# The Matrix: Escaped
# Report

## Search Tree Node Implementation

For the search tree node, we created a 5 tuple Class called Node with the following attributes:

a. currentState of type State which is an 8 tuple of the following attributes:
    i. Int neoX: carries the x value of the location of the state.
    ii. Int neoY: carries the Y value of the location of the state.
    iii. Int neoDamage: carries the value of the current damage of Neo
    iv. ArrayList allHostages: an array list of all hostages on the map with their information such as their x and y values (location), damage, and state (Alive, Carried, Mutated, Dead, and Saved).
    v. ArrayList allPills: an array list of all pills on the map with their following information: each pill's x and y (location)
    vi. ArrayList allAgents: an array list of all agents on the map with information of each agent's x and y (location) and state (Alive or Dead)
    vii. Int kills: the number of kills Neo got
    viii. Int C: the maximum number of hostages Neo can carry at any given time (capacity)

b. Parent of type Node which points to the node that came prior to the current node.

c. Operator of type Operators which is an enum that contains the following: left, right, up, down, carry, drop, kill, takePill, fly. These are the operators required according to the project description.
d. Depth of type int which indicates the depth of the node.
e. Cost of type int which indicates the cost of the node which is used in A* and Uniform Cost searches.

---

## Matrix Problem Implementation

We implemented the matrix problem in the following steps:

f. We split the grid string with the semicolon and create separate arrays of hostages, pills, agents, and pads all with their corresponding features (such as positions, damage, state, for pads -> start positions and end positions).

g. We create a 2d array of strings that contain the content of the grid.

h. We make a type of queue dependent on the type of search it is and a hashset of states, a starting state initialized with of all the necessary variables, and the start node

```
int[] dx = { -1, 1, 0, 0 };
int[] dy = { 0, 0, 1, -1 };
int nodesExpanded = 0;
HashSet<String> hashedStates = new HashSet<String>();
hostNo = hostages.size();
HashSet<State> set=new HashSet<State>();
Queue<Node> mainQueue = new LinkedList<Node>();
State startState = new State(neo.getX(), neo.getY(),c,0,0, hostages ,Pill, agents);
Node start = new Node(startState, null, null , 0, 0);
mainQueue.add(start);
```

**CELL WE ARE ON**

i. We remove nodes from the queue and check what position Neo is in **on** the map.

j. If Neo is **on** a hostage cell as the following if condition states, then we deep clone the hostage array since changes might happen to a specific hostage(i.e carried), we then loop to find out which hostage it is in our hostage array that we are standing on right now and see if he is alive, if so carry him if the capacity is not full by using the CanCarry method, finally you initialize the the new node with the same X Y position and every other variable along with the cloned hostage array and the operator used(which is carry)

```
if (grid[x][y].charAt(0) == 'H' ) {
```

k. The following block of code is the incur damage, it clones a new hostage array, and incurs damage that needs to be added when adding a new node(aka time step)

```
ArrayList<Hostage> newHostages2 = new ArrayList<Hostage>();
for (Hostage h2 : newHostages) {
    newHostages2.add(h2.clone());
}
for(int h3=0; h3<newHostages2.size(); h3++){
    newHostages2.get(h3).setDamage(newHostages2.get(h3).getDamage()+2);
    if(newHostages2.get(h3).getDamage()>=100  )

        if(newHostages2.get(h3).getState().equals("Carried"))
            newHostages2.get(h3).setState("Dead");
        else if(newHostages2.get(h3).getState().equals("Alive"))
        newHostages2.get(h3).setState("Mutated");

}
```

l. The next if-condition checks if we are on a telephone booth, if there are any hostages that are carried then execute a drop action, and change their state string to “Saved”.

m. There is also an if condition for if we are on a pill or on a pad. Note: if we are on a pill the incur damage block of code is not placed since that action does not incur damage.

n. If the current node we are on does not pass the goal test(goal test to be explained shortly) then we move on to the neighbor check of the current cell we are on

**THE FOLLOWING IS THE NEIGHBOR CHECK**

o. We then loop around all 4 possible directions for Neo to move in (north, south, east, west)

p. For the first if condition in the neighbor check, we check to see If a neighboring cell is a hostage, we see if that hostage is **Mutated** first, if so we need to kill it and then loop over all the neighboring cells, to see if there is another mutated hostage or an agent, since we need to

kill everything around neo that can possibly be killed, and then initialize a new node with same X Y position of neo and with kill operator.

q. If the hostage that is the neighbor we check is not mutated however, we execute a move action to that cell by position Col, Row which is the neighbor cell

r. However before moving to that hostage cell, if that hostage cell is about to turn (illegal case) then we do not move to the cell which is covered in the  block of code below.

```
    }
    boolean isHeGoingToTurn = false;
    for(int h4 = 0; h4 < newHostages.size(); h4++) {
       if (newHostages.get(h4).getX() == col && newHostages.get(h4).getY() == row) //get hostage
           ////carry the dead hostage as well i think?
            if (newHostages.get(h4).getDamage()>=98 && newHostages.get(h4).getState().equals("Alive"))
               isHeGoingToTurn = true;
    }
    if(!isHeGoingToTurn) {
```

**s. For any move action, we add it to the hashed set of states that we made, just so we do not make that same move action again.**

t. If a neighboring cell is an agent, you kill that agent as one of the possible moves that you can make and do the same thing as in step P, by checking if any other neighbors can be killed.

u. If that agent is already dead, then move to that cell.

v. As for any other neighbor that is either a Pill, TB, Pad, or an empty spot, the action is just a move action with incur damage added.

w. For any action here that is done an incur damage is also done.

x. We repeat steps p to u until all the neighboring cells are added and then go back to step i by removing the front node in the queue.

**GOAL TEST**

y. The first image in the figure below, checks to see if any hostages are still Alive,Carried or Mutated, if so then this is not a goal state and allHostagesSaved(Saved or Dead) is set to false.
z. If this is true, then the neighbor check is not executed as this could be a goal state

We then skip all the way to the second image code, and see whether we are on a telephone booth right now with all hostages saved or dead, if so we are on a goal node and we would like to return that node.

```
for(int h=0; h<currentNode.currentState.allHostages.size(); h++){
    ///////////lazem n3mel condition law hostage mat fe edena
    if (currentNode.currentState.allHostages.get(h).state.equals("Alive")
        ||currentNode.currentState.allHostages.get(h).state.equals("Carried") ||currentNode.currentState.allHostages.get(h).state.equals("Mutated")
                allHostagesSaved = false;

        break;
    }
}
```

```
//System.out.println(c);
int deadHostages = 0;
for (int h = 0; h < currentNode.currentState.allHostages.size(); h++) {
    if (currentNode.currentState.allHostages.get(h).getState().equals("Dead"))
        deadHostages++;

}
if(currentNode.currentState.neoX == telephone.x &&currentNode.currentState.neoY == telephone.y) {

    if(visualization) {
        System.out.println("fneofnwoelfne");
        printSteps(currentNode);
    }
    System.out.println("Solution reached at depth" + currentNode.depth);
    String output = (finalPrint(currentNode) + ";" +deadHostages  + ";" +  currentNode.currentState.kills+ ";" + nodesExpanded);
    System.out.println(output);
    return output;
```

## Main Functions Implemented

The following functions are the most important functions that we use, aside from the main search functions because they will be explained in the next section.

- makeHashedState(Node node, int direction, *Operators* e)

  This method creates and returns a string describing a state so it can be hashed in a hashset. It takes in a node and an integer "direction" describing the direction (1->North, 2-South, 3->East, 4->West) which the method "operationFinder(int direction)" handles. It will also be described in the next fragment. It loops around the array of hostages and saves their states information in a string. Same goes for the array of pills and agents. It also then includes the operation used in the node. Lastly, it includes information about Neo's position and damage. Below is a snippet of the method.

```
public static String makeHashedState(Node node, int direction, Operators e)
{
    String stateToBeHashed ="";

    for(int h=0; h<node.currentState.allHostages.size(); h++){
        stateToBeHashed += node.currentState.allHostages.get(h).getState() +",";
    }

    for(int h=0; h<node.currentState.allPills.size(); h++){
        stateToBeHashed +=  node.currentState.allPills.get(h).getState() +",";
    }
    for(int h=0; h<node.currentState.allAgents.size(); h++){
        stateToBeHashed += node.currentState.allAgents.get(h).getState() +",";
    }

    if(e== null)
        stateToBeHashed += operationFinder(direction) +", ";
    else
        stateToBeHashed += e +", ";
    stateToBeHashed += node.currentState.neoX + "," +node.currentState.neoY+"," ;
    stateToBeHashed += node.currentState.neoDamage;

    return stateToBeHashed;
}
```

- OperationFinder(int i)

  This method takes in an integer that corresponds to one of the following

integers (1->North, 2-South, 3->East, 4->West) and returns an object of type Operators which is an Enum. This method is used to find which direction Neo has moved in according to the counter in the loop that is described above. A snippet of the method is shown below.

```
public static Operators operationFinder(int i) {
    //north, south, east, west
    if(i==0)
        return Operators.up;
    if(i==1)
        return Operators.down;
    if(i==2)
        return Operators.right;
    if(i==3)
        return Operators.left;
    return null;
}
```

- canCarry(ArrayList<Hostage> hostages)

  This method takes in an arraylist of hostages and loops around it, counts the number of carried hostages, and compares that number to the max number of hostages Neo can carry. If the maximum capacity is greater than the number of carried hostages, the method will return true, otherwise it will return false. Below is a snippet of the code of this method.

```
public static boolean canCarry(ArrayList<Hostage> hostages) {
    int carried = 0;
    for(int i=0; i<hostages.size(); i++) {
        if(hostages.get(i).getState().equals("Carried"))
            carried+=1;
    }
    return c > carried;
}
```

- heuristic1(Node node)

  This method is responsible for creating the first heuristic used in our searches. It simply takes a node, loops around all of its hostages, counts the number of all of the dead ones (both the mutated ones and the ones who died while being carried), adds it to the damage of Neo, and returns that value.

```
public static int heuristic1(Node node) {
    //Taking into account Neos Damage and Hostage deaths
    int dead = 0;
    for(int i=0; i<node.currentState.allHostages.size(); i++) {
        if(node.currentState.allHostages.get(i).getState().equals("Dead") ||
                node.currentState.allHostages.get(i).getState().equals("Mutated")) {
            dead++;
        }
    }
    return node.currentState.neoDamage + dead;
}
```

- heuristic2(Node node)

  This method is responsible for creating the second heuristic used in our searches. It simply takes a node, loops around all of its hostages, counts the number of all dead ones (both the mutated ones and the ones who died while being carried), adds it to the number of kills Neo got, and returns that value.
  Note that more information about the heuristics will be provided in the next section.

```
public static int heuristic2(Node node) {
    //Count of hostage deaths and agent kills
    int dead = 0;
    for(int i=0; i<node.currentState.allHostages.size(); i++) {
        if(node.currentState.allHostages.get(i).getState().equals("Dead") ||
                node.currentState.allHostages.get(i).getState().equals("Mutated")) {
            dead++;
        }
    }
    return node.currentState.kills + dead;
}
```

## Search Problem Implementation

NOTE: any search uses the same search algorithm with some changes tailored to every search algorithm, they use the search algorithm explained in **Matrix Problem Implementation.**

- **Breadth First**

  For breadth first search, we loop considering each and every situation possible and pick the first one that meets the goal test; which is killing all mutated hostages and saving all alive agents.

  To do that we created a queue and enqueued the very first node to it which is the Node in which Neo starts. We then loop around all the possible directions for Neo to move in {(x+1,y),(x,y+1),(x-1,y),(x,y-1)} and enqueue each and every case to the queue while making sure we set a parent to each new node created so we can keep track of all the parents of the node and therefore find the final route of Neo. The Breadth First Search works in a First-In-First-Out manner. It must be noted that since we create a new node for every iteration, we must deep clone each and every attribute that is to be changed to that new node then make whatever changes required.

  It must also be noted that we have created a hashset to store in the covered situations including all the attributes of the node.

  As illustrated in Figure 1, Neo expands his movement to all possible directions with each path becoming a new branch extended from its parent. Each new step taken will expand the tree and will be added to the FIFO queue.

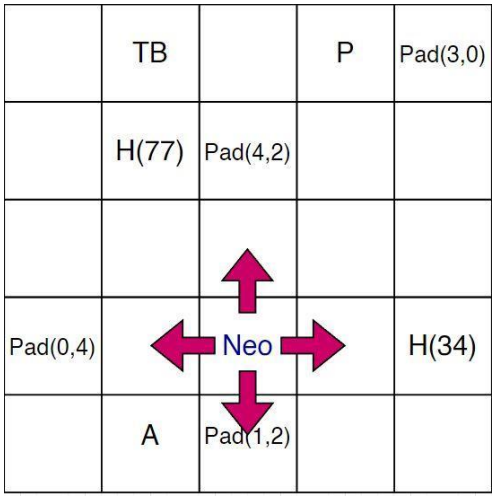



*Figure 1*

- **Depth First**

  Depth First Search works in the same manner as Breadth First Search, but instead of having a FIFO queue, it works using a Last-in-First-out queue. This was implemented by using a stack.

- **Iterative Deepening**

  We implemented the iterative deepening search by creating a variable "level" that starts at 10 and a stack instead of a queue, just like we did in Depth First Search. And with every newly created node (step by Neo) we set its depth to x where x = y + 1 where y is the depth of its parent's depth. But we only add the node to the stack if and only if its depth is less than or equal to the variable "level" to make sure Neo does not explore nodes that are out of the depth limit.

- **Uniform Cost**

  To implement the Uniform Cost Search, we created a priority queue of nodes that takes a comparator that sorts the queue by the lowest cost of the nodes, as shown in figure 2. The costs are the following

```
PriorityQueue<Node> mainQueue = new PriorityQueue<Node>( new Comparator<Node>() {

    @Override
    public int compare(Node o1, Node o2) {
        if(o1.cost > o2.cost) {
            return 1;
        } else if (o1.cost < o2.cost) {
            return -1;
        } else {
            return 0;
        }
    }
});
```

*Figure 2*

**For any child node that we add that are PAD, CARRY, DROP, EMPTY CELL OR ANY MOVE ACTION (If we are on that cell or moving to it), the following costs are implemented**

- currentNode.cost+3+(3000*hostagesThatDied));

- we add the cost of that action + the cost of each hostage that died as a result of incurring the damage

**For any child node that we add that is TAKING A PILL, the following costs are implemented:**

- currentNode.cost+3
- here hostages do not die so they are not taken into account

**For any child node that we add that is killing an agent or killing a hostage in the neighboring, the following costs are implemented:**

- currentNode.cost+(3000 * agentKills)+(3*hostageKills) +(3000*hostagesThatDied));
- This calculates the agents that are killed in all neighboring cells as a result of executing kill action
- Added onto the hostages that are Mutated (Since this is a goal action its given a cost of 3)
- Added onto the Hostages that died as a result of incurring damage by doing that action

- **Greedy**

To implement the greedy search, we took into account the heuristic only with no cost taken in consideration. We created a priority queue that sorts the nodes by one of the two heuristics we created.

- **Heuristic 1**
  For this heuristic we chose to take into account the number of dead hostages, which includes both mutated ones and the ones who just died in Neo's arms. We add that number to Neo's damage. Therefore it is clear that the nodes where Neo's damage is high as well as contains several dead hostages will be placed last in the priority queue. Below, in Figure 3, is a snippet of the heuristic as well as a snippet, in Figure 4, of how the comparator sorts the items in the priority queue using our heuristic1 method.

```java
public static int heuristic1(Node node) {
    //Taking into account Neos Damage and Hostage deaths
    int dead = 0;
    for(int i=0; i<node.currentState.allHostages.size(); i++) {
        if(node.currentState.allHostages.get(i).getState().equals("Dead") ||
                node.currentState.allHostages.get(i).getState().equals("Mutated")) {
            dead++;
        }
    }
    return node.currentState.neoDamage + dead;
}
```

*Figure 3*

```java
//  heuristic1
  PriorityQueue<Node> mainQueue = new PriorityQueue<Node>(10000, new Comparator<Node>() {
      public int compare (Node n1, Node n2) {

          if(heuristic1(n1) > heuristic1(n2)) {
              // the greater the number returned from our heuristic method, the further the
              // node will be placed in the priority queue
              return 1;
          } else if ( heuristic1(n1) < heuristic1(n2)) {
              return -1;
          } else {
              return 0;
          }
      }
  });
```

*Figure 4*

- **Heuristic 2**
  For this heuristic we chose to take the number of killed hostages, the number of dead hostages, and the number of Neo's kills into consideration. Since the most optimal solution is to have the least number of casualties in the map, we place the nodes that have a high number of kills, dead hostages, and mutated hostages last in the queue. Below, in Figure 5, is a snippet of the heuristic as well as a snippet, in Figure 6, of how the comparator sorts the items in the priority queue using our heuristic2 method.

```java
public static int heuristic2(Node node) {
    //Count of hostage deaths and agent kills
    int dead = 0;
    for(int i=0; i<node.currentState.allHostages.size(); i++) {
        if(node.currentState.allHostages.get(i).getState().equals("Dead") ||
                node.currentState.allHostages.get(i).getState().equals("Mutated")) {
            dead++;
        }
    }
    return node.currentState.kills + dead;
}
```

*Figure 5*

```
//  heuristic2
  PriorityQueue<Node> mainQueue = new PriorityQueue<Node>(10000, new Comparator<Node>() {
      public int compare (Node n1, Node n2) {
          // the greater the number returned from our heuristic method, the further the
          // node will be placed in the priority queue
          if(heuristic2(n1) > heuristic2(n2)) {
              return 1;
          } else if ( heuristic2(n1) < heuristic2(n2)) {
              return -1;
          } else {
              return 0;
          }
      }
  });
```

*Figure 6*

- **A***

  To implement the A* search, we had to take both the heuristic and the cost of each node in consideration. We created a priority queue of nodes that sorts its items by the value returned from the heuristic function **added** to the cumulative cost of the node. Note that the costs here have the same value as the costs mentioned in the Uniform Cost Search Section. Note that the higher the cost AND the heuristic function value, the further in the queue the node is placed. Which logically makes sense because what we are trying to achieve is reaching the goal with the least cost possible as well as the least number of casualties and damage (in case of heuristic 1) or kills (in case of heuristic 2).

  - **Heuristic 1**
    A*1 utilizes the same heuristic function discussed above in the Greedy Search section. Below, in Figure 7, is a snippet of the comparator of the priority queue created for A*1.

```
PriorityQueue<Node> mainQueue = new PriorityQueue<Node>(1000, new Comparator<Node>() {
    public int compare (Node n1, Node n2) {

        if((heuristic1(n1)+n1.cost) > (heuristic1(n2)+n2.cost)) {
            // The result of the heuristic function is added to the cumulative
            // cost of the node
            return 1;
        } else if ((heuristic1(n1)+n1.cost) < (heuristic1(n2)+n2.cost)) {
            return -1;
        } else {
            return 0;
        }
    }
});
```

*Figure 7*

- **Heuristic 2**
  A*2 utilizes the same heuristic function discussed above in the Greedy Search section. Below, in Figure 8, is a snippet of the comparator of the priority queue created for A*2.

```
PriorityQueue<Node> mainQueue = new PriorityQueue<Node>(1000, new Comparator<Node>() {
    public int compare (Node n1, Node n2) {
        // The result of the heuristic function is added to the cumulative
        // cost of the node
        if((heuristic2(n1)+n1.cost) > (heuristic2(n2)+n2.cost)) {
            return 1;
        } else if ((heuristic2(n1)+n1.cost) < (heuristic2(n2)+n2.cost)) {
            return -1;
        } else {
            return 0;
        }
    }
});
```

*Figure 8*

## Heuristic Functions

The heuristic function "heuristic1" explained above lets the node that contains the least number of casualties and the least number of damage dealt to Neo be a #1 priority in the queue. Which makes sense because it corresponds to the goal test since it basically is to save all the hostages and kill all the mutated ones. The same goes for heuristic2 except that it puts the nodes with the most number of kills and most number of casualties in the end of the queue which makes such nodes the least optimal.

Concerning the A* heuristic2, let's say we have a grid that Neo needs to traverse. With every step, he picks the step with the least cost as well as the least number of kills it will get him as well as the least number of dead and mutated agents. Keeping in mind that the cost also gets higher the more undesirable a move is (killing an agent and having a hostage die). Therefore Neo will be forced to take the moves with the lowest costs (carrying a hostage, taking a pill, and dropping hostages).

Our heuristics are admissible also because they get the same or a better result than Uniform Cost search does. Below is proof of this statement.

---

**Uniform Cost search solution for grid**
"5,5;2;0,4;1,4;0,1,1,1,2,1,3,1,3,3,3,4;1,0,2,4;0,3,4,3,4,3,0,3;0,0,30,3,0,80,4,4,80"

**H(30) A - Pad(4,3) Neo**

**P A - - TB**

**- A - - P**

**H(80) A - A A**

**- - - Pad(0,3) H(80)**

**Solution reached at depth37**

**down,down,takePill,left,left,down,down,left,left,up,carry,up,up,takePill,down,down,down,right,right,right,right,carry,left,fly,down,right,drop,left,up,fly,left,left,left,up,up,up,up,carry,down,down,down,down,right,right,right,fly,down,right,drop;0;0;1129**

---

**A*1 search solution for same grid**

**H(30) A - Pad(4,3) Neo**

**P A - - TB**

**- A - - P**

**H(80) A - A A**

**- - - Pad(0,3) H(80)**

**Solution reached at depth37**

**down,down,takePill,left,left,down,down,left,left,up,carry,up,up,takePill,down,down,down,right,right,right,right,carry,left,fly,down,right,drop,left,up,fly,left,left,left,up,up,up,up,carry,down,down,down,down,right,right,right,fly,down,right,drop;0;0;1083**

---

**A*2 search solution for same grid**

**H(30) A - Pad(4,3) Neo**

**P A - - TB**

**- A - - P**

**H(80) A - A A**

**- - - Pad(0,3) H(80)**

**Solution reached at depth37**

**down,down,takePill,left,left,down,down,left,left,up,carry,up,up,takePill,down,down,down,right,right,right,right,carry,left,fly,down,right,drop,left,up,fly,left,left,left,up,up,up,up,carry,down,down,down,down,right,right,right,fly,down,right,drop;0;0;1129**

---

The results above show that Neo reached his goal in 1129 traverses by Uniform Cost search. Similarly, he reached the goal as well in the same number of traverses with A* search using our second heuristic. Finally, he reached his goal in only 1083 traverses by A* with our first heuristic. This proves the admissibility of our heuristics since they are similar to, if not more optimal than, Uniform Cost search which is optimal.

## Performance Comparison

Testing on grid #1:
"5,5;2;0,4;1,4;0,1,1,1,2,1,3,1,3,3,3,4;1,0,2,4;0,3,4,3,4,3,0,3;0,0,30,3,0,80,4,4,80"

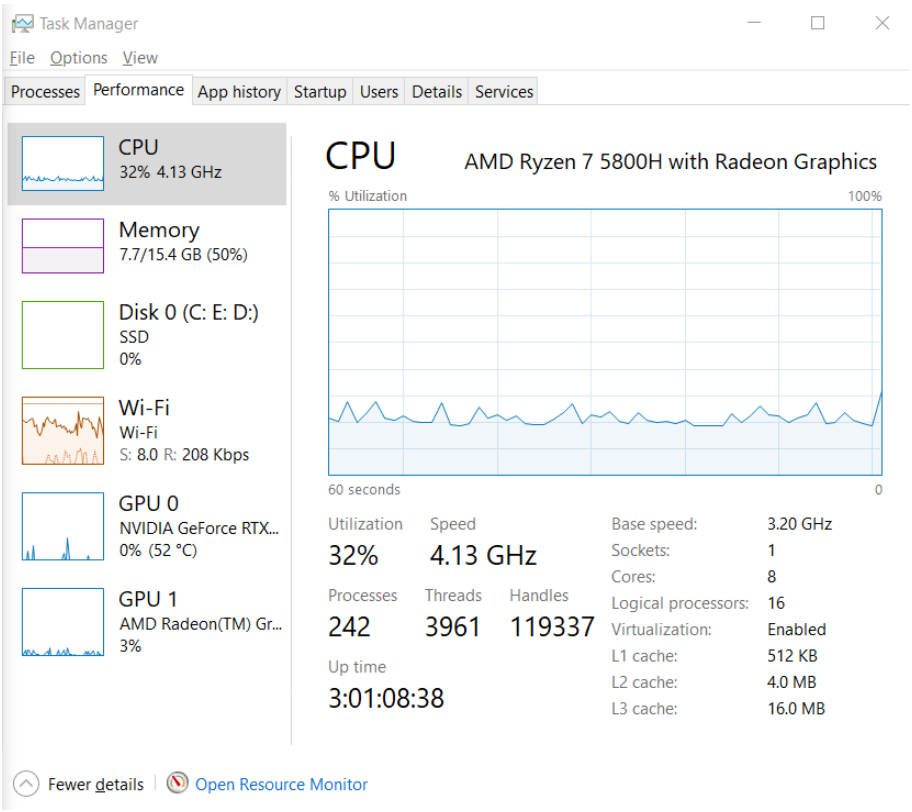
| Performance Comprison | | | |
|---|---|---|---|
| Search Type | Completeness | Optimality | No. of expanded nodes |
| Breadth First | Complete | Not Optimal | 219136 |
| Depth First | not Complete - But can be considered complete since we hash the repeated states, and so this prevents infinite paths since an infinite path has repeated states, an example is moving back and forth as a path in DFS | Not Optimal | 1782 |
| Iterative Deepening | Complete | Not Optimal | 1782 |
| Uniform Cost | Complete | Optimal | 1129 |
| Greedy 1 | Complete | Not Optimal | 584 |
| Greedy 2 | Complete | Not Optimal | 807 |
| A* 1 | Complete | Optimal | 1083 |
| A* 2 | Complete | Optimal | 1129 |

## RAM Usage Comparison

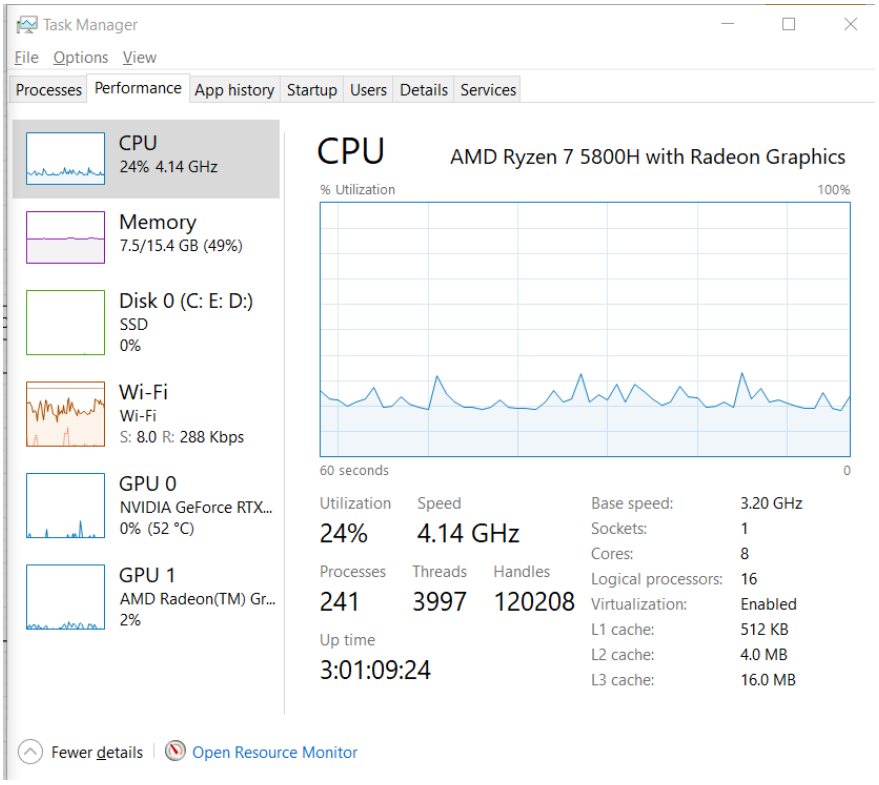
We have noticed that the RAM usage remains unchanged for all runs

# CPU Utilization Comparison

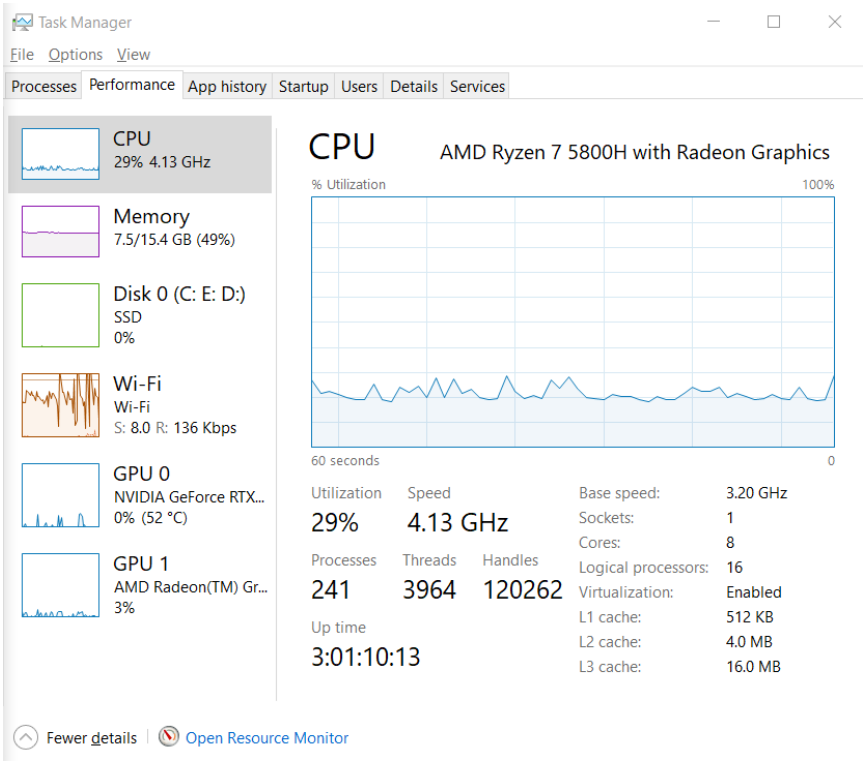
## ☐ Breadth First



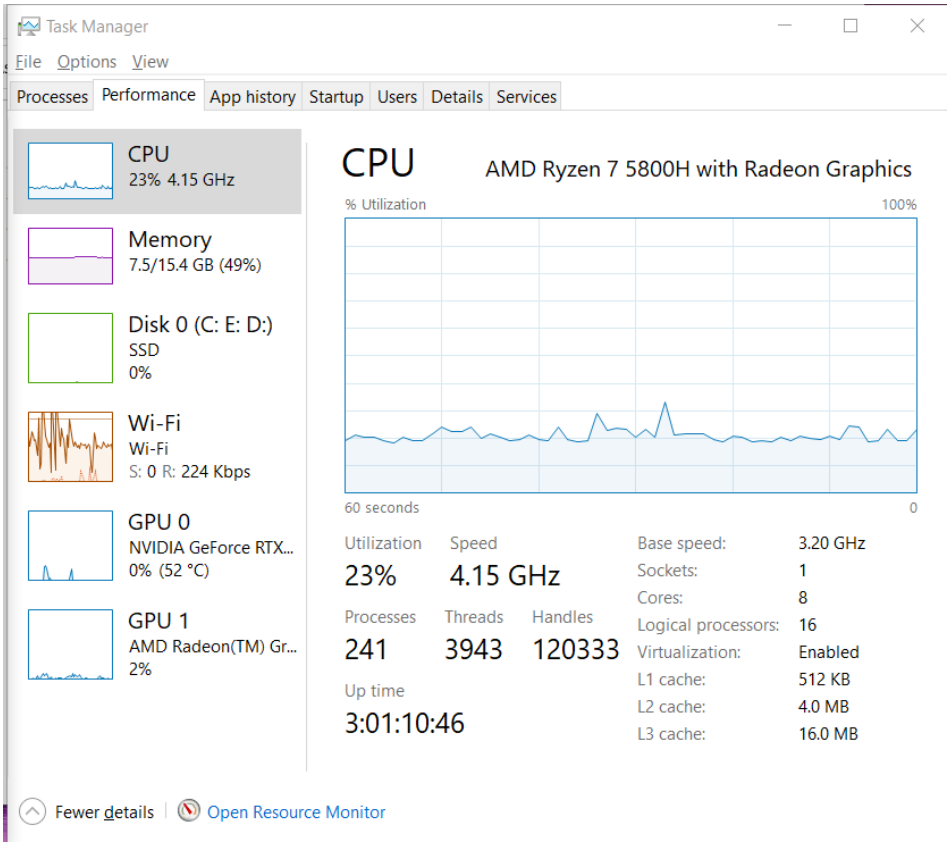

## ☐ Depth First



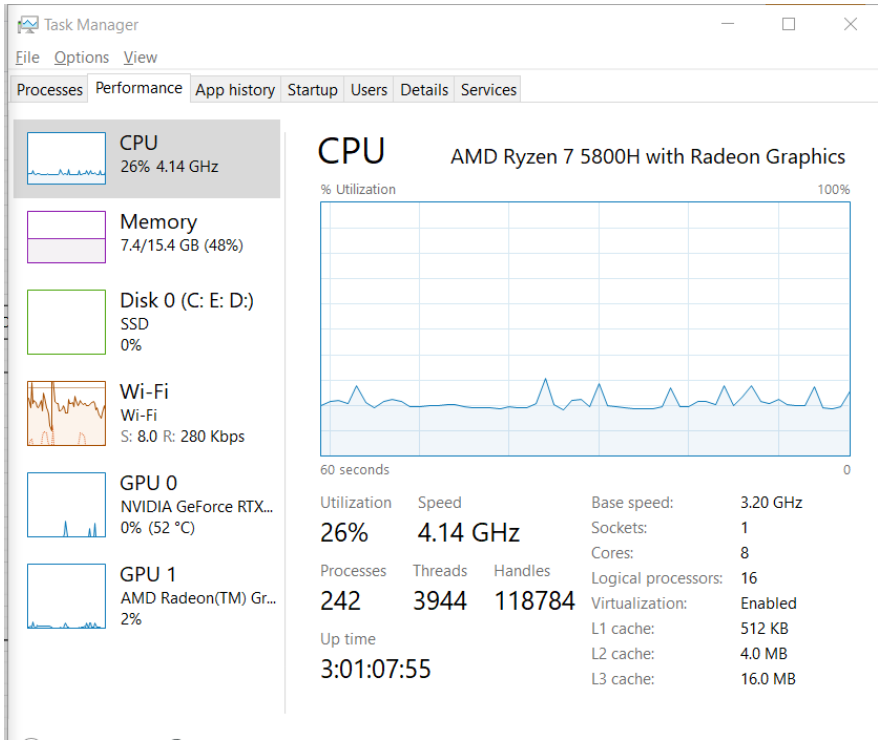
☐ Iterative Deepening



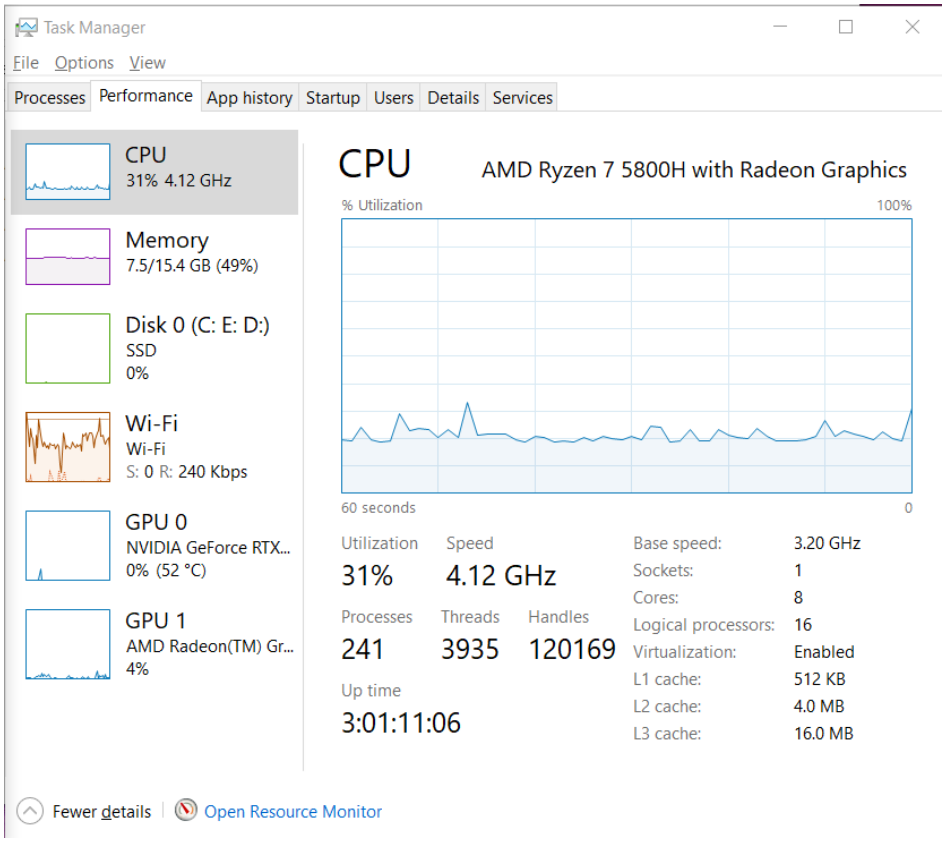

☐ Uniform Cost



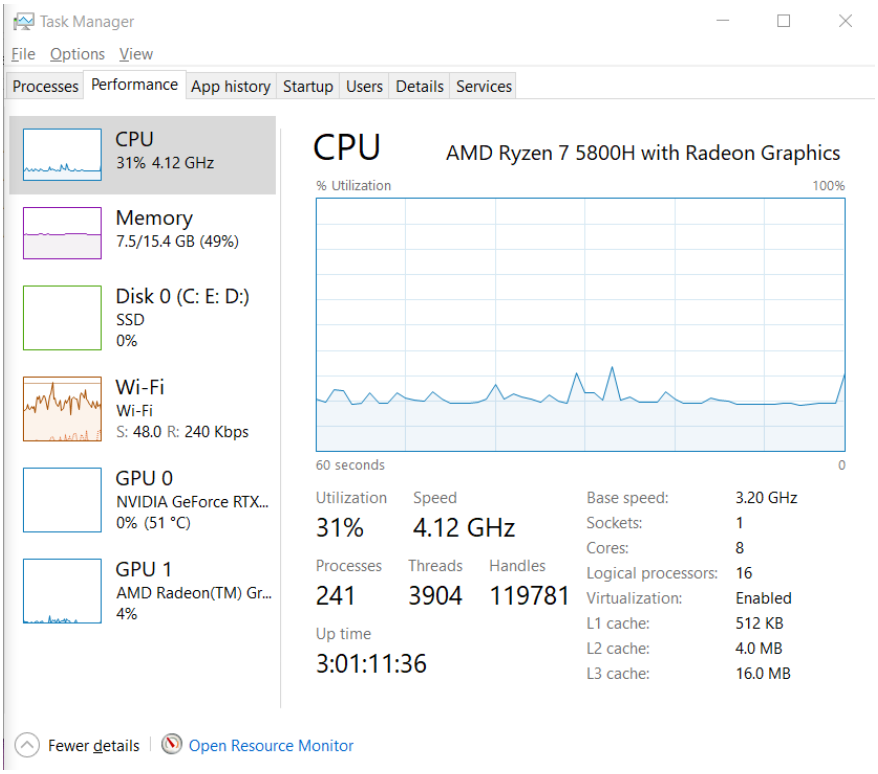
## ☐ Greedy 1



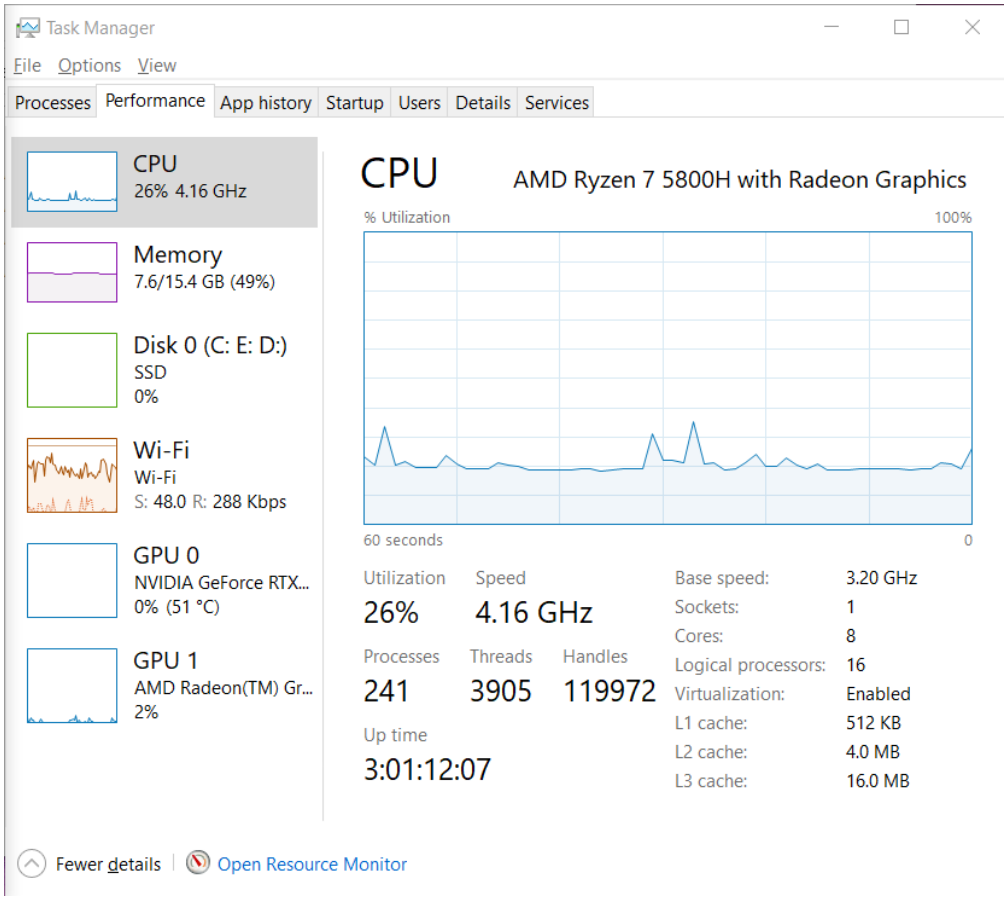

## ☐ Greedy 2



☐ A* 1

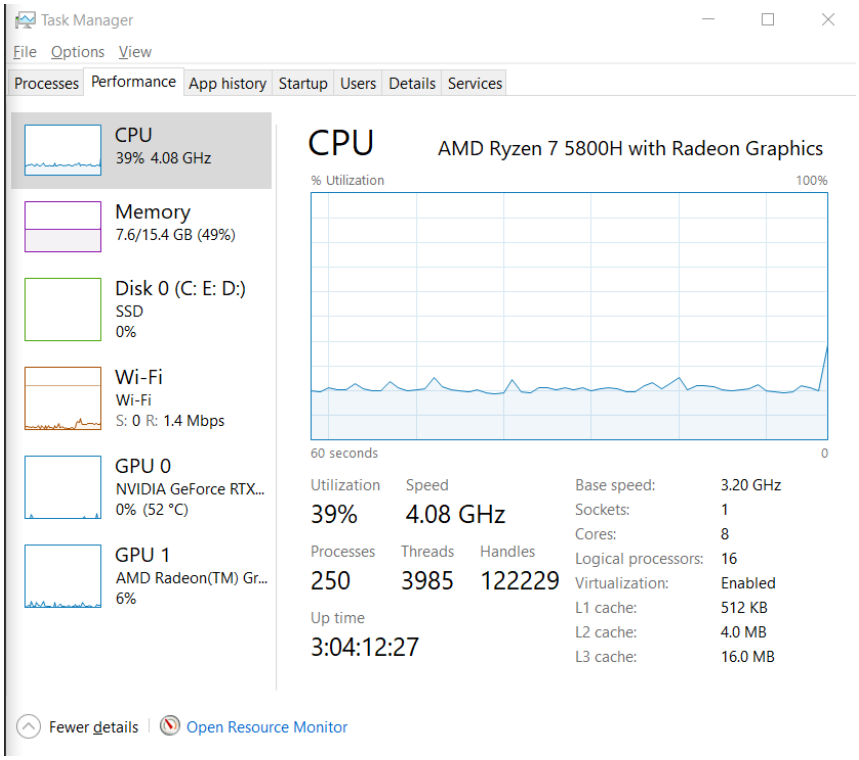

Task Manager
File Options View
Processes Performance App history Startup Users Details Services
CPU
31% 4.12 GHz
Memory
7.5/15.4 GB (49%)
Disk 0 (C: E: D:)
SSD
0%
Wi-Fi
Wi-Fi
S: 48.0 R: 240 Kbps
GPU 0
NVIDIA GeForce RTX...
0% (51 °C)
GPU 1
AMD Radeon(TM) Gr...
4%
CPU
AMD Ryzen 7 5800H with Radeon Graphics
% Utilization
100%
60 seconds
0
Utilization
31%
Speed
4.12 GHz
Processes
241
Threads
3904
Handles
119781
Up time
3:01:11:36
Base speed: 3.20 GHz
Sockets: 1
Cores: 8
Logical processors: 16
Virtualization: Enabled
L1 cache: 512 KB
L2 cache: 4.0 MB
L3 cache: 16.0 MB
Fewer details
Open Resource Monitor

☐ A* 2

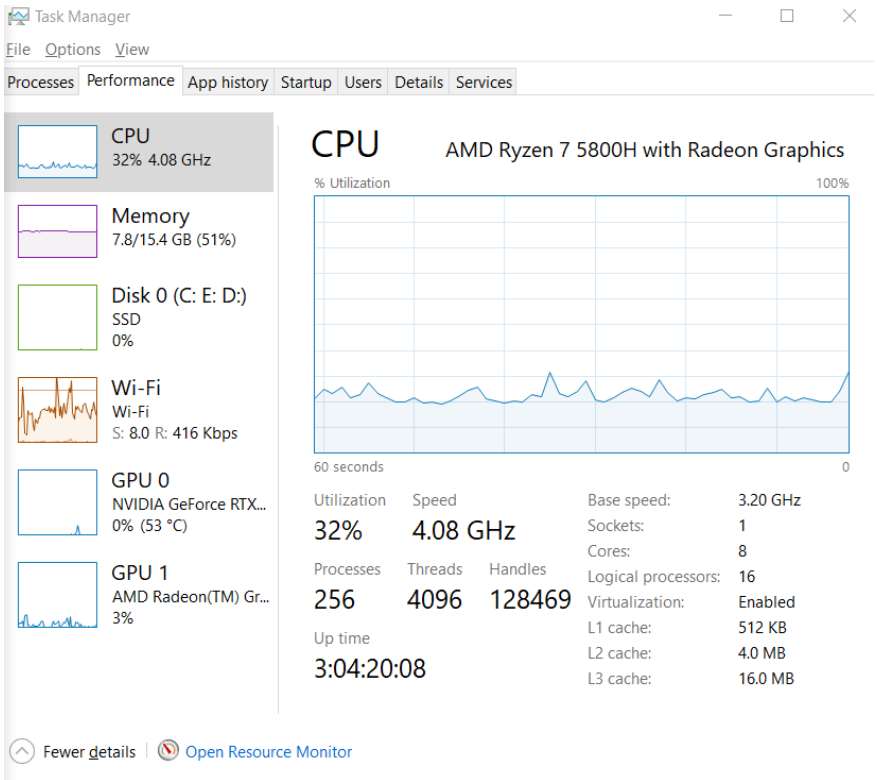

Task Manager
File Options View
Processes Performance App history Startup Users Details Services
CPU
26% 4.16 GHz
Memory
7.6/15.4 GB (49%)
Disk 0 (C: E: D:)
SSD
0%
Wi-Fi
Wi-Fi
S: 48.0 R: 288 Kbps
GPU 0
NVIDIA GeForce RTX...
0% (51 °C)
GPU 1
AMD Radeon(TM) Gr...
2%
CPU
AMD Ryzen 7 5800H with Radeon Graphics
% Utilization
100%
60 seconds
0
Utilization
26%
Speed
4.16 GHz
Processes
241
Threads
3905
Handles
119972
Up time
3:01:12:07
Base speed: 3.20 GHz
Sockets: 1
Cores: 8
Logical processors: 16
Virtualization: Enabled
L1 cache: 512 KB
L2 cache: 4.0 MB
L3 cache: 16.0 MB
Fewer details
Open Resource Monitor

Testing on grid #2:
"5,5;4;1,1;4,1;2,4,0,4,3,2,3,0,4,2,0,1,1,3,2,1;4,0,4,4,1,0;2,0,0,2,0,2,2,0;0,0,62,4,3,45,3,3,39,2,3,40"

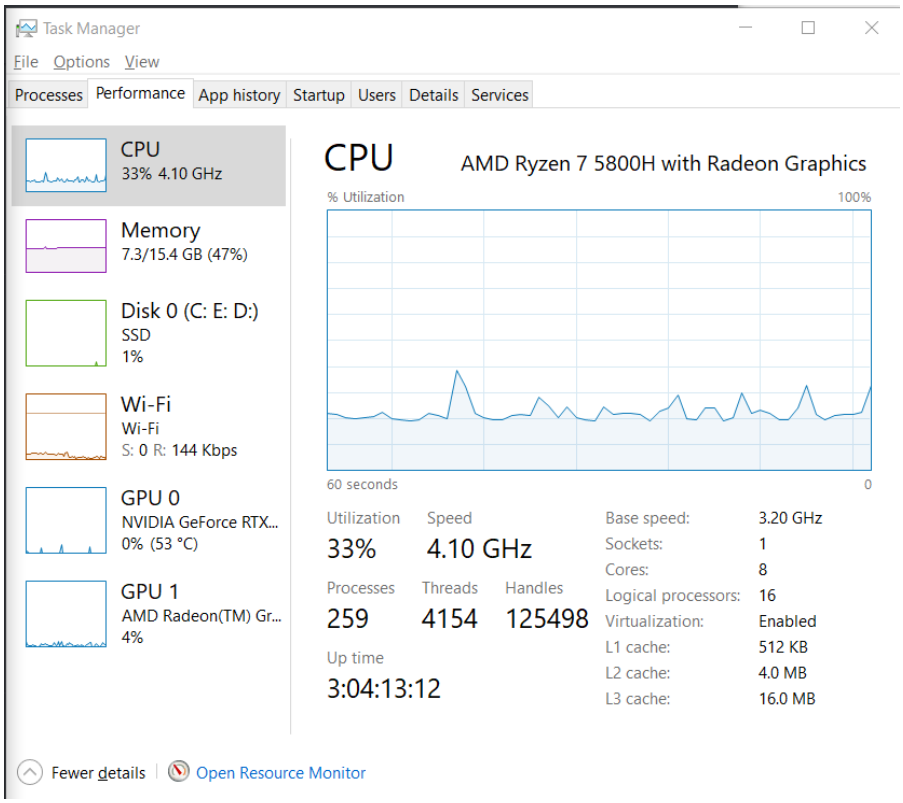
☐ Breadth First



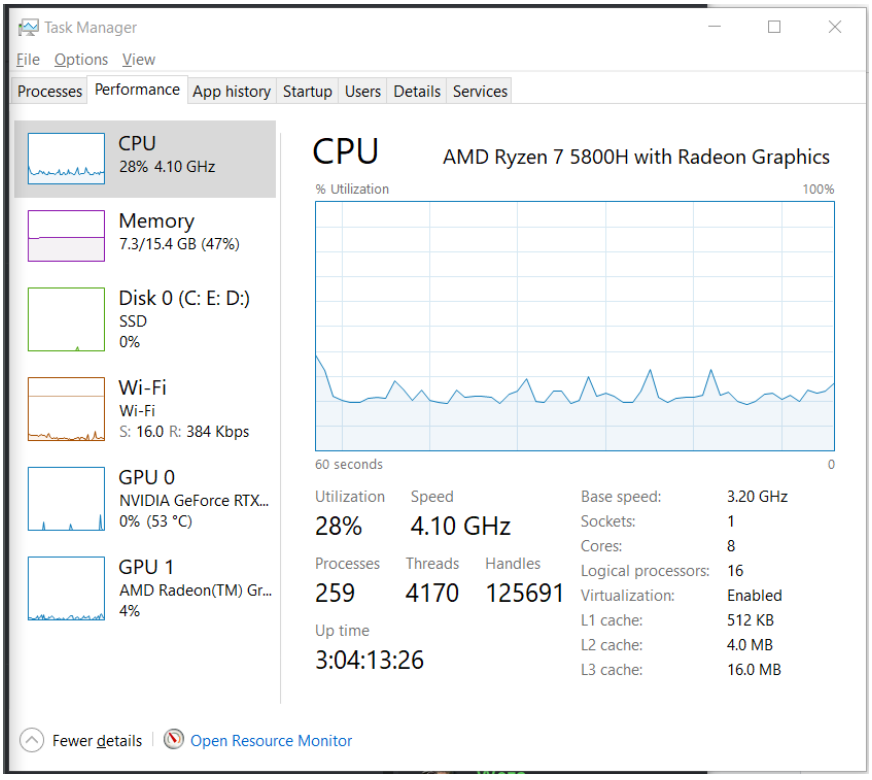

☐ Depth First



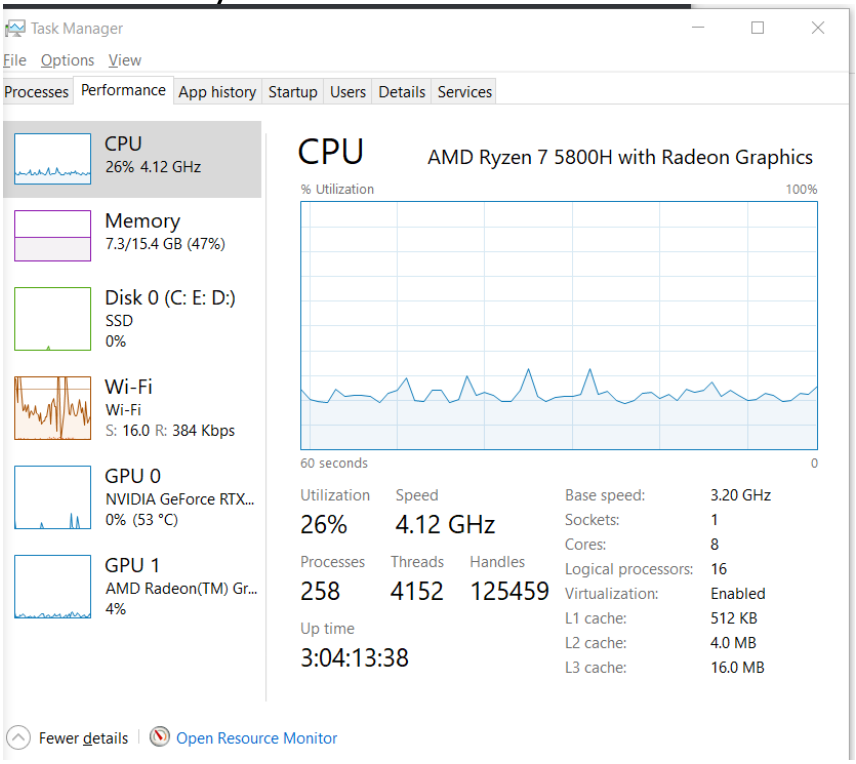
☐ Iterative Deepening



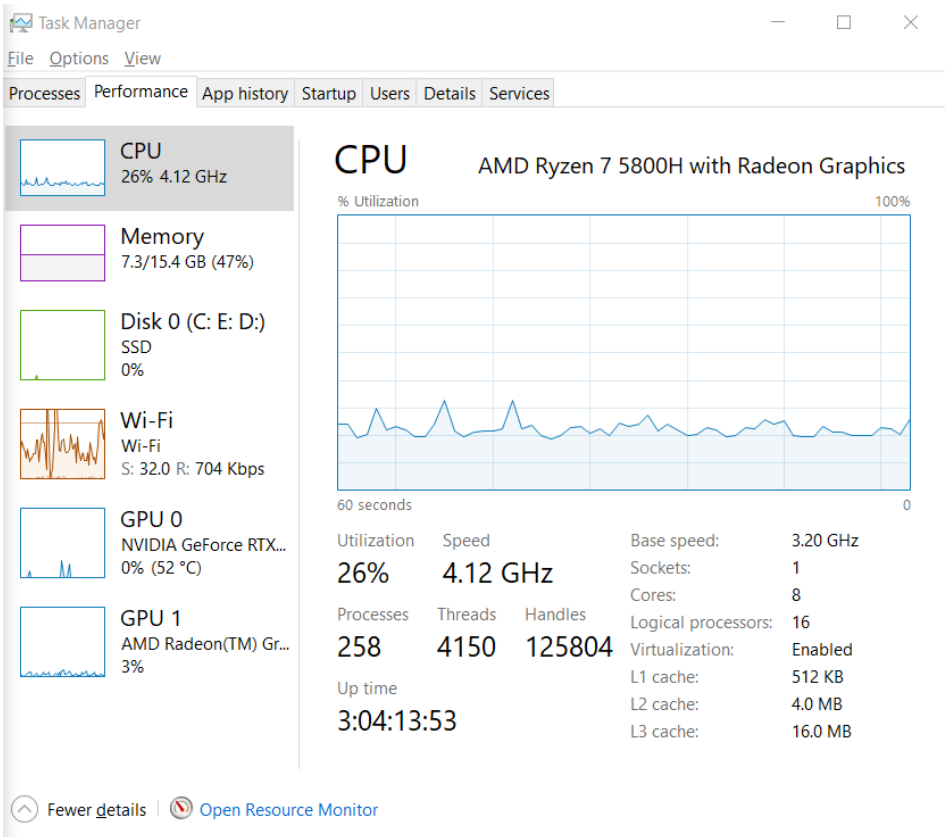

☐ Uniform Cost



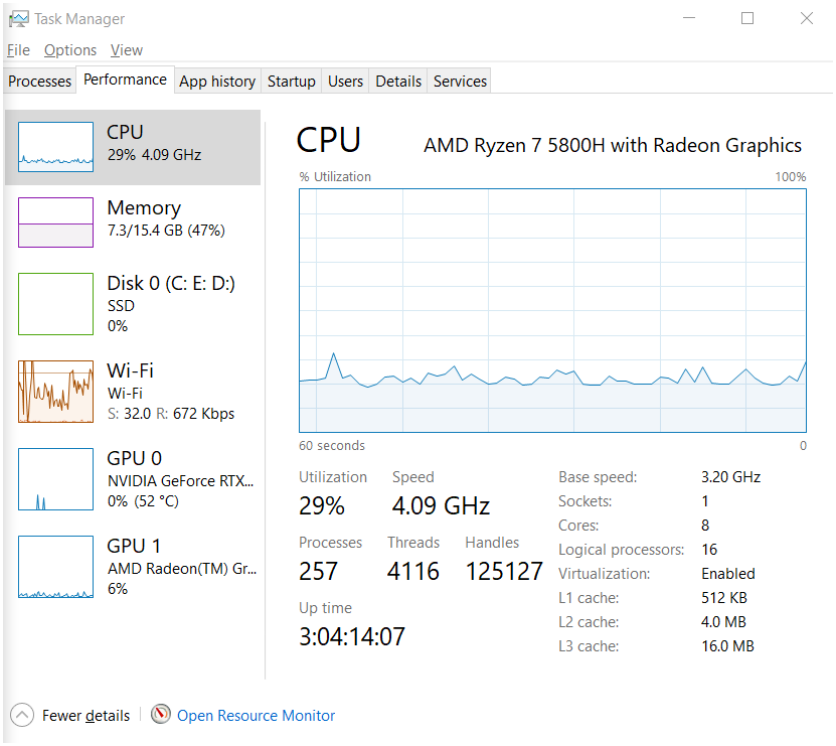
## ☐ Greedy 1



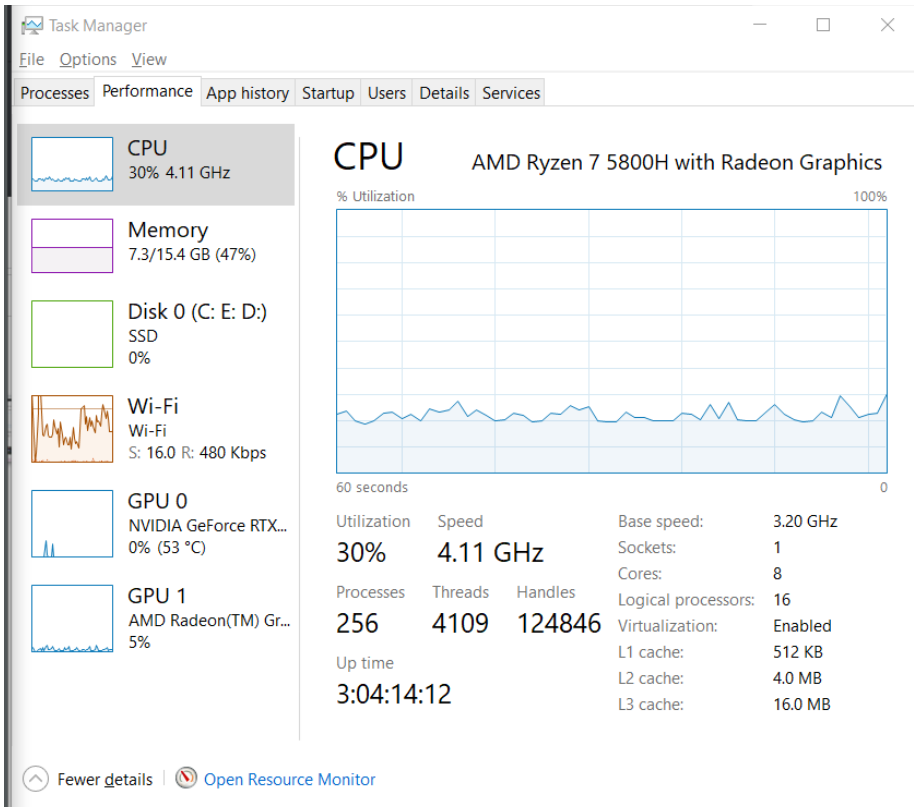

## ☐ Greedy 2



☐ A* 1


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
CPU
29% 4.09 GHz
Memory
7.3/15.4 GB (47%)
Disk 0 (C: E: D:)
SSD
0%
Wi-Fi
Wi-Fi
S: 32.0 R: 672 Kbps
GPU 0
NVIDIA GeForce RTX...
0% (52 °C)
GPU 1
AMD Radeon(TM) Gr...
6%
CPU
AMD Ryzen 7 5800H with Radeon Graphics
% Utilization
100%
60 seconds
0
Utilization
29%
Speed
4.09 GHz
Processes
257
Threads
4116
Handles
125127
Up time
3:04:14:07
Base speed: 3.20 GHz
Sockets: 1
Cores: 8
Logical processors: 16
Virtualization: Enabled
L1 cache: 512 KB
L2 cache: 4.0 MB
L3 cache: 16.0 MB
Fewer details
Open Resource Monitor

☐ A* 2


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
CPU
30% 4.11 GHz
Memory
7.3/15.4 GB (47%)
Disk 0 (C: E: D:)
SSD
0%
Wi-Fi
Wi-Fi
S: 16.0 R: 480 Kbps
GPU 0
NVIDIA GeForce RTX...
0% (53 °C)
GPU 1
AMD Radeon(TM) Gr...
5%
CPU
AMD Ryzen 7 5800H with Radeon Graphics
% Utilization
100%
60 seconds
0
Utilization
30%
Speed
4.11 GHz
Processes
256
Threads
4109
Handles
124846
Up time
3:04:14:12
Base speed: 3.20 GHz
Sockets: 1
Cores: 8
Logical processors: 16
Virtualization: Enabled
L1 cache: 512 KB
L2 cache: 4.0 MB
L3 cache: 16.0 MB
Fewer details
Open Resource Monitor

**Our Comment on the collected data:**

We have noticed that the greater the number of nodes expanded, the less optimal the search is, and therefore the greater CPU utilization used.

**Notes:**

- ➔ We tested our program using the public test files provided in the cms. Although they all ran flawlessly, test 6 for both Uniform Cost search and A* did not run like expected and gave us unexpected results, while all other tests produced the intended output.
- ➔ Our genGrid() method does not create a random grid but instead converts the input grid string to a grid.

## Citations:


GeeksForGeeks 18 Oct, 2021
<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

GeeksForGeeks 20 Aug, 2021

<https://www.geeksforgeeks.org/search-algorithms-in-ai/>