Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name:Stroie Anca Gabriela
Group:30432
Email:Stroie.Io.Anca@student.utcluj.ro

Teaching Assistant: Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Table 1: Lab scheduling

Activity	**Deadline**
Searching agents, Linux, Latex, Python, Pacman	W_1
Uninformed search	W_2
Informed Search	W_3
Adversarial search	W_4
Propositional logic	W_5
First order logic	W_6
Inference in first order logic	W_7
Knowledge representation in first order logic	W_8
Classical planning	W_9
Contingent, conformant and probabilistic planning	W_{10}
Multi-agent planing	W_{11}
Modelling planning domains	W_{12}
Planning with event calculus	W_{14}

Lab organisation.

1. Laboratory work is 25% from the final grade.
2. There are three deliverables in total: 1. Search, 2. Logic, 3. Planning.
3. Before each deadline, you have to send your work (latex documentation/code) at moodle.cs.utcluj.ro
4. We use Linux and Latex
5. Plagiarism: Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more.

Chapter 1

A1: Search

1.1 Introduction

Artificial Intelligence (AI) has revolutionized the way machines perceive and interact with their environment, enabling them to make intelligent decisions and solve complex problems. Search algorithms play a fundamental role in AI, helping systems navigate vast decision spaces to find optimal solutions. To illustrate the power and versatility of search algorithms, we can turn to the world of gaming, and specifically, the iconic Pac-Man game.

1.2 Algorithms

Search algorithms in artificial intelligence are computational techniques used to find optimal or near-optimal solutions to complex problems by systematically exploring a search space. These algorithms are a fundamental component of AI and have applications in various domains, including route planning, game playing, robotics, natural language processing, and more.

1.2.1 A* Search

The A* algorithm is a widely used and highly effective graph search algorithm. It is commonly employed for pathfinding and graph traversal in various domains, including artificial intelligence, robotics, video games, and more. A* is renowned for its ability to find the shortest path from a starting point to a goal while efficiently exploring a search space.

The algorithm operates in a loop, continuing until the open_list is empty or a goal state is found. The loop has the following steps:

- Pop the node with the lowest f-value from the open_list. This node represents the most promising state to explore next.
- Check if the current state is the goal state using `problem.isGoalState(state)`. If it is, a solution has been found, and the algorithm returns the list of actions, which is the path to the goal state.
- If the current state has not been visited, mark it as visited by adding it to the visited set.
- Generate successors of the current state using `problem.getSuccessors(state)`. These successors are states that can be reached from the current state, along with the corresponding actions and step costs.

  For each successor:

- Calculate the new cost (new_cost) by adding the step cost to the cost of the current state.
- Calculate the f-value (f_value) of the successor using the formula $f\_value = new\_cost +$ `heuristic(successor, problem)`. The heuristic function estimates the cost to reach the goal state from the successor.
- Push the successor onto the open_list with the new actions (the current actions plus the new action) and the new cost.

1.2.2 A* Weighted Search

Weighted A* is a variation of the A* algorithm that allows you to control the trade-off between the heuristic component (h-cost) and the actual cost to reach a node (g-cost) when determining the priority of nodes in the open set. In standard A*, the priority is based on the sum of g-cost and h-cost. In weighted A*, you introduce a weight factor which is used to adjust the relative importance of g-cost and h-cost in determining the priority.

The weight factor in weighted A* determines how much emphasis is placed on the heuristic estimate. By increasing its value you make the algorithm more greedy, meaning it focuses more on the heuristic estimate and is more likely to explore fewer nodes but may not guarantee an optimal path. Conversely, by decreasing it, you make the algorithm more similar to standard A*, where it explores more nodes but has a higher chance of finding the optimal path.

- When the weight is set to 1, the algorithm behaves like traditional A*. It balances the actual cost and heuristic estimate equally. This usually results in an optimal solution, assuming the heuristic is admissible and consistent.
- Setting the weight to 0 essentially eliminates the heuristic component, making it equivalent to uniform cost search (Dijkstra's algorithm).
- When you increase the weight, the algorithm gives more importance to the heuristic estimate (h(n)) relative to the actual cost (g(n)). This means the algorithm prioritizes exploring paths that seem more promising according to the heuristic.
- Conversely, when you decrease the weight, the algorithm places more emphasis on the actual cost of the path (g(n)) compared to the heuristic estimate (h(n)). Lower weights make the algorithm focus on exploring paths that are cheaper in terms of the actual path cost.

Layout	Weight=0	Weight=1	Weight=3	Weight=100	A*
expanded nodes-medium	269	221	211	78	221
expanded nodes-big	620	549	510	466	549

Table 1.1: Results for A* weighted depending on the value of the weight

1.2.3 Iterative Deepening A*

Iterative deepening A* is a graph traversal and path search algorithm that can find the shortest path between a start node and a goal node in a weighted graph. Since it is a depth-first search algorithm, its memory usage is lower than in A*, but unlike ordinary iterative deepening search, it concentrates on exploring the most promising nodes and thus does not go to the same depth everywhere in the search tree. Unlike A*, IDA* does not utilize dynamic programming and therefore often ends up exploring the same nodes many times. The code consists of two functions:

iterativeDeepeningAStar:

It starts with a depth limit of 1. It enters a while loop that continues indefinitely (until a solution is found). In each iteration, it calls the `aStarSearchLimited` function with the current depth limit. If the result of the limited A* search is not None (i.e., a solution is found), it returns the result. If no solution is found at the current depth limit, it increments the depth limit by 1 and continues to the next iteration.

aStarSearchLimited:

- It uses a priority queue (min-heap) called `open_list` to manage the nodes to be explored. It starts by pushing the start state onto the `open_list` with an empty list of actions and a cost of 0. The priority is set to 0 because it's the initial depth. It maintains a set called `visited` to keep track of explored states. It enters a loop that continues until the `open_list` is empty. In each iteration, it pops the state, actions, and cost with the lowest priority (min f-value) from the `open_list`. If the popped state is the goal state, it returns the list of actions, which is the solution. If the state is not the goal state and has not been visited, it adds it to the `visited` set and explores its successors. It calculates the new cost and f-value for each successor and pushes the successor onto the `open_list` only if the length of the actions is less than or equal to the current depth limit.

1.3 Algorithm Comparison

Algorithm	Layout	Expanded nodes	Cost	Score
A*	small maze	53	19	491
A* weighted	small maze	39	29	481
Iterative Deepening A*	small maze	907	19	491
A*	medium maze	221	68	442
A* weighted	medium maze	78	74	436
Iterative Deepening A*	medium maze	8481	68	442
A*	big maze	549	210	300
A* weighted	big maze	466	210	300
Iterative Deepening A*	big maze	61014	210	300

Table 1.2: Comparison between algorithms

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Bibliography

[1] "A* and Weighted A* Search" [Online]. Available: https://www.cs.cmu.edu/~motionplanning/lecture/Asearch_v8.pdf

[2] "A* Search" [Online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm

[3] "Variants of A*" [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/Variations.html

[4] "Iterative Deepening A*" [Online]. Available: https://en.wikipedia.org/wiki/Iterative_deepening_A*

Appendix A

Your original code

Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more. This section should contain only code developed by you, without any line re-used from other sources. This section helps me to correctly evaluate your amount of work and results obtained.

A.1 A* Search code

```
def aStarSearch(problem, heuristic=nullHeuristic):

    open_list = util.PriorityQueue()
    start = problem.getStartState()
    open_list.push((start, [], 0), 0)
    visited = set()

    while not open_list.isEmpty():
        state, actions, cost = open_list.pop()

        if problem.isGoalState(state):
            return actions

        if state not in visited:
            visited.add(state)
            successors = problem.getSuccessors(state)
            for successor, action, step_cost in successors:
                new_cost = cost + step_cost
                f_value = new_cost + heuristic(successor, problem)
                open_list.push((successor, actions + [action], new_cost), f_value)

    return None
```

A.2 A* Weighted Search code

```
def aStarWeighted(problem, heuristic=nullHeuristic, weight=0):

    open_list = util.PriorityQueue()
    start = problem.getStartState()
    open_list.push((start, [], 0), 0)
    visited = set()

    while not open_list.isEmpty():
        state, actions, cost = open_list.pop()

        if problem.isGoalState(state):
            return actions

        if state not in visited:
            visited.add(state)
            successors = problem.getSuccessors(state)
            for successor, action, step_cost in successors:
                new_cost = cost + step_cost
                f_value = new_cost + weight*heuristic(successor, problem)
                open_list.push((successor, actions + [action], new_cost), f_value)

    return None
```

A.3 Iterative Deepening A* code

```
def iterativeDeepeningAStar(problem, heuristic):
    depth_limit = 1
    while True:
        result = aStarSearchLimited(problem, heuristic, depth_limit)
        if result is not None:
            return result
        depth_limit += 1

def aStarSearchLimited(problem, heuristic, depth_limit):
    open_list = util.PriorityQueue()
    start = problem.getStartState()
    open_list.push((start, [], 0), 0)
    visited = set()

    while not open_list.isEmpty():
        state, actions, cost = open_list.pop()

        if problem.isGoalState(state):
            return actions

        if state not in visited and len(actions) <= depth_limit:
            visited.add(state)
            successors = problem.getSuccessors(state)
```

```
        for successor, action, step_cost in successors:
            new_cost = cost + step_cost
            f_value = new_cost + heuristic(successor, problem)
            if len(actions) + 1 <= depth_limit:
                open_list.push((successor, actions + [action], new_cost), f_value)

return None
```