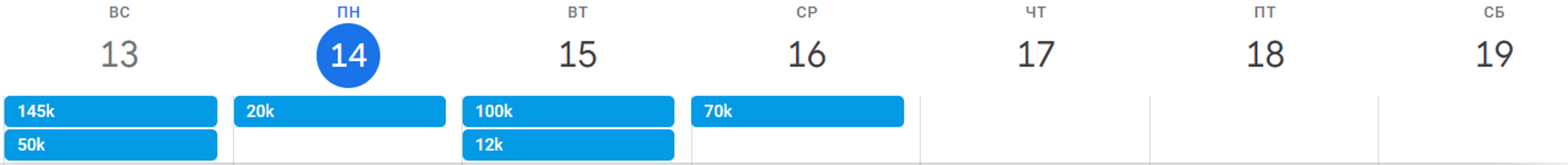
- Start with a search space from which a candidate solution will be taken from
- There is a number of decisions to be made in sequence to find a solution, at each step make a choice which maximizes the outcome
- Done

Example – job-shop scheduling

- You are given *n* tasks
- Each task has deadline, as well as its cost
- You take one day to complete the task
- Maximize the profit

Solution

- Do the most “expensive tasks” first
- Tasks will be sorted in descending order of cost and fill the schedule as follows:
  - if there is at least one more free place in the schedule for the order before its deadline, then we will put it in the very last of these places
  - otherwise we cannot complete it on time



When can you be greedy?

- **Dijkstra's algorithm** (search for the shortest paths from one of the vertices of the graph to all the others).
- **Huffman algorithm** (adaptive algorithm for optimal prefix coding of the alphabet with minimal redundancy).
- **Kruskal's algorithm** (search for a spanning tree of minimum weight in a graph).
- **Prim's algorithm** (search for a minimum weight spanning tree in a connected graph).

When is greedy not optimal?

- Coin exchange
- Knapsack Problem
- Traveling salesman problem
- Graph coloring problem

When does greedy give you optimum?

- If the structure of the problem is given by a **matroid**, then applying the greedy algorithm will produce a global optimum (Rado-Edmonds theorem)

- In terms of independence, a finite matroid M is a pair (E,I), where E is a finite set (called the ground set) and I is a family of subsets of E (called the independent sets) with the following properties:
  1. The empty set is independent, i.e., $\emptyset \in I$. Alternatively, at least one subset of E is independent
  2. Every subset of an independent set is independent, i.e., for each $A' \subseteq A \subseteq E$ if $A \in I$ then $A' \in I$
  3. If A and B are two independent sets and A has more elements than B, then there exists $x \in A \setminus B$ such that $B \cup \{x\}$ is in I.

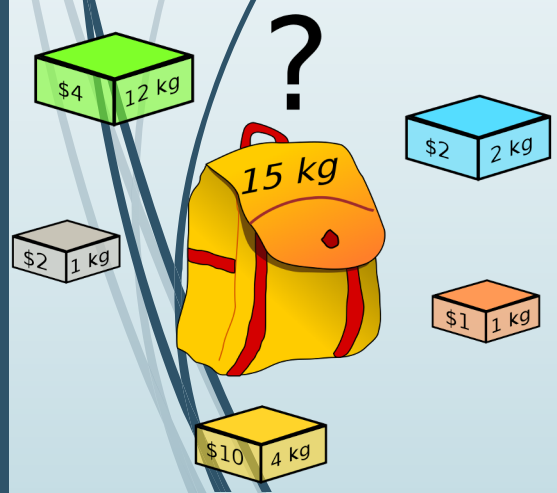
Is job-shop scheduling problem matroid?

- The ground set - a set of tasks
- Independent sets - successfully completed tasks.
- The weight of each application is its cost

Is the following pair a matroid?

- (1) The empty set of completed tasks is included in our set
- (2) If we sort the successfully completed tasks in order of increasing deadline. In this order, they will still succeed. Any subset of successfully completed tasks will be successfully completed.
- (3) Suppose we have two sets of successfully completed tasks A and B, and it is known that $|A| < |B|$. Let's standardly sort the tasks in order of increasing deadline in both sets. Let's take a task from B, which is not in A, and try to add it to set A. We will succeed, because if there were no gap in A, then this task should have been present.

Knapsack Problem



- The most common problem being solved is the **0-1 knapsack problem**, which restricts the number x_i of copies of each kind of item to zero or one. Given a set of *n* items numbered from 1 up to *n*, each with a weight w_i and a value v_i, along with a maximum weight capacity *W*,
  - maximize $\sum_{i=1}^{n} \vartheta_i x_i$
  - subject to $\sum_{i=1}^{n} \omega_i x_i \leq W \; and \; x_i \in \{0, 1\}$
- Here x_i represents the number of instances of item *i* to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

Exercise 1

- Write a program for Knapsack problem using C/C++/C#/Java/Python.
- Inputs:
  - Item values – v[ ] = {10, 120, 100, 10, 60}
    Weights – w[ ] = {5, 30, 5, 20, 10}
    Knapsack's capacity (size) – S = 50
- Output: summary value of a knapsack

A* Search

- Fast well-established algorithm for game AI
  - Pathways
  - Movement of NPC
  - RTS movement
    - Usually one member of a flock
- Works in areas where we do not know the connectivity
  - Makes it very suitable for PCG – because we do not need nodes defined by the generator –but can still understand the measurements
- Can take into account multiple different connection topologies and styles of movement
- Large number of iterative improvements and problem domain specific implementations with heuristics

Heuristics

- Rough estimate how far is the goal from current grid cell

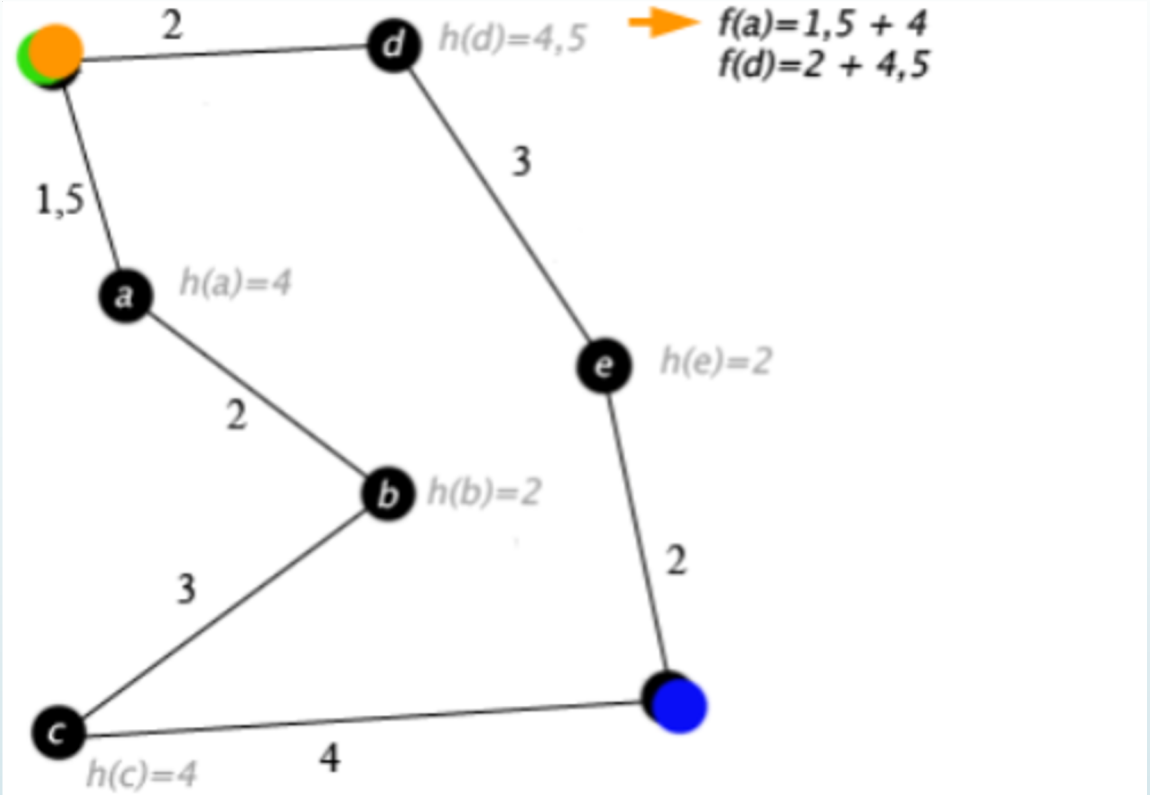
START	D=3.6	D=2.8	D=2.2	D=2
D=4.1	D=3.2	D=2.2	D=1.4	D=1
D=4	D=3	D=2	D=1	GOAL

Combine Cost and Heuristics: A*

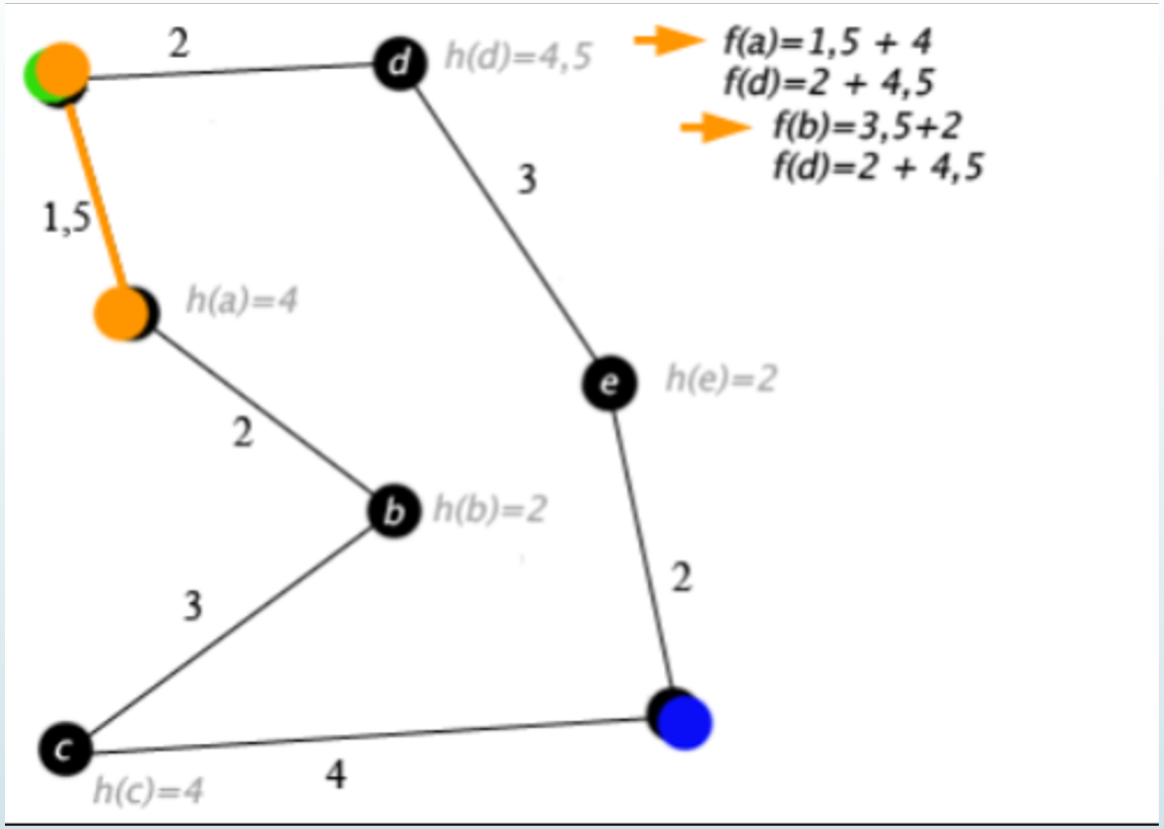
- G – cost function
- H – heuristics
- F = G + H – estimate of total cost

A* Coding Example

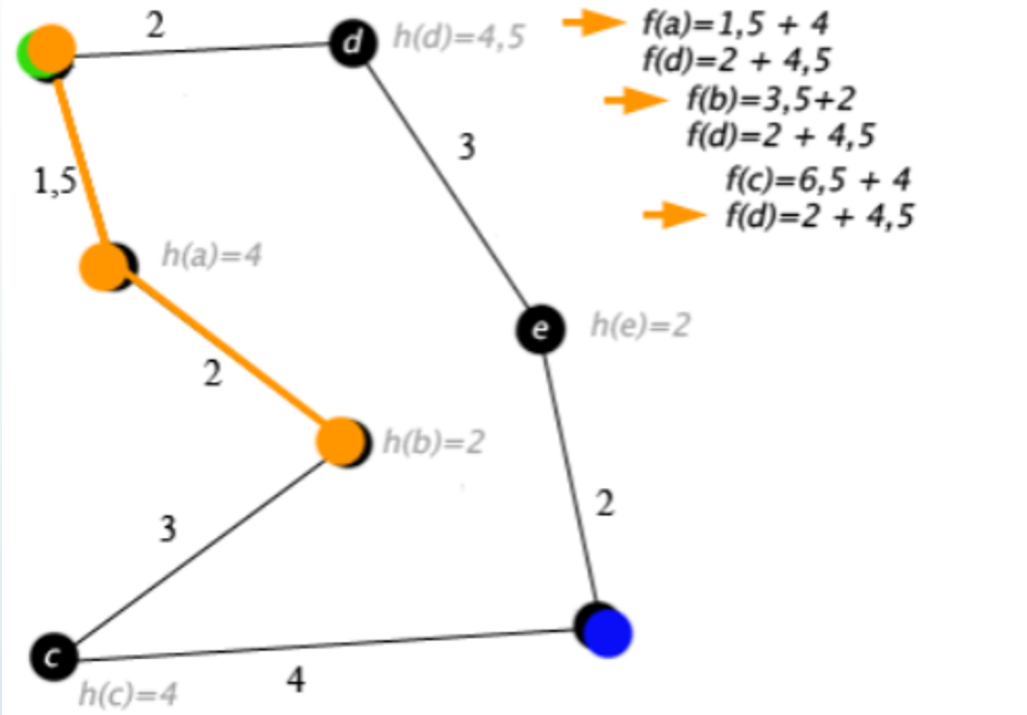
```
function A*(start, goal, f)
    % set of visited verticies
    var closed := the empty set
    % set of local solutions
    var open := make_queue(f)
    enqueue(open, path(start))
    while open is not empty
        var p := remove_first(open)
        var x := the last node of p
        if x in closed
            continue
        if x = goal
            return p
        add(closed, x)
        % add adjacent vertices
        foreach y in successors(x)
            enqueue(open, add_to_path(p, y))
    return failure
```



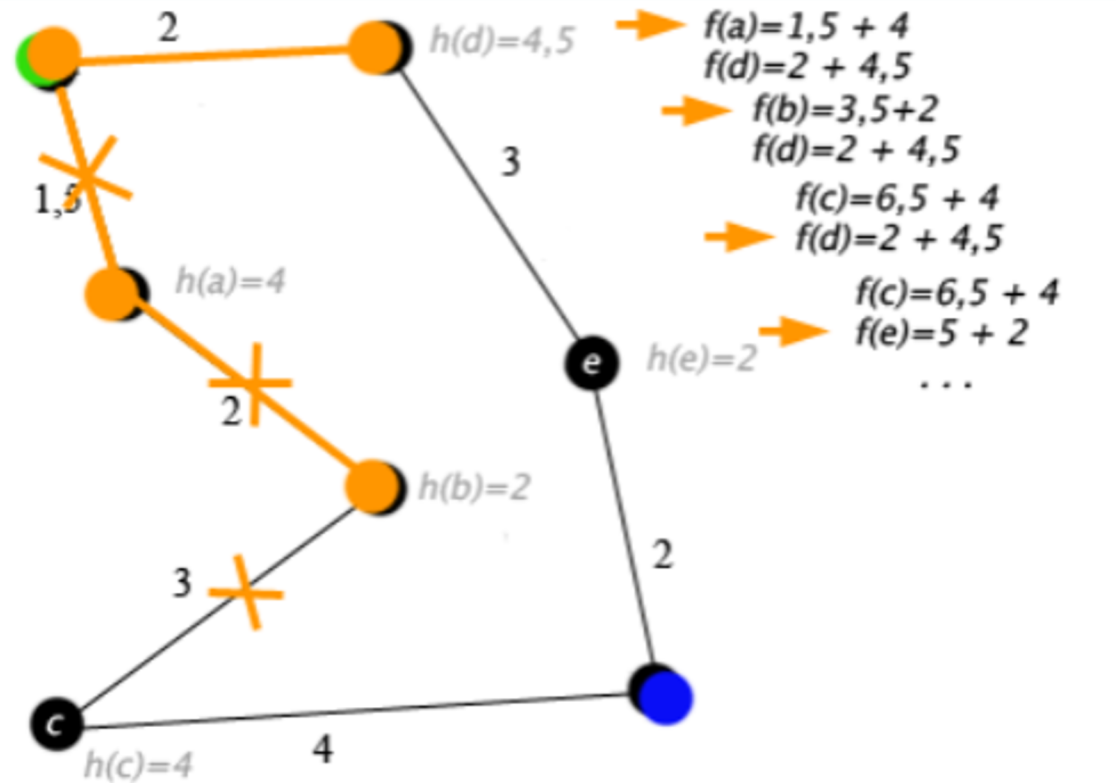
A* Coding Example



A* Coding Example



A* Coding Example

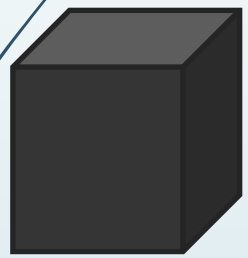


Heuristics (How to calculate h?)

- Either calculate the exact value of h (pre-compute the distance between each pair)
- OR
- Approximate the value of h using some heuristics

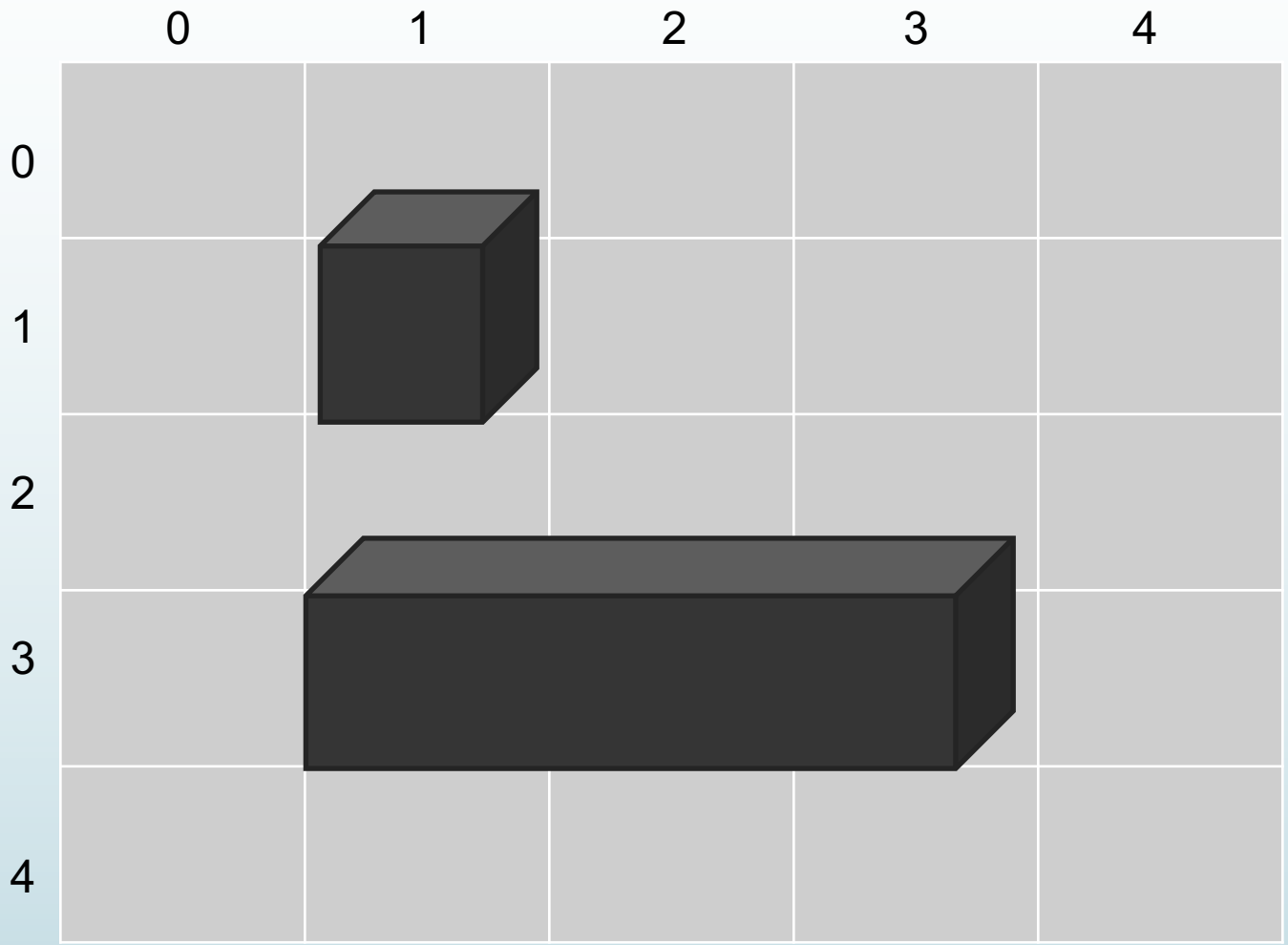
- **Manhattan Distance -** sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively. Used when we are allowed to move only in four directions only (right, left, top, bottom)
- **Diagonal Distance -** maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively. Used when we are allowed to move in eight directions only
- **Euclidean Distance -** used when we are allowed to move in any directions

World

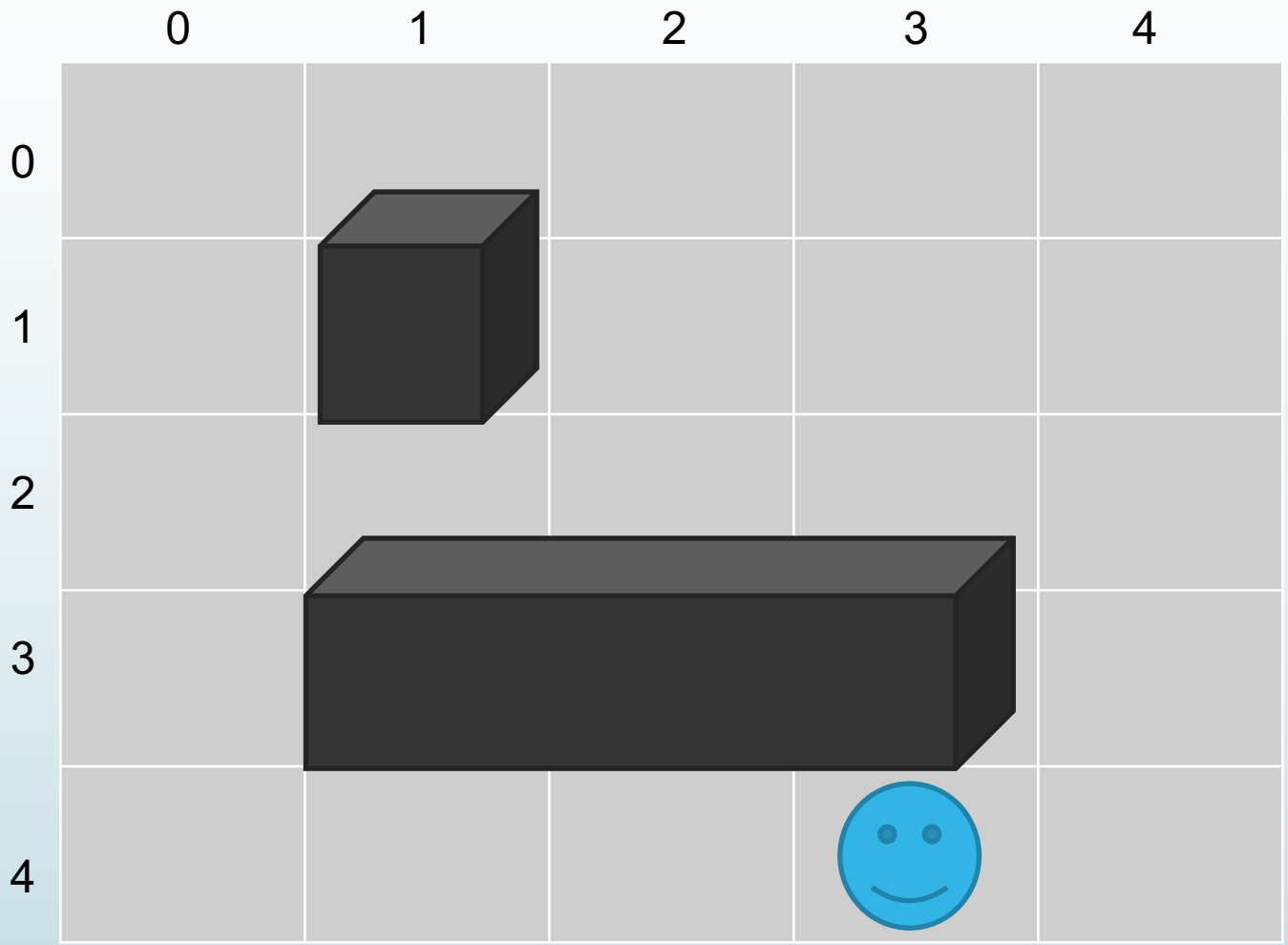


	0	1	2	3	4
0					
1					
2					
3					
4					

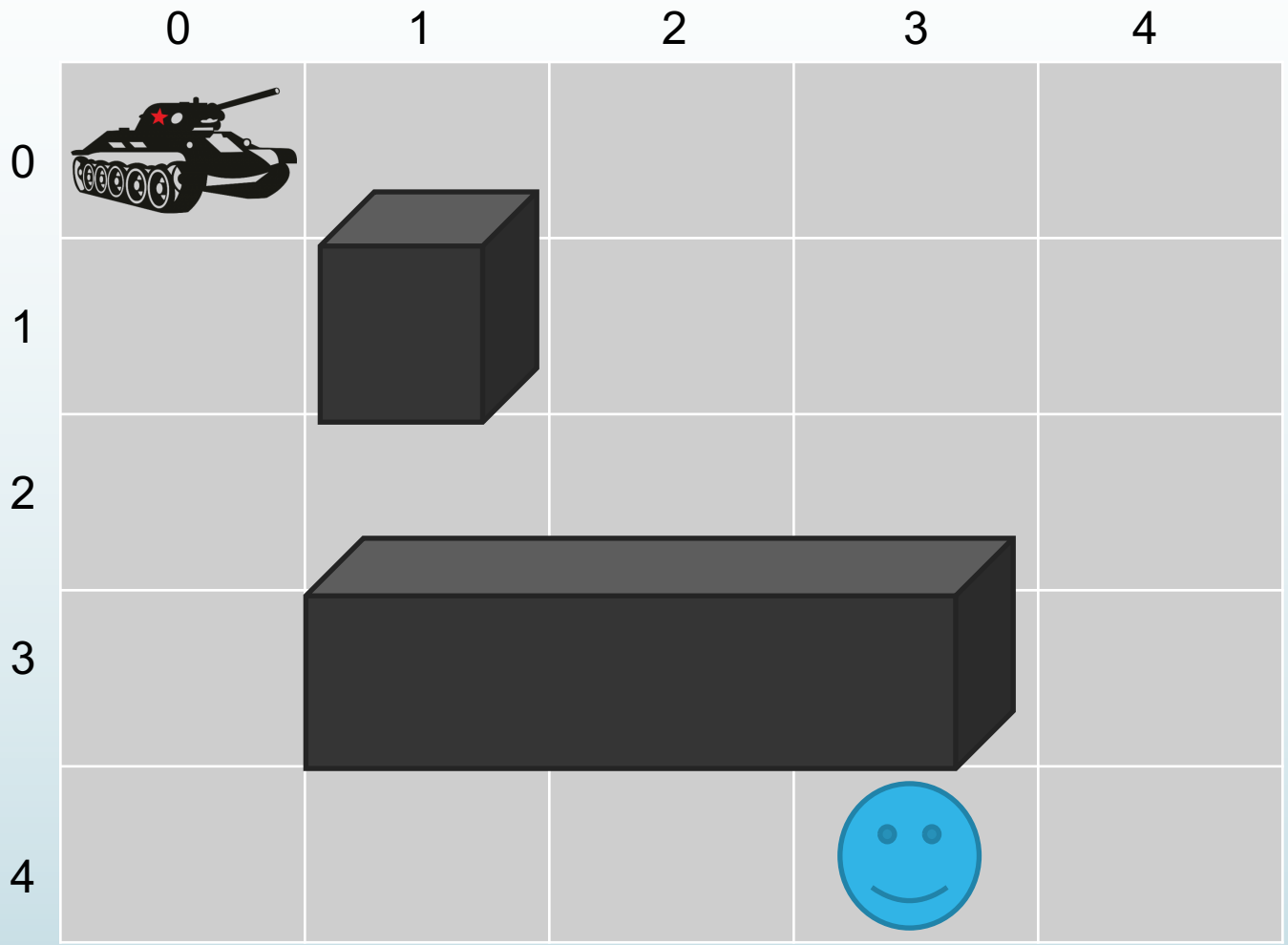
World With Obstacles



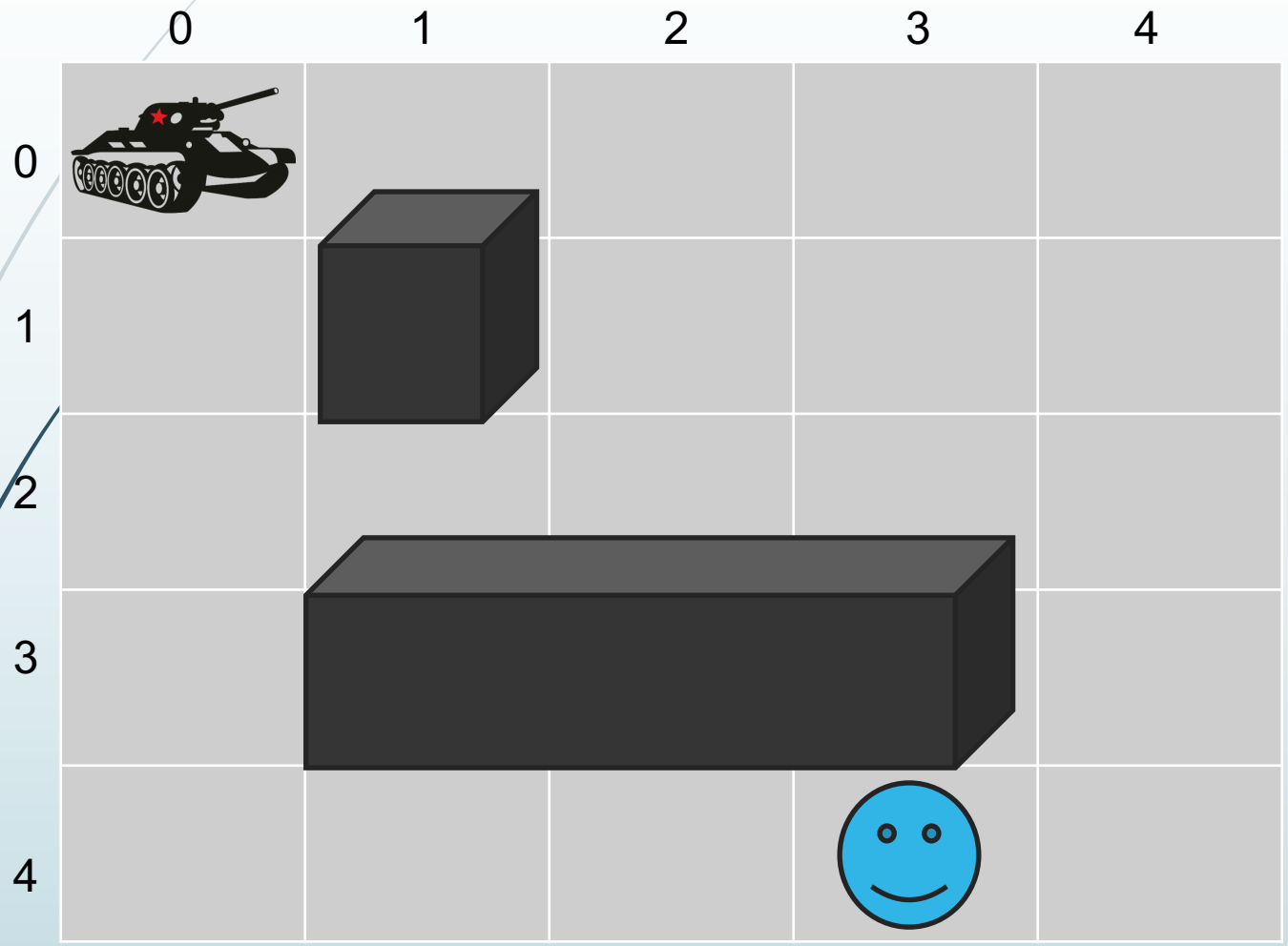
World With Obstacles and Goal



World With Obstacles, Goal and Start Position



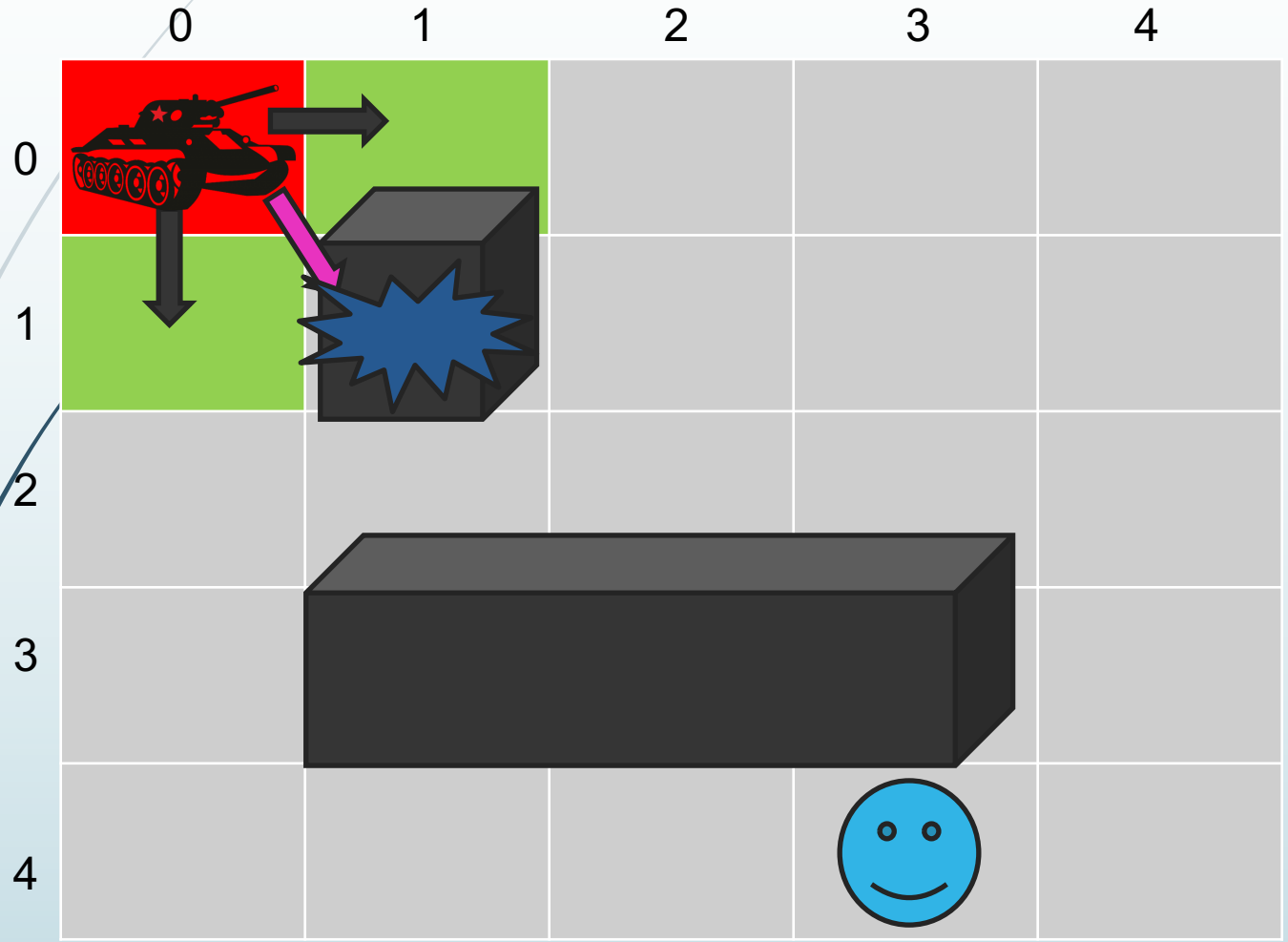
Add initial Point to Open List and Assign Cost



CLOSED:
NULL

OPEN:
(0,0)-0

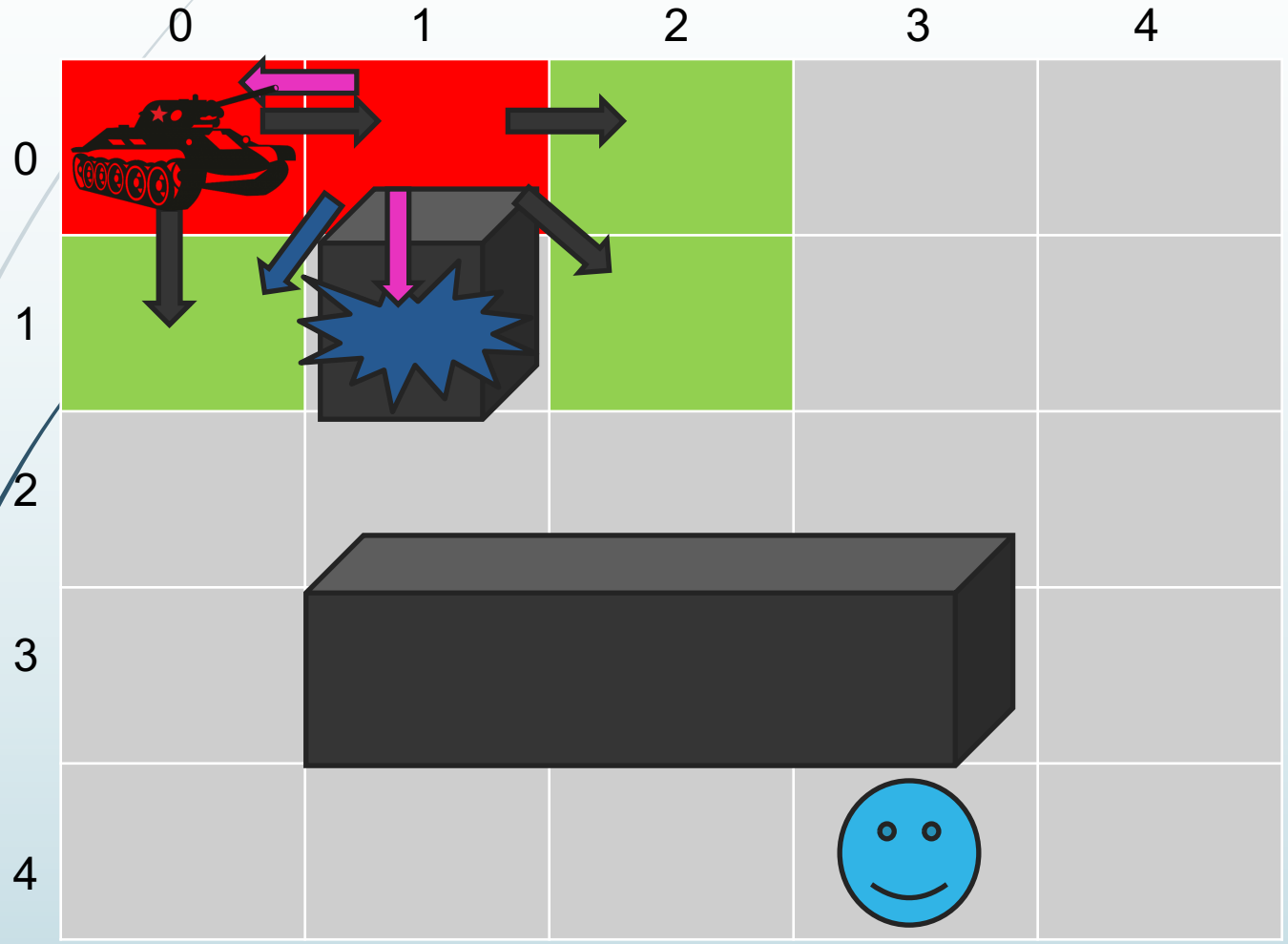
Move Lowest Cost to Closed and Search Neighbours



CLOSED:
(0,0)-0-NEW

OPEN:
(1,0)-1-NEW
(0,1)-1-NEW
(1,1)-COLLISION!

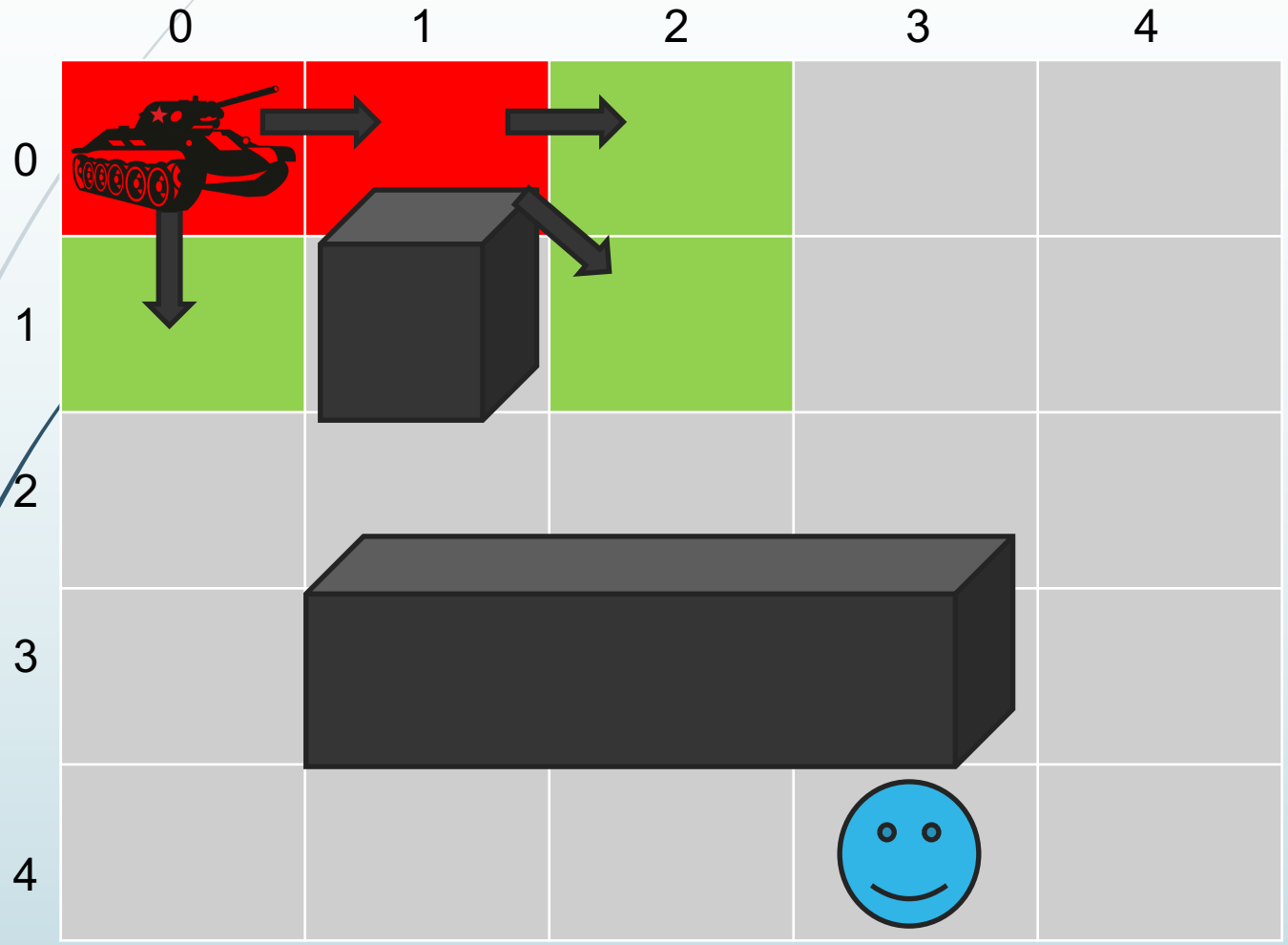
Move Lowest Cost to Closed and Search Neighbours



CLOSED:
(0,0)-0
(1,0)-1-NEW

OPEN:
(0,1)-1
(2,0)-2-NEW
(2,1)-2-NEW
(1,1)-COLLISION!
(0,1)-2-Cost is greater!
(0,0)-CLOSED!

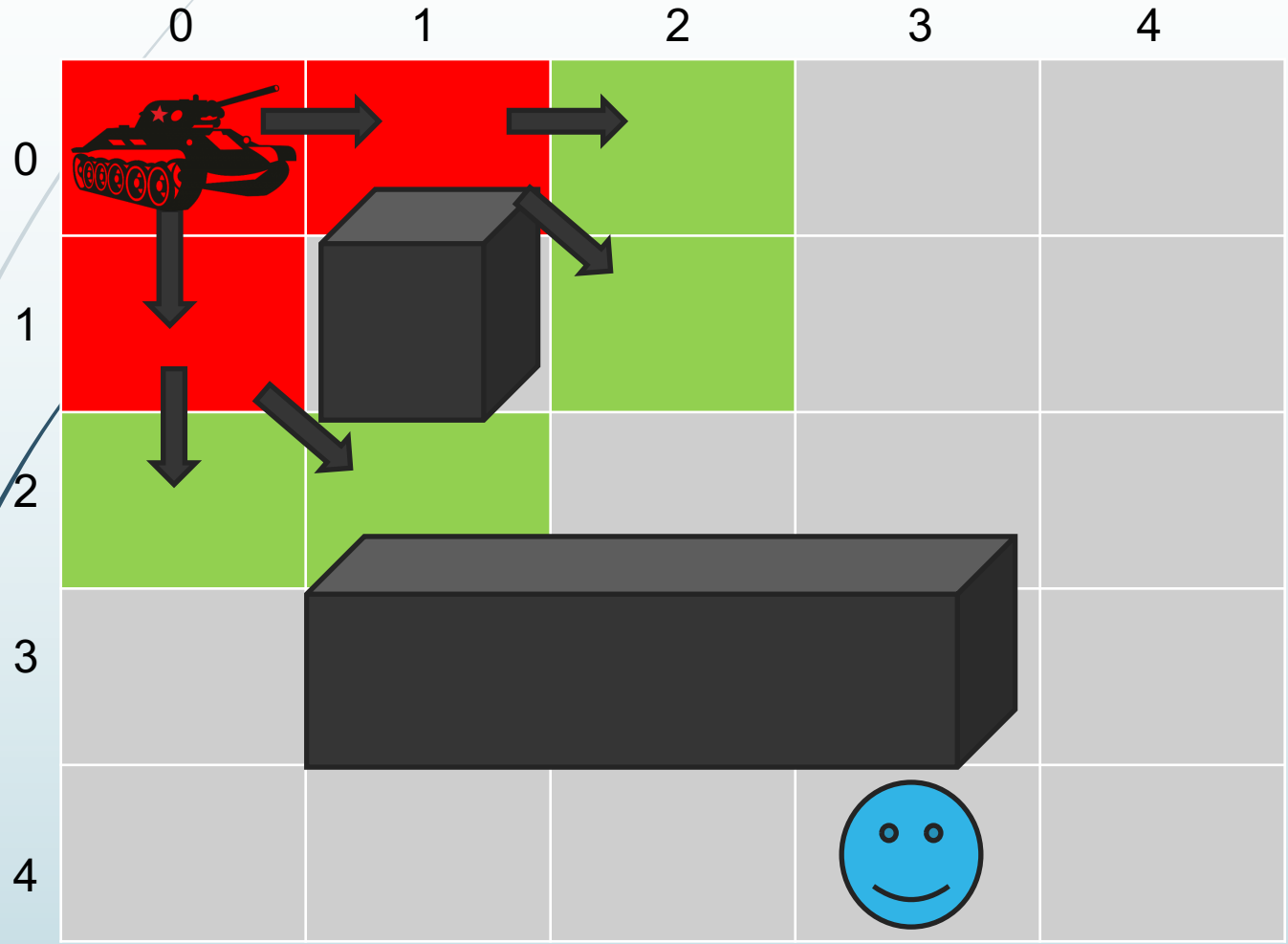
Open and Closed after Two Steps



CLOSED:
(0,0)-0
(1,0)-1

OPEN:
(0,1)-1
(2,0)-2
(2,1)-2

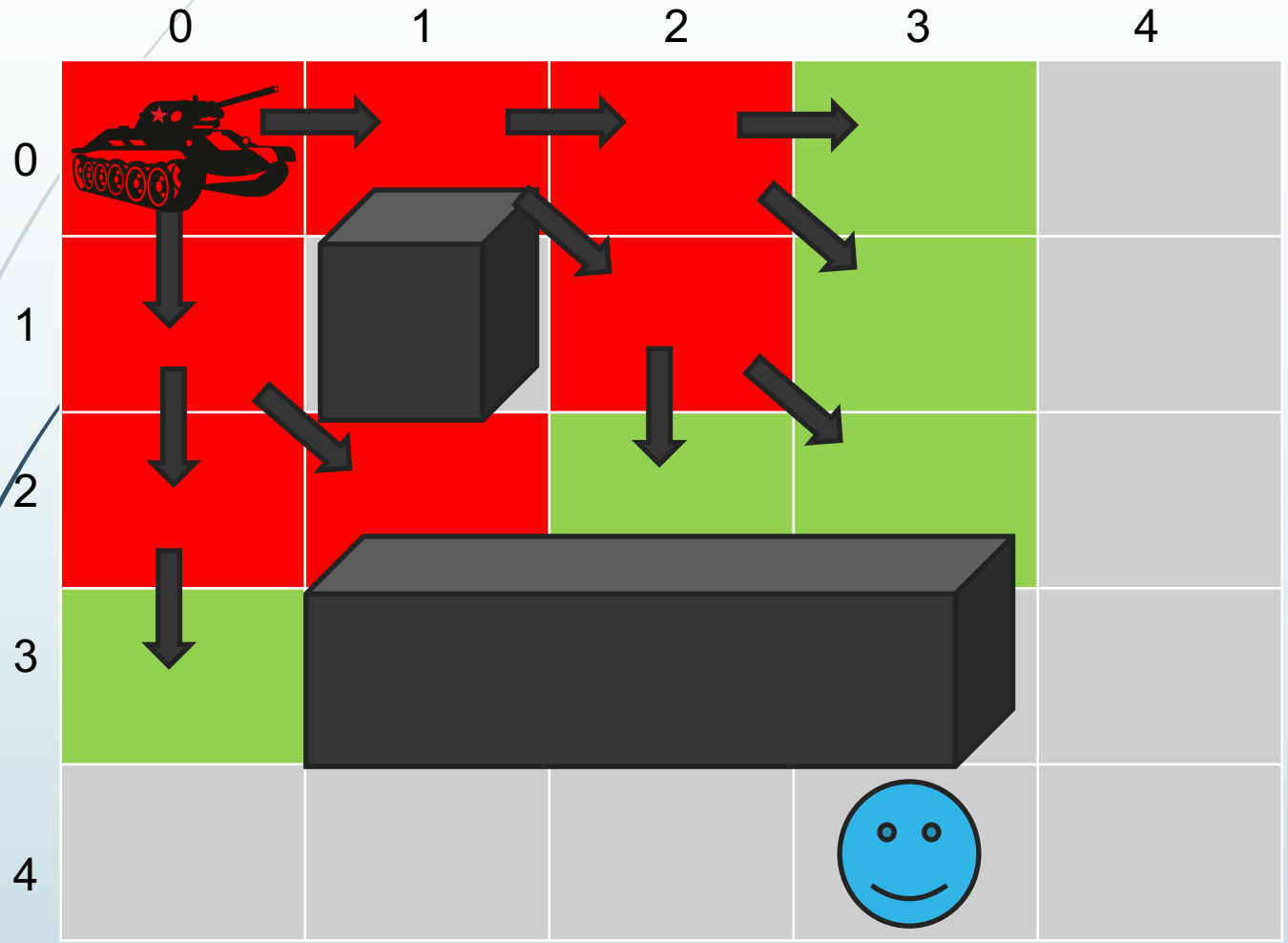
Move Lowest Cost to Closed and Search Neighbours (Only Valid)



CLOSED:
(0,0)-0
(1,0)-1
(0,1)-1-NEW

OPEN:
(2,0)-2
(2,1)-2
(0,2)-2-NEW
(1,2)-2-NEW

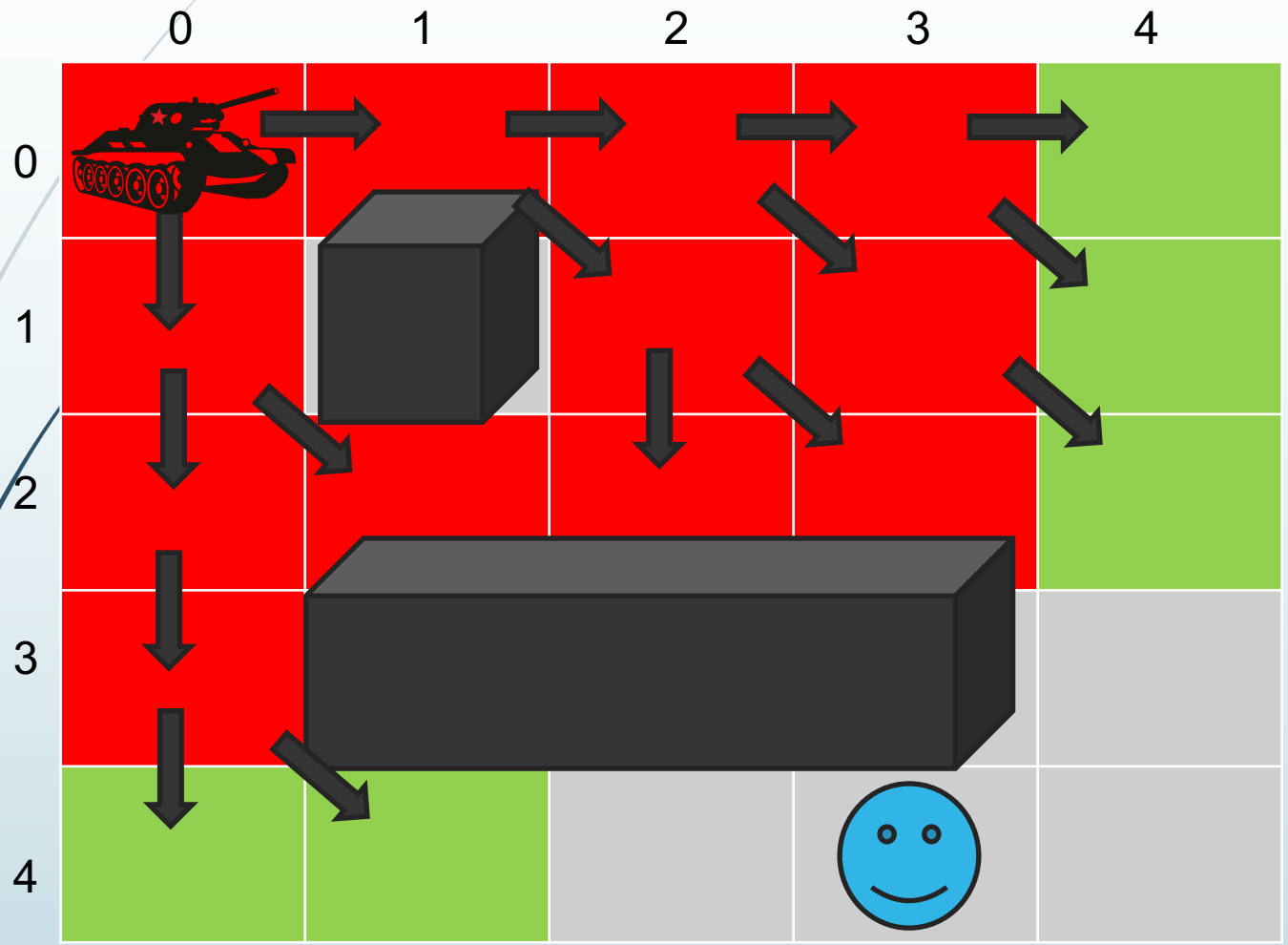
After All Distance 2 Locations Checked



CLOSED:
(0,0)-0, (1,0)-1, (0,1)-1, (2,0)-2, (2,1)-2, (0,2)-2, (1,2)-2

OPEN:
(3,0)-3, (3,1)-3, (3,2)-3, (2,2)-3, (0,3)-3

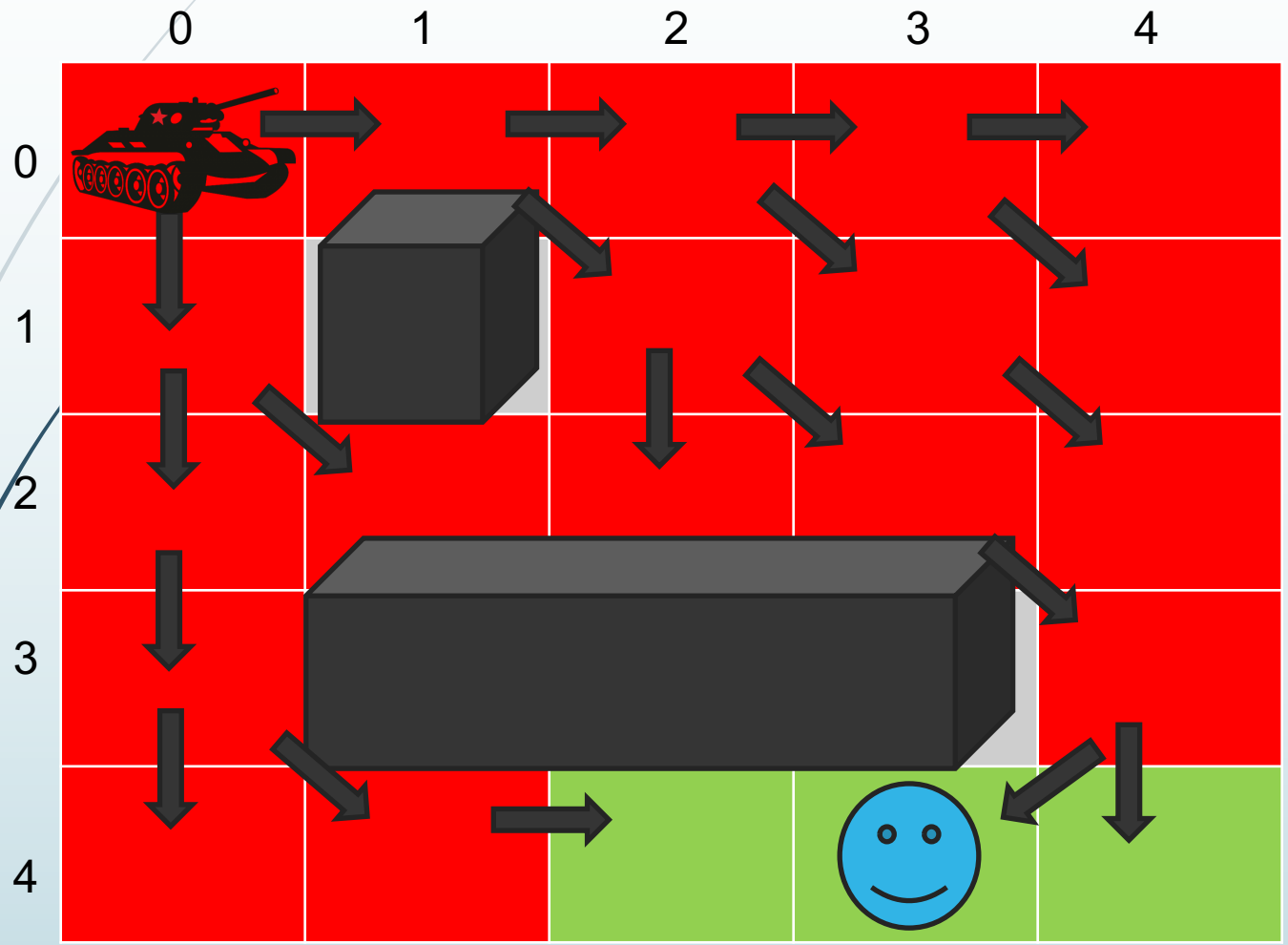
After All Distance 3 Locations Checked



CLOSED:
(0,0)-0, (1,0)-1, (0,1)-1, (2,0)-2, (2,1)-2, (0,2)-2, (1,2)-2, (3,0)-3, (3,1)-3, (3,2)-3, (2,2)-3, (0,3)-3

OPEN:
(4,0)-4, (4,1)-4, (4,2)-4, (0,4)-4, (1,4)-4

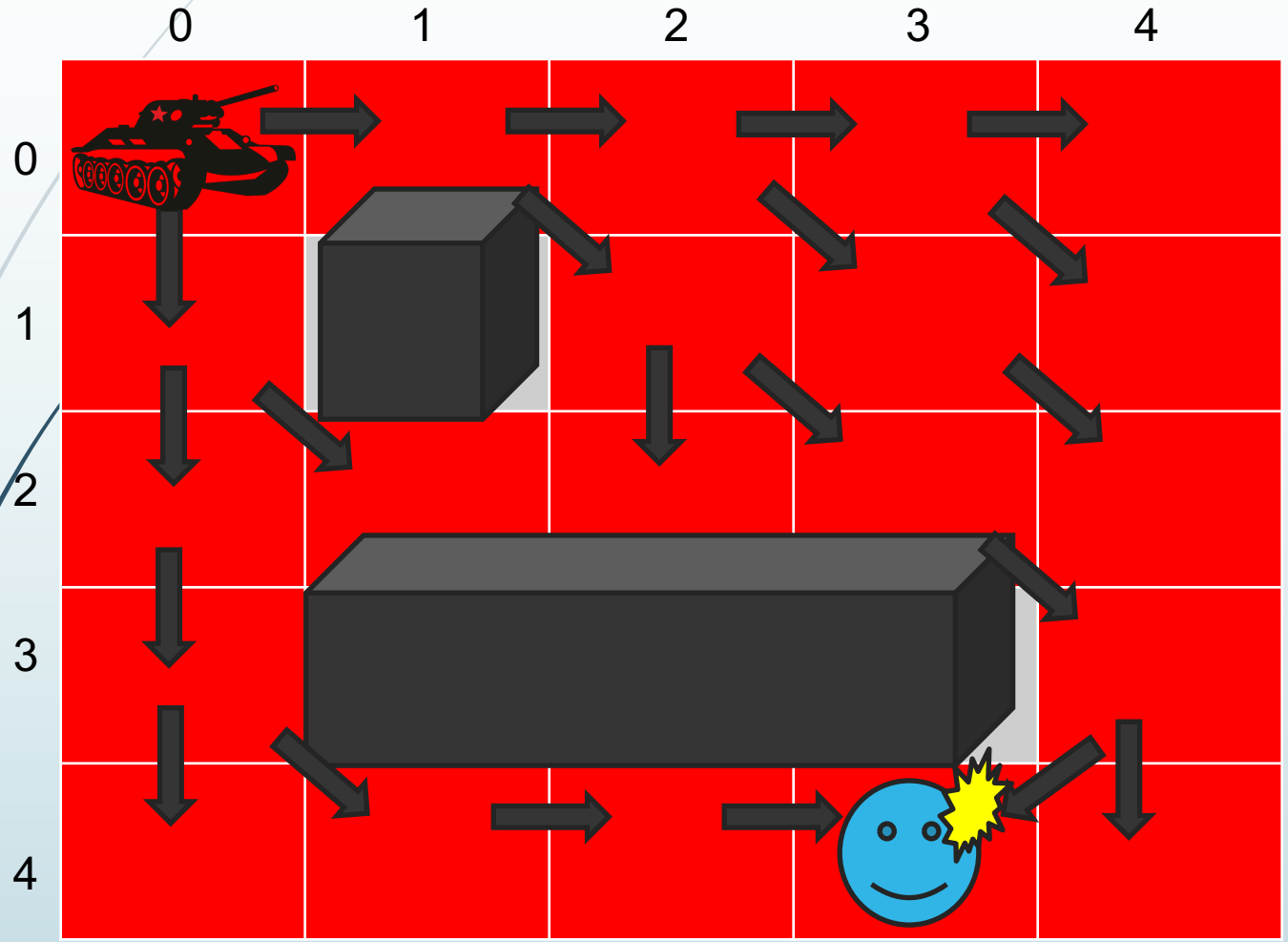
After All Distance 4 Locations Checked



CLOSED:
(0,0)-0, (1,0)-1, (0,1)-1, (2,0)-2, (2,1)-2, (0,2)-2, (1,2)-2, (3,0)-3, (3,1)-3, (3,2)-3, (2,2)-3, (0,3)-3, (4,0)-4, (4,1)-4, (4,2)-4, (0,4)-4, (1,4)-4, (4,3)-4

OPEN:
(2,4)-5, (3,4)-5, (4,4)-5

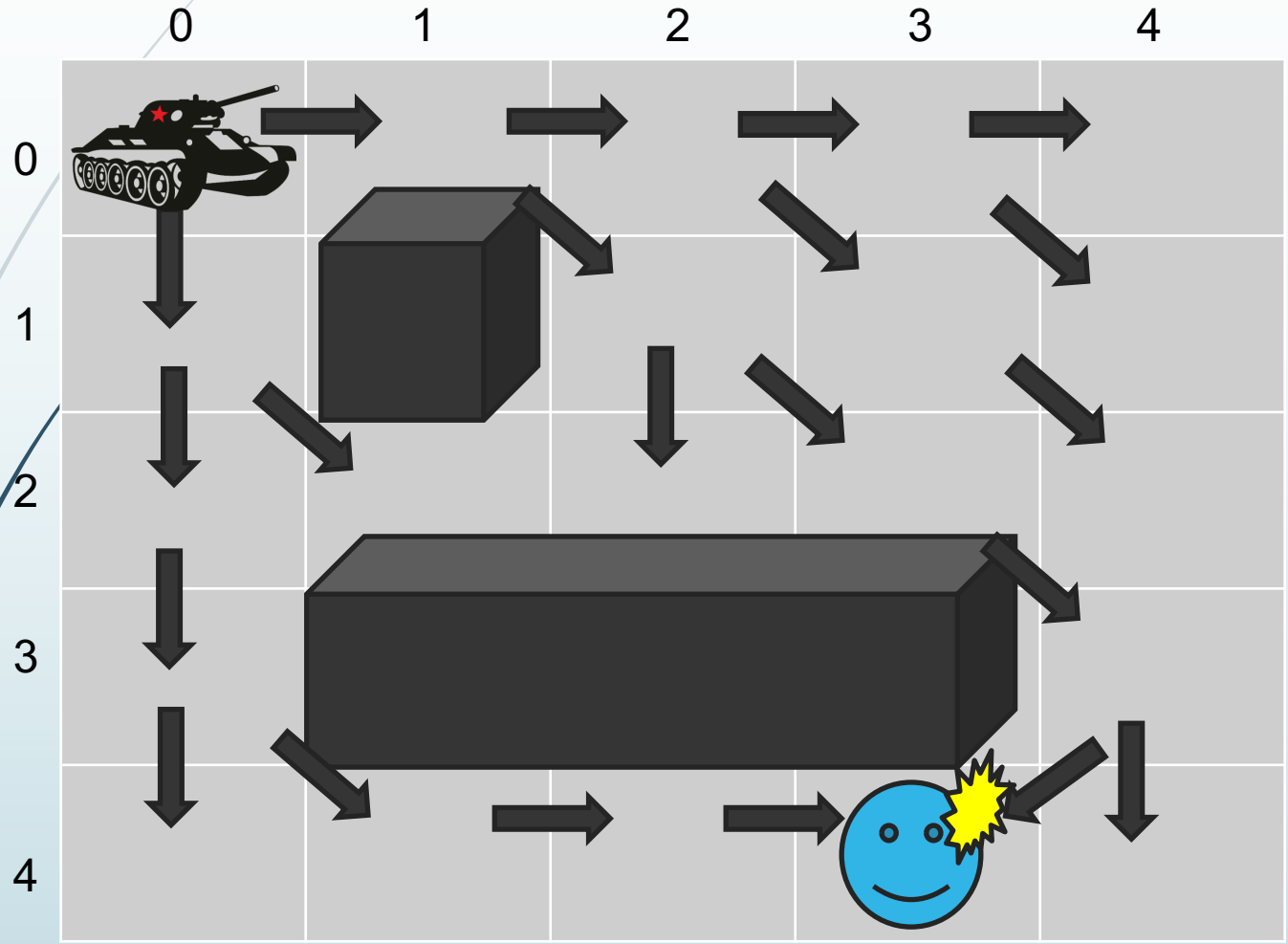
A* after All Distance 5 Locations Checked



CLOSED:
(0,0)-0, (1,0)-1, (0,1)-1, (2,0)-2, (2,1)-2, (0,2)-2, (1,2)-2, (3,0)-3, (3,1)-3, (3,2)-3, (2,2)-3, (0,3)-3, (4,0)-4, (4,1)-4, (4,2)-4, (0,4)-4, (1,4)-4, (4,3)-4 (2,4)-5, (4,4)-5, (3,4)-5

OPEN:
NULL

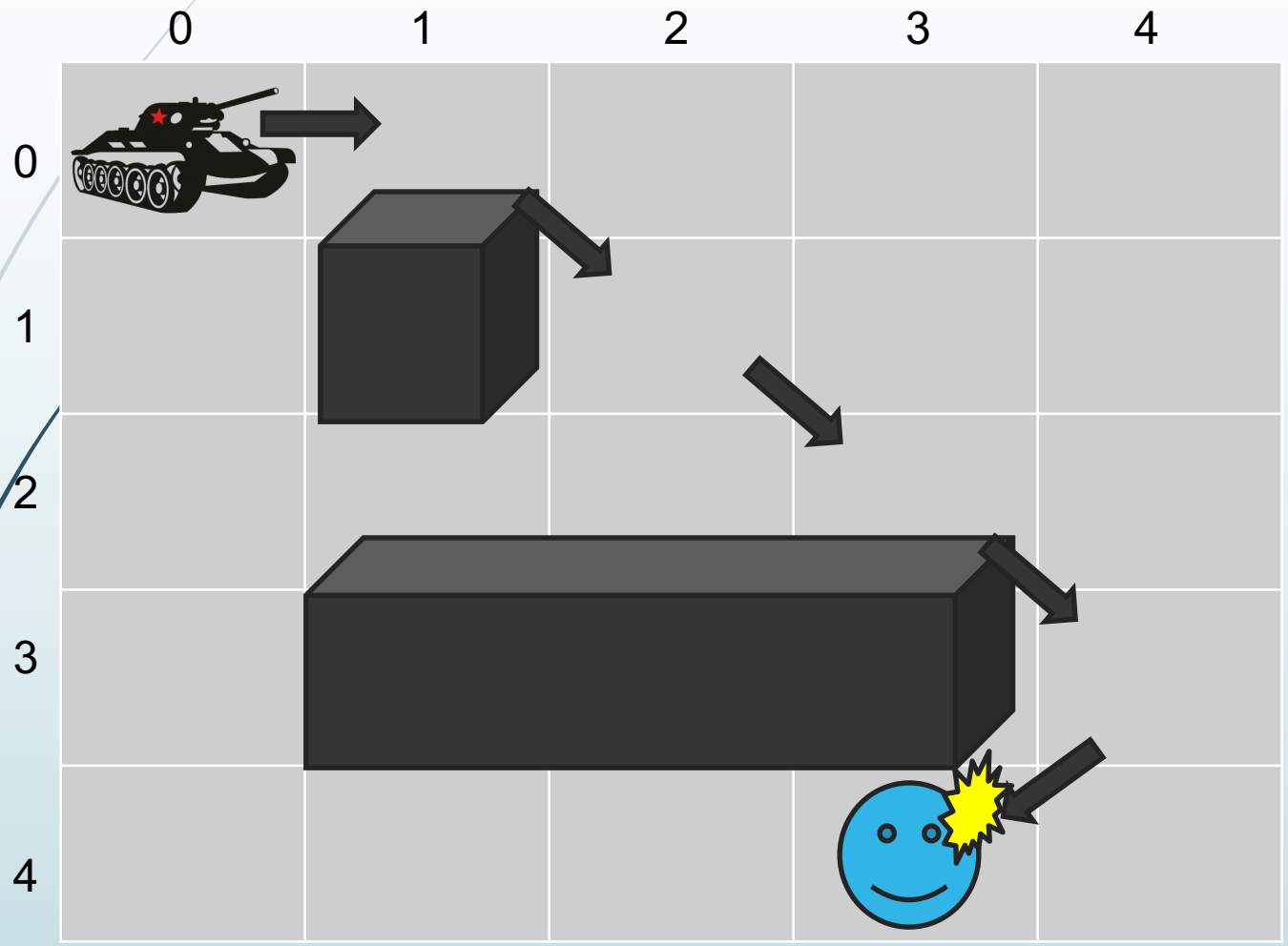
Finished



CLOSED:
(0,0)-0, (1,0)-1, (0,1)-1, (2,0)-2, (2,1)-2, (0,2)-2, (1,2)-2, (3,0)-3, (3,1)-3, (3,2)-3, (2,2)-3, (0,3)-3, (4,0)-4, (4,1)-4, (4,2)-4, (0,4)-4, (1,4)-4, (4,3)-4 (2,4)-5, (4,4)-5, (3,4)-5

OPEN:
EMPTY

A* Path



Improvements A*

- Dealing with other units
  - Treat stopped units as unpassable
  - Treat moving units nearby as unpassable
  - More complexity – prediction of their movement for avoidance – query their path
- Cost of movement
  - Taken into account in the cost function
- Non-grid setting
- Open World
  - Enforce a grid or graph (Valve Developer Commentary HL2:EP1)
  - http://youtu.be/op6aMDNOyQ8?t=24m33s
  - Develop pathways explicitly for longer movements as per before

Exercise 2

- Implement A* algorithm for considered world with obstacles using C/C++/C#/Java/Python.

References

- A* video link