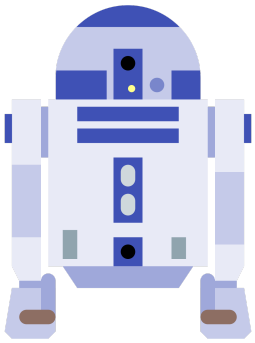
CSEN 901: Artificial Intelligence Course

Project 1



Help R2-D2 Escape!

George Moheb	**31- 2149**	**T-11**
Hisham Zahran	**31-1997**	**T-08**
Rania Wael	**31-2560**	**T-08**

Problem Definition :

- ➢ R2D2 is required to traverse a 2D grid to reach a portal and successfully escape the Death Star.
- ➢ The 2D portal have several kinds of objects placed within its cells :
  - R2D2 himself (movable)
  - Rocks (movable)
  - Pressure Pads (unmovable)
  - Obstacles (unmovable)
  - Portal (unmovable)
- ➢ However in order to **activate** the portal, R2D2 would have to push rocks on all unactivated pressure pads.
- ➢ Possible movement directions are {North, South, East, West} (hence the search tree branching factor would be 4).
- ➢ Rocks are **pushed** whenever :
  1. R2D2 destination cell (in the movement direction) is occupied by a rock.
  2. The destination cell with respect to the **rock** (also in the movement direction) is not an *obstacle* nor a *border boundary* nor *another rock*.

➢ Below is the **visualization** of the world surrounding R2D2 while solving a specific problem :

*Figure above shows the result of pushing a rock on a pressure pad to **activate** it turning it green*

*Figure above shows a **goal** state where all the pressure pads are activated and R2D2 escapes from the portal*

Abstract Data Types:

➢Node ADT :

A node acts as a wrapper for representing a state while keeping track of :

1. Parent node : representing previous state.
2. Operator : action taken whether moved north, south, east or west.
3. Depth : current state space tree level.
4. Path Cost : total cost from root till the current node.
5. Path List : total path from root till the current node.

➢Generic Search Problem ADT :

A generic search problem is defined by a 5-tuple consisting of :

1. Operators : a list of all moves with the costs of doing them.
2. Initial State : our starting state that will be stored in the root node.
3. State Space : the space of all the possible states that we can reach from the initial root node.
4. Goal Test : evaluating whether a state is considered a goal state or not.
5. Path Cost : total path cost needed to go from the initial state to the goal.

HelpR2-D2 Problem:

It's a subclass of the generic search problem ADT implementing its main functions :

1. Actions : defines the 4 possible movements alongside their costs which are

   {(**North**, 1), (**South**, 1), (**East**, 1), (**West**, 1)}

2. Initial State : a **state** is defined by 2 parameters

   {Current R2D2 Position, Current Rocks Positions}

   So in our case the initial state is

   {position (0,0), Initial Rock Positions}

3. Expansion Function : responsible for constructing the state space by expanding the current node, then returning a new list of nodes containing the possible states that we can move to from the current state using our 4 operators.
4. Goal Test : the goal here is to put every single rock on a single corresponding pad in order to **activate** it, followed by R2D2 **reaching** the portal for him to escape.
5. Path Cost : initially the path cost value is 0, then increases with expansion by adding it to the cost of doing one of the operators.
6. Static grid variables : rows count, columns count, pressure pads positions, unmovable obstacles positions and the teleportal position.
7. Memoization : a list storing all the visited nodes with the aim of avoiding replicas between the different states in order not to loop forever whenever there's no solution.

Main Functions :

The most significant functions for implementing the searching mechanism are :

- ➢ GenGrid () : A function generating and returning a top view grid representing the world surrounding R2D2.
  - The dimensions of the grid (m,n) are generated randomly with a minimum value of 3.
  - R2D2 and the teleportal are positioned randomly within the bounds of m and n.
  - A random list of non-overlapping unmovables are then generated.
  - 2 equal-sized random lists are generated for the rocks and pressure pads, pads can only overlap with R2D2 position, however rocks can't overlap with any object in general at the same cell.
- ➢ Search (grid , strategy, visualize) : A function creating a new problem from the given grid, then calls the general search with the problem defined alongside the queuing function related to the required strategy to be implemented.
  This function returns a 3-tuple of the path list from the initial state to the goal if a solution is found, the total path cost, the number of expanded nodes. **Visualisation** is also possible if a solution exists using *Visualize(grid, path_list)* function.
- ➢ general_search (problem, q_function) : the generic function implementing the search on a given problem by frequently expanding the first node in a queue containing the nodes representing the states, according to a given queuing function related to one of the search strategies. Details about how each search algorithm utilizes the queue is discussed in the next section of the 'Search Algorithms Queuing Functions'.

Finally returning a **goal node** if a solution exists or a **None** if no solution found.

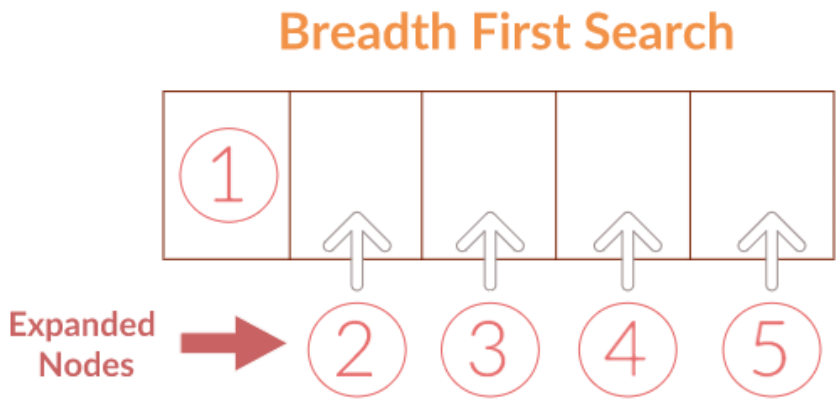
- ➢ state_space (node) : the expansion function which exploits the current positions of R2D2 and the rocks from the state, in order to check for surrounding obstacles (unmovables, consecutive rocks or grid bounds) before performing an action, followed by adding a new node for a possible movement to the expansion node list.
  Finally returning this **node list** for the queueing function of a specific search strategy to resume the searching process.
- ➢ direct_path (state) : a function returning a heuristic cost based on the **Direct Path** heuristic discussed in details in the 'Heuristics' section.
- ➢ sum_distances (state) : a function returning a heuristic cost based on the **Sum Distances** heuristic also discussed in details in the 'Heuristics' section.

Search Algorithms Queuing Functions :

There are 6 different approaches of searching implemented using a queueing function for each search algorithm, utilizing the state space expansion node list, path cost and a specific heuristic cost.
Let's discuss how each strategy works :

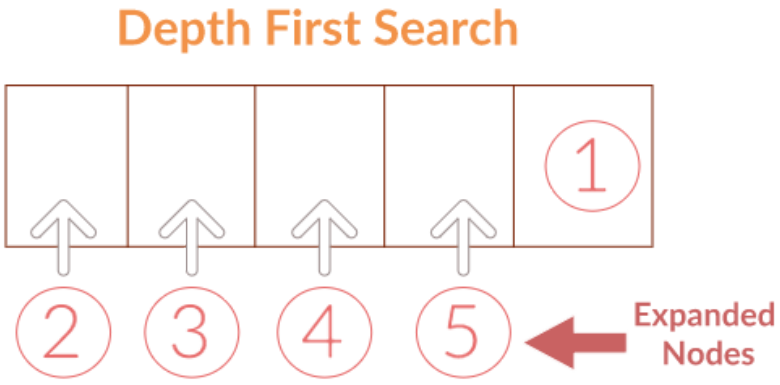
➢ Breadth-First Search :

Adjusted the queuing function, so that the children of each expanded node are inserted at the **end** of the queue.



➢ Depth-First Search :

Adjusted the queuing function, so that the children of each expanded node are inserted at the **beginning** of the queue.



➢Iterative Deepening Search :

This type of search requires an external **iterator** to be implemented in order to keep track of the current maximum level that can be reached. So in order to handle this issue, we insert respectively at the end of the queue :

1. Dollar sign ($) splitting the main queue from the variables implementing the iterator logic.
2. Maximum depth added to the queue till then, for handling memoization issues that could arise when removing replicas.
3. Root of the tree.
4. Current iterative depth.

When a node is expanded, the depth of their children are compared to the iterative depth. Only if the iterative depth is greater than their depth, these nodes are added to the beginning of the queue and the maximum depth added to the queue till then is updated with the children's depth.

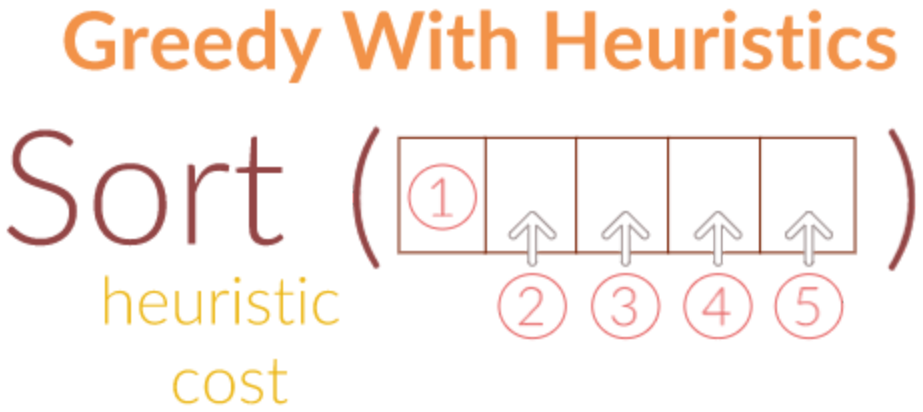
When the dollar sign ($) is reached to be the first element in the queue, the iterative depth is incremented by 1 and the root is inserted at the beginning of the queue to start all over again.

➢Uniform-Cost Search :

The queuing function is maintaining the ascending order of the nodes according to their path costs. Whenever a node is expanded, their children are inserted in the queue according to their path costs from the root.

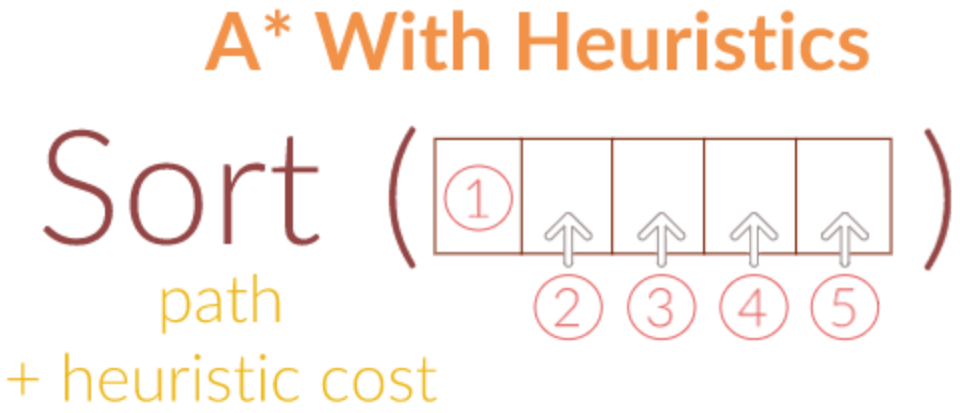
➢Greedy Search with 2 heuristics :

In the greedy approach, nodes after every expansion are sorted according to how far they are from reaching a goal state, which is based on one of the heuristic costs discussed in the 'Heuristics' section below.



➢ A* Search with 2 heuristics :

The A* search algorithm gets the best of the 'Uniform-Cost' and 'Greedy' approaches by sorting the queue after every expansion according the minimal **path cost** considered from the start till the current state, in addition to the minimal **heuristic cost** that will make R2D2 reach the goal faster if the heuristic is **admissible**.



Heuristic Functions :

1. Direct Path :
    a. Problem Relaxation :
        i. Ignores the existence of obstacles.
        ii. Assume a path to a cell is the direct block distance (absolute(deltaX) + absolute(deltaY)) between the current position and that cell.
        iii. Assume a simpler rock pushing mechanism; R2D2 would have to stand on top of the rock and simply move carrying it with it.

b. Methodology :
    i. Constructs a path by alternating between finding the nearest rock then finding the nearest pressure pad selection.
    ii. After all Pads are activated the shortest path between the last Pressure Pad and the Portal is constructed.
    iii. All the block distance moved is accumulated to be returned as the estimated cost to the goal.
c. Admissibility :
    i. The constructed path will always ignore detours required by R2D2 to reach other cells; which would underestimate in case of existing obstacles but never overestimate a direct path between cells (block distance is merely a combination of north, east, west, south movements).
    ii. The constructed path uses a greedy approach (not optimum) when looking for closest Rocks/Pads to reach first; which would underestimate in cases where an optimum path exists that would make pushing rocks overlap with a path to the portal.
    iii. Simplified Rock pushing mechanism ignores the need for R2D2 to adjust while pushing rocks; hence underestimating whenever R2D2 pushes a rock or simply yielding the actual cost in case R2D2 is moving in the direction of the Rock pushing.

2. Sum Distances :

a. Problem Relaxation :
    I. Ignores the existence of obstacles.
    II. Ignores the path R2D2 have to go through in order to reach the portal.

III. Assume a path between 2 cells is the direct block distance (absolute(deltaX) + absolute(deltaY)).
IV. Assume that our goal is just to place every rock on a pressure pad to activate it. So R2D2 will always stick to a rock till it reaches the nearest pressure pad.

b. Methodology :

The heuristic cost in this case is the accumulated sum of the block distances between a rock and its nearest unactivated pressure pad, favoring a state where each rock is placed on the closest pad to it.

c. Admissibility :

I. Just like the first heuristic, we are ignoring the part where R2D2 has to adjust himself in order to push the rock.
II. As we are ignoring the path R2D2 should take in order to reach the portal, it will move in a trivial BFS way as the cost will be constant for all paths in case of no nearby rocks, so we are not overestimating the actual cost anyway.
III. For a direct path from R2D2 to the portal having an overlapping set of rocks next to some pressure pads, we will always be underestimating the actual cost as we are ignoring the path taken from R2D2 to reach the first rock, in addition to ignoring the path between the last pressure pad to be activated and the portal.

Performance Analysis :

In this section we'll be discussing the performance it terms of some metrics like completeness, optimality, no. of expanded nodes of each search algorithm implemented.

➢ Breadth-First Search :

- Completeness : As the state space expands level by level as if R2D2 is searching the area around it in a spiral-like manner, it's guaranteed to find a solution; hence it's complete, even without using the memoization step for avoiding replicas.
- Optimality : In general as it isn't considering the best path cost when moving in any of the 4 directions, it's not guaranteed to find the best solution for reaching the goal. However in our case as the cost of making any of the actions is the same and non-decreasing across the different levels, we can say that it's optimal.
- Expansion : In the case of no repeated states, it will have a worst space complexity of $O(b^d)$ where b is the branching factor which is 4 in our case and d is the maximum depth reached.
  However handling and removing replicas decreases the number of expanded nodes in order to consider the fail state whenever there's no solution, so our worst space complexity in that case will be $O(s)$ where s is the number of all **unique** possible states.

➢ Depth-First Search :

- Completeness : The DFS takes the approach of sticking to a specific path till it reaches the end of it whether it's a goal

state or a fail state with no other possibilities. For that reason it can reach a point where R2D2 is just going back and forth forever in the same area, not reaching the goal state if there's a possible solution, hence it's not complete. However, by handling and removing the replicas, it will never revisit a state which means that it will never be stuck in a loop. So if there is a solution, finding it is guaranteed. Therefore, our DFS search is complete.

- Optimality : It's not optimal in general as it doesn't consider the path cost just like the BFS, and also in our case as it 's favoring to try paths that can get R2D2 to make more detours that could have been avoided if it followed the optimal path.
- Expansion : In the case of no repeated states, it will have a worst space complexity of $O(bd)$ where b is the branching factor which is 4 in our case and d is the maximum depth reached. In the case of handling and removing replicas, the complexity is the same as the BFS.

➢ Iterative Deepening Search :

- Completeness : It is doing a DFS while searching however mimicking the behaviour of BFS in going level by level, so it's also complete.
- Optimality : Optimal as it satisfied the same BFS optimality condition in which the path cost is non-decreasing as it expands.
- Expansion : Same complexity as the DFS per each iteration.

➢ Uniform-Cost Search :

- Completeness : In order for the uniform cost search to be complete, the cost of each node should always be less than the cost of its successors. Since we only deal here with

positive costs, then it's guaranteed to find a solution, hence it's complete.

- Optimality : In order for the uniform cost search to be complete, the cost of each node should always be less than or equal the cost of its successors. Since the path costs are the same with value of 1, then we can say that it's also optimal.
- Expansion : In general the UC algorithm has a complexity of $O(b^d)$ where d was the total path cost over the minimum path cost, however as in our case the cost of each action is the same, it will be acting as a BFS search so d will be the maximum length or depth reached.
  In case of handling replicas, the worst space complexity will be also $O(s)$.

➢ Greedy Search with 'Direct Path' Heuristic :

- Completeness : In general the greedy algorithm can't be considered as complete in the cases of not handling replicas as there may exist 2 states where their heuristic costs are the same so it will keep alternating in an infinite loop.
  For instance in our case, imagine that R2D2 is surrounded by a set of obstacles standing in the way of the nearest rock, so R2D2 will not think of the option of taking a detour in order to reach the rock however it will favor going in the rock direction as we are ignoring the obstacles in the heuristic cost. The same is applied whenever going for a pressure pad or the portal.
  On the other hand removing replicas will remove this redundancy and will be complete in this case.

- Optimality : The greedy search in general gets the local optimal solution at each step, however, that does not indicate a global optimal solution, so it is not optimal.
- Expansion : The worst space complexity is $O(b^d)$ just like the BFS.

➢Greedy Search with 'Sum Distances' Heuristic :

- Completeness : Same as the greedy algorithm with the first heuristic, however it will fail in the case of already reaching a rock where R2D2 has to deliver it on the nearest pressure pad, however there's a set of obstacles in between forcing him to make a detour with the rock in order to reach the pad. Without handling repetitions it will just try to push the rock in the same direction forever. Whereas if we handled the repeated states, it will be complete.
- Optimality : Same as the first heuristic case, it doesn't guarantee optimality.
- Expansion : The worst space complexity is $O(b^d)$ just like the BFS. However it expands more than the first heuristic because while there's no rock around that needs pushing to the nearest pad, it will move in a BFS way as the heuristic cost will be the same for all expanded nodes.

➢A* Search with 'Direct Path' Heuristic :

- Completeness : A* search in general is complete because it considers also the path cost with the heuristic cost. So in our case it guarantees that R2D2 will never get trapped inside an infinite loop.
- Optimality : As we proved that the heuristic cost is admissible, so by considering the path cost we can say that it's optimal.

- Expansion : The worst space complexity is $O(b^d)$ just like the BFS.

➢ A* Search with 'Sum Distances' Heuristic :

- Completeness : Complete for the same reason as the A* approach with the first heuristic.
- Optimality : Also as we proved already that this heuristic is also admissible, so we can say that it's optimal.
- Expansion : The same issue with the greedy utilizing the second heuristic, that whenever there's no rock around it will just keep trying every possible movement around till it reaches one in a BFS manner, hence it expands nodes more than the A* with the first heuristic.

How To Run :

➢ The project is implemented using python, consisting of 5 separate files containing the main functions discussed throughout the report.

➢ The testing file is called **main.py**. In order to run it use

python main.py

➢ You will be prompted to select which type of strategy to use separated by commas, you can type :

- BF for breadth-first search,
- DF for depth-first search,
- ID for iterative deepening search,
- UC for uniform cost search,
- GRi for greedy search, with i ∈ {1, 2} distinguishing the two heuristics, and
- ASi for A∗ search, with i ∈ {1, 2} distinguishing the two heuristics

- ➢ The output will be the Search function output for each strategy which was :
    - The path list from initial state to the goal state if there's a solution, None otherwise.
    - The total path cost.
    - The number of expanded nodes.
- ➢ As for the visualization, it is implemented using a python library called **pygame**. To install it, **pip** package manager of python can be used

  pip install pygame
- ➢ To run the visualization, **type "Y"** after running the main.py whenever prompted if visualization is needed.
  If there's a solution, the visualization will show the path list generated after each search strategy.
- ➢ In order to repeat one of those visualizations, just **press "R"**.
- ➢ In order to try new values for the grid dimensions, change the range of generating random **m** & **n** values in **grid.py**.