Usman Institute of Technology

Department of Computer Science
Course Code: CS-321
Artificial Intelligence
Spring 2023

Lab 06

Instructor: Dr. Nasir Uddin
E-mail: nuddin@uit.edu

Objective

This experiment demonstrates the use of A* as an efficient Search Strategy. It also develops the concept of admissible and un-admissible heuristics. Also, the experiment provides introduction to adversarial search schemes and uses MiniMax algorithm to plan in such scenarios.

Student Information

Student Name	
Student ID	
Date	

Assessment

Marks Obtained	
Remarks	
Signature	

Usman Institute of Technology

Department of Computer Science
Course Code: CS-321
Artificial Intelligence
Spring 2023

Objective

This experiment demonstrates the use of A* as an efficient Search Strategy. It also develops the concept of admissible and un-admissible heuristics. Also, the experiment provides introduction to adversarial search schemes and uses MiniMax algorithm to plan in such scenarios.

Instructions

You have to perform the following tasks yourselves. Raise your hand if you face any difficulty in understanding and solving these tasks. **Plagiarism** is an abhorrent practice and you should not engage in it.

How to Submit

- Submit lab work using TEAMS.

1 Optimal and Adversarial Search

1.1 A* Search

A* is an informed search algorithm, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied. At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes f(n) = g(n) + h(n), where n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the cheapest path from n to the goal. A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function is problem-specific. If the heuristic function is admissible, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.

1.2 Admissibility Measures:

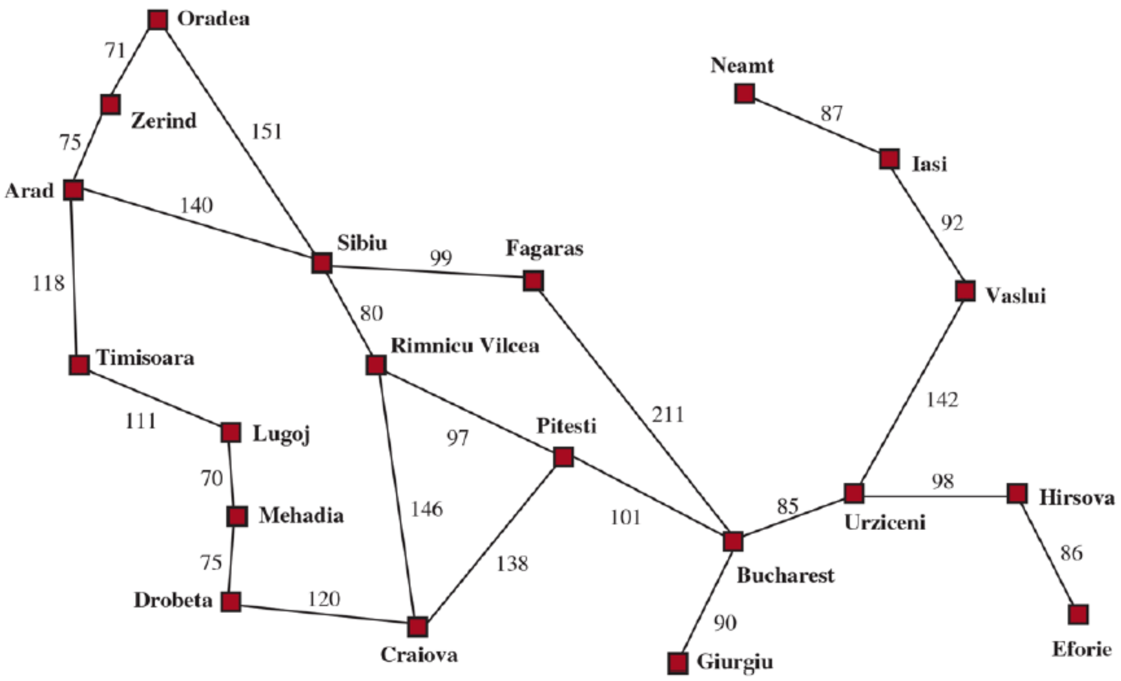
Using the evaluation function f(n) = g(n) + h(n) that was introduced in the last section, we may characterize a class of admissible heuristic search strategies. If n is a node in the state space graph, g(n) measures the depth at which that state has been found in the graph, and h(n) is the heuristic estimate of the distance from n to a goal. In this sense f(n) estimates the total cost of the path from the start state through n to the goal state. In determining the properties of admissible heuristics, we define an evaluation function f*: f*(n) = g*(n) + h*(n) where g*(n) is the cost of the shortest path from the start to node n and h* returns the actual cost of the shortest path from n to the goal. It follows that f*(n) is the actual cost of the optimal path from a start node to a goal node that passes through node n. The resulting search strategy is admissible. Although oracles such as f* do not exist for most real problems, we would like the evaluation function f to be a close estimate of f*. In algorithm A, g(n), the cost of the current path to state n, is a reasonable estimate of g*, but they may not be equal: g(n) >= g*(n). These are equal only if the graph search has discovered the optimal path to state n. Similarly, we replace h*(n) with h(n), a heuristic estimate of the minimal cost to a goal state. Although we usually may not compute h*, it is often possible to determine whether or not the heuristic estimate, h(n), is bounded from above by h*(n), i.e., is always less than or equal to the actual cost of a minimal path. If algorithm A uses an evaluation function f in which h(n) h*(n), it is called algorithm A*.

Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty).

1.3 Pseudo Code for A*

```
    Input: - QUEUE: Path only containing root
    Algorithm: -
      WHILE (QUEUE not empty && first path not reach goal) DO
        Remove first path from QUEUE
        Create paths to all children
        Reject paths with loops
        Add paths and sort QUEUE (by f = cost + heuristic)
        IF QUEUE contains paths: P, Q
          AND P ends in node Ni && Q contains node Ni AND
          cost_P >= cost_Q
          THEN remove P
      IF goal reached THEN success ELSE failure

```

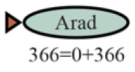
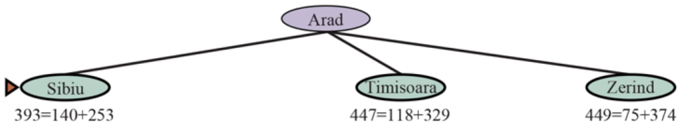
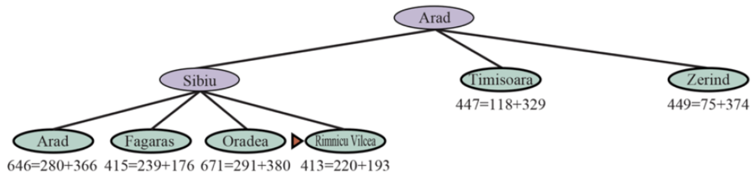
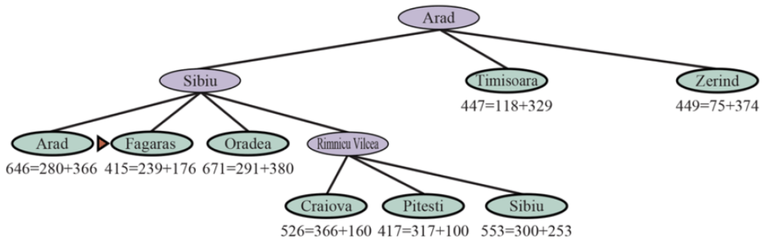
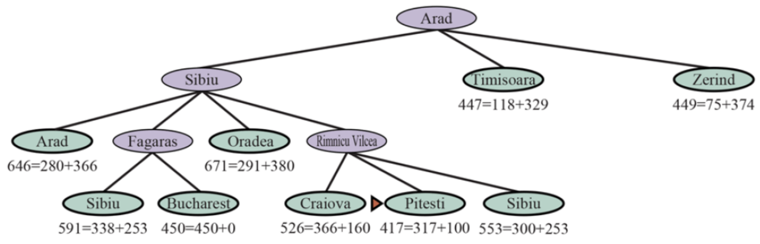


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}—straight-line distances to Bucharest.

Usman Institute of Technology

Department of Computer Science
Course Code: CS-321
Artificial Intelligence
Spring 2023

Step #	Fringe	Explored
1	1. Arad	
2	1. Sibiu 2. Timisoara 3. Zerind	 Arad Sibiu 393=140+253 Timisoara 447=118+329 Zerind 449=75+374
3	1. Rimnicu Vilcea 2. Fagaras 3. Timisoara 4. Zerind 5. Arad 6. Oradea	 Arad Sibiu Timisoara 447=118+329 Zerind 449=75+374 Arad 646=280+366 Fagaras 415=239+176 Oradea 671=291+380 Rimnicu Vilcea 413=220+193
4	1. Fagaras 2. Pitesti 3. Timisoara 4. Zerind 5. Craiova 6. Sibiu 7. Arad 8. Oradea	 Arad Sibiu Timisoara 447=118+329 Zerind 449=75+374 Arad 646=280+366 Fagaras 415=239+176 Oradea 671=291+380 Rimnicu Vilcea Craiova 526=366+160 Pitesti 417=317+100 Sibiu 553=300+253
5	1. Pitesti 2. Timisoara 3. Zerind 4. Bucharest 5. Craiova 6. Sibiu(553) 7. Sibiu(591) 8. Arad 9. Oradea	 Arad Sibiu Timisoara 447=118+329 Zerind 449=75+374 Arad 646=280+366 Fagaras Oradea 671=291+380 Rimnicu Vilcea Sibiu 591=338+253 Bucharest 450=450+0 Craiova 526=366+160 Pitesti 417=317+100 Sibiu 553=300+253

Usman Institute of Technology

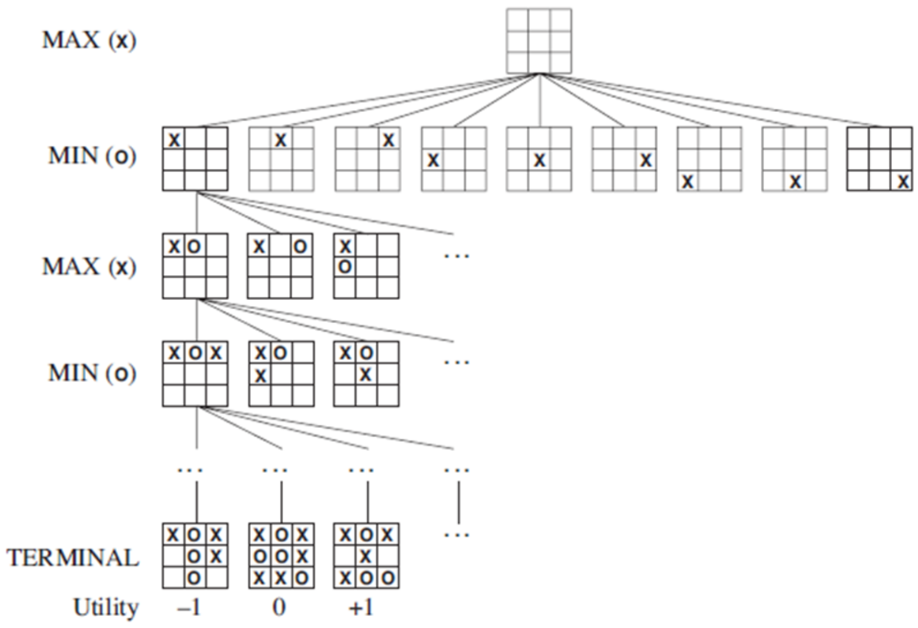
Department of Computer Science
Course Code: CS-321
Artificial Intelligence
Spring 2023

1.4 Adversarial Search

Multi-agent environments, in which each agent needs to consider the actions of other agents and how they affect its own welfare. The unpredictability of these other agents can introduce contingencies into the agent's problem-solving process. If the agents' goals are in conflict it gives rise to adversarial search problems where the adversary is planning against the agent. In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, zero-sum games of perfect information (such as chess). In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial.

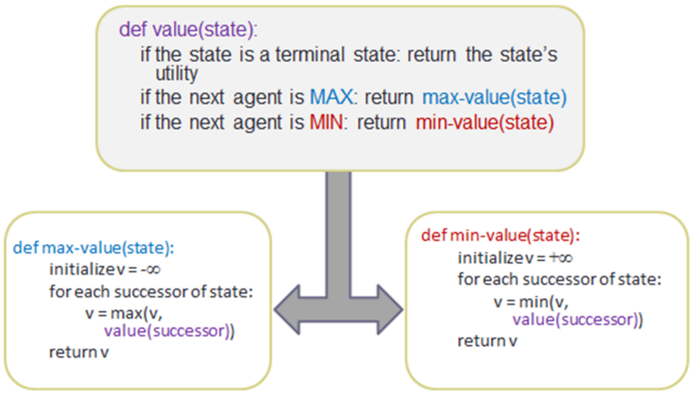
1.5 Minimax Search

The minimax search is especially known for its usefulness in calculating the best move in two player games where all the information is available, such as chess or tic tac toe. It consists of navigating through a tree which captures all the possible moves in the game, where each move is represented in terms of loss and gain for one of the players. It follows that this can only be used to make decisions in zero-sum games, where one player's loss is the other player's gain. Theoretically, this search algorithm is based on von Neumann's minimax theorem which states that in these types of games there is always a set of strategies which leads to both players gaining the same value and that seeing as this is the best possible value one can expect to gain, one should employ this set of strategies.



1.6 Minimax Algorithm

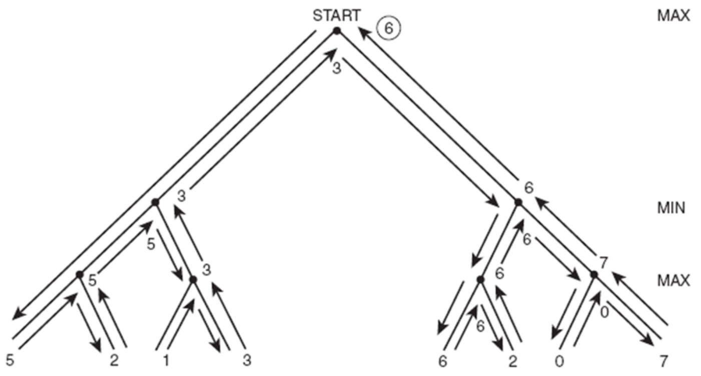
The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is O(bm). The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.



```
def value(state):
    if the state is a terminal state: return the state's
      utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v,
            value(successor))
    return v

def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v,
            value(successor))
    return v
```



1.7 Minimax Algorithm For Tic Tac Toe

For any given board status you will require to check all possible moves you can take from the specific location To check the best move we take the help of minimax() function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally. The code for the maximizer and minimizer in the minimax() function will return a value for the next move. Here is the pseudocode : https://www.youtube.com/watch?v=trKjYdBASyQ

```
function minimax(board, depth, isMaximizingPlayer):

  if current board state is a terminal state :
    return value of the board

  if isMaximizingPlayer :
    bestVal = -INFINITY
    for each move in board :
      value = minimax(board, depth+1, false)
      bestVal = max( bestVal, value)
    return bestVal

  else :
    bestVal = +INFINITY
    for each move in board :
      value = minimax(board, depth+1, true)
      bestVal = min( bestVal, value)
    return bestVal

```

Usman Institute of Technology

Department of Computer Science

Course Code: CS-321

Artificial Intelligence

Spring 2023

Exercise 1

Develop code to implement the A* algorithm in order to find the optimal path in the Travel in Romania problem. Use the heuristic given in the text above. Furthermore, propose an admissible heuristic of your own and compare the two heuristics utilized. Suggest which of the two heuristics is a better choice for the travel in Romania problem and why?

Exercise 2

For the game of Tic Tac Toe developed in lab 4, improve the agent designed by you for playing the game by using the Minimax Algorithm.