International Islamic University Chittagong

Lab Report On

Course Code: CSE-3636

Course Title: Artificial Intelligence

Spring 2024

Submitted By

Name: Maimuna Akter Shawon

ID: C213236

Semester: 6th

Section: 6BF

Submitted To

Tashin Hossain

Adjunct Lecturer

Department of CSE, IIUC

Lab Report on Artificial Intelligence Lab

Maimuna Akter Shawon Id:C213236

Artificial intelligence, or AI, refers to the simulation of human intelligence by softwarecoded heuristics. Nowadays this code is prevalent in everything from cloud-based, enterprise applications to consumer apps and even embedded firmware.

1 Breadth First Search

1.1 Introduction

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It involves visiting all the connected nodes of a graph in a level-by-level manner.

1.2 Methodology

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors.

1.3 Application of BFS

Here are some common applications of BFS:

1. Shortest Path in Unweighted Graphs
2. Finding Connected Components
3. Finding Minimum Spanning Tree
4. Detecting Bipartite Graphs

1.4 Algorithm of BFS

The algorithm for the BFS:

1. Initialization: Enqueue the starting node into a queue and mark it as visited.
2. Exploration: While the queue is not empty: Dequeue a node from the queue and visit it . For each unvisited neighbor of the dequeued node: Enqueue the neighbor into the queue. Mark the neighbor as visited.

3. Termination: Repeat step 2 until the queue is empty.

1.5 Working of BFS

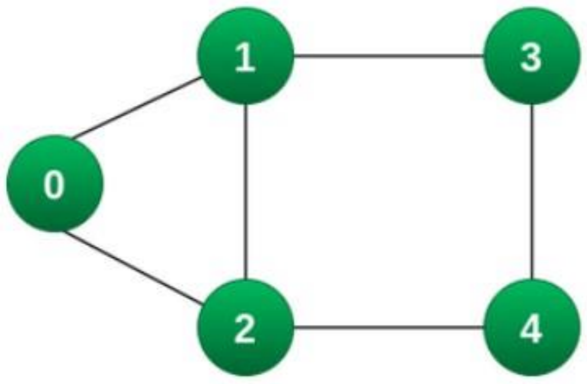


Figure 1: Breadth First Search

1. Initially queue and visited arrays are empty.
2. Push node 0 into queue and mark it visited.
3. Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.
4. Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue
5. Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.
6. Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue
7. Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.
8. Now, Queue becomes empty, So, terminate these process of iteration

1.6 Implementation

Listing1:BreadthFirstSearch

```
graph=    {
      '0':    [ '1',    '2'],
      '1':    [ '2','3','0'],
      '2':    [ '0',    '1','4'],
      '3':    [ '1',    '4'],
      '4':    [ '3','2'],
}


def  bfs(node):


      visited=[False]          *  ( len ( graph  ))


      queue=[]


      visited.append(node)
      queue.append(node)

      while   queue:

            v=queue.pop(0)
            print  ( v,end=" "     )



            for   neigh   in  graph[v]:
                  if   neigh   notin    visited:
                        visited.append(neigh)
                        queue.append(neigh)


#DriverCode
if   --name --  ==   " --main  --":
      print  ( "Breadth First Traversal starting from vertex 0:"                    )
      bfs('0')
```

1.7 Output

Input graph is given below:

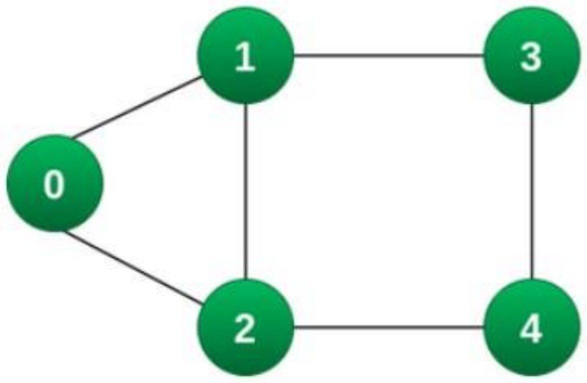


Figure 2: Input Graph

Breadth First Traversal starting from vertex 0: 0 1 2 3 4

1.8 Discussion

The time complexity of BFS is O(V + E), where V and E are the number of vertices and edges in the given graph.The space complexity of BFS is O(V), where V and E are the number of vertices and edges in the given graph.BFS is optimal for finding the shortest path in unweighted or unit-weighted graphs and in terms of the number of nodes visited within a certain depth bound, but it is not optimal for finding the shortest path in weighted graphs with varying edge weights.

2 Uniform Cost Search

2.1 Introduction

The Uniform Cost Search Algorithm is a search algorithm to find the minimum cumulative cost of the path from the source node to the destination node. It is an uninformed algorithm i.e. it doesn’t have prior information about the path or nodes and that is why it is a bruteforce approach.

2.2 Methodology

In this algorithm from the starting state, we will visit the adjacent states and will choose the least costly state then we will choose the next least costly state from the all unvisited and adjacent states of the visited states, in this way we will try to reach the goal state.

2.3 Application of UCS

Here are some of the applications of uniform cost search:

- Path finding in graphs
- Finding the shortest route between two points
- Finding the cheapest way to travel from one city to another
- Scheduling tasks

2.4 Algorithm Of UCS

The algorithm for the above algorithm is given as below:

- Create a priority queue, a boolean array visited of the size of the number of nodes, and a mincost variable initialized with maximum value. Add the source node to the queue and mark it visited.
- Pop the element with the highest priority from the queue. If the removed node is the destination node, check the mincost variable, if the value of the mincost variable is greater than the current cost then update the variable.
- If the given node is not the destination node then add all the unvisited nodes to the priority queue adjacent to the current node.

2.5 Working of UCS

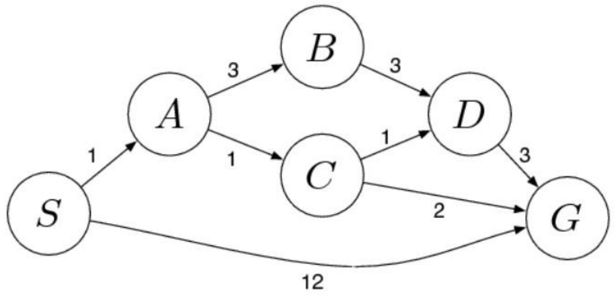


Figure 3: Uniform Cost Search

Here S is the starting state and G is the goal state. We visit the start state S. Then we explore the adjacents state of S and choose the least costly state and keep the track in the open list.

Initialization: $[S,0]$ we explore s and add the adjacent state of s in open list with their cost from s.

Iteration1: $[S->A,1],[S->G,12]$

Then we explore the least costly state A and add the adjacent states of A in the list which are C and B.

Iteration2: $[S->A->C,2],[S->A->B,4],[S->G,12]$

Then we explore least costly state C and add the adjacent state of C in the list which are D and G.

Iteration3: $[S->A->C->D,3],[S->A->B,4],[S->A->C->G,4],[S->G,12]$

Again we explore least costly state D and add the adjacent state of D in the list which is G. Iteration4: $[S->A->B,4],[S->A->C->G,4],[S->A->C->D->G,6],[S->G,12]$

At last we visit the unvisited state B and add the adjacent state of B in the list which is D.

Iteration5: $[S->A->C->G,4],[S->A->C->D->G,6],[S->A->B->D,7]$, $[S->G,12]$

As we our iteration is over ,we choose the least costly path From S to G.

Iteration6 gives the final output as $S->A->C->G$

2.6 Implementation

Listing2:UniformCostSearch

```
  from queue import PriorityQueue

def uniform -cost -search(graph, start, goal):
    visited= set ()
    pq=PriorityQueue()
    pq.put((0, start, []))
    whilenot pq.empty():
        cost,node, path=pq.get()

        if node in visited:
            continue

        visited.add(node)
        path=path+[node]

        if node==goal:
            return path, cost #Return both pathand cost

        for neighbor, neighbor -cost in graph[node]:
            if neighbor notin visited:
                pq.put((cost+neighbor -cost, neighbor, path))

    return None,None

#Example usage:
graph= {
    'S': [( 'A', 1) ,('G', 12)] ,
    'A': [( 'C', 1) ,('B', 3)] ,
    'B': [( 'D', 3)] ,
    'C': [( 'D', 1) ,('G', 2)] ,
    'D': [( 'G', 3)]
}
start -node='S'
goal -node='G'
result -path, result -cost=uniform -cost -search(graph,
start -node, goal -node)
if result -path:
    print ( f"UCS path from { start -node } to { goal -node } :
    { result -path } ")
    print ( f"Cost of the path: { result -cost } ")
else :
    print ( f"No path found from { start -node } to { goal -node } ")
```

2.7 Output

For this input graph, UCS path from S to G: ['S', 'A', 'C', 'G']

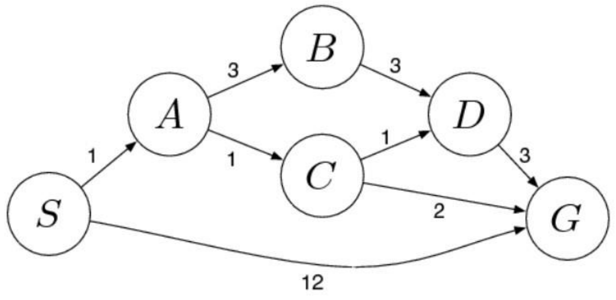


Figure 4: Uniform Cost Search

Cost of the path is: 4

2.8 Discussion

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.Uniformcost search is complete, such as if there is a solution, UCS will find it.The worst-case time complexity of Uniform-cost search is $O(b1 + [C * /])$.The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is O(b1 + [C*/]).

3 Greedy Best First Search

3.1 Introduction

Greedy Best-First Search is an informed search algorithm that attempts to find the most promising path from a given starting point to a goal..

3.2 Methodology

Greedy Best-First Search is an AI search algorithm that attempts to find the most promising path from a given starting point to a goal. It prioritizes paths that appear to be the most promising, regardless of whether or not they are actually the shortest path. The algorithm works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.

3.3 Application of GBFS Algorithm

Pathfinding: Greedy Best-First Search is used to find the shortest path between two points in a graph. It is used in many applications such as video games, robotics, and navigation systems.

Machine Learning: Greedy Best-First Search can be used in machine learning algorithms to find the most promising path through a search space.

Optimization: Greedy Best-First Search can be used to optimize the parameters of a system in order to achieve the desired result.

Game AI: Greedy Best-First Search can be used in game AI to evaluate potential moves and chose the best one.

3.4 Algorithm of GBFS

- Greedy Best-First Search works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.
- The algorithm uses a heuristic function to determine which path is the most promising.
- The heuristic function takes into account the cost of the current path and the estimated cost of the remaining paths.
- If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.

3.5 Working of GBFS

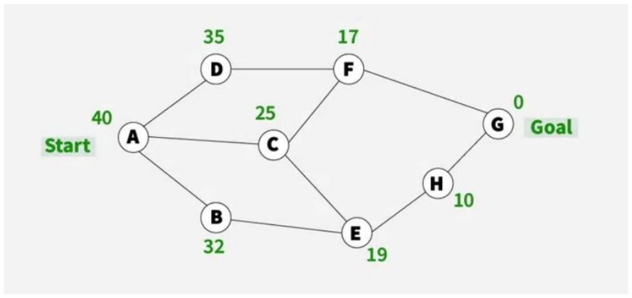


Figure 5: Greedy Best First Search

1) We are starting from A , so from A there are direct path to node B(with heuristics value of 32) , from A to C (with heuristics value of 25) and from A to D(with heuristics value of 35) .
2) So as per best first search algorithm choose the path with lowest heuristics value , currently C has lowest value among above node . So we will go from A to C.
3) Now from C we have direct paths as C to F(with heuristics value of 17) and C to E(with heuristics value of 19) , so we will go from C to F.
4) Now from F we have direct path to go to the goal node G (with heuristics value of 0) , so we will go from F to G.
5) So now the goal node G has been reached and the path we will follow is A,C,F,G.

3.6 Implementation

Listing3:GreedyBestFirstSearch

```
import heapq
romania -graph= {
    'A': { 'B': 9,'C':10,'D' :8 } ,
        'B': { 'E':8 } ,
        'C': { 'E': 16,'F' : 14} ,
        'D': { 'F': 10} ,
        'F': { 'G': 1} ,
        'E': { 'H': 9} ,
        'H': { 'G':4 }}
heuristic -values= {
   'A': 40,'B' : 32,'C' : 25,'D' : 35,'E' : 19,'F' : 16,'H' : 10,
   'G':0
}
#GreedyBest −First Search algorithm with path tracking
def greedy -best -first -search(start, goal):
    open -list=[(heuristic -values[start], start, 0)]
#Initial state: heuristic value, city, path cost
    closed -set= set ()
    parent -nodes= {} #For backtracking
    while open -list:
        - , current -city, current -cost=heapq.heappop(open -list)
#Popthe city with lowest heuristic value
        if current -city==goal:
            path=construct -path(parent -nodes, current -city)
            return path, current -cost
        closed -set.add(current -city)
```

```
            for   next -city,    cost   inromania   -graph[current    -city].items():
                  if   next -city    notin     closed  -set:
                        heapq.heappush(open     -list,
                        ( heuristic    -values[next     -city],
                        next  -city,     current   -cost+cost))
                        parent  -nodes[next    -city]=current         -city
#Updateparentnodefor              backtracking

      returnNone,None          #Goalnotreachable
#Functiontobacktraceandconstruct                    thepath
def   construct   -path(parent    -nodes,    goal):
      path=[goal]
      while    goal    in   parent  -nodes:
            goal=parent       -nodes[goal]
            path.insert(0,        goal)
      returnpath

#Exampleusage:
if   --name --  ==   ' --main -- ':
      start   -city='A'
      goal  -city='G'
      path,    path  -cost=greedy       -best  -first  -search(start      -city,
      goal  -city)
      if   path    is   notNone:
            print("Pathfrom        {}   to   {}  :   {}  ".format(start      -city,
            goal  -city,     path))
            print("Pathcost:          {}  ".format(path      -cost))
      else:
            print("Goalcity          notreachable.")
```

3.7 Output

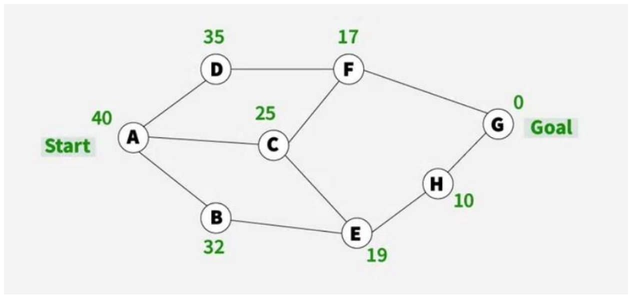


Figure 6: Input Graph

Path from A to G: ['A', 'C', 'F', 'G']

Path cost: 25

3.8 Discussion

Greedy Best-First Search has several advantages, including being simple and easy to implement, fast and efficient, and having low memory requirements. However, it also has some disadvantages, such as inaccurate results, local optima, and requiring a heuristic function. Greedy Best-First Search is used in many applications, including pathfinding, machine learning, and optimization. It is a useful algorithm for finding the most promising path through a search space.

4 A* Search

4.1 Introduction

A* Search is an informed best-first search algorithm that efficiently determines the lowest cost path between any two nodes in a directed weighted graph with non-negative edge weights. This algorithm is a variant of Dijkstra's algorithm. A slight difference arises from the fact that an evaluation function is used to determine which node to explore next.

4.2 Methodology

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

4.3 Application of A* search

Here are some of the applications of A* search:

- Resource allocation
- Job shop scheduling
- Vehicle routing
- Network flow problems

4.4 Algorithm of A* Search

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

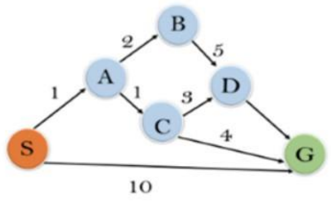
Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

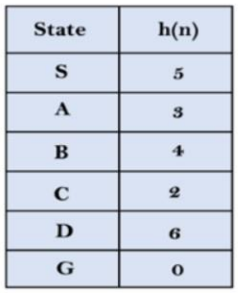
Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to Step 2.

4.5 Working of A* search





State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Figure 7: A* Search

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state. Here we will use OPEN and CLOSED list.

Initialization: (S, 5)

Iteration1: $(S- > A,4),(S- > G,10)$

Iteration2: $(S- > A- > C,4),(S- > A- > B,7),(S- > G,10)$

Iteration3: $(S- > A- > C- > G,6),(S- > A- > C- > D,11),(S- > A- > B,7),(S- > G,10)$

Iteration 4 will give the final result, as $S- > A- > C- > G$it provides the optimal path with cost 6.

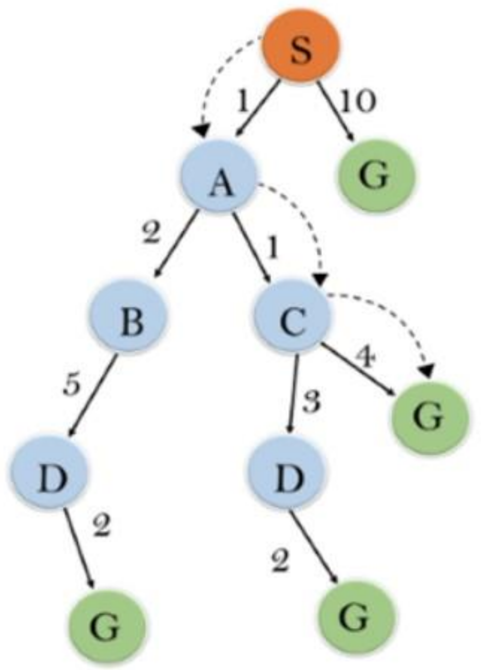


Figure 8: Solution tree of A* Search

4.6 Implementation

Listing4:A*Search

```
  from  queue  import  PriorityQueue
GRAPH=  {
      'S':   { 'A':   1 ,  'G':   10} ,
      'A':   { 'C':   1 ,  'B':   2} ,
      'B':   { 'D':   5} ,
      'D':   { 'G':   2} ,
      'C':   { 'D':   3 ,  'G':   4} ,
      'G':   { 'S':   10 ,'D'  :2  }}
 def  dfs -paths(source,     destination,     path=None):
      """All   possible   paths  from  source  to  destination     using
      depth −first   search"""
      if  path  is  None:
           path=[source]
      if  source==destination:
           yield   path
      for  next -node  in  GRAPH[source]:
           if  next -node  notin   path:
                yield  from  dfs -paths(next   -node,   destination,
                path+[next    -node])
```

def ucs (source ,   destination):

```
    ”””Cheapest path from source to destination using uniform cost
    search ””” priority queue , visited = PriorityQueue () , {} priority
    queue . put ((0 , source , [ source ])) visited [ source ] = 0 while not
    priority queue . empty (): cost , vertex , path = priority queue . get ()
        i f      vertex == destination :
            return    cost ,    path
        for    next node ,      edge cost       in GRAPH[ vertex ] . items ():
# Iterate directly over key−value pairs current cost = cost + edge cost i f next node not in
            visited or visited [ next node ] >=
            current cost :
                visited [ next node ] = current cost priority queue . put ((
                current cost , next node ,
                path + [ next node ]))
def    a star ( source ,       destination ):
    ”””Optimal path from source to destination using straight line
    distance heuristic ””” straight line = {
        ’S ’ :   5 ,   ’A’ :   3 ,   ’B’ :   4 ,   ’C’ :   2 ,   ’D’ :    6 , ’G’ :     0 }

    priority queue , visited = PriorityQueue () , {} priority queue . put ((
    straight line [ source ] , 0 , source ,
    [ source ])) visited [ source ] = straight line [ source ] while not priority queue
    . empty (): heuristic , cost , vertex , path = priority queue . get ()
        i f      vertex == destination :
            return     heuristic ,     cost ,    path
        for    next node ,      edge cost       in GRAPH[ vertex ] . items ():
            current cost = cost + edge cost heuristic = current cost + straight line [
            next node ] i f next node not in visited or visited [ next node ] >= heuristic
            :
                visited [ next node ] = heuristic priority queue . put ((
                heuristic , current cost , next node , path + [ next node ]))
```

```
defmain():
        ”””Mainfunction”””
      #print(’ENTERSOURCE:’,end=’                    ’)
        source=’S’
        goal=’G’
        if    sourcenotinGRAPHorgoalnotinGRAPH:
                print(’ERROR:CITYDOESNOTEXIST.’)
        else:
                print(’    \nALLPOSSIBLEPATHS:’)
                paths=dfs        -paths(source,         goal)
                forpathinpaths:
                        print(’      −>    ’.join(city          for    city     inpath))
                print(’    \nCHEAPESTPATH:’)
                cost,      cheapest   -path=ucs(source,              goal)
                if    cheapest   -path:
                        print(’PATHCOST=’,cost)
                        print(’      −>    ’.join(city          for    city     in    cheapest   -path))
                else:
                        print(’Nopathfound.’)

                print(’    \nOPTIMALPATH:’)
                heuristic,        cost,      optimal   -path=a       -star(source,         goal)
                if    optimal   -path:
                        print(’HEURISTIC=’,             heuristic)
                        print(’PATHCOST=’,cost)
                        print(’      −>    ’.join(city          for    city     in    optimal   -path))
                else:
                        print(’Nopathfound.’)
if    --name --  ==    ’ --main  -- ’:
        main()
```

4.7 Output

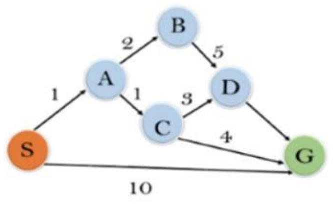
Input Graph:

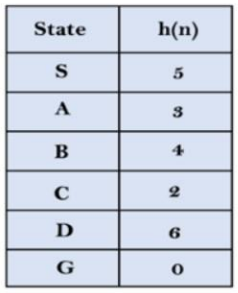
ALL POSSIBLE PATHS: $S- > A- > C- > D- > G//S- > A- > C- > G//S- > A- > B- > D- > G//S- > G$

CHEAPEST PATH: PATH COST = 6 $S- > A- > C- > G$

OPTIMAL PATH: HEURISTIC = 6

PATH COST = 6 $S- > A- > C- > G$





State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Figure 9: Input Graph

4.8 Discussion

The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is $O(b^d)$, where b is the branching factor.The space complexity of A* search algorithm is O(bd).A* algorithm is complete as long as: 1)Branching factor is finite. 2) Cost at every action is fixed.A* search algorithm is optimal if it follows below two conditions:

Admissible: the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.

Consistency: Second required condition is consistency for only A* graph-search.

5 Greedy Hill Climbing Searching Algorithm

5.1 Introduction

The hill climbing search algorithm is a local search algorithm used for optimization problems. It is designed to find the highest point or the best solution within a given search space by iteratively exploring neighboring solutions.

5.2 Methodology

A hill-climbing algorithm has four main features:

1. It employs a greedy approach: This means that it moves in a direction in which the costfunction is optimized. The greedy approach enables the algorithm establish local maxima or minima.

2. No Backtracking: A hill-climbing algorithm only works on the current state and succeedingstates (future). It does not look at the previous states.

3. Feedback mechanism: The algorithm has a feedback mechanism that helps it decide on the direction of movement (whether up or down the hill). The feedback mechanism is enhanced through the generate-and-test technique. 4. Incremental change: The algorithm improves the current solution by incremental changes.

5.3 Application of Hill Climbing

Hill Climbing technique can be used to solve many problems, where the current state allows for an accurate evaluation function, such as

- Network-Flow
- Traveling Salesman problem
- 8-Queens problem
- Integrated Circuit design

5.4 Algorithm of Hill Climbing

The hill climbing algorithm follows a simple iterative process to search for the best solution:

1. Initialization: Start with an initial solution within the search space.
2. Evaluation: Evaluate the quality of the current solution using an objective function or fitness measure.
3. Neighbor Generation: Generate neighboring solutions by making small modifications to the current solution.
4. Selection: Choose the best neighboring solution based on its objective function value.
5. Comparison: Compare the objective function value of the best neighboring solution with the current solution.
6. Iteration: Repeat the process until a termination condition is met.

.

5.5 Working of Hill Climbing

Here I implement a solution to the 8-Queens problem using the Hill Climbing algorithm. I'll provide an example of how it works.

1. generaterandomboard(): This function generates a random initial state for the 8Queens problem. It creates a list representing the positions of the queens on the board, with each element representing the column index of a queen.
2. computecost(board): This function calculates the number of conflicts (queens threatening each other) in the board. It iterates over all pairs of queens and checks if they are in the same row, same column, or same diagonal. If they are, it increments the conflict count.
3. hillclimbing(): This function solves the 8-Queens problem using the Hill Climbing algorithm. It starts with a random initial board state and iteratively improves it until it

reaches a state where no queens threaten each other. In each iteration, it generates all neighboring states (boards) by moving one queen to another column in its row. It selects the best neighboring state (the one with the lowest number of conflicts) and moves to it if it improves the current state. If no better state is found, it restarts the search with a new random board.

4. printsolution(board): This function prints the solution board in a human-readable format, with 'Q' representing a queen and '.' representing an empty space.

5.6 Implementation

Listing5:GreedyHillClimbingSearch

```
import random
def generate_initial_state():
    return random.sample( range (8) , 8)
def evaluate(state):
    conflicts=0
    for i inrange (8):
        for j inrange ( i+1, 8):
            if state[i]==state[j] orabs ( state[i ] − state[j])
            == abs ( i − j):
                conflicts+=1
    return conflicts
def hill_climbing():
    current_state=generate_initial_state()
    current_conflicts=evaluate(current_state)
```

Listing6:GreedyHillClimbingSearch

```
        while current-conflicts > 0:
            next-states=[]
            next-conflicts=[]

            for col inrange (8):
                for row inrange (8):
                    if current-state[col]!=row:
                        new-state= list ( current-state)
                        new-state[col]=row
                        next-states.append(new-state)
                        next-conflicts.append(evaluate(new-state))

            min-conflicts= min( next-conflicts)
            if min-conflicts >= current-conflicts:
                current-state=generate-initial-state()
                current-conflicts=evaluate(current-state)
                continue
#print(current-state)
            min-indices=[i for i,x inenumerate ( next-conflicts)
            if x==min-conflicts]
            min-index=random.choice(min-indices)
            current-state=next-states[min-index]
            current-conflicts=min-conflicts
          #print(current-state)
           #print(current-conflicts)

        return current-state, current-conflicts

 def print-board(state):
        for row inrange (8):
            line=””
            for col inrange (8):
                if state[col]==row:
                    line+=”Q ”
                else :
                    line+=”. ”
            print ( line )
        print ()

 if --name-- == ”--main--”:
        solution, conflicts=hill-climbing()
        print (”Solution found with”, conflicts,”conflicts:”)
        print-board(solution)
```

5.7 Output

Solution found with 0 conflicts:

```
. . . . . Q . .
. . . Q . . . .
. . . . . . Q . Q . . . .
. . .
. . Q . . . . .
. . . . Q . . .
. Q . . . . . .
. . . . . . . Q
```

5.8 Discussion

Completeness: Hill Climbing is not complete because it can get stuck in local optima, failing to find the global optimum solution.

Optimality: Hill Climbing is not guaranteed to find the optimal solution. It may settle for a suboptimal solution if it gets trapped in a local maximum without exploring further.

Time Complexity: The time complexity of Hill Climbing depends on the problem and the state space. In the worst case, it can be exponential, especially if the number of possible states is large and there are many local optima.

Space Complexity: The space complexity of Hill Climbing depends on how states are represented and stored. However, it typically requires storing only a small number of states at a time.

6 Min-Max Algorithm

6.1 Introduction

The minimax algorithm in game theory helps the players in two-player games decide their best move. It is crucial to assume that the other player is also making the best move while determining the best course of action for the present player

6.2 Methodology

The Min Max algorithm is a decision-making algorithm used in the field of game theory and artificial intelligence. It is used to determine the optimal move for a player in a two-player game by considering all possible outcomes of the game. The algorithm helps in selecting the move that minimizes the maximum possible loss.

6.3 Application of Min-Max Algorithm

The Min Max algorithm has many applications in game AI, decision-making, and optimization.

6.4 Algoritm of Min-Max Algorithm

Algorithm of Min-Max Algorithm is given below:

1. Define the Game State: Represent the current state of the game, including the positions of pieces, scores, or any other relevant information.

2. Generate Possible Moves: For the current player, generate all possible moves that can be made from the current game state.

3. Evaluate Terminal States: If the game has ended (e.g., someone has won or it's a draw), evaluate the utility or score of the terminal state.

4. Recursive Search: If the game has not ended, recursively evaluate the possible outcomes of each move. This involves switching between players (one player tries to maximize the score while the other tries to minimize it) and exploring deeper into the game tree.

5. Backtrack: Once the recursion reaches a terminal state or a certain depth limit, backtrack to the previous level of the tree, updating scores and choosing the move that leads to the best outcome for the current player (maximizing or minimizing depending on whose turn it is).

6. Repeat: Repeat steps 2-5 until a certain depth is reached or a time limit is exceeded.

6.5 Working of Min-Max Algorithm

The Min Max algorithm recursively evaluates all possible moves the current player and the opponent player can make. It starts at the root of the game tree and applies the MinMax algorithm to each child node. At each level of the tree, the algorithm alternates between maximizing and minimizing the node's value. The player who will be winning the game is the maximizing player, whereas the player who will be losing the game is the minimizing player. The maximizing player chooses the child node with the highest value, while the minimizing player chooses the child node with the lowest value. This move is considered the optimal move for the player. The algorithm evaluates the child nodes until it reaches a terminal node or a predefined depth. When it reaches a terminal node, the algorithm returns the heuristic value of the node. The heuristic value is a score that represents the value of the game's current state

6.6 Implementation

Listing7:Min-MaxAlgorithm

```
        maximum,minimum=1000,     −1000

def fun -minimax(d,node,maxP,v):
    if d==3:
        return v[node]
    if maxP:
        best=minimum
        for i inrange (0 , 2):
            value=fun -minimax(d+1,node * 2+ i, False,v)
            best= max( best, value)
        return best
    else :
        best=maximum
        for i inrange (0 , 2):
            value=fun -minimax(d+1,node * 2+ i,True,v )
            best= min ( best, value)
        return best

scr=[]
x= int ( input ( ”Enter total number of leaf node: ” ))
for i inrange ( x ):
    y= int ( input ( ”Enter node value: ” ))
    scr.append(y)

d= int ( input ( ”Enter depth value: ” ))
node= int ( input ( ”Enter node value: ” ))

print ( ”The optimal value is:”,fun -minimax(d,node,True, scr))
```

6.7 Output

```
Enter total number of leaf node: 4
Enter node value: 3
Enter node value: 5
Enter node value: 2
Enter node value: 9
Enter depth value: 1
Enter node value: 0 The optimal value is: 3
```

6.8 Discussion

The time complexity of the Minimax algorithm depends on the branching factor of the game tree and the maximum depth of the search. In the worst-case scenario, where the entire game tree needs to be explored, the time complexity is exponential in the depth of the tree.The space complexity of Minimax depends on the size of the game tree and the amount of memory required to store the nodes and their associated values during the search. It is also exponential in the depth of the tree in the worst case.Minimax is optimal under the assumption that both players play optimally and have perfect information about the game

7 Alpha Beta Pruning

7.1 Introduction

Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.Alpha-beta pruning is a search algorithm commonly used in game-playing scenarios,such as chess or tic-tac-toe. Its primary goal is to reduce the number of nodes evaluated in the search tree, thereby improving the efficiency of the search process.

7.2 Methodology

Alpha Beta Pruning is an optimization technique of the Minimax algorithm. This algorithm solves the limitation of exponential time and space complexity in the case of the Minimax algorithm by pruning redundant branches of a game tree using its parameters Alpha and Beta.

7.3 Application of Alpha Beta Pruning

It is an adversarial search algorithm used commonly for machine playing of two-player combinatorial games (Tic-tac-toe, Chess, Connect 4, etc.).

7.4 Algorithm of Alpha Beta Pruning

1. Alpha: At each point along the Maximizer path, Alpha is the best option or the highest value we've discovered. -(infinity) is the initial value for alpha.
2. Beta: At every point along the Minimizer route, Beta is the best option or the lowest value we've identified. The value of beta is initially set to +infinity.
3. The condition for Alpha-beta Pruning is Alpha=>Beta.
4. Alpha is updated only when it's MAX's time, and Beta can only be updated when it's MIN's turn.

5. The MAX player will only update alpha values, whereas the MIN player will only update beta values.

6. During the reversal of the tree, node values will be transferred to upper nodes instead of alpha and beta values.

7. Only child nodes will get Alpha and Beta values.

7.5 Working of Alpha Beta pruning

I implement alpha beta pruning code which is given in Implementation section. Here,I explain the code:

- MIN and MAX are initialized to very large negative and positive values, respectively, to represent negative and positive infinity.

- The minimax function is defined recursively. It evaluates the optimal value for the current player at a given depth in the game tree. The base case (depth == 3) returns the value of the leaf node.

- If it's the maximizing player's turn, it iterates over the child nodes and selects the one with the maximum value. It updates alpha and beta values accordingly.

- If it's the minimizing player's turn, it does the opposite, selecting the child node with the minimum value.

- Alpha-beta pruning is applied by breaking out of the loop if the beta value is less than or equal to the alpha value, as this indicates that further exploration of child nodes is unnecessary.

- The driver code initializes the values array and calls the minimax function with the initial parameters.

7.6 Implementaion

Listing8:AlphaBetaPruning

```
        #Initial   values  of  AlphaandBeta
MAX,MIN=10000000,    −100000000
#Defining   the  minimax  function   that  returns  the  optimal
 value  for  thecurrent   player
def  minimax(depth,   index,   maximizingPlayer,    values,   alpha,   beta):
    #Terminating    condition
    if  depth==3:
        return  values[index]

    if  maximizingPlayer:
        optimum=MIN

        #Recursion    for  left  and  right  children
        for  i  inrange  (0 ,  2):
            val=minimax(depth+1,index        *  2+  i,  False,
            values,   alpha,   beta)
            optimum=   max( optimum,  val)

            alpha=   max( alpha,optimum   )
            #AlphaBetaPruning       condition
            if  beta  <=  alpha  :
                break
        return  optimum
     else  :
        optimum=MAX
        #Recursion    for  left  and  right  children
        for  i  inrange  (0 ,  2):
            val=minimax(depth+1,index        *  2+  i,True,
            values,   alpha,   beta)
            optimum=   min ( optimum,  val)
            beta=   min ( beta,optimum   )
            #AlphaBetaPruning
            if  beta  <=  alpha  :
                break
        return  optimum
#DriverCode
 if  --name --  ==  ” --main  --”:
    values=[2,3,5,9,0,1,7,5]
    print  ( ”The value is :”,minimax(0,0,True,
    values,MIN,MAX))
```

7.7 Output

Input Graph:

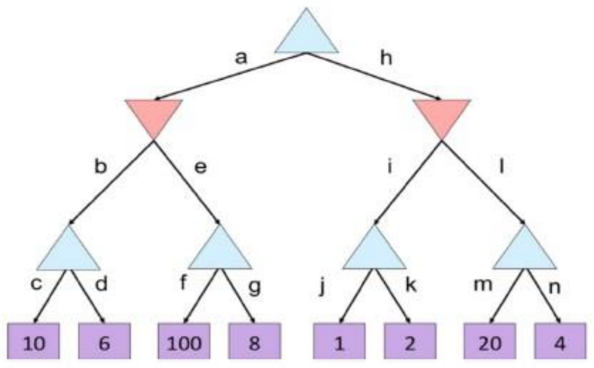


Figure 10: Input Tree

The value is : 10

7.8 Discussion

Alpha-Beta Pruning maintains completeness in the sense that it guarantees to find the optimal solution if one exists within the search space defined by the game tree.Alpha-Beta Pruning preserves optimality by correctly identifying and retaining the best moves for the current player while discarding inferior moves.

8 Genetic Algorithm

8.1 Introduction

Genetic algorithms in the domain of artificial intelligence are heuristic search and optimization techniques inspired by the process of natural selection observed in biological populations. Utilizing the principles of genetics and evolution, genetic algorithms iteratively improve candidate solutions to solve complex problems. These algorithms are particularly effective for optimization, search, and machine learning tasks.

8.2 Methodology

The basic processes which are involved in genetic algorithms are as follows:

- A population of solutions is built to any particular problem. The elements of the population compete with each other to find out the fittest one.
- The elements of the population that are fit are only allowed to create offspring (better solutions).
- The genes from the fittest parents (solutions) create a better offspring. Thus, future solutions will be better and sustainable.

8.3 Application of Genetic Algorithm

Genetic Algorithms are most commonly used in optimization problems wherein we have to maximize or minimize a given objective function value under a given set of constraints.

8.4 Algorithm of Genetic Algorithm

1. Initialization: A population of potential solutions is randomly generated to represent the first generation.
2. Fitness Evaluation: Each solution in the population is evaluated based on a predefined fitness function, which measures how well it solves the problem at hand.
3. Selection: Solutions are selected for reproduction based on their fitness. The fitter individuals are more likely to be chosen, simulating the "survival of the fittest."
4. Crossover: Pairs of selected solutions undergo genetic crossover, exchanging parts of their genetic information to create new offspring. Mutation: Some of the new solutions undergo random changes, or mutations, to introduce genetic diversity.
5. Replacement: The new generation, now composed of both parents and offspring, replaces the previous generation.
6. Termination: The algorithm repeats these steps for multiple generations or until a satisfactory solution is found.

8.5 Working of Genetic Algorithm

The working of a genetic algorithm in AI is as follows:

- The components of the population, i.e., elements, are termed as genes in genetic algorithms in AI. These genes form an individual in the population (also termed as a chromosome).
- A search space is created in which all the individuals are accumulated. All the individuals are coded within a finite length in the search space. Each individual in the search space (population) is given a fitness score, which tells its ability to compete with other individuals.
- All the individuals with their respective fitness scores are sought and maintained by the genetic algorithm and the individuals with high fitness scores are given a chance to reproduce.
- The new offspring are having better 'partial solutions' as compared to their parents. Genetic algorithms also keep the space of the search space dynamic for accumulating the new solutions (offspring).
- This process is repeated until the offsprings do not have any new attributes/features than their parents (convergence). The population converges at the end, and only the fittest solutions remain along with their offspring (better

solutions). The fitness score of new individuals in the population (offspring) are also calculated.

8.6 Implementation

```
            #Python3programto          create     target     string,      startingfrom
#randomstring          usingGeneticAlgorithm

 importrandom


 POPULATION    SIZE=200

#Validgenes
GENES='''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ'''

#Target       string     tobegenerated
TARGET="GeneticAlgorithm"

 class     Individual(object):
       '''
       Class     representing       individual       in    population
       '''
       def    - - init   - - ( self,chromosome        ):
             self.chromosome=chromosome
             self.fitness=self.cal                    - fitness()

       @classmethod
       defmutated      -genes(self):
             '''
             createrandomgenes           for    mutation
             '''
             globalGENES
             gene=random.choice(GENES)
            #print("GENE:",gene)
             returngene

       @classmethod
       defcreate     -gnome(self):
             '''
             createchromosomeor           string      of    genes
             '''
             globalTARGET
             gnome  -len=len(TARGET)
             return     [ self.mutated      -genes()     for    -    inrange(gnome       -len)]
```

Listing9:GeneticAlgorithm

```
        def   mate(self,        par2):
              ’’’
              Performmatingandproducenew                  offspring        ’’’
              #chromosome       for    offspring
              child  -chromosome=[]
              for    gp1,gp2       inzip    ( self.chromosome,         par2.chromosome):
                  #random      probability
                    prob=random.random()
                    if    prob  <    0.45:
                          child  -chromosome.append(gp1)
                    elif    prob  <    0.90:
                          child  -chromosome.append(gp2)
                          #otherwise         insert      randomgene(mutate),
                    #for       maintaining       diversity
                    else  :
                          child  -chromosome.append(self.mutated              -genes())

              #createnew          Individual(offspring)              using
              #generated        chromosome     for    offspring
              return    Individual(child        -chromosome)

      def    cal  -fitness(self):

            #Calculate         fitness       score,       it    is    the    number    of
            characters       in    stringwhich         differ      from    target      string.

              global    TARGET
              fitness=0
              for    gs,    gt    inzip    ( self.chromosome,TARGET        ):
                    if    gs!=gt:          fitness+=1
              return    fitness

#Driver        code
def   main():
      global    POPULATION    SIZE

      #current      generation
      generation=1

      found=False
      population=[]

      #create        initial        population
      for    -   inrange    (POPULATION    SIZE):
                      gnome=Individual.create             -gnome()
                      population.append(Individual(gnome))
        whilenotfound:

             #sort     thepopulation       in   increasing     order   of   fitness     score
              population=sorted(population,                 key=lambdax:x.fitness)

             #if     the   individual     havinglowest       fitness     score    ie.
             #0thenweknowthatwehavereachedtothe                          target
             #andbreaktheloop
              if   population[0].fitness            <=0:
                    found=True
                    break

             #Otherwisegeneratenewoffsprings                    fornewgeneration
              new -generation=[]

             #PerformElitism,            thatmean10%of         fittest       population
             #goestothenextgeneration
              s=int((10       * POPULATION   SIZE)/100)
              new -generation.extend(population[:s])

             #From50%of        fittest      population,      Individuals
             #willmatetoproduce                 offspring
              s=int((90       * POPULATION   SIZE)/100)
              for    -   inrange(s):
                    parent1=random.choice(population[:50])
                    parent2=random.choice(population[:50])
                    child=parent1.mate(parent2)
                    new -generation.append(child)

              population=new         -generation

              print("Generation:          {} \ tString:      {} \ tFitness:      {}  ".
                    format(generation,          "".join(population[0].chromosome),
                    population[0].fitness))

              generation+=1
    print("Generation:          {} \ tString:      {} \ tFitness:      {}  ".
              format(generation,
              "".join(population[0].chromosome),
              population[0].fitness))

  if   --name --  ==   ' --main  -- ':
        main()
```

8.7 Output

```
Generation: 1 String: NQnrizjgeiOWho L Fitness: 15
```

```
Generation: 2 String: TrpwPpOpAlFonGbLq Fitness: 14
Generation: 3 String: GoyiricFDYOXWqgm Fitness: 13
Generation: 4 String: Pxnewic HyocoDIm Fitness: 10
Generation: 5 String: teILPic AlgMuWKAm Fitness: 9
Generation: 6 String: GxneJic Ki miphm Fitness: 8
Generation: 7 String: GNnewic DlgouJjhm Fitness: 6
Generation: 8 String: GNnewic DlgouJjhm Fitness: 6
Generation: 9 String: Genetic UlgorJtIm Fitness: 3
Generation: 10 String: Genetic UlgorJtIm Fitness: 3 Generation:
11 String: Genetic UlgorJtIm Fitness: 3
Generation: 12 String: Genetic Ulgorkthm Fitness: 2 Generation:
13 String: Genetic AlUorithm Fitness: 1
Generation: 14 String: Genetic AlUorithm Fitness: 1
Generation: 15 String: Genetic Algorithm Fitness: 0
```

8.8 Discussion

The space complexity includes the memory required to store the population of individuals, as well as additional data structures needed for operations such as selection, crossover, and mutation.Generally, the time complexity of a genetic algorithm can be high, especially for large or complex problem spaces. However, they can be effective in finding approximate solutions to complex optimization problems in reasonable time frames.Genetic algorithms are generally used for optimization problems where finding the global optimal solution is not always feasible or necessary due to the complexity of the problem space.

9 Map Coloring Problem

9.1 Introduction

The Map Coloring Problem is a classic problem in graph theory and computer science. The goal is to assign colors to the regions of a map in such a way that no two adjacent regions share the same color, using as few colors as possible

9.2 Methodology

9.3 Application Of Map Coloring Problem

The Map Coloring Problem arises in various real-world scenarios, such as scheduling tasks without conflicts, assigning frequencies to wireless devices without interference, and designing circuit boards.

9.4 Algorithm Of Map Coloring Problem

One of the commonly used algorithms to solve the Map Coloring Problem is the backtracking algorithm.

1. Initiate all the vertices in the graph.
2. Select the node with the highest degree to colour it with any colour.
3. Choose the colour to be used on the graph with the help of the selection colour function so that no adjacent vertex is having the same colour.
4. Check if the colour can be added and if it does, add it to the solution set.
5. Repeat the process from step 2 until the output set is ready

9.5 Working

The algorithm works by iteratively assigning colors to states while ensuring that no neighboring states have the same color. It explores the search space using a backtracking approach:

It starts with the first state in the states list. For each state, it attempts to assign colors until it finds a promising color.

If a promising color is found, it proceeds to the next state.

If no promising color is found, it backtracks to the previous state and tries a different color.

It continues this process until all states are assigned colors or until no valid color assignments are possible

9.6 Implementation

Listing10:MapColoring

```
colors=['Red', 'Blue', 'Green']

states=['wa', 'nt', 'sa', 'q', 'nsw', 'v']

neighbors= {}
neighbors['wa']=['nt', 'sa']
neighbors['nt']=['wa', 'sa', 'q']
neighbors['sa']=['wa', 'nt', 'q', 'nsw', 'v']
neighbors['q']=['nt', 'sa', 'snw']
neighbors['nsw']=['q', 'sa', 'v']
neighbors['v']=['sa', 'nsw']


colors_of_states= {}

def promising(state, color):
    for neighbor in neighbors.get(state):
        color_of_neighbor=colors_of_states.get(neighbor)
        if color_of_neighbor==color:
            return False

    return True

def get_color_for_state(state):
    for color in colors:
        if promising(state, color):
            return color

def main():
    for state in states:
        colors_of_states[state]=get_color_for_state(state)

    print ( colors_of_states)


main()
```

9.7 Output

```
'wa':'Red','nt':'Blue','sa':'Green','q':'Red','nsw':'Blue','v':'Red'
```

9.8 Discussion

The backtracking algorithm used to solve the Map Coloring Problem is complete, meaning it will always find a valid coloring if one exists.The backtracking algorithm may not always find the optimal solution in terms of minimizing the number of colors used. It finds a feasible solution but not necessarily the most efficient one. The time and space complexity of the backtracking algorithm for the Map Coloring Problem depend on the size of the map and the specific implementation. The space complexity also grows with the size of the problem, but it is typically polynomial.