



L-Università ta' Malta
Faculty of Information &
Communication Technology

Department
of Artificial
Intelligence

Fundamentals of Automated Planning Documentation

Isaac Muscat* (0265203L), Matthias Bartolo* (0436103L), Jerome Agius* (0353803L)

*B.Sc. (Hons) Artificial Intelligence (2nd Years)

Study-unit: **Fundamentals of Automated Planning**

Code: **ARI2101**

Lecturer: **Dr Josef Bajada**

Table of Contents

Introduction	2
8-tile Puzzle Representation	2
Methods	5
General Methods:	5
Heuristics:	6
Breadth First Search.....	7
Greedy Best First Search	9
A* Search	11
Enforced Hill Climbing.....	13
Evaluation	15
PDDL.....	22
Domain.....	22
Code:	22
Explanation:	23
Problem.....	24
Code:	24
Explanation:	24
Evaluation	28
References	31
Distribution of Work	32
Plagiarism Declaration Form.....	33

Introduction

The assignment involved representing an 8-tile sliding puzzle, and creating multiple algorithms consisting of **Breadth First Search**, **Greedy Best First Search**, **A* Search** and **Enforced Hill Climbing** which would resolve said problem and output their respective solution. In doing so, we in turn can compare the efficiency of all previously mentioned algorithms. The programming language chosen to implement such a problem and its solution was Python as it provided a variety of methods, as well as generally being convenient which was deemed useful for such an assignment. Please note that the above searching algorithms, were heavily inspired from sources [1] and [2].

8-tile Puzzle Representation

Firstly, prior to implementing any of the algorithms mentioned above we had to represent the 8-tile puzzle which we achieved via the following code seen Figures 1 and 2:

Code:

```
#Declaring Puzzle class which holds an instance of a board
class Puzzle:

    #Creating Shared variable which stores the number of created states
    stateCounter = 0
    #Creating Shared variable which stores the number of unique explored/unexplored states
    noOfUniqueStates=0

    #Puzzle Constructor which includes the following variables:
    # 1. currentState - which holds the current state of the board
    # 2. parentState - which holds the previous/parent state of the board
    # 3. action - which holds the action applied to achieve the current state from the parent state of the board
    # 4. nodeId - which holds a unique node Id used to distinguish between nodes (mainly utilised for A* Search)
    # 5. costToReachNode - which holds the actual cost to reach the current node, in other words, g(n)
    def __init__(self, currentState, parentState, action, nodeId, costToReachNode):
        self.currentState = currentState
        self.parentState = parentState
        self.action = action
        self.nodeId = nodeId
        self.costToReachNode = costToReachNode;
        #Incrementing the state counter every time a Puzzle instance is created
        Puzzle.stateCounter += 1

    #Declaring a Method which generates new children nodes
    def generateChildren(self):
        #Declaring a list which holds new child instances of the board
        childrenStates = []

        #Retrieving the Location of the empty slot
        emptyLocation = self.currentState.index(0)

        #Checking if the empty Location (0) is at the right column
        if(int(emptyLocation%width) != (width-1)):
            #Making a copy of the board & swapping values
            temp = self.currentState.copy()
            temp[emptyLocation], temp[emptyLocation+1] = temp[emptyLocation+1], temp[emptyLocation]
            #Appending the valid Puzzle action to the list
            childrenStates.append(Puzzle(temp, self, "Slide Left", Puzzle.stateCounter, self.costToReachNode + 1))
```

Figure 1: 8-tile puzzle representation part 1

```

#Checking if the empty Location (0) is at the Left column
if(int(emptyLocation%width) != 0):
    #Making a copy of the board & swapping values
    temp = self.currentState.copy()
    temp[emptyLocation], temp[emptyLocation-1] = temp[emptyLocation-1], temp[emptyLocation]
    #Appending the valid Puzzle action to the List
    childrenStates.append(Puzzle(temp, self, "Slide Right", Puzzle.stateCounter, self.costToReachNode + 1))

#Checking if the empty Location (0) is at the top row
if(int(emptyLocation/height) != 0):
    #Making a copy of the board & swapping values
    temp = self.currentState.copy()
    temp[emptyLocation], temp[emptyLocation-width] = temp[emptyLocation-width], temp[emptyLocation]
    #Appending the valid Puzzle action to the List
    childrenStates.append(Puzzle(temp, self, "Slide Down", Puzzle.stateCounter, self.costToReachNode + 1))

#Checking if the empty Location (0) is at the Lowest row
if(len(temp)>emptyLocation+width):
    #Making a copy of the board & swapping values
    temp = self.currentState.copy()
    temp[emptyLocation], temp[emptyLocation+width] = temp[emptyLocation+width], temp[emptyLocation]
    #Appending the valid Puzzle action to the List
    childrenStates.append(Puzzle(temp, self, "Slide Up", Puzzle.stateCounter, self.costToReachNode + 1))

#Returning the childrenStates List
return childrenStates

#Declaring a Method which backtracks the graph when the goal is found, to output the whole solution
def backtrackSolution(self):
    #Creating empty array to hold solution of the problem
    solution = []
    #Adding the current state of the board to the solution
    solution.append(self)
    #Declaring a variable used to backtrack to the initial node, and initialising it to self
    previousState = self
    #Looping until the initial state is found, since initial state does not have any parents
    while(previousState.parentState != None):
        #Resetting previousState to hold the parentState of the previous State
        previousState = previousState.parentState
        #Adding the previous state of the board to the solution
        solution.append(previousState)
    #Since at every step we are appending to the back of the list, instead of appending to
    #the beginning, we reverse the list to obtain the correct solution order
    solution.reverse()
    #Returning solution
    return solution

```

Figure 2: 8-tile puzzle representation part 2

Explanation:

A Puzzle was created to be able to represent the 8-tile board and all its possible moves in a tree-like structure, which would then be utilised by the requested algorithms. This class is composed of the following instance variables denoted below:

- **currentState**
 - This variable holds a 1D array which represents the current puzzle layout ex: [1,2,3,4,5,6,7,0,8].
- **parentState**
 - This variable holds a reference to the parent node of the current node. The parent node would be of type Puzzle, and it would refer to the action preceding the current one.
- **action**
 - This variable holds one of four strings: “Slide Up”, “Slide Down”, “Slide Left”, “Slide Right” which represent the move carried out from the previous state, to get to the current state.
- **nodeId**
 - This variable holds a unique value assigned to each node starting from one and increasing onwards.
- **costToReachNode**
 - This variable holds an integer value associated with the layer upon which the node was discovered, thus the initial node would have a value of 0 and its children a value of 1.

The Puzzle class also contains two global variables, these being **stateCounter** which is incremented each time a new Puzzle is instantiated and **noOfUniqueState** which holds the number of unique states present in the solution.

Besides the constructor, the Puzzle class also employs the following two methods:

- **generateChildren()**
 - When called upon, this method instantiates a Puzzle which first finds the location of 0 (used to represent the empty cell). Following this, it proceeds to determine all possible executable actions. Upon doing so, the resulting states are generated in accordance with said actions. In other words, new Puzzle instances are created to represent all the possible moves. These new puzzle instances are then appended to the childrenStates array which is returned upon method completion.
 - Every Puzzle instance can only have a minimum of two actions and a maximum of four actions. The former occurs whenever the empty cell is in any of the four corners whilst the latter occurs if the empty cell is in the middle of the board.
- **backtrackSolution()**
 - When called upon, this method instantiates a Puzzle which first looks at the parentState variable of the Puzzle instance and appends the parentState.currentState to the solution array. Then, the parentState is accessed and its parent state is added to the solution array. This keeps looping until a state which has no parent state is found, this being the initial state. In other words, said method backtracks from a given board state to reach the initial state.

Methods

General Methods:

Code:

```
#Declaring a Method which checks if goal state has been found
def goalFound(currentState):
    #Checking if current state matches the goals
    goal=list(range(width*height))
    goal.remove(0)
    goal.append(0)
    if(currentState == goal):
        return True
    return False

#Declaring a Method which counts the number of inversions
def countInversions(puzzleboard):
    #Declaring and initialising the counter to 0
    counter = 0
    #Looping through all board contents, and for every cell, we are checking whether there are any inversions
    for i in range(0, width*height):
        for j in range(i + 1, width*height):
            #Checking if puzzleboard[i]>puzzleboard[j], then there exists an inversion, and we must increment counter by 1
            if (puzzleboard[i] != 0 and puzzleboard[j] != 0 and puzzleboard[i] > puzzleboard[j]):
                counter += 1
    return counter
```

Figure 3: General methods

Explanation:

As can be seen in Figure 3:

- goalFound()
 - This method compares the array passed representing a specific 8-tile layout to the goal state. If they match, true is returned. Otherwise, false is returned.
- countInversion()
 - This method detects unsolvable board state configurations and thus, avoids solving invalid 8-tile representations. This is achieved by checking for inversions. Such an invalid case occurs when two adjacent cells have consecutive values where the larger value precedes the smaller one. For example, if Cell 1 has value 6 and Cell 2 has value 5, the puzzle is invalid. Such method was inspired from [3].

Heuristics:

Code:

```
#Declaring a Heuristic function which counts the number of misplaced tiles, and returns a heuristic value
def CountMisplacedTiles(puzzleboard):
    goal=list(range(width*height))
    goal.remove(0)
    goal.append(0)
    HeuristicValue = 0
    #Looping through all the board locations and checking that the Location is not an empty Location,
    #checking whether the tiles
    #are in their correct position as stated in the goal, if not the heuristic value is updated
    for x in range(0,width*height):
        if (puzzleboard[x] != 0 and puzzleboard[x] != goal[x]):
            HeuristicValue += 1
    return HeuristicValue

#Declaring a Heuristic function which counts the sum of how far misplaced tiles are from their target position,
#and returns a heuristic value
def ManhattanDistance(puzzleboard):
    goal=list(range(width*height))
    goal.remove(0)
    goal.append(0)
    HeuristicValue = 0
    #Looping through all the board locations and checking that the Location is empty
    for x in range(width*height):
        if (puzzleboard[x] != 0):
            #Retrieving row and column position of current node
            rowboard = int(x / height)
            columnboard = int(x % width)

            #Finding out the row and column position of current node in the goal
            nodeindex = goal.index(puzzleboard[x])
            rowgoal = int(nodeindex / height)
            columngoal = int(nodeindex % width)
            #Performing the Manhattan Distance formula to obtain the result
            HeuristicValue += abs(rowboard-rowgoal) + abs(columnboard - columngoal)
    return HeuristicValue

#Declaring a Method to choose the heuristic function
def heuristicChoice(puzzleboard, heuristicNumber):
    #Invoking the appropriate method according to the past parameter
    if heuristicNumber == 1:
        return CountMisplacedTiles(puzzleboard)
    elif heuristicNumber == 2:
        return ManhattanDistance(puzzleboard)
```

Figure 4: Heuristic methods

Explanation:

As can be seen in Figure 4:

- CountMisplacedTiles()
 - This method calculates the misplaced tiles heuristic. It does so by counting the number of tiles which do not coincide with their correct cell location in the goal state i.e. [1,2,3,4,5,6,7,8,0]. Thus, [1,2,3,0,5,6,4,7,8] has a heuristic value of 3 since 4, 7 and 8 are out of place.
- ManhattanDistance()
 - This method calculates the Manhattan distance heuristic. It does so by iterating through each of the cells and working out the number of moves required to move them to their correct position, to match the goal state. Via division and modulus, we can obtain the row and column of every cell in the array, and thus, we can work out the Manhattan distance by subtracting the current position of a cell with the position of that cell in the goal state. The use of division and modulus is required since the 8-tile board is represented as a 1D array.
- heuristicChoice()
 - This method takes in an array (representation of the 8-tile board) and a value between 1 and 2. It then calls the respective method from the two mentioned above, based on whether 1 or 2 is passed as a parameter and returns the corresponding heuristic value.

Breadth First Search

Definition:

Breadth First Search (BFS) is a graph traversal algorithm which evaluates the shortest path to the goal state in an unweighted graph. This is achieved by restructuring the input graph $G = (V, E)$ of n nodes and m edges into a series of n levels where each level is composed of all the nodes that can be reached from the previous level. Each of said layers is fully explored prior to moving on to the next layer. Hence, BFS may end up expanding states which lead to dead ends or suboptimal solutions, in turn also increasing its execution time. Although BFS has several shortcomings, it is very easy to implement. [4]

Code:

```
#Declaring a Method which performs Breadth First Search on a Puzzle
def BFS(puzzleboard,startTime):
    #Declaring a List to hold all explored nodes
    exploredList = set()
    unexploredBoardStates = set()
    #Checking if the initial state is a goal
    if(goalFound(puzzleboard.currentState)):
        Puzzle.noOfUniqueStates+=1
        return puzzleboard.backtrackSolution()

    #Initialising a queue to store all the children
    unexploredList = queue.Queue()
    #Appending the initial puzzle to the queue
    unexploredList.put(puzzleboard)
    unexploredBoardStates.add(str(puzzleboard.currentState))

    #Looping until unexploredList is empty
    while not (unexploredList.empty()):
        #Checking if the time limit of 15 minutes has been exceeded, if so, the algorithm is stopped
        currentTime=time.time()
        if((currentTime-startTime)>15*60):
            print("Exceeded Time limit!\nAlgorithm Stopped")
            #Setting the number of unique states to the number of explored nodes and unexplored nodes
            Puzzle.noOfUniqueStates=unexploredList.qsize()+len(exploredList)
            return puzzleboard.backtrackSolution()
        #Retrieving the front node from the queue and appending its current state to exploredList
        node = unexploredList.get()
        unexploredBoardStates.remove(str(node.currentState))
        exploredList.add(str(node.currentState))

        #Checking if the current state is the goal state, if so we backtrack from that current state
        if(goalFound(node.currentState)):
            #Setting the number of unique states to the number of explored nodes and unexplored nodes
            Puzzle.noOfUniqueStates=unexploredList.qsize()+len(exploredList)
            return node.backtrackSolution()

        #Retrieving the next layer of children for the current node
        childrenStatesList = node.generateChildren()

        #Checking if any of the current children are the goal state if so we backtrack from that child state
        for child in childrenStatesList:
            if((str(child.currentState) not in exploredList) and (str(child.currentState) not in unexploredBoardStates)):
                if(goalFound(child.currentState)):
                    #Setting the number of unique states to the number of explored nodes and unexplored nodes
                    Puzzle.noOfUniqueStates=unexploredList.qsize()+len(exploredList)
                    return child.backtrackSolution()
                #Appending the child puzzle to unexploredList
                unexploredList.put(child)
                unexploredBoardStates.add(str(child.currentState))
```

Figure 5: BFS method

Explanation:

As can be seen in Figure 5, the BFS() method holds the functionality of the BFS algorithm. This method takes a puzzle object and startTime as inputs. The latter is used to calculate the time taken for the algorithm to execute whilst the former serves as the starting node for the search tree.

Firstly, the initial state is checked to make sure that it is not the goal state. If so, it returns the solution via the backtrackSolution() method. Otherwise, it proceeds to generate its children. This repeats until all the nodes in the current layer are explored. If none of them are the goal state, it moves to the next layer and proceeds to repeat the same process. Every time a node is generated via the generateChildren() method, it is checked to make sure that it is not the goal state. If so, it returns the solution via the backtrackSolution() method. Otherwise, it is appended to the unexploredList queue. Each time a node is explored, it is removed from said queue and appended to the exploredList set. Every time a child is generated, the child node is checked to see that it has not been explored before and that it is not already present in the unexploredList. If this is the case, the child node is compared to the goal state and if they match the algorithm terminates. Otherwise, it is added to the unexploredList queue. This keeps on repeating until a goal is found or the allocated time of fifteen minutes expires.

Upon finding a solution, the backtrackSolution() method is called on the child node (which matches the goal state), that then returns the path of nodes from the initial state to the goal state.

Greedy Best First Search

Definition:

Greedy Best First Search (GBFS) builds upon Breadth First Search by including a heuristic $h(n)$ which estimates the cost of reaching the goal node from the current node. The nodes are explored in accordance with their heuristic, smallest first. GBFS has similar behaviour to depth-first search. Its advantages are delivered via the use of a quality heuristic function to direct the search. [5]

Code:

```
#Declaring a Method which performs Greedy Best First Search on a Puzzle
def GBFS(puzzleboard, heuristicNumber, startTime):
    #Declaring a List to hold all explored nodes
    exploredList = set()

    #Checking if the initial state is a goal
    if(goalFound(puzzleboard.currentState)):
        Puzzle.noOfUniqueStates=1
        return puzzleboard.backtrackSolution()

    #Initialising a List to store all the children
    unexploredList = []
    unexploredBoardStates = set()

    #Appending the initial puzzle to the queue
    heapq.heappush(unexploredList, (heuristicChoice(puzzleboard.currentState, heuristicNumber), puzzleboard.nodeId,
                                         puzzleboard))
    unexploredBoardStates.add(str(puzzleboard.currentState))

    #Looping until unexploredList is empty
    while (unexploredList != []):
        #Checking if the time limit of 15 minutes has been exceeded, if so, the algorithm is stopped
        currentTime=time.time()
        if((currentTime-startTime)>15*60):
            print("Exceeded Time limit!\nAlgorithm Stopped")
            #Setting the number of unique states to the number of explored nodes and unexplored nodes
            Puzzle.noOfUniqueStates=len(unexploredList)+len(exploredList)
            return puzzleboard.backtrackSolution()

        #Retrieving front node from the queue and appending its current state to exploredList
        heuristic,stateCounter,node = heapq.heappop(unexploredList)
        unexploredBoardStates.remove(str(node.currentState))
        exploredList.add(str(node.currentState))

        #Checking if the current state is the goal state, if so we backtrack from that current state
        if(goalFound(node.currentState)):
            #Setting the number of unique states to the number of explored nodes and unexplored nodes
            Puzzle.noOfUniqueStates=len(unexploredList)+len(exploredList)
            return node.backtrackSolution()

        #Retrieving next Layer of children for node
        childrenStatesList = node.generateChildren()

        #Checking if any of the current children are the goal state if so we backtrack from that child state
        for child in childrenStatesList:
            if((str(child.currentState) not in exploredList) and (str(child.currentState) not in unexploredBoardStates)):
                if(goalFound(child.currentState)):
                    #Setting the number of unique states to the number of explored nodes and unexplored nodes
                    Puzzle.noOfUniqueStates=len(unexploredList)+len(exploredList)
                    return child.backtrackSolution()
                #Appending the optimal child puzzle to unexploredList
                heapq.heappush(unexploredList, (heuristicChoice(child.currentState, heuristicNumber), child.nodeId, child))
                unexploredBoardStates.add(str(child.currentState))
```

Figure 6: GBFS method

Explanation:

As can be seen in Figure 6, the GBFS() method holds the functionality of the GBFS algorithm. This method takes a puzzle object, startTime and heuristicNumber as inputs. The former two parameters have the same use as that denoted in the previous section whilst the heuristicNumber parameter is used to determine which heuristic function is to be used whether it be Manhattan distance or misplaced tiles.

The code for the GBFS() function uses the BFS() code as its basis, the main differences being that the unexploredList is now a heap queue instead of a normal queue as well as that the node now not only includes the Puzzle instance but is also accompanied by a heuristic value and a unique nodeId. The heuristic value is used to retrieve the node which is assumed to be closest to the goal node, the smaller the value, the closer the node presumably is.

Since some tile configurations might result in the same heuristic value, a secondary index is needed. The nodeId serves as a secondary index. Since every nodeId is unique, this problem should not happen.

A* Search

Definition:

A* Search builds upon Greedy Best First Search by using a cost functionality used to measure the cost of traversing nodes. Essentially, it is the number of nodes required to traverse from the initial state to the current state. The result of adding together this cost function ($g(n)$) with the heuristic function ($h(n)$) is $f(n)$. The nodes are explored in accordance with their smallest $f(n)$ value first. Its advantages include that it guarantees optimality given that the heuristic function is admissible, which means that the heuristic function is always smaller or equal to the actual cost to reach the goal state, and consistent, which means that the heuristic of a node is always smaller than or equal to the sum of the cost to reach a successor node and its corresponding heuristic. [6]

Code:

```
#Declaring a Method which performs Greedy Best First Search on a Puzzle
def Astar(puzzleboard, heuristicNumber, startTime):
    #Declaring a List to hold all explored nodes
    exploredList = dict()

    #Checking if the initial state is a goal
    if(goalFound(puzzleboard.currentState)):
        Puzzle.noOfUniqueStates=1
        return puzzleboard.backtrackSolution()

    #Initialising a list to store all the children
    unexploredList = []
    unexploredBoardStates = set()
    #Appending the initial puzzle to the queue
    heapq.heappush(unexploredList, (heuristicChoice(puzzleboard.currentState, heuristicNumber)
    + puzzleboard.costToReachNode, puzzleboard.nodeId, puzzleboard))
    unexploredBoardStates.add(str(puzzleboard.currentState))
    #Looping until unexploredList is empty
    while (unexploredList != []):
        #Checking if the time limit of 15 minutes has been exceeded, if so, the algorithm is stopped
        currentTime=time.time()
        if((currentTime-startTime)>15*60):
            print("Exceeded Time limit!\nAlgorithm Stopped")
            #Setting the number of unique states to the number of explored nodes and unexplored nodes
            Puzzle.noOfUniqueStates=len(unexploredList)+len(exploredList)
            return puzzleboard.backtrackSolution()

        #Retrieving front node from the queue and appending its current state to exploredList
        heuristic,stateCounter,node = heapq.heappop(unexploredList)
        unexploredBoardStates.remove(str(node.currentState))
        exploredList[str(node.currentState)]=node.costToReachNode
        #Checking if the current state is the goal state, if so we backtrack from that current state
        if(goalFound(node.currentState)):
            #Setting the number of unique states to the number of explored nodes and unexplored nodes
            Puzzle.noOfUniqueStates=len(unexploredList)+len(exploredList)
            return node.backtrackSolution()

        #Retrieving next layer of children for node
        childrenStatesList = node.generateChildren()

        #Checking if any of the current children are the goal state if so we backtrack from that child state
        for child in childrenStatesList:
            if(str(child.currentState) not in exploredList and str(child.currentState) not in unexploredBoardStates):
                if(goalFound(child.currentState)):
                    #Setting the number of unique states to the number of explored nodes and unexplored nodes
                    Puzzle.noOfUniqueStates=len(unexploredList)+len(exploredList)
                    return child.backtrackSolution()
                #Appending the optimal child puzzle to unexploredlist
                heapq.heappush(unexploredList, (heuristicChoice(child.currentState, heuristicNumber)
                + child.costToReachNode, child.nodeId, child))
                unexploredBoardStates.add(str(child.currentState))
            #Updating cost of node, if current child node cost is less
            elif(str(child.currentState) in exploredList):
                cost=exploredList[str(child.currentState)]
                if(cost>child.costToReachNode):
                    exploredList[str(child.currentState)]=child.costToReachNode
```

Figure 7: Astar method

Explanation:

As can be seen in Figure 7, the Astar() method holds the functionality of the A* algorithm. This method takes a puzzle object, startTime and heuristicNumber as inputs. The former two parameters have the same use as that denoted in the previous section whilst the heuristicNumber parameter is used to determine which heuristic function is to be used whether it be Manhattan distance or misplaced tiles.

The code for the Astar() function uses the GBFS() code as its basis, the main differences being that costToReachNode is being used as a $g(n)$. The sum of the costToReachNode and the heuristic value is used to determine the order at which nodes are explored.

Enforced Hill Climbing

Definition:

Enforced Hill Climbing attempts to find a solution without incurring the space complexity of A* search. Firstly, it computes the heuristic of the initial node and stores it as the best heuristic value. Following this it computes a breadth first search on the child nodes and if it finds one whose cost is smaller than the current best heuristic value, breadth first search restarts on that child node. Also, the best heuristic value is updated to that of the child node. [7]

Code:

```
def EHCS(puzzleboard, heuristicNumber, startTime):
    exploredList = set()
    unexploredBoardStates = set()
    Puzzle.noOfUniqueStates=1
    #Checking if first state is a goal
    if(goalFound(puzzleboard.currentState)):
        Puzzle.noOfUniqueStates=1
        return puzzleboard.backtrackSolution()
    #Initialising a queue to store all the children
    unexploredList = queue.Queue()
    #Calculating the heuristic of the current state and setting it as the bestHeuristic
    bestHeuristic = heuristicChoice(puzzleboard.currentState, heuristicNumber)
    #Append the initial puzzle to the queue
    unexploredList.put(puzzleboard)
    unexploredBoardStates.add(str(puzzleboard.currentState))

    #Loop until unexploredList is empty
    while not (unexploredList.empty()):
        #Checking if the time limit of 15 minutes has been exceeded, if so, the algorithm is stopped
        currentTime=time.time()
        if((currentTime-startTime)>15*60):
            print("Exceeded Time limit!\nAlgorithm Stopped")
            return puzzleboard.backtrackSolution()
        #Retrieving front node from the queue and appending its current state to exploredList
        node = unexploredList.get()
        exploredList.add(str(node.currentState))
        #Checking if the current state is the goal state, if so we backtrack from that current state
        if(goalFound(node.currentState)):
            #Setting the number of unique states to the number of explored nodes and unexplored nodes
            return node.backtrackSolution()

        #Retrieving next layer of children for node
        childrenStatesList = []
        childrenStatesList = node.generateChildren()
        #Checking if any of the current children are the goal state if so we backtrack from that child state
        for child in childrenStatesList:
            if((str(child.currentState) not in exploredList) and (str(child.currentState) not in unexploredBoardStates)):
                #Calculating the heuristic of the child state and assigning it to cost
                cost = heuristicChoice(child.currentState, heuristicNumber)
                Puzzle.noOfUniqueStates+=1
                if(goalFound(child.currentState)):
                    return child.backtrackSolution()
                #Checking if the cost of the child is less than that of the current state, assigning the cost to bestHeuristic,
                #clearing unexploredList, appending the child to it and breaking from the loop
                if (cost < bestHeuristic):
                    bestHeuristic = cost
                    unexploredList.queue.clear()
                    unexploredList.put(child)
                    unexploredBoardStates.add(str(child.currentState))
                    break
                #Appending the child to unexploredList
                unexploredList.put(child)
                unexploredBoardStates.add(str(child.currentState))
        #Checking if Enforced Hill Climbing fails and if so falling back to A* Search
        print("Fallback Solution")
    return Astar(puzzleboard, heuristicNumber, time.time())
```

Figure 8: EHCS method

Explanation:

As can be seen in Figure 8, the EHC() method holds the functionality of the Enforced Hill Climbing algorithm. This method takes a puzzle object, startTime and heuristicNumber as inputs. The former two parameters have the same use as that denoted in the previous section whilst the heuristicNumber parameter is used to determine which heuristic function is to be used whether it be Manhattan distance or misplaced tiles.

The code for the EHC() function incorporates the BFS() functionality, the main difference being that when a new child node with a heuristic value smaller than the current node is found, it is immediately explored. When this occurs, the unexploredList is cleared and the breadth first algorithm is called on the new child node.

The above reasoning is being done by checking if the child nodes generated are in the explored and unexploredBoardStates list (used to store all the unexplored nodes), to make sure no duplicate child nodes are considered. If it is not found in either set, it is compared to the goal state. If it is not a goal state, its cost is compared to that of the current node's heuristic, and if it is smaller, the heuristic is updated to the cost of the child, the unexploredList is cleared, and the child is appended to the unexploredList.

In the case that none of the children node's costs are smaller than the bestHeuristic, EHC fails and so A* search is called as a fallback solution.

Evaluation

To evaluate the two hard problems given in the brief, a bar chart was utilised to show the differences between the seven combinations of search techniques and heuristic functions. Hence, a bar chart was created for every problem given. This section uses the evaluate function, which calls each algorithm and then creates a bar chart from the returned values, these being: the states generated, the number of actions taken as well as the time taken.

Test Initial State 1:

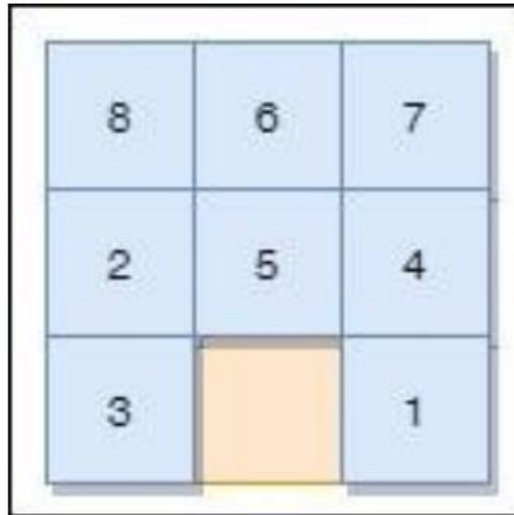


Figure 9: Test Initial State 1

The following was displayed when evaluating the above initial state:

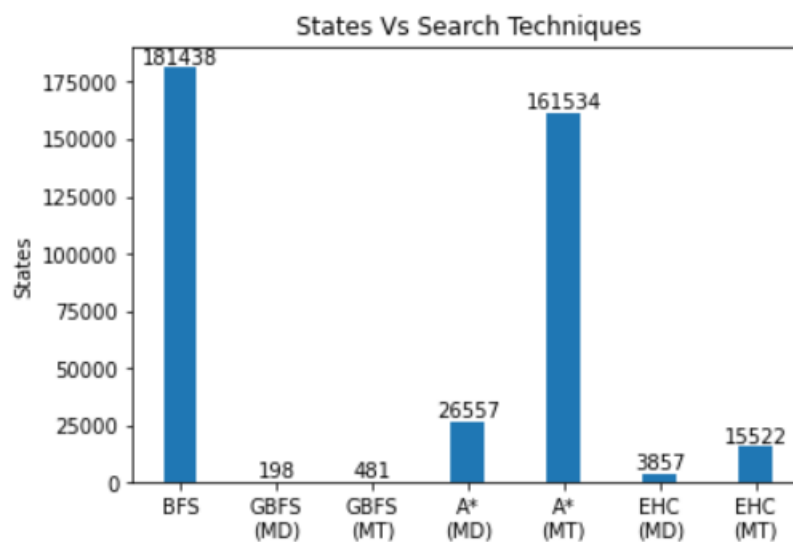


Figure 10: Test Initial State 1, States vs Search Techniques Graph

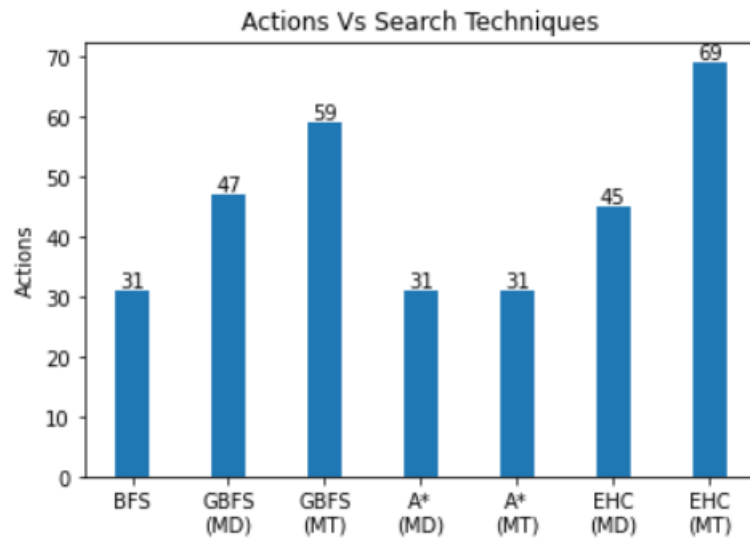


Figure 11: Test Initial State 1, Actions vs Search Techniques Graph

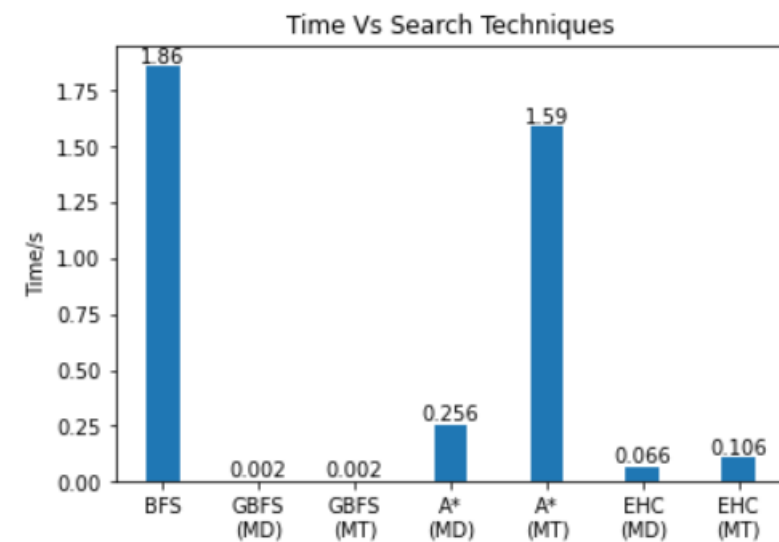


Figure 12: Test Initial State 1, Time vs Search Techniques Graph

Test Initial State 2:

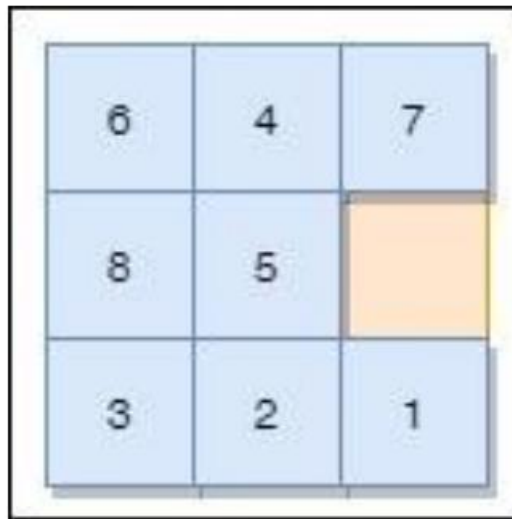


Figure 13: Test Initial State 2

The following was displayed when evaluating the above initial state:

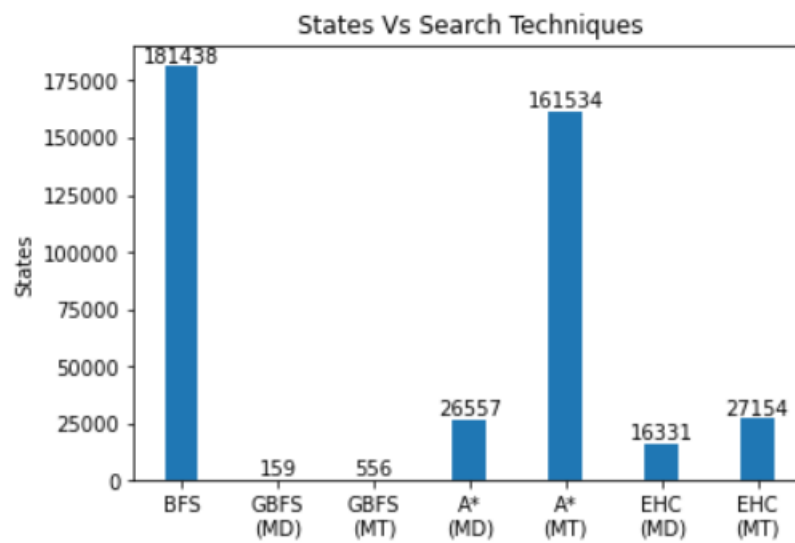


Figure 14: Test Initial State 2, States vs Search Techniques Graph

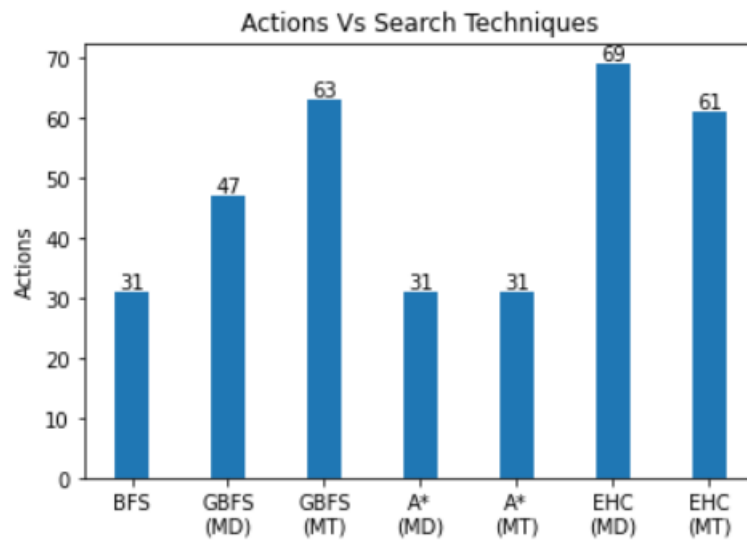


Figure 15: Test Initial State 2, Actions vs Search Techniques Graph

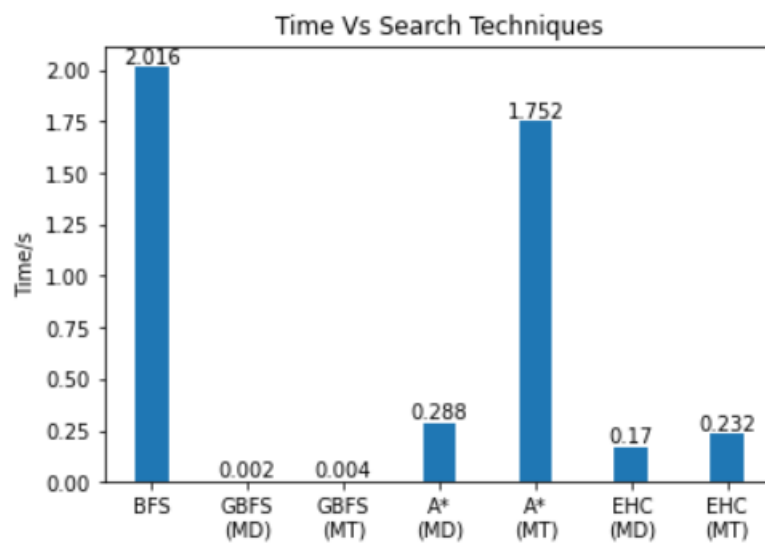


Figure 16: Test Initial State 2, Time vs Search Techniques Graph

To be able to clearly note the differences in performance statistics, an easier problem was evaluated, this being:

Test Initial State 3:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Figure 17: Test Initial State 3

The following was displayed when evaluating the above initial state:

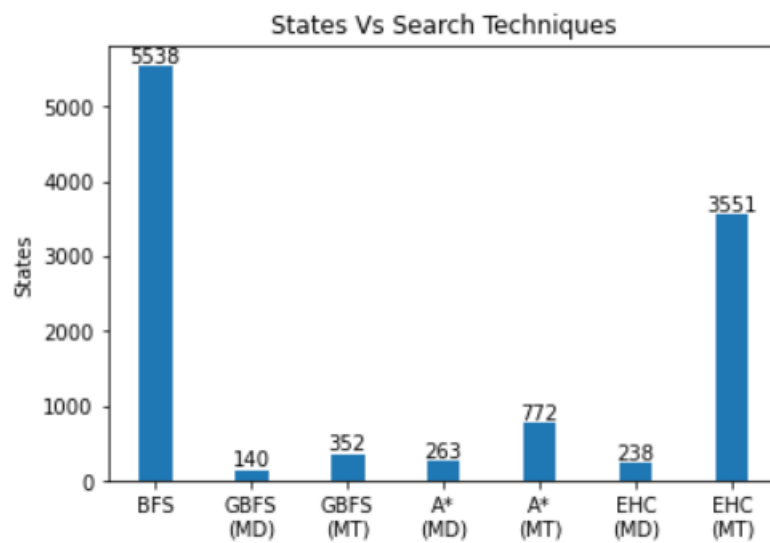


Figure 18: Test Initial State 3, States vs Search Techniques Graph

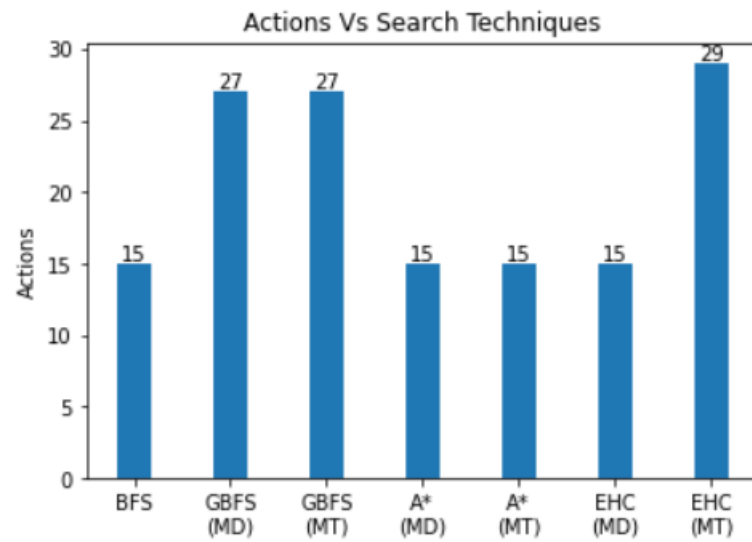


Figure 19: Test Initial State 3, Actions vs Search Techniques Graph

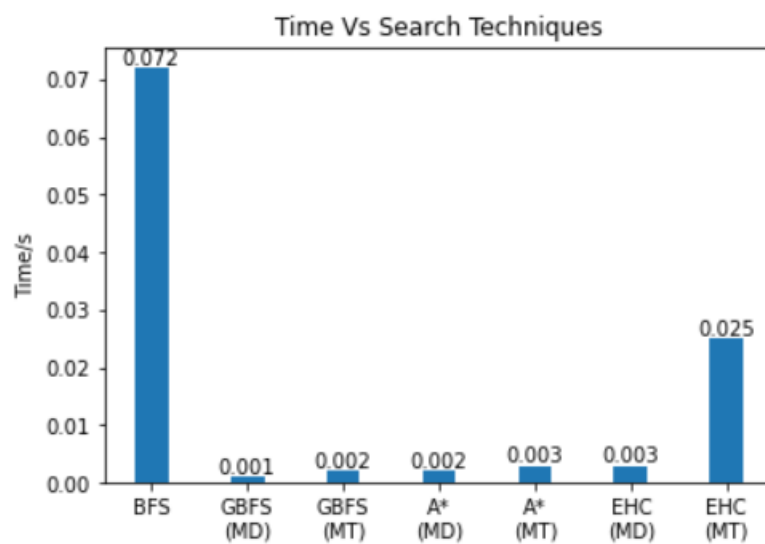


Figure 20: Test Initial State 3, Time vs Search Techniques Graph

In general, from the above graphs we can note the importance of having a good heuristic. The Manhattan distance heuristic consistently computes much faster than the Misplaced tiles heuristic. In fact, in the hard problem examples provided, the A* Search when paired with the Misplaced Tiles heuristic, takes substantially longer than when paired with the Manhattan Distance heuristic.

It was also noted that GBFS, whilst being the fastest algorithm to compute and generally generating the least number of nodes, does not guarantee an optimal plan. Moreover, GBFS is complete given that its search space is finite. Contrary to this, A* Search requires more time to compute and generates more states than GBFS. However, it will always provide the optimal plan of actions, provided that the heuristic function given is admissible and consistent, and ideally informative too, to give accurate guidance. Similarly, BFS will also result in the optimal plan, however, it takes a lengthy amount of time to compute, and it will also generate a greater number of states since no heuristic function is being utilised as it is an uninformed search. This type of search becomes unfeasible with hard problems, as noted from the graphs above. The EHC algorithm whilst not always providing an optimal plan, requires much less states than A* Search. Furthermore, this type of search algorithm is susceptible to getting stuck in local minima. Additionally, there are cases where EHC fails, and so it is not a complete search algorithm.

An interesting characteristic observed was that when given a simple problem, such as Test Initial State 3, EHC when paired with the Misplaced Tiles heuristic generated more states than A* Search. This is due to EHC being greedy and thus, since the Misplaced Tiles heuristic is less informative, will end up exploring a sizable number of states.

PDDL

Domain

Code:

```
1 (define (domain SlidingTilePuzzle)
2   (:requirements :typing)
3
4   (:types
5     tile tile_location - object ;tile is the location in the grid / v_n is the nth value i.e v_1 = 1
6   )
7
8   (:predicates
9     (tile-at ?t - tile ?tl - tile_location) ;checks if a tile has a value
10
11     (to-left ?current-tile ?left-tile - tile_location) ;denotes the tile to the left of the current tile
12     (to-right ?current-tile ?right-tile - tile_location) ;denotes the tile to the right of the current tile
13     (to-up ?current-tile ?up-tile - tile_location) ;denotes the tile above of the current tile
14     (to-down ?current-tile ?down-tile - tile_location) ;denotes the tile below of the current tile
15
16     (is-empty ?tl - tile_location) ;checks if a tile is empty
17   )
18
19   (:action SLIDE-LEFT
20     :parameters (?current-tile - tile_location ?tile - tile ?left-tile - tile_location)
21     :precondition (and
22       (tile-at ?tile ?current-tile)
23       (is-empty ?left-tile)
24       (not (is-empty ?current-tile))
25       (to-left ?current-tile ?left-tile)
26       (to-right ?left-tile ?current-tile)
27     )
28     :effect (and
29       (not (tile-at ?tile ?current-tile))
30       (not (is-empty ?left-tile))
31       (tile-at ?tile ?left-tile)
32       (is-empty ?current-tile)
33     )
34   )
35
36   (:action SLIDE-RIGHT
37     :parameters (?current-tile - tile_location ?tile - tile ?right-tile - tile_location)
38     :precondition (and
39       (tile-at ?tile ?current-tile)
40       (is-empty ?right-tile)
41       (not (is-empty ?current-tile))
42       (to-right ?current-tile ?right-tile)
43       (to-left ?right-tile ?current-tile)
44     )
45     :effect (and
46       (not (tile-at ?tile ?current-tile))
47       (not (is-empty ?right-tile))
48       (tile-at ?tile ?right-tile)
49       (is-empty ?current-tile)
50     )
51   )
52
53   (:action SLIDE-UP
54     :parameters (?current-tile - tile_location ?tile - tile ?up-tile - tile_location)
55     :precondition (and
56       (tile-at ?tile ?current-tile)
57       (is-empty ?up-tile)
58       (not (is-empty ?current-tile))
59       (to-up ?current-tile ?up-tile)
60       (to-down ?up-tile ?current-tile)
61     )
62     :effect (and
63       (not (tile-at ?tile ?current-tile))
64       (not (is-empty ?up-tile))
65       (tile-at ?tile ?up-tile)
66       (is-empty ?current-tile)
67     )
68   )
69
70   (:action SLIDE-DOWN
71     :parameters (?current-tile - tile_location ?tile - tile ?down-tile - tile_location)
72     :precondition (and
73       (tile-at ?tile ?current-tile)
74       (is-empty ?down-tile)
75       (not (is-empty ?current-tile))
76       (to-down ?current-tile ?down-tile)
77       (to-up ?down-tile ?current-tile)
78     )
79     :effect (and
80       (not (tile-at ?tile ?current-tile))
81       (not (is-empty ?down-tile))
82       (tile-at ?tile ?down-tile)
83       (is-empty ?current-tile)
84     )
85   )
86 )
```

Figure 21: PDDL Domain

Explanation:

As can be seen in Figure 21, the first thing defined in the code are the name and requirements of the domain itself. These are `SlidingTilePuzzle` and `:typing` respectively. Following this are the definition of the types, which include:

- `tile` – This type is used to denote a movable tile i.e., one with a value ranging from 1 to 8.
- `tile_location` – This type is used to denote the current location of a tile object in relation to the nine possible board locations.

Secondly, the predicates are defined which include:

- `tile-at` – This predicate is used to determine if a tile object is at a specific `tile_location` object i.e., tile with number 1 is at location 5 on the board.
- `to-left/to-right/to-up/to-down` – These predicates are used to denote which tiles are to the left/right/above/below the current tile.
- `is-empty` – This predicate is used to denote a specific tile location as being empty i.e., having no tile.

Finally, the actions are defined, which include:

- `SLIDE-LEFT` – This action is used to slide a tile one location to the left, this is achieved as follows:

Firstly, three parameters are established as follows:

- `?current-tile` – This denotes the current tile position.
- `?tile` – This denotes the tile that to be moved one location to the left.
- `?left-tile` – This denotes the tile location that is to the left of the tile that to be moved.

Then the following preconditions are checked:

1. `?tile` must be at the `?current-tile` location.
2. `?left-tile` must be empty.
3. `?tile` must not be empty.
4. `?left-tile` must be to the left of `?tile`.
5. `?tile` must be to the right of `?left-tile`.

When all these preconditions are met the following effects will take place:

1. `?tile` is no longer at `?current-tile` location.
2. `?tile` is now at `?left-tile` location.
3. `?left-tile` location is no longer considered empty.
4. `?current-tile` location is now considered empty.

The remaining actions, `SLIDE-RIGHT`, `SLIDE-UP`, `SLIDE-DOWN` all have a similar structure to that of `SLIDE-LEFT` excluding certain checks such as `to-left` and `to-right` being swapped or replaced by `to-up` and `to-bottom`. Note that the action's effects abide by the add-after-delete semantics. This can be seen in the code snippet above.

Problem

Code:

```
1 (define (problem Problem1)
2   (:domain SlidingTilePuzzle)
3
4   (:objects
5     tile-loc_1 tile-loc_2 tile-loc_3 tile-loc_4 tile-loc_5 tile-loc_6 tile-loc_7 tile-loc_8 tile-loc_9 - tile_location
6     t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 - tile
7   )
8
9   (:init
10    (tile-at t_1 tile-loc_1)
11    (tile-at t_2 tile-loc_2)
12    (tile-at t_3 tile-loc_3)
13    (is-empty tile-loc_4)
14    (tile-at t_4 tile-loc_5)
15    (tile-at t_6 tile-loc_6)
16    (tile-at t_7 tile-loc_7)
17    (tile-at t_5 tile-loc_8)
18    (tile-at t_8 tile-loc_9)
19
20    (to-right tile-loc_1 tile-loc_2)
21    (to-right tile-loc_2 tile-loc_3)
22    (to-right tile-loc_4 tile-loc_5)
23    (to-right tile-loc_5 tile-loc_6)
24    (to-right tile-loc_7 tile-loc_8)
25    (to-right tile-loc_8 tile-loc_9)
26
27    (to-left tile-loc_3 tile-loc_2)
28    (to-left tile-loc_2 tile-loc_1)
29    (to-left tile-loc_6 tile-loc_5)
30    (to-left tile-loc_5 tile-loc_4)
31    (to-left tile-loc_9 tile-loc_8)
32    (to-left tile-loc_8 tile-loc_7)
33
34    (to-up tile-loc_4 tile-loc_1)
35    (to-up tile-loc_5 tile-loc_2)
36    (to-up tile-loc_6 tile-loc_3)
37    (to-up tile-loc_7 tile-loc_4)
38    (to-up tile-loc_8 tile-loc_5)
39    (to-up tile-loc_9 tile-loc_6)
40
41    (to-down tile-loc_1 tile-loc_4)
42    (to-down tile-loc_2 tile-loc_5)
43    (to-down tile-loc_3 tile-loc_6)
44    (to-down tile-loc_4 tile-loc_7)
45    (to-down tile-loc_5 tile-loc_8)
46    (to-down tile-loc_6 tile-loc_9)
47  )
48
49   (:goal (and
50     (tile-at t_1 tile-loc_1)
51     (tile-at t_2 tile-loc_2)
52     (tile-at t_3 tile-loc_3)
53     (tile-at t_4 tile-loc_4)
54     (tile-at t_5 tile-loc_5)
55     (tile-at t_6 tile-loc_6)
56     (tile-at t_7 tile-loc_7)
57     (tile-at t_8 tile-loc_8)
58     (is-empty tile-loc_9)
59   ))
60 )
61 )
```

Figure 22: PDDL Problem 1

Explanation:

As can be seen in Figure 22. Firstly, the domain that said uses, is specified at the top, in this case it is the SlidingTilePuzzle domain mentioned prior.

Secondly, the objects required are defined, these include tile_locations from tile-loc_1 to tile-loc_9 which denote the 9 possible locations that a tile might be in and tile from t_1 to t_8 which represent the tiles themselves.

The initial state of the problem is then defined this includes the starting location of each tile (this changes according to the problem that we want to solve) and the relative location of each tile_location in relation to every other tile_location (to-right, to-left, to-up, to-down relations – this is constant irrelevant of the problem).

Finally, the goal state is defined which denotes the required location of each tile in relation to the 9 tile_locations.

The structure of the problem should stay the same between problem instances excluding the initial states which differ according to the initial board configuration. This can be seen in Figure 23.

```

1 (define (problem Problem3)
2   (:domain SlidingTilePuzzle)
3
4   (:objects
5     tile-loc_1 tile-loc_2 tile-loc_3 tile-loc_4 tile-loc_5 tile-loc_6 tile-loc_7 tile-loc_8 tile-loc_9 - tile_location
6     t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 - tile
7   )
8
9   (:init
10    (tile-at t_1 tile-loc_1)
11    (tile-at t_2 tile-loc_2)
12    (tile-at t_3 tile-loc_3)
13    (tile-at t_4 tile-loc_4)
14    (tile-at t_8 tile-loc_5)
15    (tile-at t_5 tile-loc_6)
16    (tile-at t_7 tile-loc_7)
17    (is-empty tile-loc_8)
18    (tile-at t_6 tile-loc_9)
19
20    (to-right tile-loc_1 tile-loc_2)
21    (to-right tile-loc_2 tile-loc_3)
22    (to-right tile-loc_4 tile-loc_5)
23    (to-right tile-loc_5 tile-loc_6)
24    (to-right tile-loc_7 tile-loc_8)
25    (to-right tile-loc_8 tile-loc_9)
26
27    (to-left tile-loc_3 tile-loc_2)
28    (to-left tile-loc_2 tile-loc_1)
29    (to-left tile-loc_6 tile-loc_5)
30    (to-left tile-loc_5 tile-loc_4)
31    (to-left tile-loc_9 tile-loc_8)
32    (to-left tile-loc_8 tile-loc_7)
33
34    (to-up tile-loc_4 tile-loc_1)
35    (to-up tile-loc_5 tile-loc_2)
36    (to-up tile-loc_6 tile-loc_3)
37    (to-up tile-loc_7 tile-loc_4)
38    (to-up tile-loc_8 tile-loc_5)
39    (to-up tile-loc_9 tile-loc_6)
40
41    (to-down tile-loc_1 tile-loc_4)
42    (to-down tile-loc_2 tile-loc_5)
43    (to-down tile-loc_3 tile-loc_6)
44    (to-down tile-loc_4 tile-loc_7)
45    (to-down tile-loc_5 tile-loc_8)
46    (to-down tile-loc_6 tile-loc_9)
47  )
48
49   (:goal (and
50     (tile-at t_1 tile-loc_1)
51     (tile-at t_2 tile-loc_2)
52     (tile-at t_3 tile-loc_3)
53     (tile-at t_4 tile-loc_4)
54     (tile-at t_5 tile-loc_5)
55     (tile-at t_6 tile-loc_6)
56     (tile-at t_7 tile-loc_7)
57     (tile-at t_8 tile-loc_8)
58     (is-empty tile-loc_9)
59   ))
60 )
61 )

```

Figure 23: PDDL Problem 3

```

1 (define (problem Problem5)
2   (:domain SlidingTilePuzzle)
3
4   (:objects
5     tile-loc_1 tile-loc_2 tile-loc_3 tile-loc_4 tile-loc_5 tile-loc_6 tile-loc_7 tile-loc_8 tile-loc_9 - tile_location
6     t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 - tile
7   )
8
9   (:init
10    (tile-at t_8 tile-loc_1)
11    (tile-at t_6 tile-loc_2)
12    (tile-at t_7 tile-loc_3)
13    (tile-at t_2 tile-loc_4)
14    (tile-at t_5 tile-loc_5)
15    (tile-at t_4 tile-loc_6)
16    (tile-at t_3 tile-loc_7)
17    (is-empty tile-loc_8)
18    (tile-at t_1 tile-loc_9)
19
20    (to-right tile-loc_1 tile-loc_2)
21    (to-right tile-loc_2 tile-loc_3)
22    (to-right tile-loc_4 tile-loc_5)
23    (to-right tile-loc_5 tile-loc_6)
24    (to-right tile-loc_7 tile-loc_8)
25    (to-right tile-loc_8 tile-loc_9)
26
27    (to-left tile-loc_3 tile-loc_2)
28    (to-left tile-loc_2 tile-loc_1)
29    (to-left tile-loc_6 tile-loc_5)
30    (to-left tile-loc_5 tile-loc_4)
31    (to-left tile-loc_9 tile-loc_8)
32    (to-left tile-loc_8 tile-loc_7)
33
34    (to-up tile-loc_4 tile-loc_1)
35    (to-up tile-loc_5 tile-loc_2)
36    (to-up tile-loc_6 tile-loc_3)
37    (to-up tile-loc_7 tile-loc_4)
38    (to-up tile-loc_8 tile-loc_5)
39    (to-up tile-loc_9 tile-loc_6)
40
41    (to-down tile-loc_1 tile-loc_4)
42    (to-down tile-loc_2 tile-loc_5)
43    (to-down tile-loc_3 tile-loc_6)
44    (to-down tile-loc_4 tile-loc_7)
45    (to-down tile-loc_5 tile-loc_8)
46    (to-down tile-loc_6 tile-loc_9)
47  )
48
49   (:goal (and
50     (tile-at t_1 tile-loc_1)
51     (tile-at t_2 tile-loc_2)
52     (tile-at t_3 tile-loc_3)
53     (tile-at t_4 tile-loc_4)
54     (tile-at t_5 tile-loc_5)
55     (tile-at t_6 tile-loc_6)
56     (tile-at t_7 tile-loc_7)
57     (tile-at t_8 tile-loc_8)
58     (is-empty tile-loc_9)
59   ))
60 )
61 )

```

Figure 24: PDDL Problem 5

When problem-5 hard is executed via online solver, the following are the returned performance statistics:

The number of actions taken to solve the puzzle were: 37

The number of states generated throughout the puzzle solution were: 244

The algorithm took the following time to execute: 0.008 s

When problem-6 hard is executed via online solver, the following are the returned performance statistics:

The number of actions taken to solve the puzzle were: 37

The number of states generated throughout the puzzle solution were: 248

The algorithm took the following time to execute: 0.004 s

Evaluation

The following bar graphs in Figures 25-30, compare our domain-specific solver, coded in Python, and our domain-independent planner, coded in PDDL.

Problem 5:

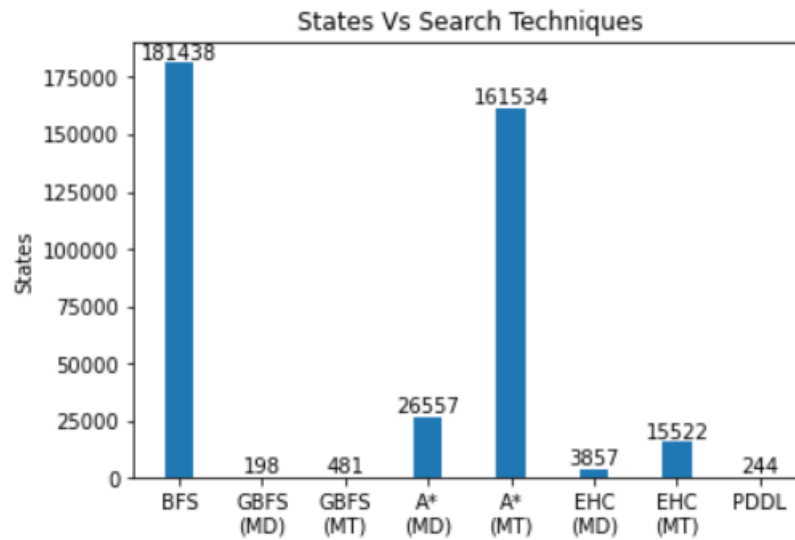


Figure 25: Problem 5, States vs Search Techniques Graph

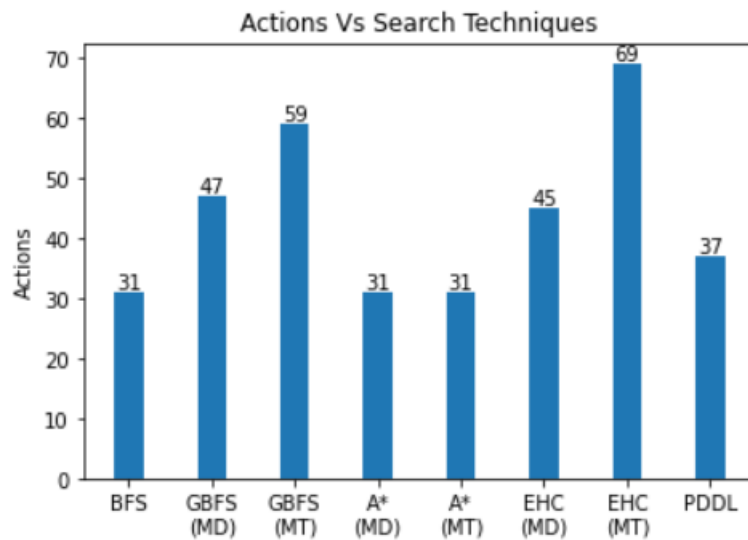


Figure 26: Problem 5, Actions vs Search Techniques Graph

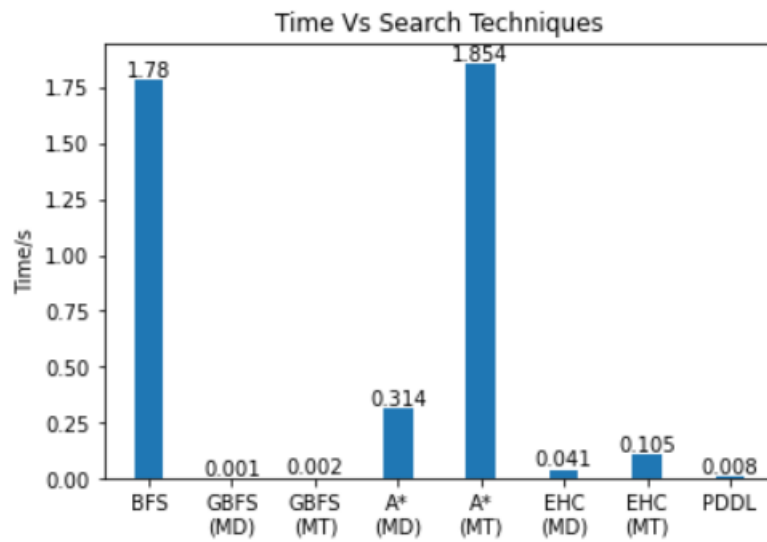


Figure 27: Problem 5, Time vs Search Techniques Graph

Problem 6:

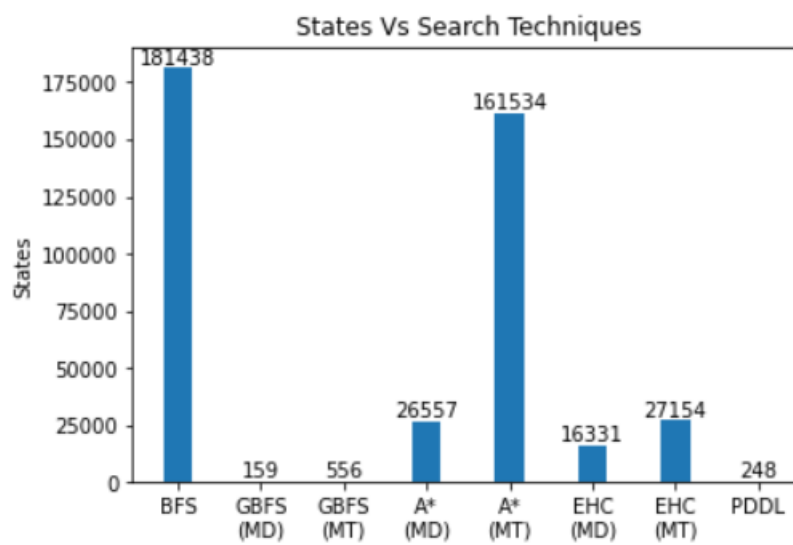


Figure 28: Problem 6, States vs Search Techniques Graph

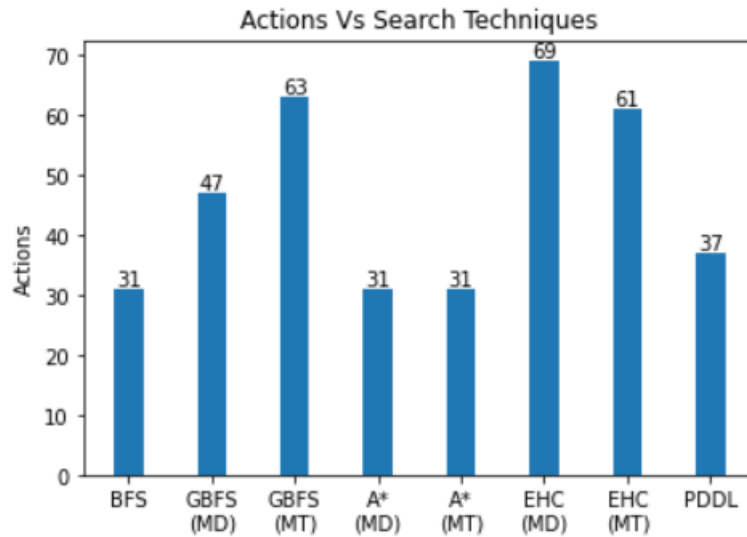


Figure 29: Problem 6, Actions vs Search Techniques Graph

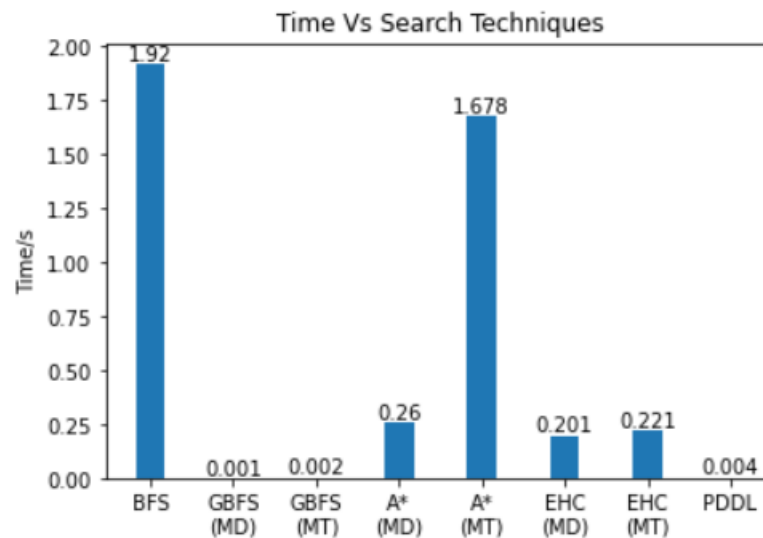


Figure 30: Problem 6, Time vs Search Techniques Graph

In addition to the previous search algorithm evaluations, from the above graphs we can note that compared to the rest of the search techniques, PDDL generates very few nodes (similar to GBFS), it will not necessarily find the most optimal plan of actions (unlike A* search) but it will take very little time to compute (similar computation time to GBFS). Nevertheless, it is not sensible to consider the time taken since we utilised an online solver, and thus the computation time will not necessarily be accurate.

References

- [1] J.Bajada, ARI2101: "Planning as State-Space Search" [Online]. Available: https://www.um.edu.mt/vle/pluginfile.php/1076248/mod_resource/content/2/Fundamentals%20of%20Automated%20Planning%20-%20Lecture%203%20-%20Planning%20as%20State-Space%20Search.pdf [Accessed: 14- Dec- 2022].
- [2] J.Bajada, ARI2101: "Planning Graph Heuristics" [Online]. Available: https://www.um.edu.mt/vle/pluginfile.php/1076262/mod_resource/content/3/Fundamentals%20of%20Automated%20Planning%20-%20Lecture%207%20-%20Planning%20Graph%20Heuristics.pdf [Accessed: 14- Dec- 2022].
- [3] Geeks for Geeks, "How to check if an instance of 15 puzzle is solvable?" 2021 [Online]. Available: <https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/> [Accessed: 14- Dec- 2022].
- [4] D.Ajwani, R.Dementiev, U.Meyer, V.Osipov, "Breadth first search on massive graphs" [Online]. Available: <http://www.diag.uniroma1.it/challenge9/papers/ajwani.pdf> [Accessed: 14- Dec- 2022].
- [5] A.Adnan, Department of Computer Engineering Techniques Artificial Intelligence , "Best-First Search Algorithm & Greedy Search Algorithm" 2014 [Online]. Available: https://www.uomus.edu.iq/img/lectures21/MUCLecture_2021_123356.pdf [Accessed: 14- Dec- 2022].
- [6] Educative Answer Team, "What is the A* algorithm?" [Online]. Available: <https://www.educative.io/answers/what-is-the-a-star-algorithm> [Accessed: 14- Dec- 2022].
- [7] R.Aware, "Solve the Slide Puzzle with Hill Climbing Search Algorithm" 2017 [Online]. Available: <https://towardsdatascience.com/solve-slide-puzzle-with-hill-climbing-search-algorithm-d7fb93321325> [Accessed: 14- Dec- 2022].

Distribution of Work

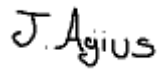
We all worked out the project together and contributed an equal amount of work.

Signatures:

Matthias Bartolo



Jerome Agius



Isaac Muscat



Plagiarism Declaration Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

☒ We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is ~~my~~ / our* work, except where acknowledged and referenced.

☒ We* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Jerome Agius
Student Name

J Agius
Signature

Matthias Bartolo
Student Name

MB Bartolo
Signature

Isaac Muscat
Student Name

IMuscat
Signature

Student Name

Signature

ARI2101
Course Code

ARI2101 Group Assignment
Title of work submitted

13/01/2023
Date