AI Project Report

- **A discussion of your implementation of the search-tree node ADT:**

We shall consider that the search-tree nodes to be 7-tuples

1. The State is a object that will defines the current state of the problem.
2. The parent node
3. Operator applied to generate this node
4. Depth of node from search tree
5. Path cost from this node to successor node in term of Operator
6. Heuristic function
7. Total Cost from initial state to this node

- **A discussion of your implementation of the search problem ADT:**

The generic Search Problem has abstract methods which are:

1. getInitialState: returns the initial node from search-tree
2. goalTest: takes a node as an input and check if it is a goal state or not and returns a Boolean value
3. getAllowedOperators: Takes a node as an input and returns a list of the allowed operators on this node
4. getNextState: Takes a Node as an input and the Operator and returns the new node if it's not repeated state.

- **A discussion of your implementation of the Olympics problem:**

The **Olympics** class is a subclass of generic Search Problem class and implemented all of the methods discussed in the Search Problem.

The Olympic State is consisted into 2- tuples:

1. Location in x & y
2. List of collected components

The initial State of Olympic problem will contains the location of Jarvis extracted from string input and an empty list of collected component.

Olympic problem has 6 operators which are:

1. UP
2. DOWN
3. RIGHT
4. LEFT
5. PICK
6. LIGHT

UP,DOWN,LEFT,RIGHT operators can only be executed if Jarvis will not exceed the grid's dimensions. Each of these operators costs 10 points to reach from the current node to next node.

PICK operator can be executed if Jarvis is at the cell where a component is existed in. This operators costs 1 points to reach from the current node to next node.

LIGHT operator can be executed if Jarvis is at the cell where a flame is existed in & if the state defines that Jarvis is in this cell & collect all components in the grid. This operators costs 1 points to reach from the current node to next node.

After calling **getAllowedOperators** on a Node, a list of valid Operators will be returned. **getNextState** is then called on the node and one of the Operators returned, check if this node is repeated or not, and assign the total cost and path cost & heuristic value of the successor node.

- **A description of the main functions you implemented:**
    1. **The genGrid method** generates a random grid with dimensions ranging from 5 to 15 in both rows and columns (m x n). The number of components is set randomly between 5 and 10. The Poisonus locations is set randomly between 2 and 25. We take care of overlapping so that no component will be same as poisonous, Jarvis or flame in location.
    2. **The visualize method** generates the path between the initial state to goal by printing all states from all nodes. To implement that, we have to take the returned node from solve method and get the state from it and after that we get its parent node and loop that until we reach to the initial state.
    3. **Solve method** generate the goal node that been reach by putting the input string and decode it & get the search strategy and solve it in Searchproblemprocedure class and finally get the goal node and print the plan that represent operators to show us how Jarvis collects all components and get to the flame location to reach the goal.

- **A discussion of how you implemented the various search algorithms:**

The Searchproblemprocedure method is a method where generic search algorithm is implemented & consist to 7 types of search algorithm.

1. **Breadth First (BF):** The breadth first search is implemented using a queue. First step is to push the initial state. After that we'll check if the queue is not empty and if it's empty it will return nothing. Then we remove the first node and see if it's goal state to be returned and terminate the search algorithm. If the first node in the queue isn't the goal one.The removed node will be an input in getAllowedOperators and get the list of operators to start expanding the node. Finally we take each of operators from list and see if there's a non-repeated successor from this operator to be pushed in queue or not using getNextNode method and repeat that until we reach to final operator. BF doesn't sort nodes in term of totalcost or heuristics.
2. **Depth First (DF):** The breadth first search is implemented using a stack. First step is to push the initial state. After that we'll check if the stack is not empty and if it's empty it will return nothing. Then we remove the first node and see if it's goal state to be returned and terminate the search algorithm. If the first node in the stack isn't the goal one. The removed node will be an input in getAllowedOperators and get the list of operators to start expanding the node. Finally we take each of operators from list and see if there's a non-repeated successor from this operator to be pushed in stack or not using getNextNode method and repeat that until we reach to final operator.
3. **Uniform Cost (UC):** The breadth first search is implemented using a priorityqueue. Priority queue will sort nodes ascendingly in term of their Total Cost for every node is pushed in. First step is to push the initial state. After that we'll check if the priorityqueue is not empty and if it's empty it will

return nothing. Then we remove the first node and see if it’s goal state to be returned and terminate the search algorithm. If the first node in the priorityqueue isn’t the goal one.The removed node will be an input in getAllowedOperators and get the list of operators to start expanding the node. Finally we take each of operators from list and see if there’s a non-repeated successor from this operator to be pushed in priorityqueue or not using getNextNode method and repeat that until we reach to final operator. BF doesn’t sort nodes in term of totalcost or heuristics.

4. **Greedy search** works exactly the same as the Uniform Cost implementation but priority queue will sort nodes ascendingly in term of their Heuristics for every node is pushed in.
5. **A* search** works exactly the same as the Uniform Cost implementation but priority queue will sort nodes ascendingly in term of their (Heuristics + Total Cost) for every node is pushed in.

- **A discussion of the heuristic functions you employed and, in the case of A*, an argument for their admissibility:**

  We have two types of Greedy Search (GRi) & two types of A* search (ASi). The two types differ in which technique of their heuristic function did use.
  We have two used techniques of heuristic function which are Manhattan: the one which measures the distance between Jarvis and first nearest component vertically and horizontally and it’s used by GR1 & AS1. And Euclidian: the one which measures the distance between Jarvis and first nearest component diagonally and used by GR2 & AS2. How these two techniques are admissible, basically they measure the shortest path that Jarvis can collect a one component which is the nearest to Jarvis. This will minimize the heuristic cost and make it smaller than or equal to the total cost and not make it overestimate.

- **At least two running examples from your implementation:**

Using String input: `“5,5;1,2;3,2;0,3,2,1,3,4,4,0,4,3;0,1,0,2,3,1,3,3,4,2”`

1. BF:
```
right,up,pick,right,down,down,down,down,left,pick,right,up,pick,up,left,left,left,left,down
,down,pick,up,up,right,pick,right,down,light;2740
```

2. DF:
```
down,right,right,down,pick,down,left,pick,right,up,up,up,up,left,pick,down,down,left,left,
pick,left,down,down,pick,up,up,right,right,down,light;130
```

- **A comparison of the performance of the implemented search strategies on your running examples in terms of completeness, optimality, RAM usage, CPU utilization, and the number of expanded nodes:**

  Lets take a certain Input string to test the performance of each search algorithm:
  Input:
  ”`15,15;5,1;14,1;3,14,11,11,12,12,4,8,8,14,5,0,13,6,0,6,5,11,11,0,7,0,12,11,6,2;6,1,14,10,6,5,6,11,4,2,9,1,2,12,2,6,1,7,2,7,2,3,2,5,8,11,4,3,2,0,5,10`”

BF:

- Output:
```
left,pick,down,down,down,down,down,down,pick,up,up,up,up,pick,right,right,up,pick,up,right,
right,up,up,up,up,up,right,right,pick,right,right,down,down,down,down,pick,right,right,righ
t,down,pick,up,up,right,right,right,pick,down,down,down,down,down,pick,left,left,down,left,
down,down,pick,right,down,pick,left,pick,left,left,left,left,left,down,pick,left,left,left,
left,left,down,light;15060078
```
- Expanded nodes : `15060078`
- CPU: 16%
- Ram: 4.1 GB
- Time: 22s

DF:

- Output:
```
right,down,pick,down,down,down,down,down,down,down,down,right,right,right,right,right,right
,right,up,right,right,down,right,right,right,up,up,up,up,up,up,pick,down,down,down,down,dow
n,down,left,left,up,up,pick,down,down,right,right,up,up,up,up,up,up,up,up,up,up,up,pick,dow
n,down,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,up,up,up,up,up
,up,left,up,up,up,left,left,down,down,down,down,down,down,down,down,down,down,down,down,dow
n,down,left,left,up,up,up,up,up,up,up,up,up,up,up,left,left,down,down,left,down,down,down,d
own,down,left,left,up,up,up,up,up,left,up,up,right,up,up,right,right,right,right,up,pick,do
wn,left,left,down,down,down,down,down,down,down,down,down,down,down,down,down,right,right,r
ight,right,right,up,right,right,down,right,right,right,up,up,up,up,up,up,up,up,up,up,up,lef
t,left,down,down,down,down,down,down,down,down,left,pick,down,pick,down,down,right,right,ri
ght,up,up,up,up,up,up,up,up,up,up,up,up,up,up,left,left,down,left,down,down,down,down,pick,
right,down,down,down,down,down,down,down,down,down,right,right,up,up,up,up,up,up,up,up,up,u
p,up,up,up,up,left,left,down,left,down,down,left,down,left,down,down,down,down,down,down,do
wn,down,down,down,left,left,up,up,up,up,up,up,up,up,up,up,up,right,up,right,up,up,left,left
,left,down,left,left,down,down,down,down,down,down,down,down,down,left,left,up,up,up,up,up,
left,left,pick,down,down,pick,down,down,down,down,pick,down,down,down,right,right,right,rig
ht,right,right,right,right,right,up,right,right,down,right,right,right,up,up,up,up,up,up,up
,up,up,up,up,up,up,up,left,left,left,left,left,down,down,down,down,left,pick,down,down,down
,down,down,down,down,down,down,down,left,left,up,pick,down,right,right,right,up,right,right
,down,right,right,right,up,up,up,up,up,up,up,up,up,up,up,up,up,up,left,left,down,left,down,
down,down,down,right,down,down,down,down,down,down,left,left,up,up,up,up,up,left,up,up,up,u
p,up,up,left,left,left,down,left,left,down,down,down,down,down,down,down,down,down,right,do
wn,down,left,down,down,left,left,left,light;2226
```

- Expanded nodes : `2226`
- CPU: 0%
- Ram: 0.3 GB
- Time: 0.005s

Despite lot's of operations are occurred by this search, it has the lowest amount of expanded node. CPU Utilization recorded to be zero due to the fast speed of execution and it cost a less memory. However it's still not complete or optimal solution.

UC:

- Output:
```
left,pick,down,down,pick,down,down,down,down,pick,up,up,up,right,right,up,up,pick,right,up,
right,up,up,up,up,up,right,right,pick,right,right,down,down,down,down,pick,right,right,righ
t,down,pick,up,up,right,right,right,pick,down,down,down,down,down,pick,down,down,down,left,
down,left,pick,left,up,pick,down,pick,down,left,left,left,left,left,pick,left,left,left,dow
n,left,left,light;13847479
```
-

- Expanded nodes : `13847479`
- CPU: 74%
- Ram: 4.1 GB
- Time: 25s

The difference between BF & UC is The UC cares about getting to the goal state with less total cost unlike the BF that doesn't care about total cost. The UC expand less than BF due to using prioity queue to sort node ascendingly in term of total cost. BF can be better in cpu utilization & time consuming than UC. The similarity of the UC & BF is they gave a complete & optimal solution and use same amount of ram.

GR1:

- Output:

```
left,down,down,right,right,right,up,right,up,right,up,right,right,right,down,right,down,rig
ht,down,down,down,right,right,up,right,right,pick,down,left,left,left,left,up,up,up,left,le
ft,up,left,left,left,left,down,left,left,pick,right,right,up,right,up,right,right,right,dow
n,right,down,right,down,down,down,right,down,down,down,right,pick,up,up,right,up,right,up,u
p,up,up,up,up,pick,left,left,left,up,up,left,up,left,left,left,left,pick,left,down,left,dow
n,down,down,down,left,down,left,down,left,left,pick,right,right,up,right,up,right,up,right,
right,right,right,down,right,down,right,down,down,down,down,left,down,down,right,right,pick
,up,left,left,left,left,left,down,down,pick,up,up,left,left,left,left,left,left,pick,up,up,
up,up,right,right,up,up,right,right,up,right,right,right,right,right,right,right,down,pick,
up,left,left,left,down,down,down,down,down,right,right,down,down,right,pick,left,left,left,
up,up,up,up,up,up,up,pick,left,left,left,left,down,left,left,left,left,pick,down,down,down,
down,down,right,down,down,down,down,light;256787
```

- Expanded nodes : `256787`
- CPU: 42%
- Ram: 0.37 GB
- Time: 0.63s

GR2:

- Output:

```
right,right,right,right,right,right,up,right,right,right,right,down,right,down,down,right,r
ight,down,pick,down,left,left,down,left,down,down,pick,up,up,right,up,up,up,up,up,left,pick
,right,up,up,left,left,left,left,down,left,down,left,left,left,left,left,down,pick,up,left,
left,down,down,pick,up,up,right,right,right,right,right,right,right,right,down,down,down,do
wn,right,right,down,down,down,left,left,left,left,down,pick,right,up,right,right,right,up,u
p,left,up,up,up,up,up,up,right,right,right,right,up,right,pick,left,left,left,left,left,dow
n,down,down,down,down,right,down,down,right,down,right,down,pick,left,up,pick,left,up,up,up
,left,up,up,up,left,up,pick,up,up,up,up,left,left,pick,left,left,left,left,down,left,down,d
own,down,left,down,pick,down,down,down,down,down,down,pick,down,down,right,down,light;24350
89
```

- Expanded nodes : `2435089`
- CPU: 22%
- Ram: 1.7 GB
- Time: 5s

Both **GR1** & **GR2** don't give an optimal solution but at least aren't worse than DF solutions. However, GR2 has better solution than GR1 because it expands more for finding best solution.

AS1:

- Output:
```
left,pick,down,down,pick,down,down,down,down,pick,up,up,up,up,right,right,up,pick,right,rig
ht,up,up,up,up,up,right,right,up,pick,right,right,down,down,down,down,pick,right,right,righ
t,down,pick,up,right,up,right,right,pick,down,down,down,down,down,pick,left,left,down,down,
left,down,pick,down,right,pick,left,pick,down,left,left,left,left,left,pick,down,left,left,
left,left,left,light;13517949
```
- Expanded nodes : `13517949`
- CPU: 28%
- Ram: 4 GB
- Time: 36s

AS2:

- Output:
```
left,pick,down,down,pick,down,down,down,down,pick,up,up,up,right,up,right,up,pick,right,rig
ht,up,up,up,up,up,right,right,up,pick,right,right,down,down,down,down,pick,right,right,righ
t,down,pick,right,right,up,up,right,pick,down,down,down,down,down,pick,down,down,left,left,
down,left,pick,down,right,pick,left,pick,left,left,left,left,down,left,pick,left,left,left,
left,down,left,light;14794758
```
- Expanded nodes : `14794758`
- CPU: 14%
- Ram: 3.9 GB
- Time: 37s