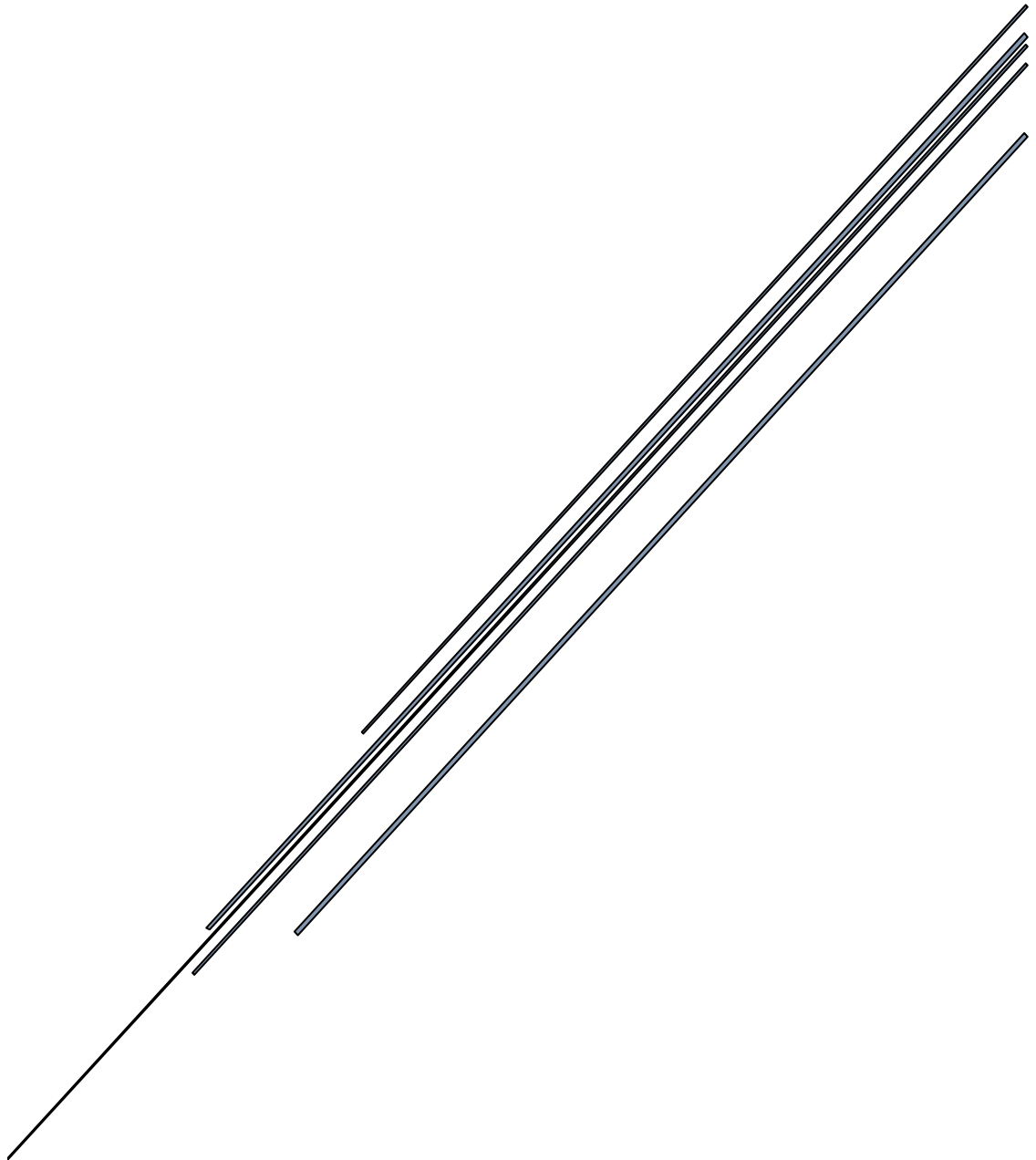


CSEN 901: AI

Project 1 Report



Instructor: Dr. Haythem O. Ismail

Done by: Raneem Wael (43-3851), Abdelrahman Khalifa (43-3878),
Heidi Talaat (43-2081), Rowan Bassem (43-4669) – Team 47

Contents

Project Overview	2
Design.....	3
Search Tree Node.....	4
State	4
Generic Search Problem	5
Matrix Problem	7
Search Algorithms	11
Breadth First Search.....	11
Depth First Search.....	11
Iterative Deepening Search	11
Uniform Cost Search.....	11
Greedy Search.....	12
A* Search.....	12
Examples	13
Example 1	13
Example 2	16
Visualize Example	20

Project Overview

In this project, we are aiming to use Java and Eclipse to develop a prototype or simply an application to formulate winning plans of a grid problem using the search tree methods we learned about in this course.

The prototype we will be implementing is a form of program where the user inputs a grid String containing information about the matrix, including its size and the placements of the variables.

The grids represent a world space identical to that of “The Matrix” movie franchise, where “Neo” tries to save as many hostages as possible from the virus “Agent Smith” without dying.

The search tree methods implemented are:

1. Breadth-first search.
2. Depth-first search,
3. Iterative deepening search.
4. Uniform-cost search.
5. Greedy search with two heuristics.
6. A* search with two admissible heuristics.

Design

The project is divided into four main classes as follows:

1. “Node.java”: which is a class representing an object of type search tree node.
2. “State.java”: which is a class representing the state of a search tree node.
3. “GenericSearchProblem.java”: which is an abstract class representing an object of type search problem.
4. “Matrix.java”: which is a class extending “GenericSearchProblem.java” and represents the main class of our program.

The rest of the classes represents the various different objects that can exist in the grid. These are:

1. Agent.
2. Hostage.
3. Neo.
4. Pad.
5. Pill.
6. TelBooth.

Each of the previous objects are represented by x and y coordinates, damage integers for both “Neo.java” and each “Hostage.java”, and the position a flying pad transports to for the “Pad.java” class.

Each class object also overrides the *toString* method to output, respectively:

1. “A”.
2. “H (*damage*)”.
3. “Neo (*damage*)”.
4. “Pad (*padNewPosX, padNewPosY*)”.

5. “P”.
6. “TB”.

The “Hostage.java” class carries more information, as attributes, regarding a hostage’s current status; whether they are carried, saved, dead or have become an agent.

Search Tree Node

We represented a search tree node having the following attributes:

1. Node parentNode: which represents the parent of this node if it exists, otherwise it is set to null.
2. Int dept: to aid us in the traversal of the iterative deepening search so we know when to step the traversal in accordance with the chosen depth.
3. Int cost: which is set to the cost of its state.
4. String operator: representing the action “Neo” has taken to reach this node; this could be any of “up, down, left, right, kill, takePill, fly, carry, drop”.
5. State state: carries data about the state “Neo” is in when he is in said node.

State

We represented a search tree node state in a class having the following attributes:

1. Int cost: which represents the cost of going to this state. We set it to be as an addition of the number of kills “Neo” has made so far and the number of hostages that died so far or to reach this state. This represents the standard uniform cost. We included four other costs, two heuristics for each of the greedy and a star searches. They are labelled “g1Cost, g2Cost, a1Cost and a2Cost”. More about each of them is explained in the coming *Search Algorithms* section of the report.
2. Int neoPosX, neoPosY: the new position of “Neo” if he transitions with this state.
3. Int neoDamage: the new damage of “Neo” if he transitions with this state.

4. Int neoPrevPosX, neoPrevPosY: the position of “Neo” before he transitions with this state.
5. Vector<Agent> agents: contains the remaining agents still in the matrix that “Neo” did not kill.
6. Vector<Pill> pills: contains the remaining pills still in the matrix that “Neo” did not take.
7. Vector<Hostage> hostages: contains the remaining hostages still in the matrix that “Neo” did not carry or those that have turned to agents.
8. Vector<Hostage> carriedHostages: contains the that “Neo” is currently carrying.
9. Int kills: which is set to the number of agents or agent hostages that “Neo” has killed so far (up to this state).
10. Int deaths: which is set to the number of hostages that have died so far (up to this state). This is calculated through the “**int** deaths()” method in the same class, which counts the number of hostages dead in both the hostages and carriedHostages vectors.

It contains one important method worth noting; in this class, we override the *toString* method to output a string containing “Neo’s” positions, his damage, the vectors of hostages, carriedHostages, pills, and agents. We will use this later to ensure no repeated states will be taken.

Generic Search Problem

This class is the one that attempts to find a solution to a given grid.

We represented a generic search problem in an abstract class having the following attributes:

1. Int expandedNodesCount: which represents the number of nodes expanded to procure a goal state.
2. State initState: carries data about the initial state of “Neo”.
3. Int expandedNodesCount: this aids in solving the problem using the iterative deepening search, it is set to the depth level and incremented if a solution is not found at it.
4. String[] operators: representing the actions “Neo” can take; “up, down, left, right, kill, takePill, fly, carry, drop”.
5. Hashtable<String, String> states: carries data the visited states, it ensures no repeated states occur.

This class contains two important abstract methods to be implemented in the “Matrix.java” class and will be explained in more details later in the *Matrix Problem* subsection:

1. **State neoAction(State state, String operator)**: this method returns a new state if the operator used on the inputted state is valid in accordance with the matrix rules and does not result in a repeated state.
2. **boolean goalState(Node node)**: this method return true if the node “Neo” is currently at represents a goal state.

It contains three important methods which are:

1. **Node genericSearch(String strategy)**: this method returns a goal node, for a given matrix with the inputted strategy, if it exists.
 - a. We first create a new vector representing the *nodesQueue* and add the root node to it. This node has no parent, 0 depth, 0 cost, no operator and an initial state.
 - b. We then iterate over the vector and while it is not empty, we remove the first node and check if it is a goal state and return it if so, otherwise we expand the node and merge the resulting nodes to the *nodesQueue* according to the search strategy inputted to the method.
 - c. We then repeat the iteration, if all nodes are expanded and no goal state is found, the method returned null.
2. **Vector<Node> expandNode(Node node)**: this method returns a vector of the possible nodes resulting from the expansion of the inputted node.
 - a. We first increment the *expandedNodesCount* variable by 1.
 - b. We then create an empty vector representing the *resultingNodes* from the expansion.
 - c. We then iterate over the operators and call the *neoAction* abstract method on the node on each operator separately then add the resulting nodes (if any) to the *resultingNodes* vector.
3. **Vector<Node> addToQueue(String strategy, Vector<Node> nodesQueue, Vector<Node> resultingNodes)**: this method returns a vector of the ordered nodes queue in accordance with the strategy inputted. More will be explained about this method in the *Search Algorithms* section.

It also contains one helper function; **Vector<Node> sort(Vector<Node> nodesQueue, String strategy)**. This function represents an insertion sort to order the nodes when using the greedy and a* search strategies according to their relative costs. It takes the nodesQueue and the search strategy as input and returns the ordered queue.

Matrix Problem

This class represents the generic search problem however tailored to the specific problem, being the grid.

We represented the matrix problem having the following attributes:

1. Int N: represents the matrix height.
2. Int M: represents the matrix width.
3. Object[[]] grid: which represents the matrix as a 2D grid of objects in their respective locations.
4. Neo neo: this represents “Neo”.
5. Int C: represents the maximum number of hostages “Neo” can carry at once.
6. Vector<Agent> agents: contains the agents in the matrix.
7. Vector<Pill> pills: contains the pills in the matrix.
8. Vector<Pad> pads: contains the pads in the matrix.
9. Vector<Hostage> hostages: contains the hostages in the matrix.

It contains two important static methods which are:

1. **String genGrid()**: this method returns a string of a matrix grid. The positions of all the matrix objects are generated randomly while maintaining no overlaps.
The conditions set are as follow:
 - The grid dimensions are between 5x5 and 15x15.
 - The number of hostages “Neo” can carry simultaneously is between 1 and 4.
 - The number of hostages in the matrix is between 3 and 10.
 - The number of pills should be less than the number of hostages.
2. **String solve(String grid, String strategy, boolean visualize)**: this method returns a string of the solution (if it exists) for the inputted grid using the

strategy inputted. If the *visualize* input is set to true, the plan is illustrated in the console one a step-by-step basis.

- a. We first create a new matrix and initialize its variables using the inputted grid string.
- b. We then implement the generic search problem on the matrix using the requested strategy and store the results in a node.
- c. If the node is not null (i.e., a goal state exists), we traverse back through parent nodes to create the plan part of solution string. Otherwise we return a string saying “No Solution”.
- d. All the while, we are storing the nodes in a vector, adding each node at the front to be able to then remove them one by one and print its contents in a grid like form.
- e. The resulting solution string includes the plan, the number of hostage death, the number of kills and the number of nodes expanded.

It contains two important abstract methods which are:

1. **State neoAction(State state, String operator)**: this method returns a new state if the operator used on the inputted state is valid in accordance with the matrix rules and does not result in a repeated state.
 - We check if a state is repeated (after its generation) by checking whether or not it is in the parent’s class (*GenericSearchProblem*) Hashtable containing the states “Neo” passed over so far.
 - For each taken action, all the living hostages damages increase by 2.
 - For an operator to be valid it needs to follow the following rules:
 - a. Up, down, left, right: for “Neo” to be able to take any of these actions from the current state, the positions of the new state first need to exist within the matrix boundaries, and no agents or agent hostages or hostages with a damage count of greater than or equal to 98 (as for each neoAction, the damage of all living hostages increases by 2, and if they are not currently carried by “Neo” they turn to agents and “Neo” cannot transport to the same cell as an agent) can exist in that location. If “Neo” takes any of these actions, his x and y positions are updated.
 - b. Carry: for “Neo” to be able to carry a hostage he needs to be in the same cell as a hostage and the hostage needs to not be carried or has become an agent or is saved. In addition to that, “Neo” needs to be carrying a number of hostages less than C (the maximum he

could simultaneously carry). If a hostage is carried, we remove it from the *hostages* vector and add it to the *carriedHostages* vector and set its carried flag to true.

- c. Drop: for “Neo” to be able to drop the carried hostages, the *carriedHostages* vector needs not be empty and “Neo” needs to be in the same cell as the telephone booth. If a hostage is dropped, we remove it from the *carriedHostages* vector and set its carried flag to false. If it is not dead, we set its saved flag to true.
 - d. Kill: for “Neo” to be able to kill, any of the four adjacent cells to that of the current one (up, down, left, or right) need to contain an agent or an agent hostage and “Neo’s” current damage count need not be equal to or greater than 80, as to take such an action, his damage increases by 20 points and “Neo” cannot die. If a hostage is killed, we remove it from the *hostages* vector and set its dead flag to true.
 - e. TakePill: for “Neo” to be able to take a pill he needs to be in the same cell as a pill. If a pill is taken, we remove it from the *pills* vector. We also decrease the damage of “Neo” and all living hostages that have not become agents by 20 points. We ensure that is the damage gets below 0, it is set to 0.
 - f. Fly: for “Neo” to be able to fly he needs to be in the same cell as a pad. We also decrease the damage of “Neo” and all living hostages that have not become agents by 20 points. We ensure that is the damage gets below 0, it is set to 0. If “Neo” flies, his x and y positions are updated.
2. **boolean goalState(Node node)**: this method return true if the node “Neo” is currently at represents a goal state. For it to be a goal state, “Neo” needs to have saved all living hostages by dropping them off at the telephone booth (*carriedHostages* vector should be empty), or if they have died before he ever got to them, then he needs to have killed them (all hostages in *hostages* vector need to be dead). Neo also needs to be at the telephone booth himself with a damage of less than 100 (i.e., he needs to be alive).

It contains some other helper functions which are stated below:

1. **int randomInt(int min, int max)**: this is a helper function to generate a random integer between two inclusive numbers.

2. **void findFreeCell(Object[][] grid, int x, int y)**: this method takes as input a position and a grid and returns a new empty position if the inputted one was not free/contained a matrix object.
3. **void initializeVariables(String gridString)**: this method takes as input a grid in a form of a string and dissects it to initialize the matrix class variables accordingly. It also sets the *initState* variable of the parent class (*GenericSearchProblem*) with “Neo’s” positions of x and y, “Neo’s” damage, the vectors of the agents, hostages, and pills, an empty vector for the carried hostages and finally a kill count of 0. Moreover, it sets the operators of the parents class to a string array containing the different actions “Neo” can take in this order { "up", "down", "left", "right", "carry", "drop", "takePill", "kill", "fly" }.
4. **void incDamageHostage(Vector<Hostage> hostages)**: this method increases the damage of all hostages in the inputted vector that are not dead or agents. It also ensures that when we add 2 points to their damage, it does not exceed 100 as well as turning the hostage into an agent if it does reach 100 and the hostage is not carried or just set their dead flag to dead if they are carried.

Search Algorithms

We implemented the search algorithms in the *GenericSearchProblem* class in the *addToQueue* method.

```
Vector<Node> addToQueue(String strategy, Vector<Node> nodesQueue,  
Vector<Node> resultingNodes)
```

Breadth First Search

To implement breadth first search, we simply concatenate the *resultingNodes* to the end of the *nodesQueue*.

Depth First Search

To implement depth first search, we simply concatenate the *nodesQueue* to the end of the *resultingNodes*.

Iterative Deepening Search

To implement iterative deepening search, we first check if we are at the maximum depth we decided on, if yes, we add the *nodesQueue* to the end of the *resultingNodes*. Otherwise, we check if we have no more nodes to expand (*nodesQueue* is empty); meaning we reached the maximum depth, so we add 1 to the *iterativeDepth* variable of the class *GenericSearchProblem*, empty the *states* hashtable of the class *GenericSearchProblem* and add the root node to the *nodesQueue*.

Uniform Cost Search

To implement depth first search, we combine the *nodesQueue* and the *resultingNodes* then sort them according to their costs.

Greedy Search

To implement greedy search, we combine the *nodesQueue* and the *resultingNodes* then sort them according to their $g(i)$ costs.

The $g1Cost$ is calculated according to the following heuristic function:
 $(\text{Math.abs}(\text{neoPosX} - \text{Matrix.telBooth.posX}) + \text{Math.abs}(\text{neoPosY} - \text{Matrix.telBooth.posY}))/4$. It is a spin on the manhattan's distance heuristic where we divide by 4 instead of 2 to maintain admissibility.

The $g2Cost$ is calculated according to the following heuristic function:
 $(\text{Math.abs}(\text{neoPosX} - \text{Matrix.telBooth.posX}) + \text{Math.abs}(\text{neoPosY} - \text{Matrix.telBooth.posY}) + \text{hostagesLeftToSave()})/8$. It is a spin on the manhattan's distance heuristic where we add the number of hostages left alive that "Neo" has not carried yet. We divide by 8 to maintain admissibility.

A* Search

To implement greedy search, we combine the *nodesQueue* and the *resultingNodes* then sort them according to their $a(i)$ costs.

The $a1Cost$ is calculated as the addition of the $g1Cost$ heuristic function and the default cost (uniform cost) of the node. As the $g1Cost$ heuristic function is admissible, so is the $a1Cost$.

The $a2Cost$ is calculated as the addition of the $g2Cost$ heuristic function and the default cost (uniform cost) of the node. As the $g2Cost$ heuristic function is admissible, so is the $a2Cost$.

Examples

RAM Usage when idle: 57M of 256M.

From the following two examples, we can deduce that DF search has the least CPU utilization.

Example 1

We will be using grid 1 to compare between the 8 different search trees.


Grid5: "5,5;2;0,4;3,4;3,1,1,1;2,3;3,0,0,1,0,1,3,0;4,2,54,4,0,85,1,0,43".

1. BF:

Output:

"left,left,left,left,down,carry,down,down,kill,down,right,right,carry,up, right, right, drop;1;2;2827"

- Completeness: Yes, as the grid is finite.
- Optimality: No, as the costs of all actions are not equal. The BFS algorithm stops at the shallowest goal found. The shallowest goal node need not compulsorily be the optimal goal node.
- RAM Usage: 57M – 67M = 10M.
- CPU Utilization:

>  eclipse.exe

5.4%


- Number of expanded nodes: 2827.

2. DF:

Output:

"down,down,down,down,left,left,up,up,up,up,left,left,down,carry,up,right, right,down,down,down,down,left,kill,up,up,left,up,up,down,down,down,down, right,kill,up,up,left,up,up,right,right,down,right,down,down,right, drop;2;3;52"

- Completeness: Yes, as the grid is finite.
- Optimality: No.
- RAM Usage: 57 – 72M = 15M.
- CPU Utilization:

>  eclipse.exe

4.1%


e. Number of expanded nodes: 52.

3. ID:

Output:

“down,down,down,down,left,left,carry,up,up,up,up,left,left,down,carry,up, right,kill,fly,kill,right,right,right,right,drop;1;3;13006”.

- Completeness: Yes, as the grid is finite.
- Optimality: No, as it may need to search through all depths iteratively if goal state is in a very far depth.
- RAM Usage: $101\text{M} - 57\text{M} = 44\text{M}$.
- CPU Utilization:

>  eclipse.exe

15.8%


e. Number of expanded nodes: 13006.

4. UC:

Output:

“down,down,left,takePill,left,left,left,up,carry,down,down,down,carry, right,up,right,right,drop,down,left,left,carry,up,right,right,drop;0;0;760”.

- Completeness: Yes, as the grid is finite.
- Optimality: Yes, as there are no negative costs.
- RAM Usage: $97\text{M} - 57\text{M} = 40\text{M}$.
- CPU Utilization:

>  eclipse.exe

7.6%

e. Number of expanded nodes: 760.


5. GR1:

Output:

“left,left,left,left,down,carry,down,down,kill,down,right,right,carry,up, right,drop;1;2;2827”.

- Completeness: Yes, as the grid is finite.
- Optimality: No, because they do not consider all the node states.
- RAM Usage: $93\text{M} - 57\text{M} = 36\text{M}$.

d. CPU Utilization:

>  eclipse.exe

7.9%


e. Number of expanded nodes: 2827.

6. GR2:

Output:

“left,left,left,left,down,carry,down,down,kill,down,right,right,carry,up, right,right,drop;1;2;2827”.

- a. Completeness: Yes, as the grid is finite.
- b. Optimality: No, because they do not consider all the node states.
- c. RAM Usage: RAM Usage: $93\text{M} - 57\text{M} = 36\text{M}$.
- d. CPU Utilization:

>  eclipse.exe

15.6%


e. Number of expanded nodes: 2827.

7. AS1:

Output:

“left,left,left,left,down,carry,down,down,kill,down,right,right,carry,up, right,right,drop;1;2;2827”.

- a. Completeness: Yes, as the grid is finite.
- b. Optimality: Yes, as different actions have different costs and so we traverse through the least costly ones first. Also, we are using an admissible heuristic function.
- c. RAM Usage: $106\text{M} - 57\text{M} = 49\text{M}$.
- d. CPU Utilization:

>  eclipse.exe

12.2%

e. Number of expanded nodes: 2827.


8. AS2:

Output:

“left,left,left,left,down,carry,down,down,kill,down,right,right,carry,up, right,right,drop;1;2;2827”.

- a. Completeness: Yes, as the grid is finite.

- b. Optimality: Yes, as different actions have different costs and so we traverse through the least costly ones first. Also, we are using an admissible heuristic function.
- c. RAM Usage: $91\text{M} - 57\text{M} = 34\text{M}$,
- d. CPU Utilization:

>  eclipse.exe

15.0%

- e. Number of expanded nodes: 2827.

Even though A* uses more memory than BF (uses least memory) and has more CPU utilization than DF (has least CPU utilization), but it guarantees that the path found is optimal. AS2 is the better option when regarding memory usage however AS1 is the better option when comparing CPU utilization.

Example 2

We will be using grid 5 to compare between the 8 different search trees.


Grid1: "5,5;1;1,4;1,0;0,4;0,0,2,2;3,4,4,2,4,2,3,4;0,2,32,0,1,38".

1. BF:

Output:

"left,up,left,carry,down,left,left,drop,up,right,carry,down,left,drop;0;0;877".

- a. Completeness: Yes, as the grid is finite.
- b. Optimality: No, as the costs of all actions are not equal. The BFS algorithm stops at the shallowest goal found. The shallowest goal node need not compulsorily be the optimal goal node.
- c. RAM Usage: $75\text{M} - 57\text{M} = 18\text{M}$.
- d. CPU Utilization:

>  eclipse.exe

4.7%

- e. Number of expanded nodes: 877.


2. DF:

Output:

"down,down,left,left,up,up,up,left,left,takePill,down,down,down,down,rig

ht, right, up, up, up, up, up, right, kill, down, down, down, down, left, left, up, up, up, up, carry, down, down, down, down, right, right, up, up, up, up, left, left, down, down, right, takePill, up, left, up, kill, down, left, drop; 1; 2; 99”.

- Completeness: Yes, as the grid is finite.
- Optimality: No.
- RAM Usage: $71\text{M} - 57\text{M} = 14\text{M}$.
- CPU Utilization:

>  eclipse.exe

3.0%


- Number of expanded nodes: 99.

3. ID:

Output:

“down, down, left, left, up, up, up, left, carry, down, left, drop, up, right, right, carry, down, left, left, drop; 0; 0; 3917”.

- Completeness: Yes, as the grid is finite.
- Optimality: Yes, as different actions have different costs and so we traverse through the least costly ones first. Also, we are using an admissible heuristic function.
- RAM Usage: $72\text{M} - 57\text{M} = 15\text{M}$.
- CPU Utilization:

>  eclipse.exe

5.8%


- Number of expanded nodes: 3917.

4. UC:

Output:

“left, up, left, carry, down, left, left, drop, up, right, carry, down, left, drop; 0; 0; 440”.

- Completeness: Yes, as the grid is finite.
- Optimality: Yes, as there are no negative costs.
- RAM Usage: $71\text{M} - 57\text{M} = 14\text{M}$.
- CPU Utilization:

>  eclipse.exe

3.4%


- Number of expanded nodes: 440.

5. GR1:

Output:

“left,up,left,carry,down,left,left,drop,up,right,carry,down,left,drop;0;0;877”.

- Completeness: Yes, as the grid is finite.
- Optimality: No, because they do not consider all the node states.
- RAM Usage: $71\text{M} - 57\text{M} = 14\text{M}$.
- CPU Utilization:

>  eclipse.exe

3.5%


- Number of expanded nodes: 877.

6. GR2:

Output:

“left,up,left,carry,down,left,left,drop,up,right,carry,down,left,drop;0;0;877”.

- Completeness: Yes, as the grid is finite.
- Optimality: No, because they do not consider all the node states.
- RAM Usage: $67\text{M} - 57\text{M} = 10\text{M}$.
- CPU Utilization:

>  eclipse.exe

3.7%


- Number of expanded nodes: 877.

7. AS1:

Output:

“left,up,left,carry,down,left,left,drop,up,right,carry,down,left,drop;0;0;877”.

- Completeness: Yes, as the grid is finite.
- Optimality: Yes, as different actions have different costs and so we traverse through the least costly ones first. Also, we are using an admissible heuristic function.
- RAM Usage: $70\text{M} - 57\text{M} = 13\text{M}$.
- CPU Utilization:

>  eclipse.exe

9.0%


- Number of expanded nodes: 877.

8. AS2:

Output:

“left,up,left,carry,down,left,left,drop,up,right,carry,down,left,drop;0;0;877”.

- Completeness: Yes, as the grid is finite.
- Optimality: Yes, as different actions have different costs and so we traverse through the least costly ones first. Also, we are using an admissible heuristic function.
- RAM Usage: $71\text{M} - 57\text{M} = 14\text{M}$.
- CPU Utilization:

>  eclipse.exe

4.1%

- Number of expanded nodes: 877.

Even though A* uses more memory than GR2 (uses least memory) and has more CPU utilization than DF (has least CPU utilization), but it guarantees that the path found is optimal. AS1 is the better option when regarding memory usage however AS2 is the better option when comparing CPU utilization.

Visualize Example

ID search on grid 5.

Step 0:

Carried hostages: 0

```

null | Pad (3, 0) | null | null | Neo |
H (43) | A | null | null | null |
null | null | null | P | null |
Pad (0, 1) | A | null | null | TB |
H (85) | null | H (54) | null | null |

```

Step 1: down

Carried hostages: 0

```

null | Pad (3, 0) | null | null | null |
H (45) | A | null | null | null, Neo (0) |
null | null | null | P | null |
Pad (0, 1) | A | null | null | TB |
H (87) | null | H (56) | null | null |

```

Step 2: down

Carried hostages: 0

```

null | Pad (3, 0) | null | null | null |
H (47) | A | null | null | null |
null | null | null | P | null, Neo (0) |
Pad (0, 1) | A | null | null | TB |
H (89) | null | H (58) | null | null |

```

Step 3: down

Carried hostages: 0

```

null | Pad (3, 0) | null | null | null |
H (49) | A | null | null | null |
null | null | null | P | null |
Pad (0, 1) | A | null | null | TB, Neo (0) |
H (91) | null | H (60) | null | null |

```
