Introduction to Artificial Intelligence

Week 5

Learning by Searching

Types of Searches

Uninformed Search

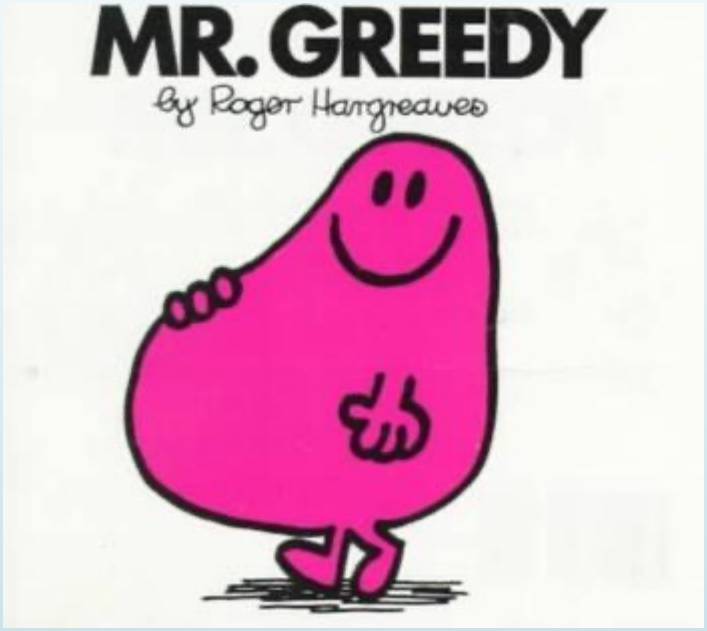
- Have no sense of the problem domain
- Generally applicable in all cases

Informed Search

- Use a heuristic function developed for the domain
- Applicable in their own domain

Greedy Approaches

- Generate and Test Methods primarily
- Current best solutions are held
- When I find something better – I get rid of what I had
- Always wanting more
- Always takes the
  Locally Optimal choice

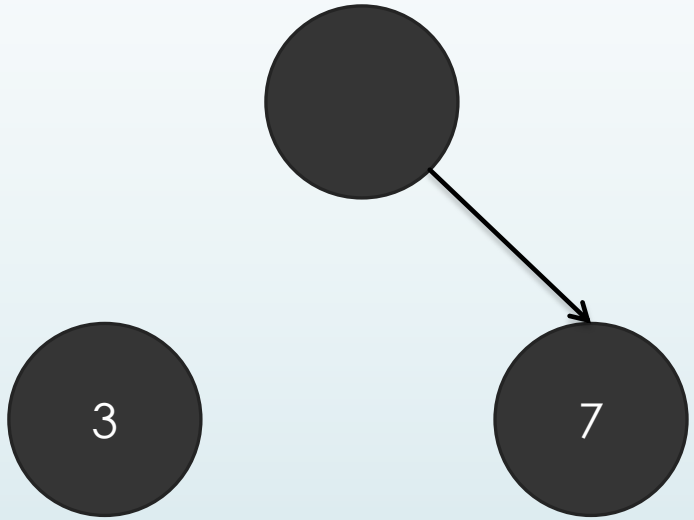


Greedy Algorithm Basics

- 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

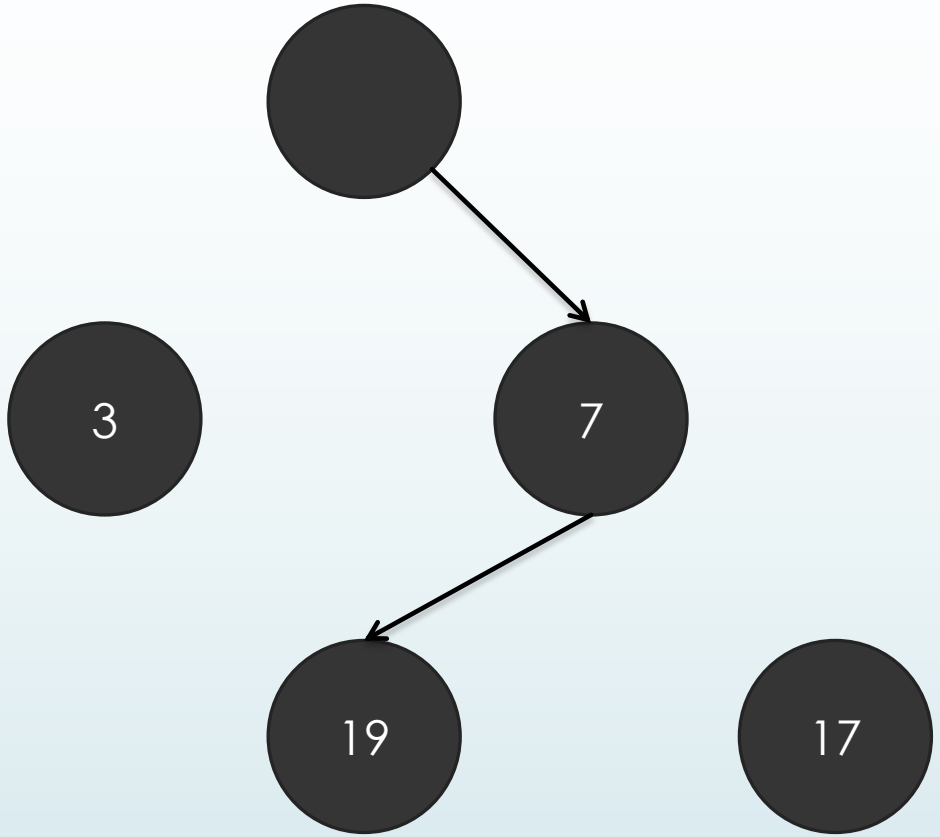
Example – Game Tree

- Two players are making moves in a game, each move gives a certain number of points to the player – goal is to make the most points



- In the first move our greedy player will take 7 points

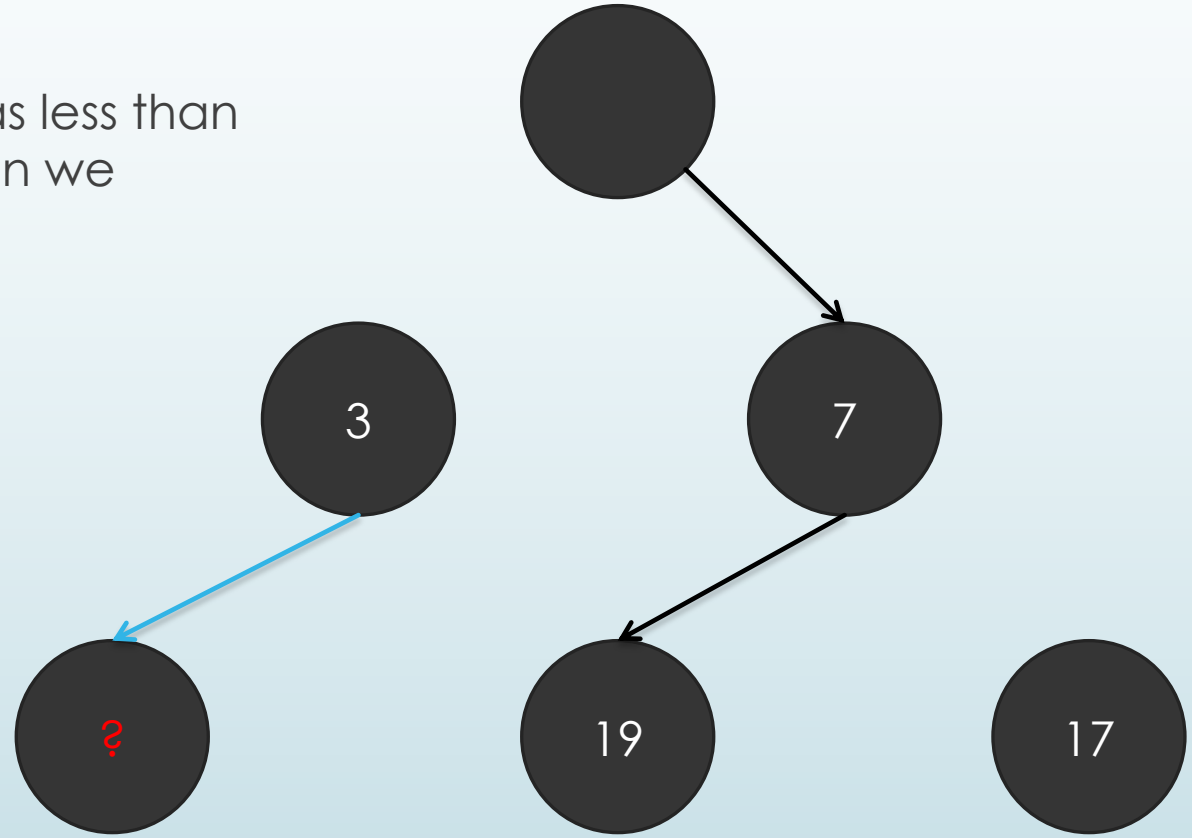
Example – Game Tree



- In the second move move our greedy player will take 19 points, for a total of 26 points

Was Greedy Good

- Maybe – Depends on the Game being played
- If the ? Position was less than or equal to 23, then we have maximized the score
- If ? was greater, then we have failed



What Ensures That Greed Will Be Rewarded

- Sub-problem optimization
  - Also called optimal substructures
  - The optimal solution to the problem contains optimal sub-problems
- Greedy Choice Property
  - We never need to consider the play until now and can move from this point only by making the current best decision. We never need to review the past moves in order to make a better decision
  - In our example, if we have played the optimal game until now, then the current point is also part of the optimal value on this move

Activities Not Clashing Problem

- We have a set of activities each with a start and end time
- We want to find the maximum number of activities that a single person can complete in the time allotted

Example

- One Operator on one machine
- Assume all parts give the same profit margin, but we have a choice of which car to build now
- Times are dependent on assembly

Greedy Algorithm

- Sort the activities by their end time from smallest to largest
- Start with Set of Activities *S = {1}*, and finish time *f = f[1]*
- For all the other activities *X* in order *I*
  - If s[x] >= f
    - S = S U X
    - f = f[x]

Why it Works?

- We order *S* from smallest to largest finish time, so *s[1]* finishes first
- Let *A* be an optimal subset of *S* and ordered by increasing finish time which does not contain *S[1]*, let *S[k]* be its first member

Why it Works? Cont…

- Then we can construct a set $B = (\{A \setminus S[k]\} \cup S[1])$, as $f[1] < f[k]$, then the activities are disjoint, note that $|B| = |A|$, therefore both are optimal
- Once this choice is made of $s[1]$, then the remaining disjoint parts of the set reduce into a smaller instance of the same problem
- We always make the right greedy choice by picking the first finishing task – note their might be other optimal solutions – we have at least shown one construction

Expand on the Problem

- However, not every part costs the same profit margin
- We are interested in profitability – not in number produced
- How do we maximize our profits?

Greed is No Longer Good

- The greedy solution is based on the number of completed parts solely
- If producing a smaller number of parts has a higher profit, or if there are multiple maximum part solutions with different profits, we no longer have the optimal solution
- We no longer have the greedy choice property!

Generate and Test

- Greedy is a constructive solution method – we construct the solution one part at a time
- Method to produce a solution may also generate the entire solution in one step; non-constructive
- Generate the problem solution
- Test the solution against the problem, or some representative amount of the problem

Hill Climbers

- Single Searching point
- Search locally for changes to the actions
- Have a representation of a solution string
- Have a test of the representation about the space
- Each solution string maps to an outcome
  - An objective function
  - A fitness on the problem

Climbing Hill in Our Example

- Representation
  - String of actions which we can take
- Objective
  - Maximization of profit value based on the actions
  - No string of actions is allowed to have an overlap
- Local Change
  - Add and/or remove some number of items from the string
- Climbing the Hill
  - Replace my position if the change is better

Hill Climbing Algorithm

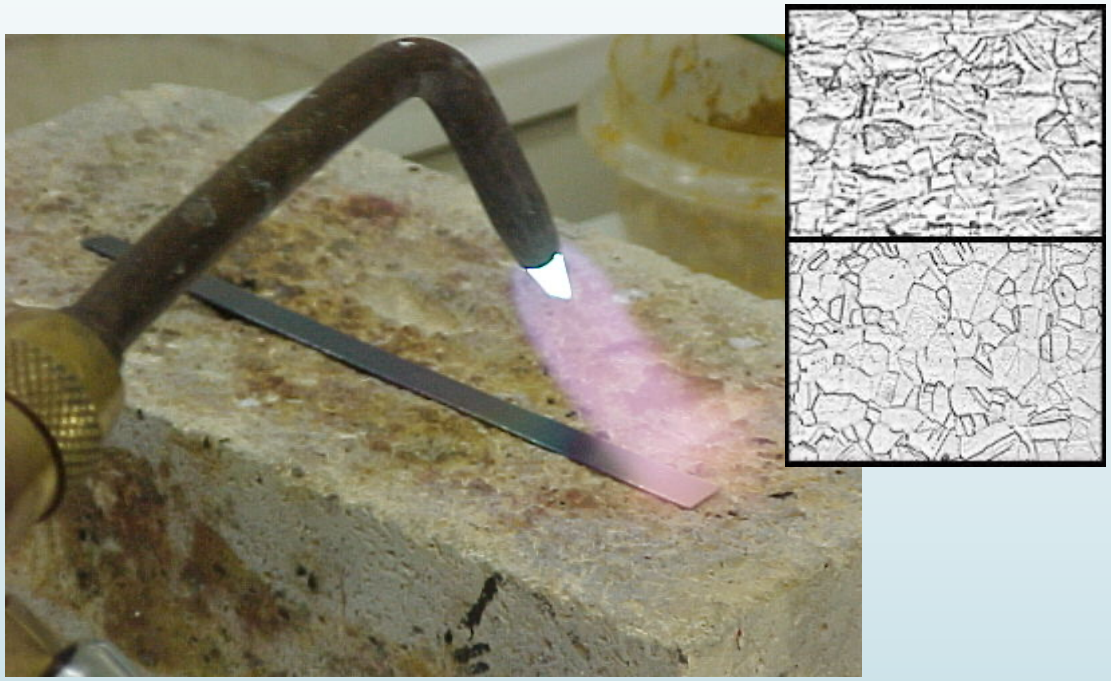
```
function HILL-CLIMBING(problem) returns a solution state
   inputs: problem, a problem
   static: current, a node
           next, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      next ← a highest-valued successor of current
      if VALUE[next] < VALUE[current] then return current
      current ← next
   end
```

Issues and Work About

- Local Maximums
  - Stuck in suboptimal places if our value is on a localized hill
    - Change the mutation
    - Change the representation
    - Random restart
- Multiple Optimums
  - Multiple Good Solutions
    - Random Restart
  - Large areas of equal solutions
    - Change the mutation
    - Random Restart
- Jumping over areas of the space
  - Selected Mutation hops over the space

Simulated Annealing

- Based on the properties of Metal working
- Metal is heated and cooled in order for it to be made workable



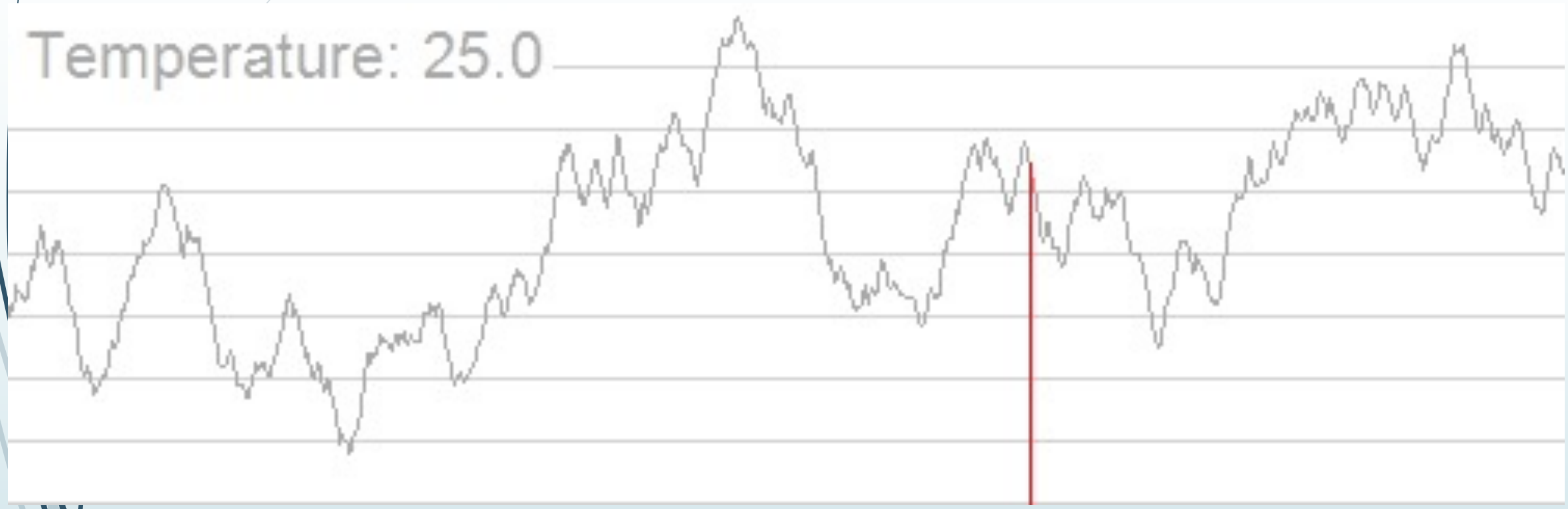
Simulation of This Process

- Set down a number of search points
- Points move dependent upon the temperature applied
- They move about the space based on a temperature which is slowly reducing over time
  - This allows for large movement at the beginning of the algorithm and reduces into smaller and smaller movements as time goes on
- Two Types of controls
  - Temperature controls size of step
  - Temperature controls the acceptance of a step

Simulated Annealing – Step Size

- Start at an initial temperature value (usually something hot) and create a random initial solution
- Until temperature is cool
  - Make a change to the current solution (neighbour) based on the temperature
    - Current location + Rand_Distribution(temperature)
  - Decrease the temperature

Temperature Controls the Step Size



Simulated Annealing

- Start at an initial temperature value (usually something hot) and create a random initial solution
- Until temperature is cool
  - Make a small change to the current solution (neighbour)
  - Decide if we move to this neighbour solution
    - Neighbour solution is better or
    - Rand_Distribution() < temp
  - Decrease the temperature

Exploring and Exploiting

- Exploration – the want for a search method to find new areas of the search space which have not been visited yet
- Exploitation – the want for a search method to locally search about the best known areas in the space in order to refine these current solutions
- All search algorithms will have some mix of both properties
- Note that if we only have a limited number of objective tests available, then we have a zero-sum game – we can explore, or we can search with a test

Exploration or Exploitation?

- We create a simulated annealing which has a reheat if we are not seeing a 10% improvement in the solution

Exploration

- The reheat increases the step size
- The reheat increases the probability we accept worst solutions

Exploration or Exploitation?

- After the application of a search we look at the least significant digit and test from 0-9 taking the best final solution

Exploitation

- Looking at the least significant digit we are looking in a close locality to the presvious, making the assumption that close parameter settings make for close outputs

Parallel

- All of these methods are trivially parallel
- We can place one of these searching units onto a different ‘core’ processing unit without issue
- Hill Climbers and Simulated Annealing use the idea of a solution neighbourhood, so we can divide the space

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
- $f(n) = g(n) + h(n)$
  - $g(n)$ is the path from the initial position to the current node
  - $h(n)$ is the path from the current node to the goal position

A* Algorithm

- A node contains a location, a cost, and a parent
- Initialize a set CLOSED to be null
- Initialize a priority queue based on cost of a node called OPEN to be the starting location, set cost = 0
- While OPEN is non-empty do
  - Current <- Pop OPEN.top
  - For all neighbouring locations X
  - If not (Blocked or Closed)
    - If not in OPEN
      - X.cost = current.cost+1
        X.parent = current
      - Push X to OPEN

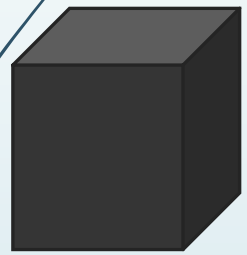
A* Algorithm

- Else if X is in OPEN
  - If current.cost+1< Open.x.cost // we found a better path
  - Open.x.cost = current.cost+1
  - Open.x.parent = current
- END FOR ALL X
- Push Current to CLOSED

A* for PATH FINDING

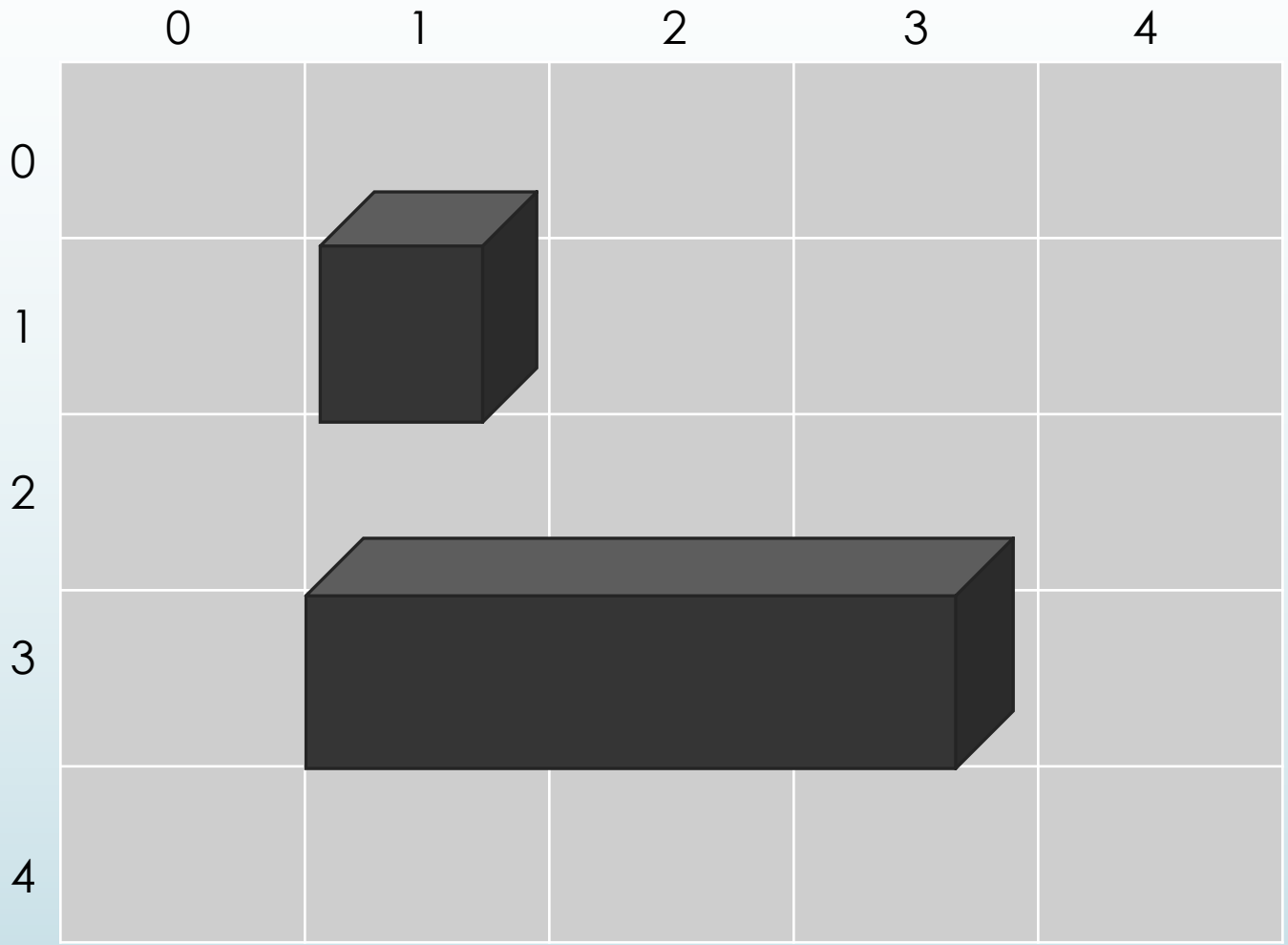
- REPORT_PATH (START, FINISH)
  - STACK PATH initialized to NULL
  - RUN A* (START)
  - Find FINISH in CLOSED
  - Current <- FINISH
  - While Current not START
    - PUSH Current to PATH
    - Current <- Current.parent
  - PRINT START
  - While STACK not NULL
    - PRINT POP PATH

World

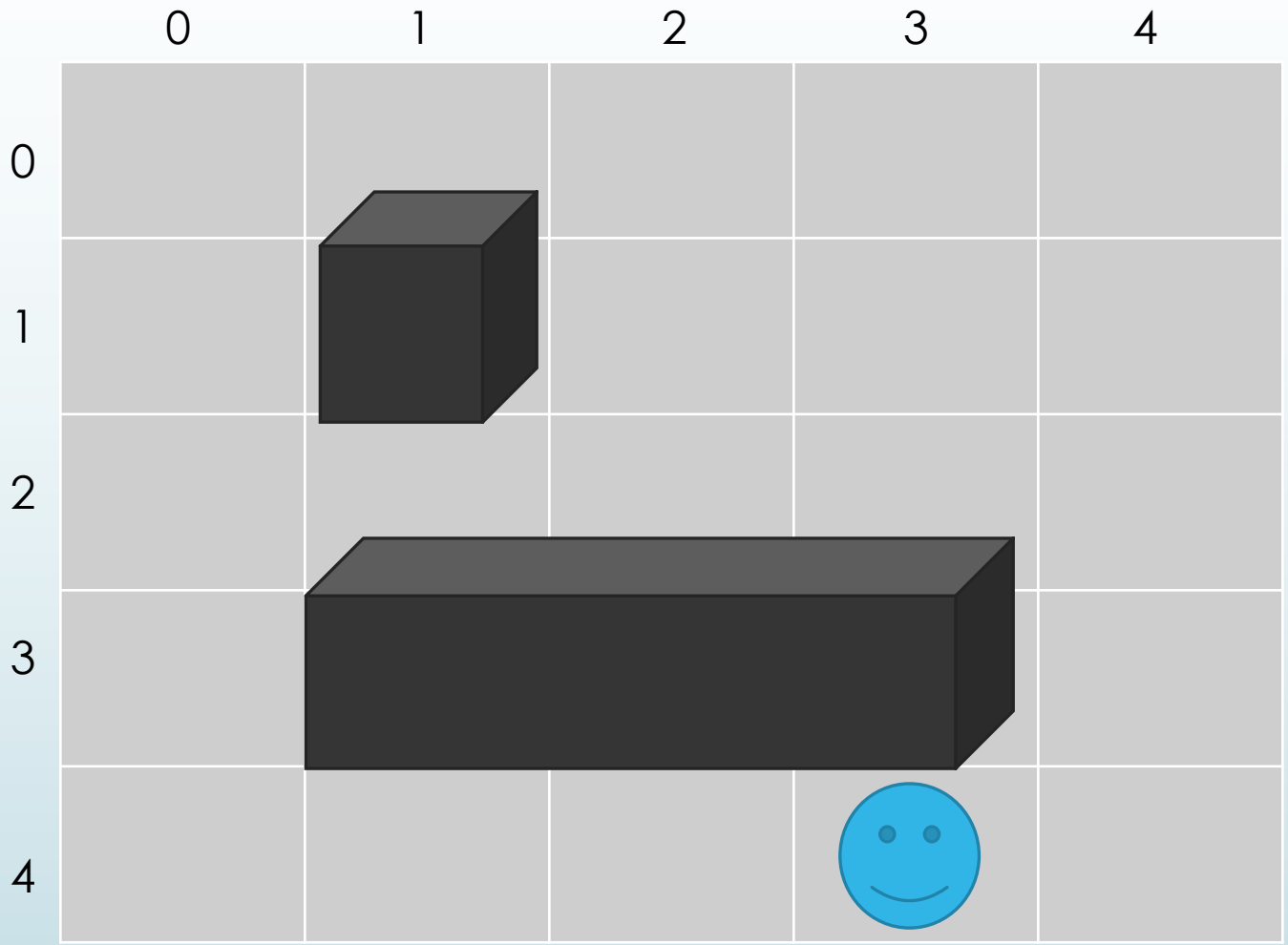


	0	1	2	3	4
0					
1					
2					
3					
4					

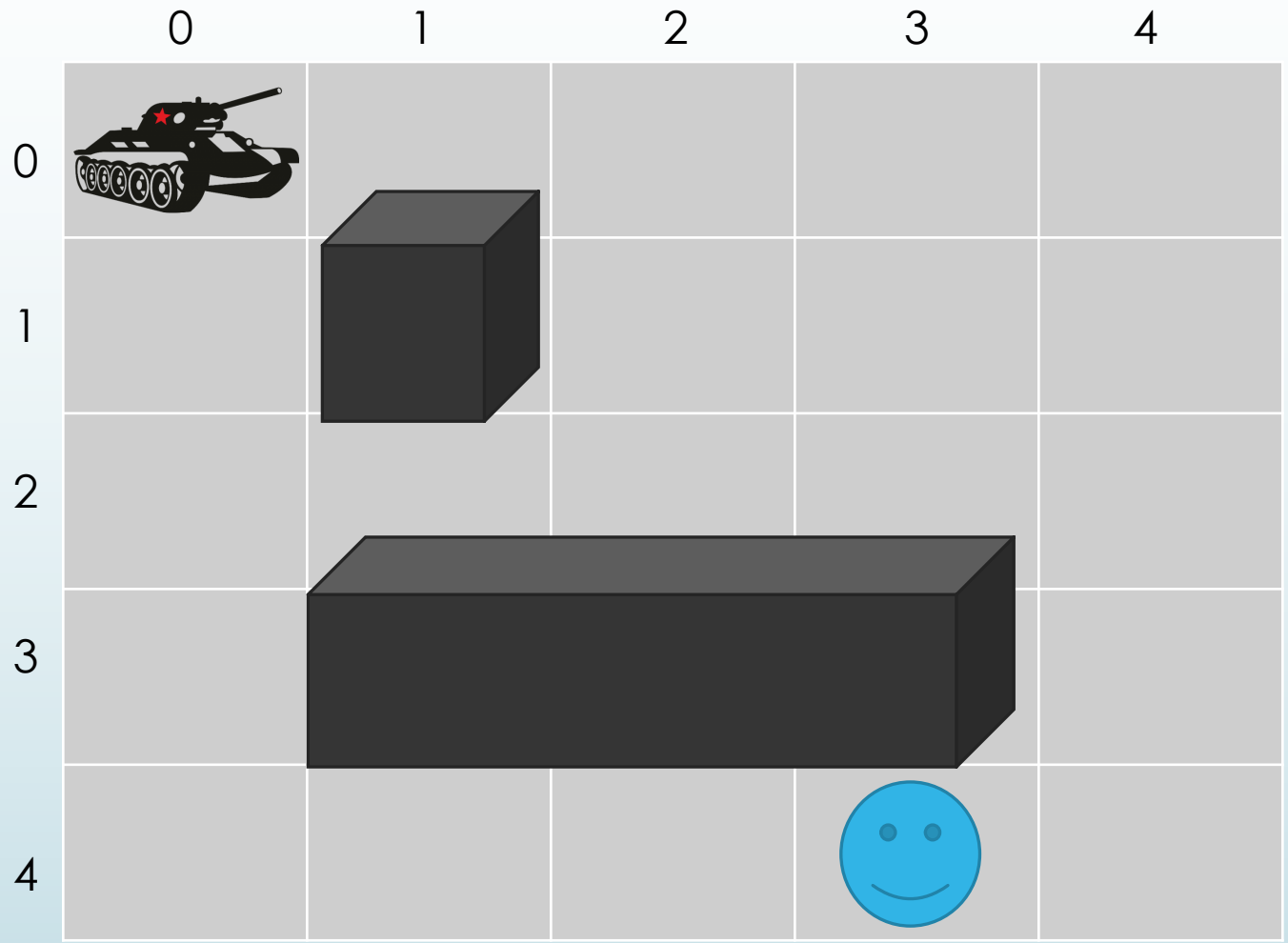
World With Obstacles



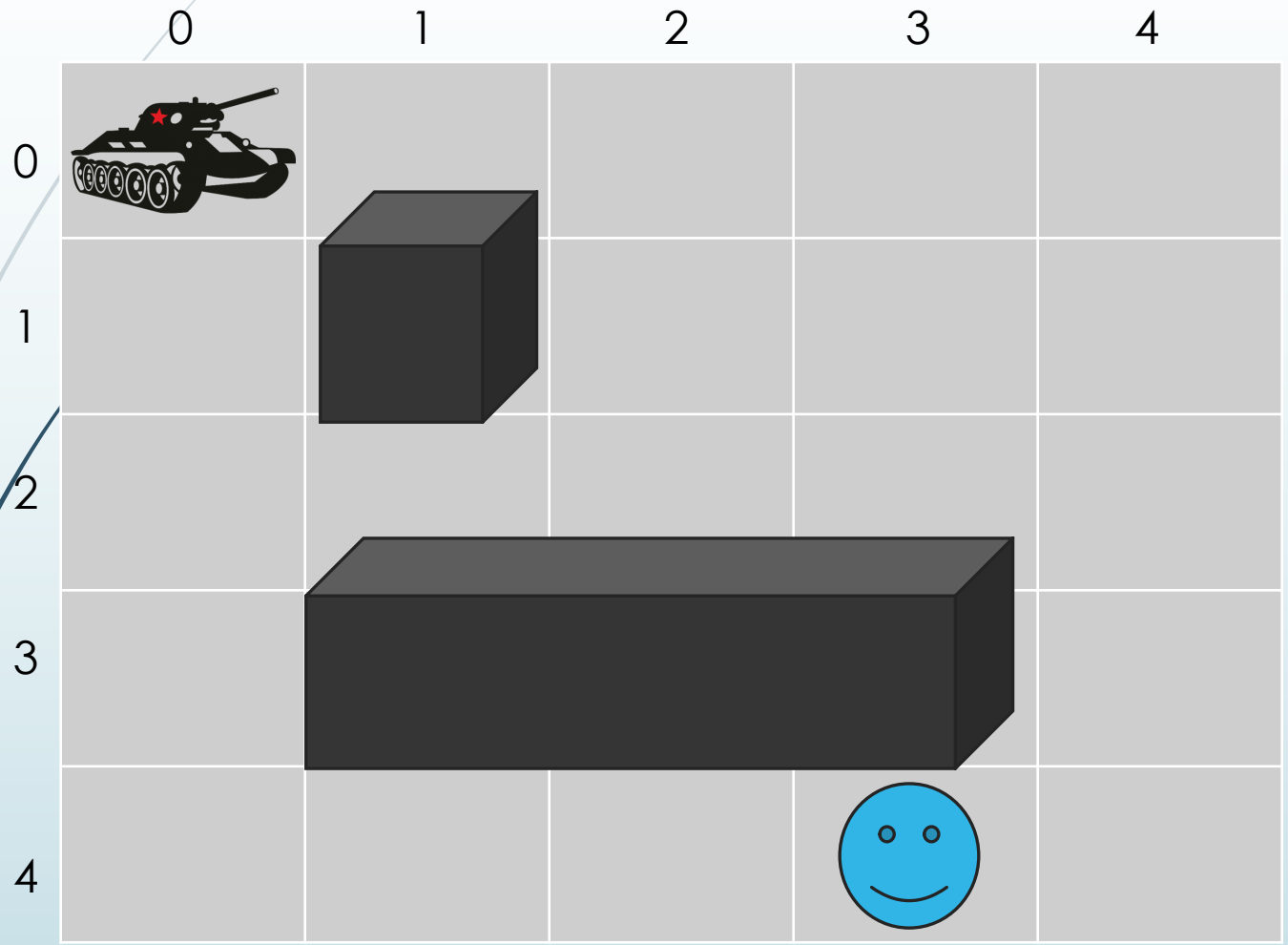
World With Obstacles and Goal



World With Obstacles, Goal and Start Position



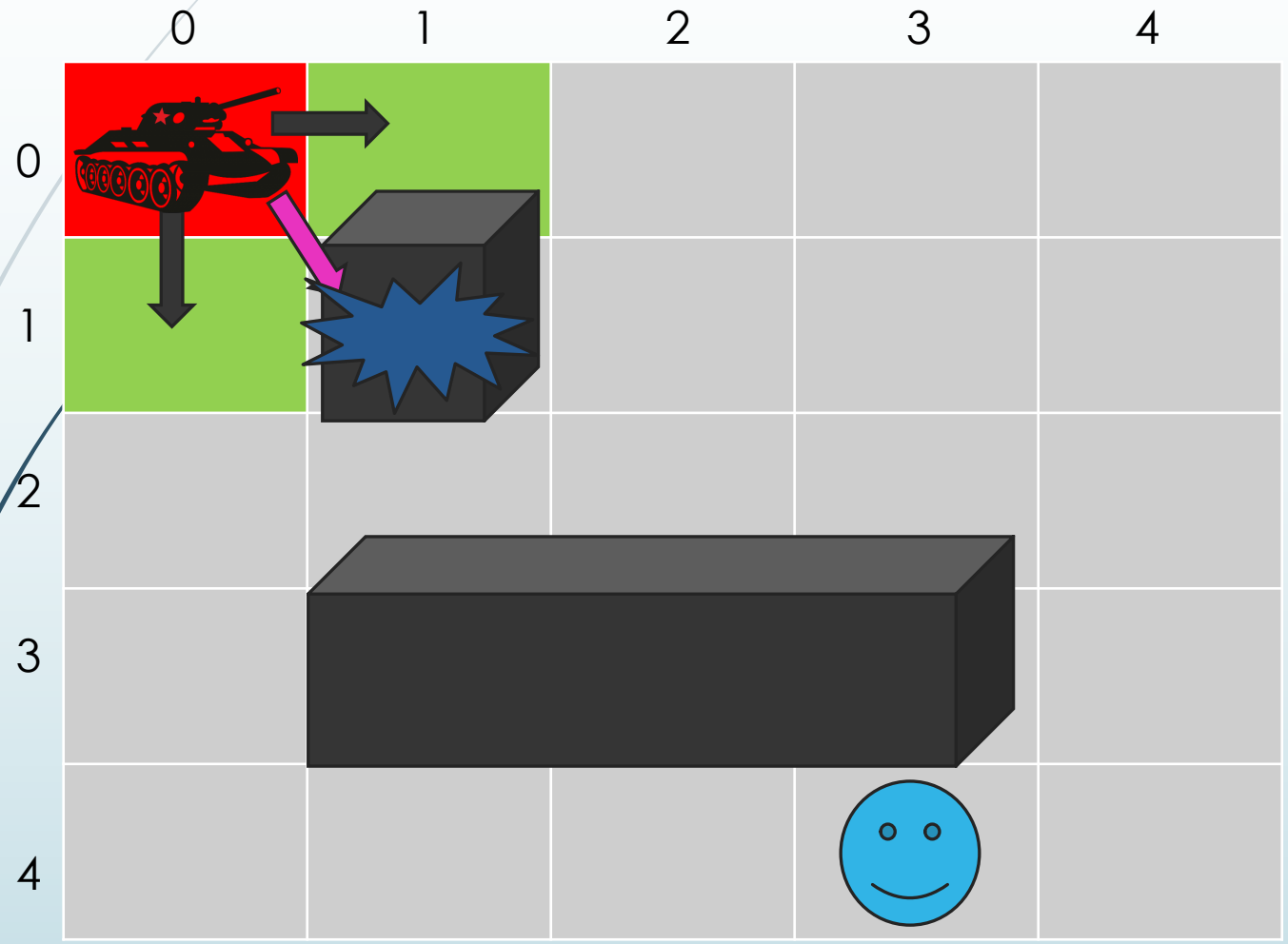
Add Initial Point to Open List and Assign Cost



CLOSED:
NULL

OPEN:
(0,0)-0

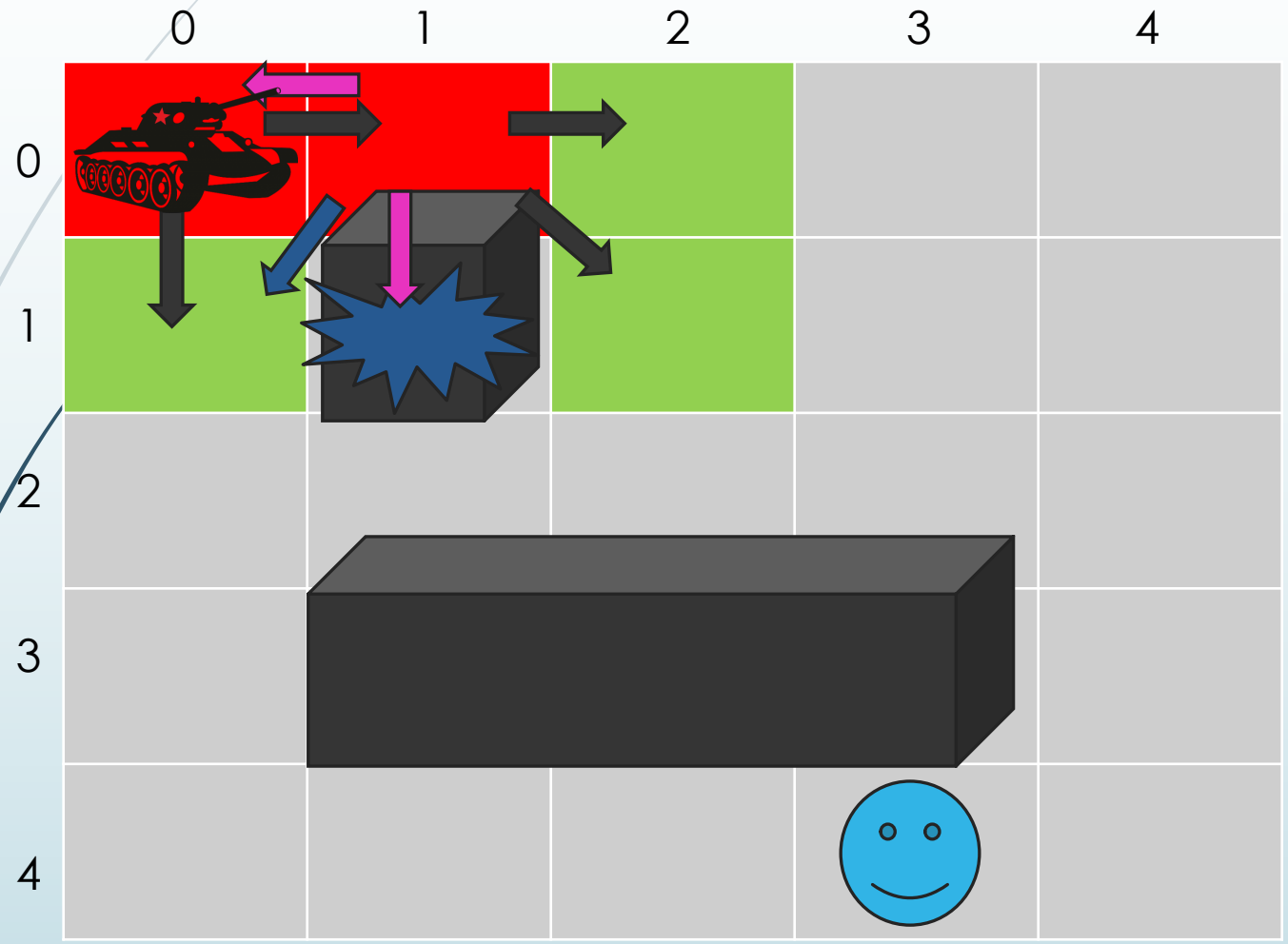
Move Lowest Cost to Closed and Search Neighbours



CLOSED:
(0,0)-0-NEW

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

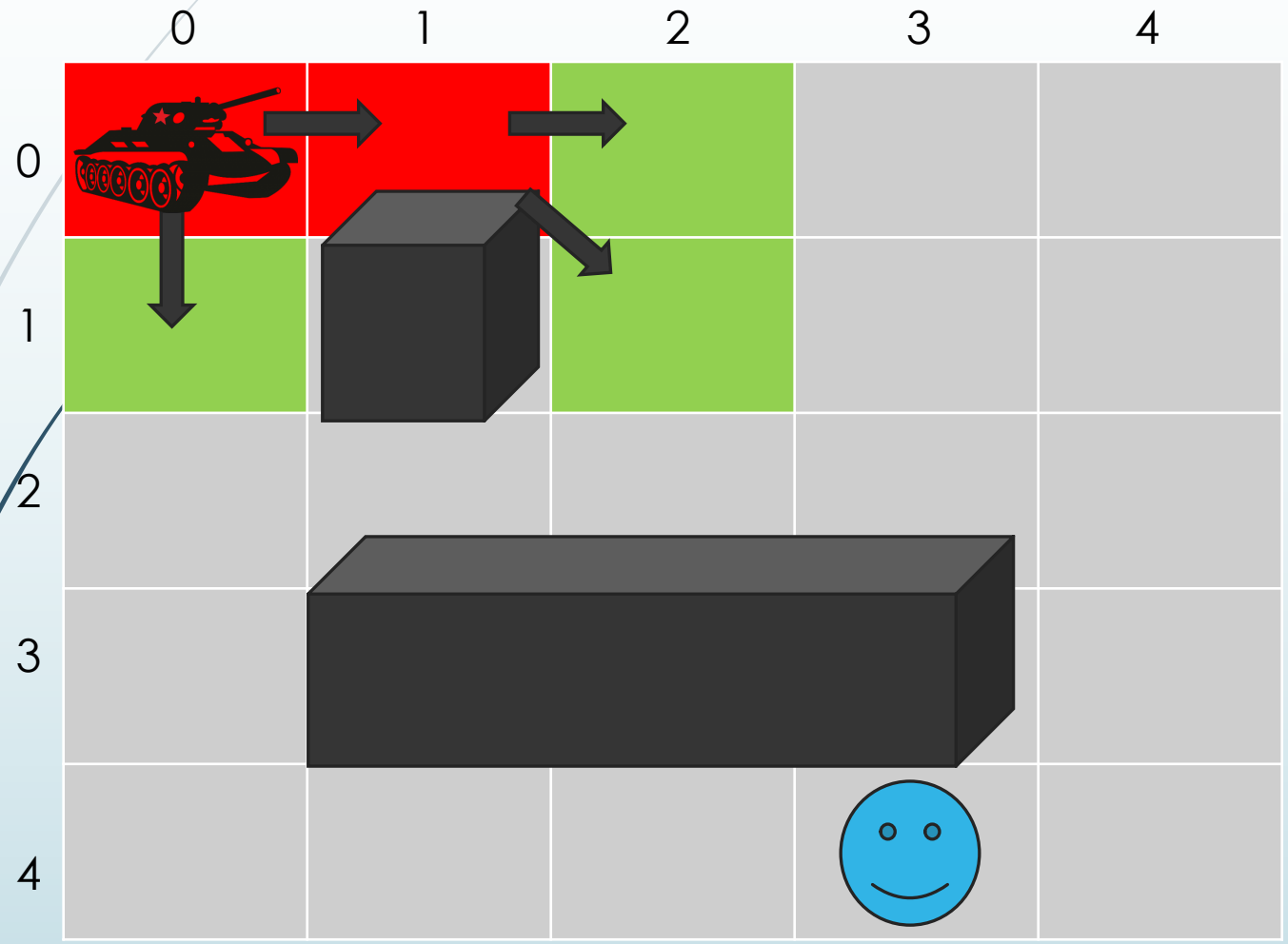
Move Lowest Cost to Closed and Search Neighbours



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!

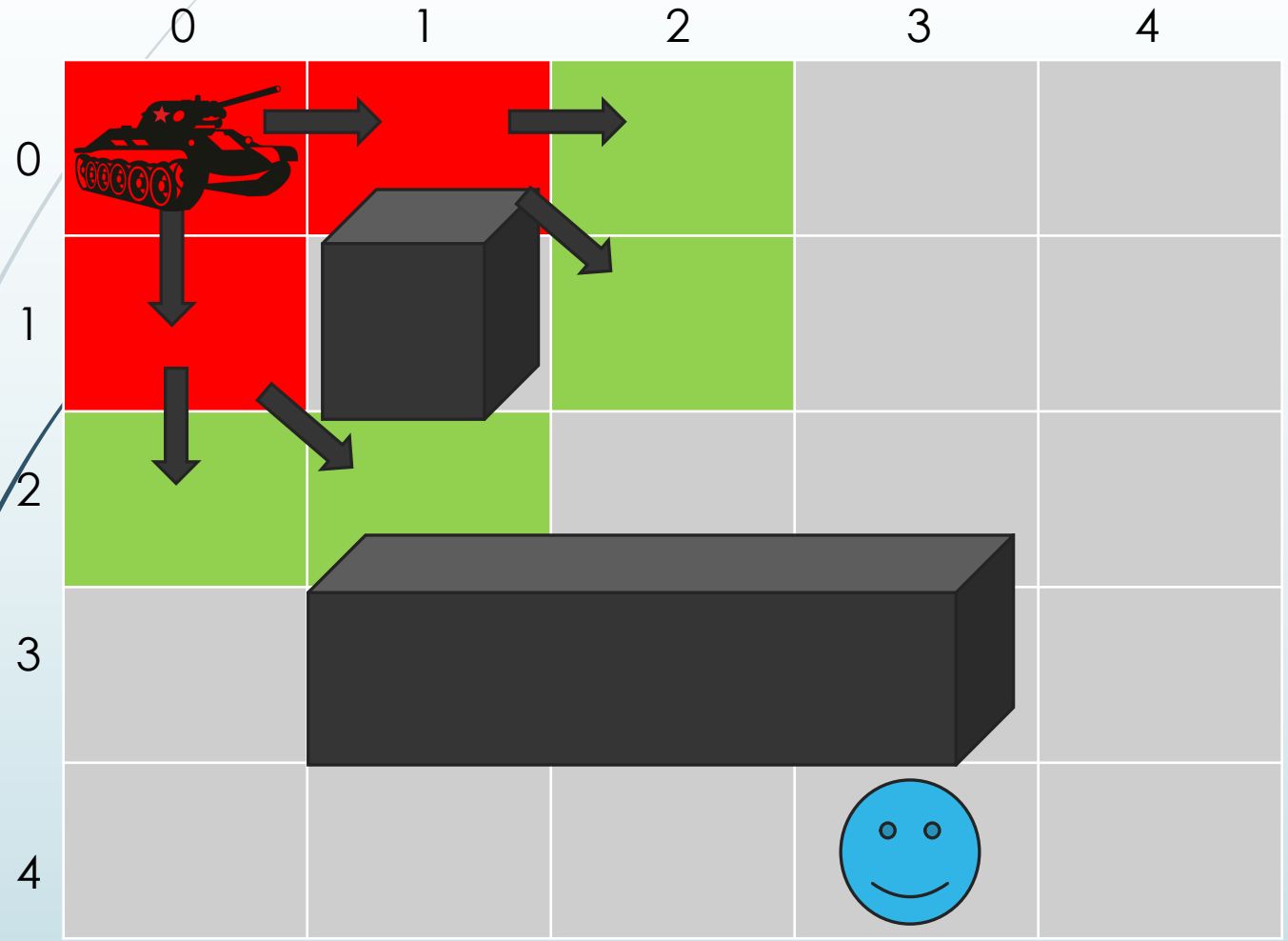
Open and Closed after Two Steps



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

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

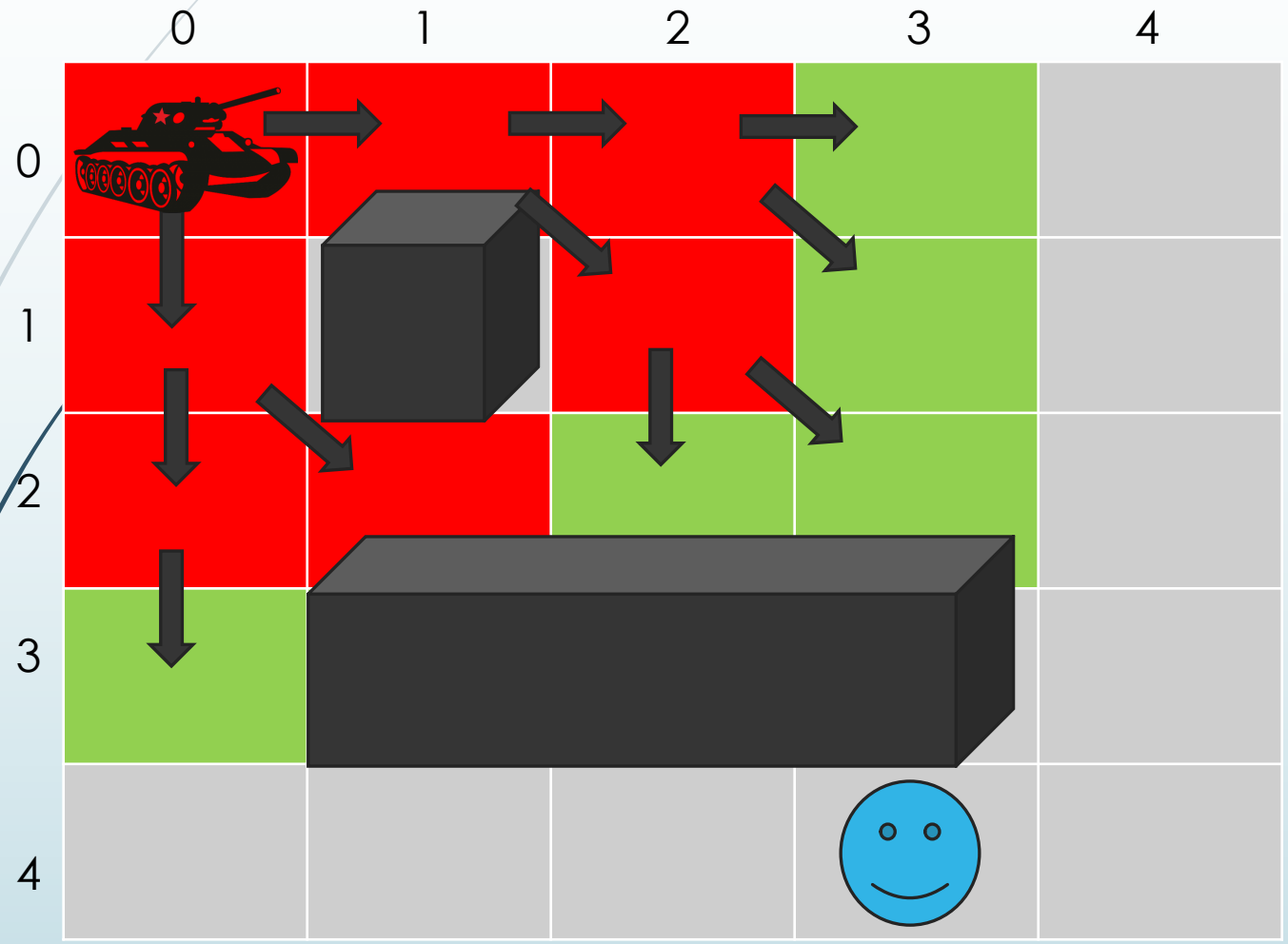
Move Lowest Cost to Closed and Search Neighbours (Only Valid)



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

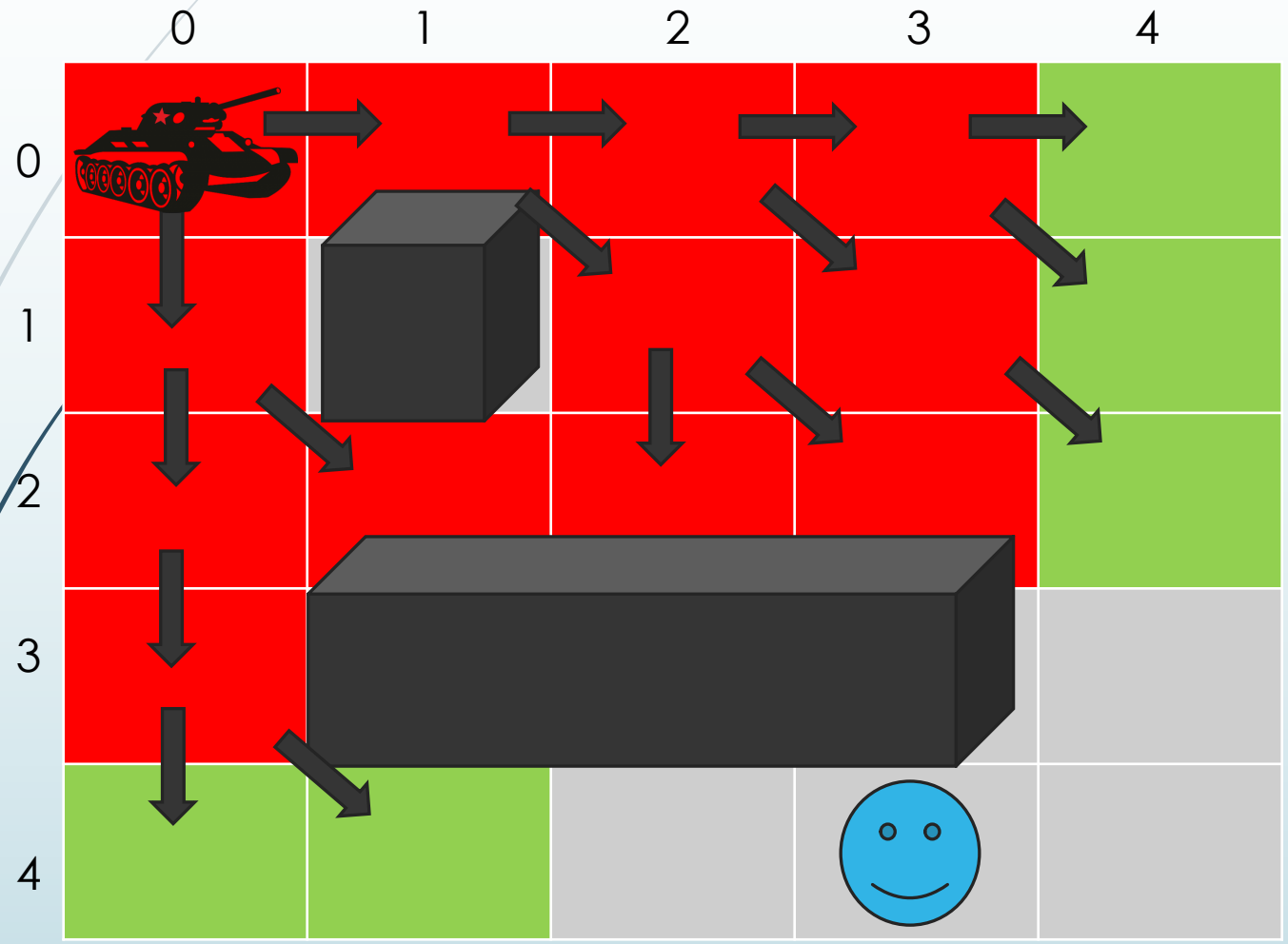
After All Distance 2 Locations Checked



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

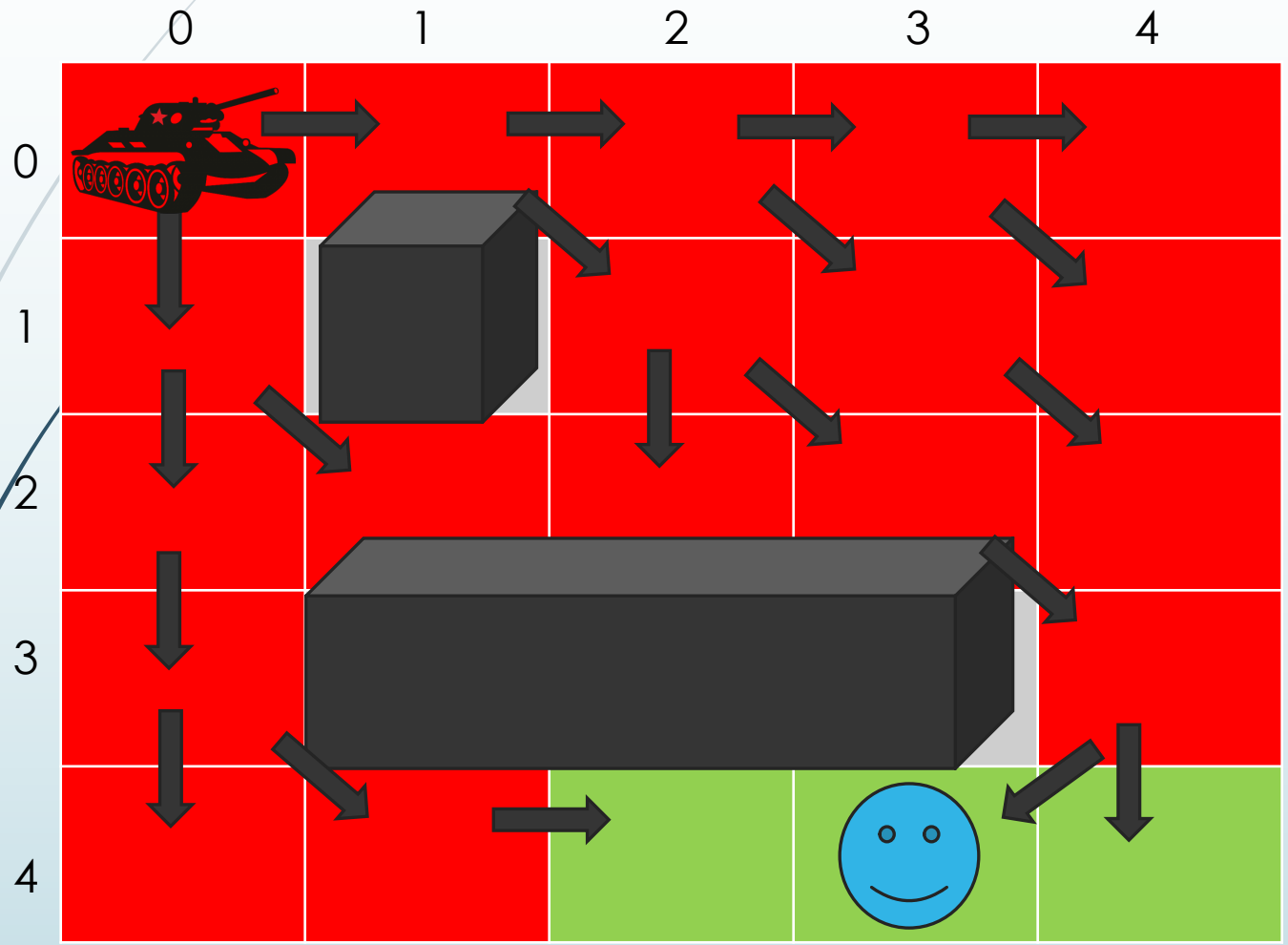
After All Distance 3 Locations Checked



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

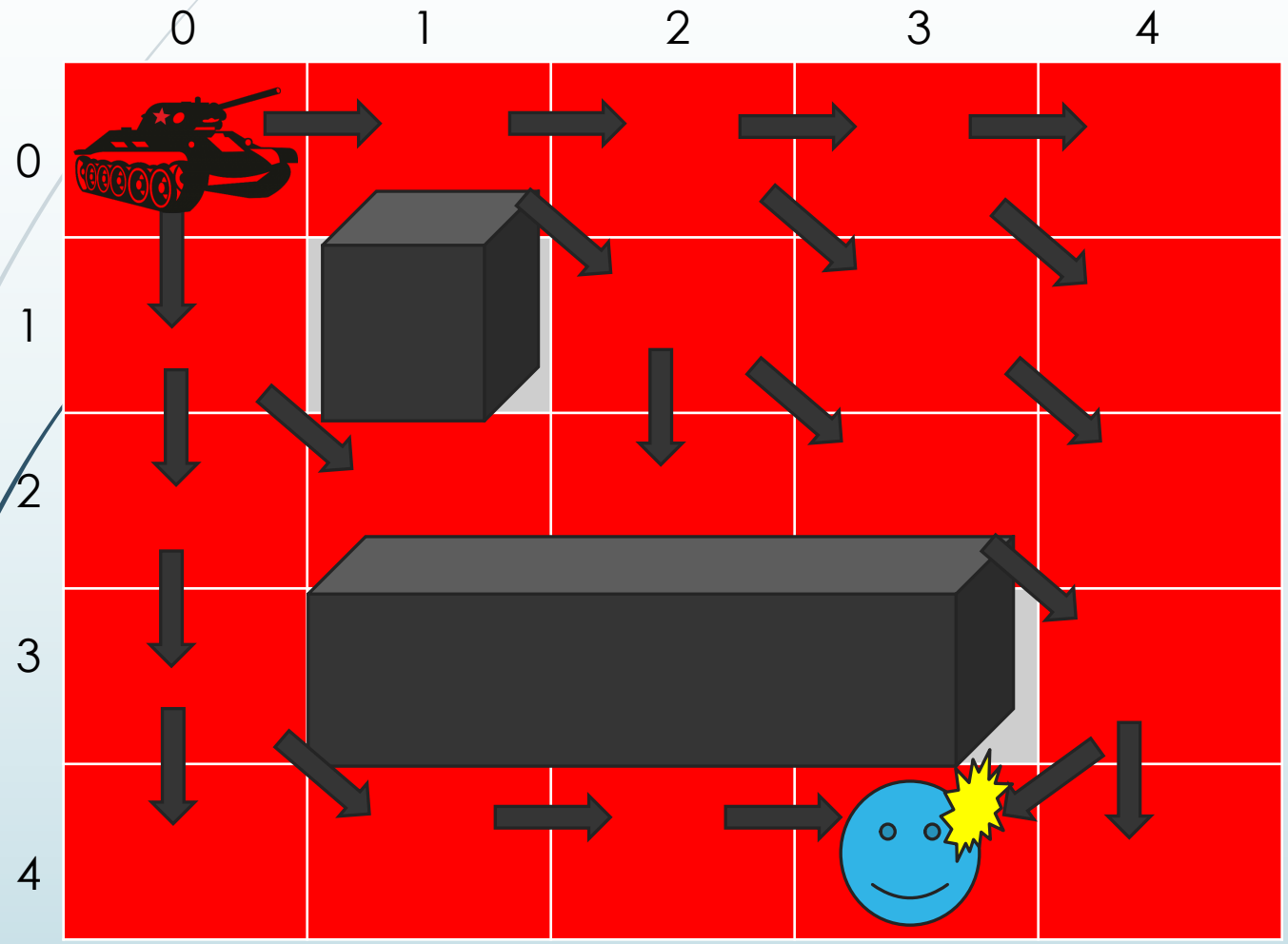
After All Distance 4 Locations Checked



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

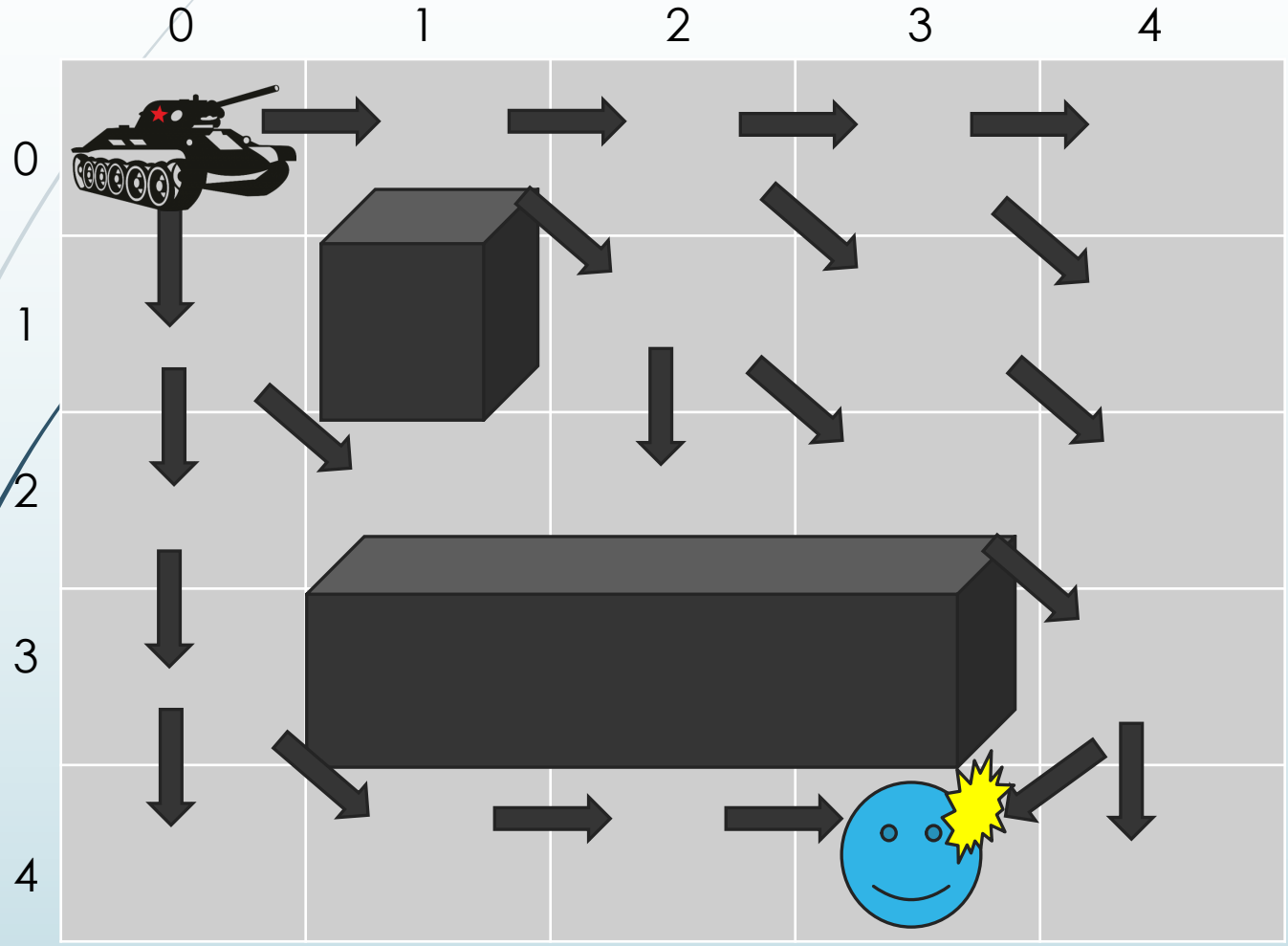
After All Distance 5 Locations Checked



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

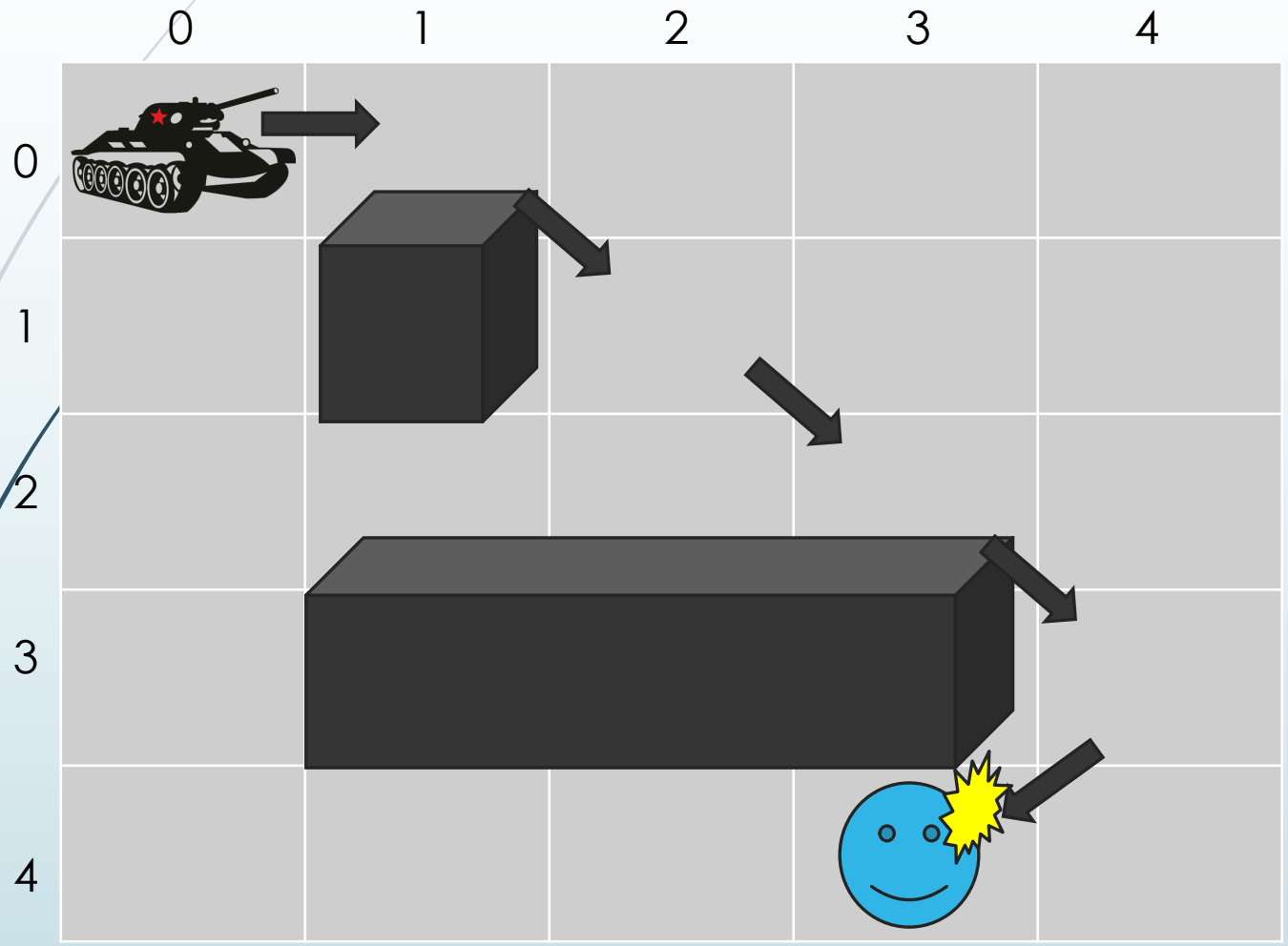
Finished



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

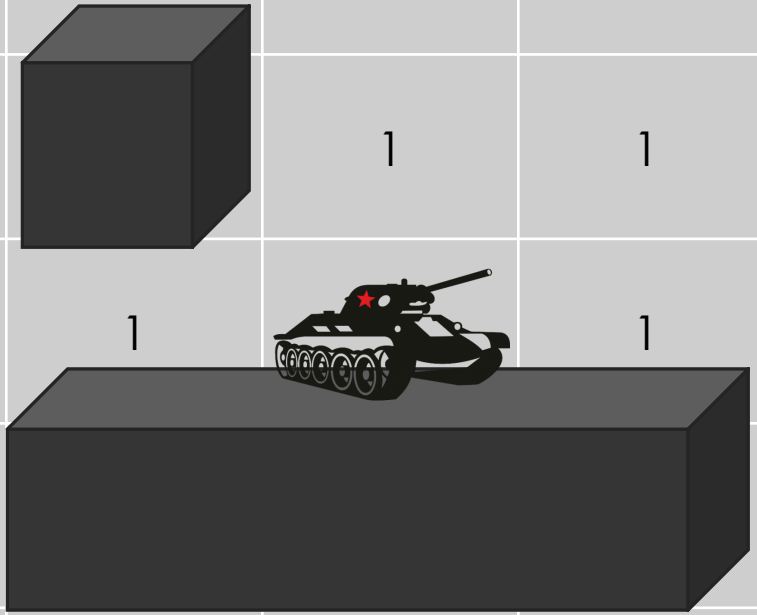
OPEN:
EMPTY

A* Path



World With Movement Values

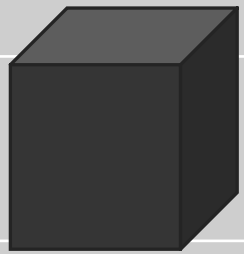
	0	1	2	3	4
0	3	2	2	2	2
1	2		1	1	2
2	2	1		1	2
3	2				2
4	3	4	5		3

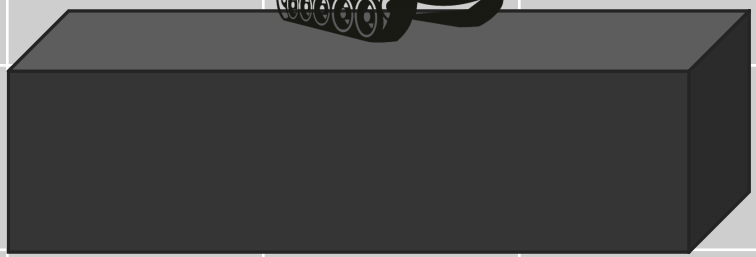


These are g(n) values

World With Movement Values with Manhattan Heuristic

	0	1	2	3	4
0	3+7	2+6	2+5	2+4	2+5
1	2+6		1+4	1+3	2+4
2	2+5	1+4		1+2	2+3
3	2+4				2+2
4	3+3	4+2	5+1		3+1

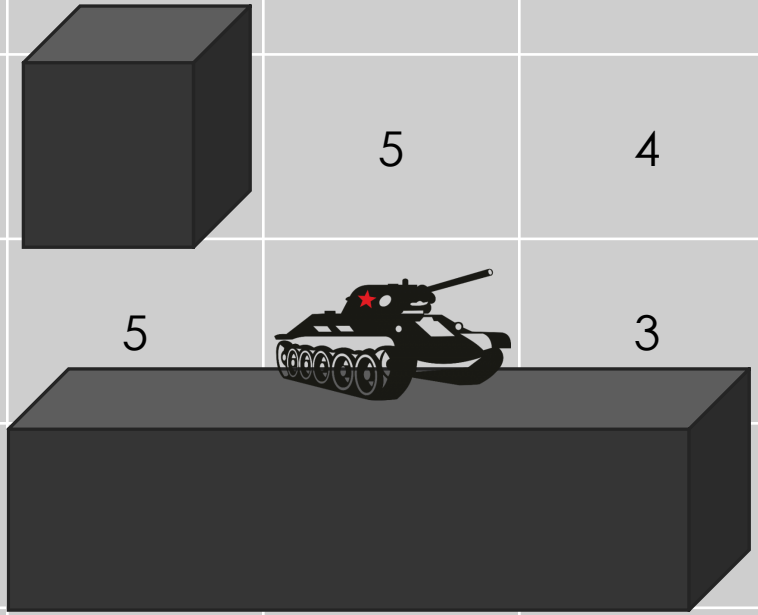




These are g(n) + h(n) values

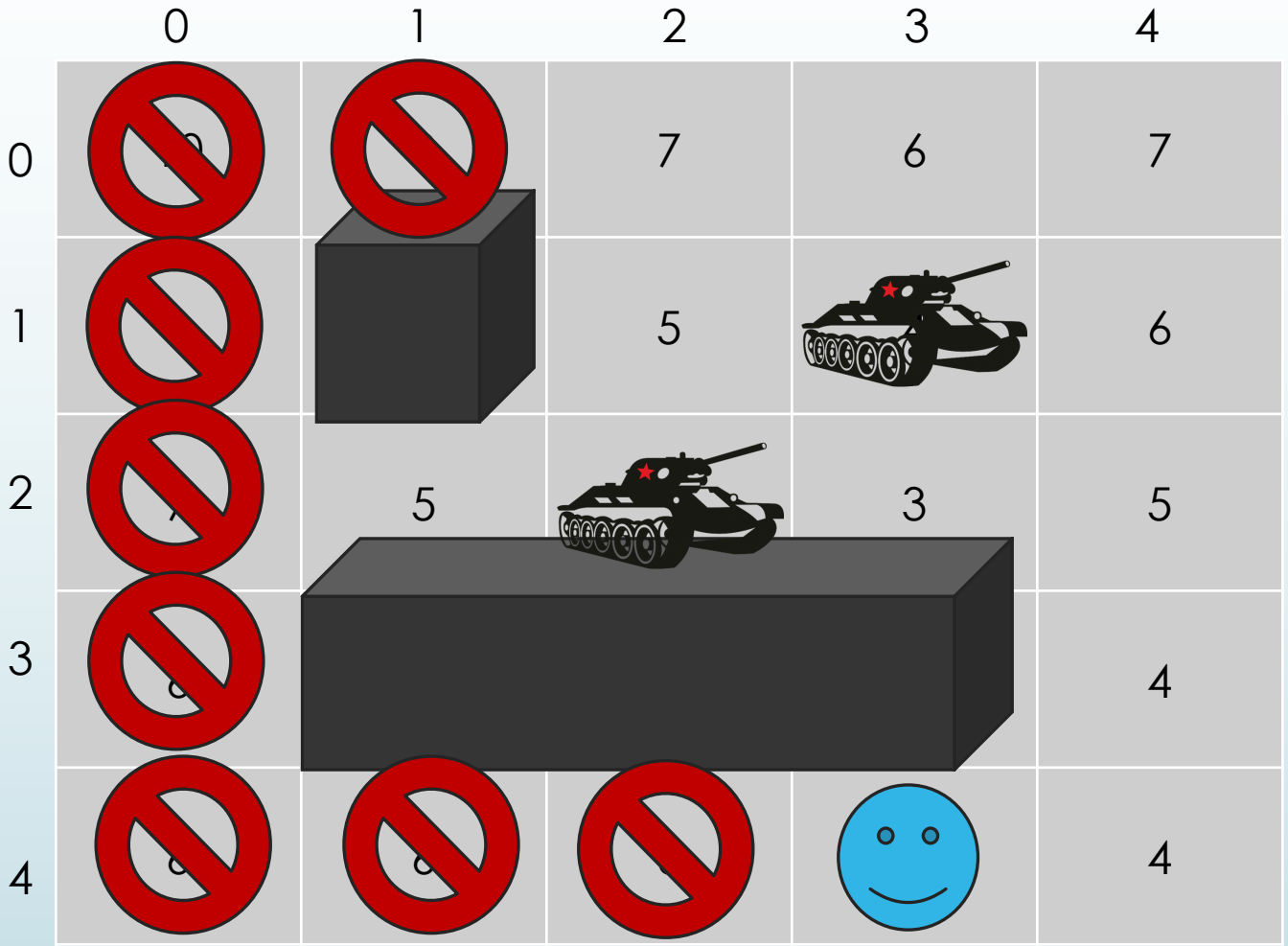
World With Movement Values with Manhattan Heuristic

	0	1	2	3	4
0	10	8	7	6	7
1	8		5	4	6
2	7	5		3	5
3	6				4
4	6	6	6		4



These are f(n) values

World With Movement Values with Manhattan Heuristic – Worst Case



Adding the Heuristic to A*

- If we only look at the number of current steps, then A* degrades into a basic breadth first search
- By adding to the cost the current straight line distance from the start and end point
  - Open.x.cost = current.cost+1+Heuristic();
  - We push the tank towards the right path faster

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
  - Develop pathways explicitly for longer movements as per before

Tabu Search

- Glover 1980s
- Idea coming from a social taboo – never return to where I once was
  - Heuristic – were I have seen is not where I want to go
- Search is conducted by single point changes in the candidate solution with a list of previous places that the search was at, a Tabu List
- The tabus are feature sets of the parameters
- Searches locally, then places the locality on the tabu areas of the search – forcing it to move away

Tabu Loop

- Set a current position, set best to this position, set a tabu list to null
- While not at a stopping condition
  - Create a candidate list from neighbours
  - Check if the candidates are on the tabu list. If they are, remove them
  - Find the best candidate
  - Move to this as current position
  - Test the best candidate against the best
    - If it is the new best – replace
    - Create a tabu by looking at the differences in the features between this new best and the previous best
    - Produce a tabu restriction based on this difference
    - Add this tabu to the tabu list
    - Check tabu list length and remove old tabus

Example - Tabu

- String of 5 parameters – containing numbers 0-9
- Tabu List is *,*,3,*,*
- New Tabus are created by adding one point where the best strings matched
- Best is 0,3,3,2,1 scoring 14
- Local neighbourhood is +/-1 from each parameter
- Generate the new list
  - 0,2,3,2,0 - 17
  - 0,3,4,2,1 - 18
  - 0,4,4,2,0 - 20

Compare and Add Tabu

- 0,3,3,2,1
- 0,4,4,2,0
- 0,*,*,2,*
- Add the tabu of 0,*,*,*,* or *,*,*,2,*
- New Tabu list of *,*,3,*,* and (0,*,*,*,* or *,*,*,2,*)

Issues with Too Many Tabus

- Small sets of tabus might make it impossible to move
- In our example with 3 tabus all happening about the same value, it would lock movement completely (lock upper and lower move, then move between and make that tabu)
- Rationalize the Tabus – Costly
- Only allow a small enough number of Tabus – where is this necessary value? Is it too low? How often will it happen?
- Aspiration criteria – Accept Tabu moves if they SIGNIFICANTLY increase fitness