Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: URDEA MARA-CRISTINA
Group: 30432
Email: mara31urdea@gmail.com

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

In the initial part of our laboratory work, we implemented various search algorithms using the Pac-Man template from Berkeley University. We started with simple algorithms such as breadth-first search and depth-first search. Subsequently, we moved on to more sophisticated approaches like uniform cost search and A* search. Additionally, we were tasked with selecting and implementing two new algorithms. For our new algorithms, we chose to implement both the Best First Search and Weighted A* Search. These two methods strike a balance in terms of performance between the basic Breadth-First Search and the advanced A* Search. Notably, Best First Search is an informed algorithm, while Breadth-First Search is uninformed. The key difference between A* and Best First Search lies in their evaluation functions. A* uses the evaluation function f(n) = h(n) + g(n), whereas Best First Search employs f(n) = h(n). In summary, Best First Search doesn't rely on past knowledge, which means it requires less memory to find a path. On the other hand, the Weighted A* algorithm shares a similar evaluation function with A*, but it introduces a bias by multiplying the heuristic with a preference for states closer to the goal. This bias results in a trade-off between optimality and speed, making it a valuable addition to our set of algorithms.

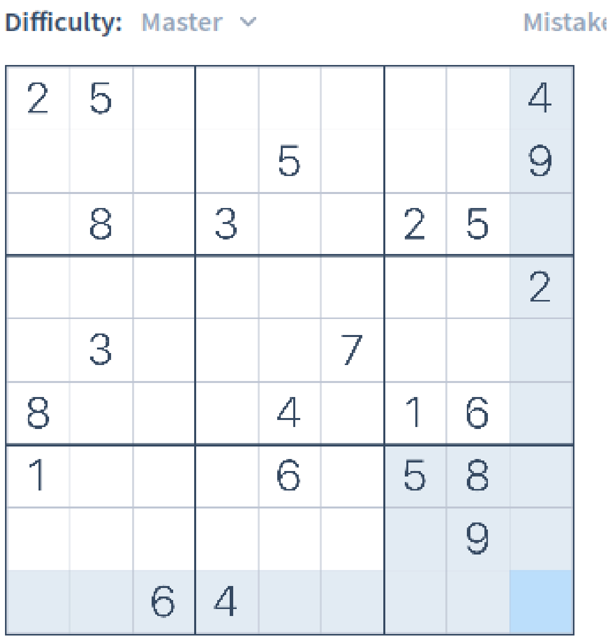
Chapter 2

A2: Logics

1. Zebra Puzzle
Einstein's puzzle is a famous logic puzzle that was supposedly created by Albert Einstein as a boy. The puzzle goes as follows:
There are five houses in a row, each with a different color. In each house lives a person of a different nationality. The five owners drink a certain type of beverage, smoke a certain brand of cigar, and keep a certain pet. No owners have the same pet, smoke the same brand of cigar, or drink the same beverage. We have to find out each of the respective persons with their respective belongings. Here are the clues:

```
– The Brit lives in the red house.
– The Swede keeps dogs as pets.
– The Dane drinks tea.
– The greenhouse is on the immediate left of the white house.
– The owner of the greenhouse drinks coffee.
– The person who smokes Pall Mall rears birds.
– The owner of the yellow house smokes Dunhill.
– The man living in the center house drinks milk.
– The Norwegian lives in the first house.
– The man who smokes Blends lives next to the one who keeps cats.
– The man who keeps the horse lives next to the man who smokes
Dunhill.
– The man who smokes Blue Master drinks beer.
– The German smokes Prince.
– The Norwegian lives next to the blue house.
– The man who smokes Blends has a neighbor who drinks water.
```

2. Sudoku Puzzle

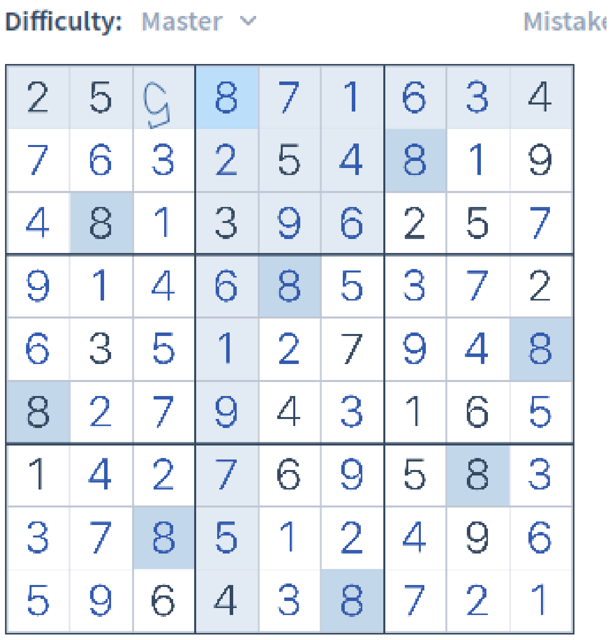


Difficulty: Master

Mistak

2	5							4
				5				9
	8		3			2	5	
								2
	3				7			
8				4		1	6	
1				6		5	8	
							9	
		6	4					

Figure 2.1: Initial shape



Difficulty: Master

Mistak

2	5	9	8	7	1	6	3	4
7	6	3	2	5	4	8	1	9
4	8	1	3	9	6	2	5	7
9	1	4	6	8	5	3	7	2
6	3	5	1	2	7	9	4	8
8	2	7	9	4	3	1	6	5
1	4	2	7	6	9	5	8	3
3	7	8	5	1	2	4	9	6
5	9	6	4	3	8	7	2	1

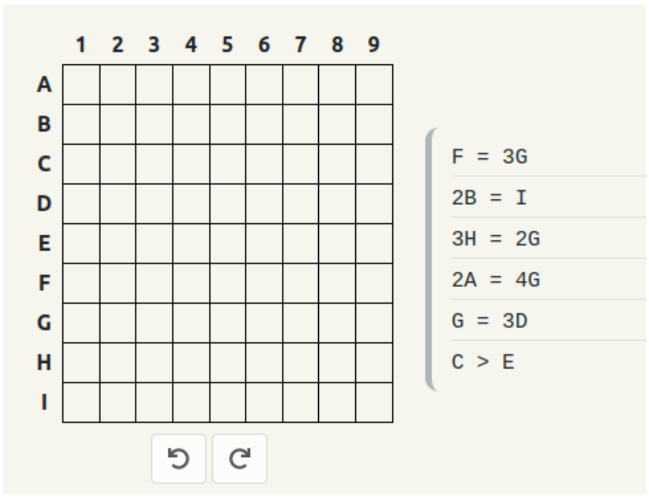
Figure 2.2: Solved shape

In Figure 2.1 can be observed the initial shape of a Sudoku Puzzle. Rules of Sudoku are:
- Sudoku grid consists of 9x9 spaces.
- You can use only numbers from 1 to 9.
- Each 3×3 block can only contain numbers from 1 to 9.
- Each vertical column can only contain numbers from 1 to 9.
- Each horizontal row can only contain numbers from 1 to 9.

- Each number in the 3×3 block, vertical column or horizontal row can be used only once.
- The game is over when the whole Sudoku grid is correctly filled with numbers.

In Figure 2.2 can be observed the solved puzzle using Mace4.

3. Logic Equation

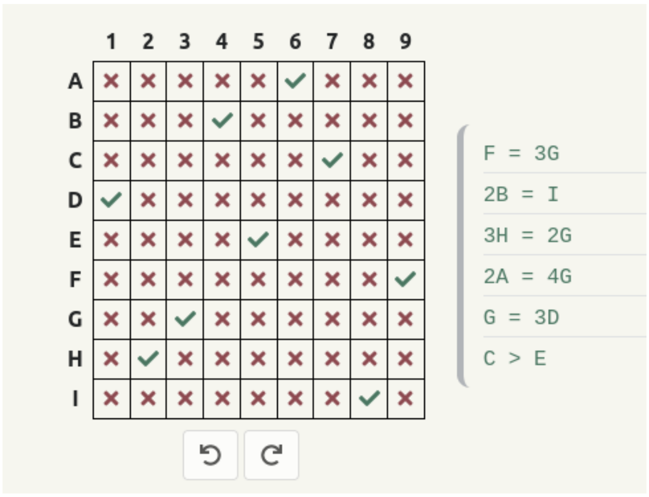


	1	2	3	4	5	6	7	8	9
A									
B									
C									
D									
E									
F									
G									
H									
I									

F = 3G
2B = I
3H = 2G
2A = 4G
G = 3D
C > E

Figure 2.3: Enter Caption

The variables represents unique integers ranging from 1 to the number of variables. Based on the clues (equations and inequations), use the grid to create relations between variables and values. The game ends when all values are correctly assigned to the variables.
In Figure 2.3 is represented the system of equations that we want to solve using Mace4.



	1	2	3	4	5	6	7	8	9
A	✗	✗	✗	✗	✗	✓	✗	✗	✗
B	✗	✗	✗	✓	✗	✗	✗	✗	✗
C	✗	✗	✗	✗	✗	✗	✓	✗	✗
D	✓	✗	✗	✗	✗	✗	✗	✗	✗
E	✗	✗	✗	✗	✓	✗	✗	✗	✗
F	✗	✗	✗	✗	✗	✗	✗	✗	✓
G	✗	✗	✓	✗	✗	✗	✗	✗	✗
H	✗	✓	✗	✗	✗	✗	✗	✗	✗
I	✗	✗	✗	✗	✗	✗	✗	✓	✗

F = 3G
2B = I
3H = 2G
2A = 4G
G = 3D
C > E

Figure 2.4: Solutions discovered using Mace4

In Figure 2.4 can be observed the solutions obtained using Mace4.

4. Logic Puzzle

There are three people (Alex, Ben and Cody), one of whom is a knight, one a knave, and one a spy. The knight always tells the truth, the knave always lies, and the spy can either lie or tell the truth. Alex says: "Cody is a knave." Ben says: "Alex is a knight." Cody says: "I am the spy." Who is the knight, who the knave, and who the spy?

Using Mace4 the final result will be:
- Alex = Knight
- Ben = Spy
- Cody = Knave

5. Crypto-Arithmetic

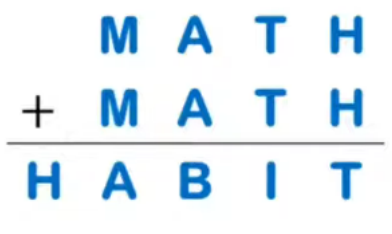


Figure 2.5: Unsolved crypto-arithmetic

Crypto-arithmetic puzzle is a mathematical exercise where the digits of some numbers are represented by letters (or symbols). Each letter represents a unique digit. The goal is to find the digits such that a given mathematical equation is verified. As with any optimization problem, we'll start by identifying variables and constraints. The variables are the letters, which can take on any single digit value. Constraints:
- Each of the ten letters must be a different digit.
- M, A, T, H, B and I can't be zero (since we don't write leading zeros in numbers).

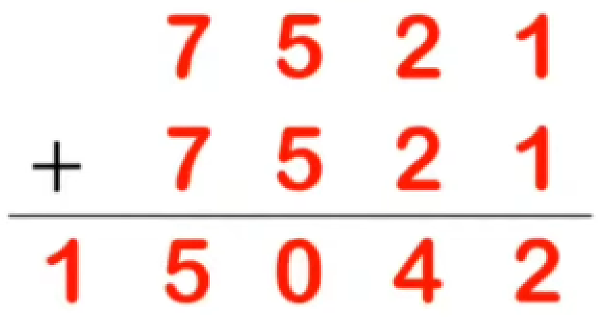


Figure 2.6: Solved crypto-arithmetic

Chapter 3

A3: Planning

Bibliography

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.

1.1 Code for Depth First Search

```
def depthFirstSearch(problem):
    #we create a stack to store the nodes to be explored
    stack = util.Stack()
    #push the start
    stack.push((problem.getStartState(), []))

    visited = set()
    while not stack.isEmpty():
        #pop the current state and the actions taken so far to
            reach this state
        current_state, actions = stack.pop()
        if problem.isGoalState(current_state):   # check if the
           current state is a goal state
            return actions
        #check if the current state has already been visited
        if current_state not in visited:
            # mark the current state as visited
            visited.add(current_state)
            #successors of the current state
            successors = problem.getSuccessors(current_state)

            for successor, action, _ in successors:
                #check if successor state is a valid move
                if problem.isGoalState(successor):
                    return actions + [action]

                stack.push((successor, actions + [action]))
    return []
```

1.2 Brief code explanation

Depth First Search explores the search tree by diving as deep as possible along one branch before backtracking. It utilizes a stack to manage the nodes to explore.

2.1 Code for Breadth First Search

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    stack = util.Queue()
    stack.push((problem.getStartState(), []))

    visited = set()
    while not stack.isEmpty():
        current_state, actions = stack.pop()
        if problem.isGoalState(current_state):
            return actions
        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)

            for successor, action, _ in successors:
                if problem.isGoalState(successor):
                    return actions + [action]

                stack.push((successor, actions + [action]))
    return []
    util.raiseNotDefined()
```

2.2 Brief code explanation

Breadth first search is similar to the depth first search, but it adds the elements to a queue, instead of a stack.

3.1 Code for Uniform Cost Search

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    """Search the node of least total cost first."""
    # priority queue to store the nodes
    pq = util.PriorityQueue()

    #start state,empty list of actions and cost onto the pq
    pq.push((problem.getStartState(), [], 0), 0)

    visited = set()

    while not pq.isEmpty():
        #current state, actions, and cost from pq
        current_state, actions, cost = pq.pop()

        # check if the current state is a goal state
        if problem.isGoalState(current_state):
            return actions

        if current_state not in visited:
            visited.add(current_state)

            successors = problem.getSuccessors(current_state)

            for successor, action, step_cost in successors:
                if problem.isGoalState(successor):
                    return actions + [action]

                #total cost for the successor
                total_cost = cost + step_cost

                pq.push((successor, actions + [action],
                    total_cost), total_cost)

    return []
    #util.raiseNotDefined()
```

3.2 Brief code explanation

Uniform cost search is similar to the breadth first search and depth first search. It selects nodes based on their cumulative path cost and uses a priority queue to prioritize nodes with lower cost.

4.1 Code for A* Search

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and
       heuristic first."""
    "*** YOUR CODE HERE ***"
    pq = util.PriorityQueue()

    pq.push((problem.getStartState(), [], 0), 0)

    visited = set()

    while not pq.isEmpty():
        current_state, actions, cost = pq.pop()

        if problem.isGoalState(current_state):
            return actions

        if current_state not in visited:
            visited.add(current_state)

            successors = problem.getSuccessors(current_state)

            for successor, action, step_cost in successors:
                if problem.isGoalState(successor):
                    return actions + [action]

                total_cost = cost + step_cost
                #cost to reach the goal from the successor using
                   heuristic
                heuristic_cost = heuristic(successor, problem)
                #sum of actual cost and heuristic cost
                total_estimated_cost = total_cost + heuristic_cost

                pq.push((successor, actions + [action],
                   total_cost), total_estimated_cost)

    return []
```

4.2 Brief code explanation

A* Search combines the path cost (g) and a heuristic function (h) to estimate the cost to the goal. It employs a priority queue to explore nodes with the lowest f(n) = g(n) + h(n) value.

5.1 Code for Weighted A* Search

```
def weightedAStarSearch(problem, heuristic=nullHeuristic, weight=1):
    pq = util.PriorityQueue()

    start_state = problem.getStartState()
    start_heuristic = weight * heuristic(start_state, problem)
    pq.push((start_state, [], 0), start_heuristic)

    visited = set()

    while not pq.isEmpty():
        current_state, actions, cost = pq.pop()

        if problem.isGoalState(current_state):
            return actions

        if current_state not in visited:
            visited.add(current_state)

            successors = problem.getSuccessors(current_state)

            for successor, action, step_cost in successors:
                if problem.isGoalState(successor):
                    return actions + [action]

                heuristic_cost = weight * heuristic(successor,
                   problem)
                total_cost = cost + step_cost
                pq.push((successor, actions + [action],
                   total_cost), total_cost + heuristic_cost)

    return []
```

5.2 Brief code explanation

Weighted A* search employs a Priority Queue to arrange nodes according to their evaluation function. Initially, it initializes the evaluation function for the start state to 0 and includes it in the queue. It proceeds to check whether the current state corresponds to the goal state. If it does not, the algorithm appends it to the list of visited states and retrieves its successor states. The evaluation function is computed by combining the heuristic value multiplied by the weight with the cost of the action. If the state hasn't been previously visited, it is then included in the queue for further exploration.

6.1 Code for Best First Search

```
def bestFirstSearch(problem, heuristic=nullHeuristic):
    pq = util.PriorityQueue()

    pq.push((problem.getStartState(), [], 0), 0)
    visited = set()

    while not pq.isEmpty():
        current_state, actions, cost = pq.pop()

        if problem.isGoalState(current_state):
            return actions

        if current_state not in visited:
            visited.add(current_state)

            successors = problem.getSuccessors(current_state)

            for successor, action, step_cost in successors:
                if problem.isGoalState(successor):
                    return actions + [action]

                heuristic_cost = heuristic(successor, problem)
                pq.push((successor, actions + [action], cost +
                   step_cost), heuristic_cost)

    return []
```

3.2 Brief code explanation

Best First Search employs a Priority Queue to organize nodes by their heuristic values. It first examines if the current node is the goal state; if it isn't, the algorithm proceeds to explore its successors and enqueues them based on their respective heuristics. It also verifies whether a successor has been visited before; if it hasn't, the algorithm includes it in the list of visited nodes.

A2: Logics

1. Zebra Puzzle no.1

```
set(arithmetic).
assign(domain_size, 5).
assign(max_models, -1).

list(distinct).
  [Yellow, Blue, Red, Green, White].
  [Norwegian, Dane, Swede, German, Brit].
  [Dunhill, PallMall, Blends, BlueMaster, Prince].
  [Coffee, Tea, Beer, Water, Milk].
  [Birds, Cats, Dogs, Horses, Fishes].
end_of_list.

formulas(utils).
  right_neighbor(x,y) <-> x+1=y.
  left_neighbor(x,y) <-> x = y+1.
  neighbor(x,y) <-> right_neighbor(x,y) | left_neighbor(x,y).
  middle(x) <-> x=2.
  border(x) <-> x=0 | x=4.
  first(x) <-> x=0.
end_of_list.

formulas(houses).
  Brit = Red.
  Swede = Dogs.
  Dane = Tea.
  left_neighbor(Green, White).
  Green = Coffee.
  PallMall = Birds.
  Yellow = Dunhill.
  middle(Milk).
  first(Norwegian).
  neighbor(Blends, Cats).
  neighbor(Horses, Dunhill).
  BlueMaster = Beer.
  German = Prince.
  neighbor(Norwegian,Blue).
  neighbor(Blends, Water).
end_of_list.
```

2. Sudoku Puzzle

```
     formulas(assumptions).
     S(x, y1) = S(x, y2) -> y1 = y2. % to have at most one on each row
     S(x1, y) = S(x2, y) -> x1 = x2. % to have at most one on each column
% let's take intervals as (0=9) {0,1,2}, {3,4,5}, {6,7,8};
same_interval(x,x).
same_interval(x,y) -> same_interval(y,x).
same_interval(x,y) & same_interval(y,z) -> same_interval(x,z).
same_interval(0,1).
same_interval(1,2).
same_interval(3,4).
same_interval(4,5).
same_interval(6,7).
same_interval(7,8).
-same_interval(0,3).
-same_interval(3,6).
-same_interval(0,6).
% to have at most one of each in each region
(S(x1, y1) = S(x2, y2) & same_interval(x1,x2) & same_interval(y1,y2) -> x1 =
% initial shape of sudoku
S(0,0) = 2. S(0,1) = 5. S(0,8) = 4.
S(1,4) = 5. S(1,8) = 0.
S(2,1) = 8. S(2,3) = 3. S(2,6) = 2. S(2,7) = 5.
S(3,8) = 2.
S(4,1) = 3. S(4,5) = 7.
S(5,0) = 8. S(5,4) = 4. S(5,6) = 1. S(5,7) = 6.
S(6,0) = 1. S(6,4) = 6. S(6,6) = 5. S(6,7) = 8.
S(7,7) = 0.
S(8,2) = 6. S(8,3) = 4.
end_of_list.
```

3. Logic Equation

```
     set(arithmetic).
assign(max_models, -1).
assign(domain_size, 10).

list(distinct).
     [0, A, B, C, D, E, F, G, H, I].
end_of_list.

formulas(assumptions).
     F=3*G.
     2*B=I.
     3*H=2*G.
     2*A=4*G.
     G=3*D.
     C>E.
end_of_list.
```

4. Logic Puzzle

```
    assign(domain_size, 3).
assign(max_models, -1).

formulas(persons).
    all x (person(x) -> knight(x) |  knave(x) | spy(x)).
    all x (knight(x) -> -knave(x) & -spy(x)).
    all x (knave(x) -> -spy(x) & -knight(x)).
    all x (spy(x) -> -knight(x) & -knave(x)).
    knight(x) -> m(x).
    knave(x) -> -m(x).
end_of_list.

formulas(puzzle).
    person(Alex) & person(Ben) & person(Cody).
    Alex = 0 & Ben = 1 & Cody = 2.
    knight(x) & knight(y) -> x = y.
    knave(x) & knave(y) -> x = y.
    spy(x) & spy(y) -> x = y.
    (exists x knight(x)) & (exists x knave(x)) & (exists x spy(x)).
   m(Alex) <-> knave(Cody).
   m(Ben) <-> knight(Alex).
   m(Cody) <-> spy(Cody).
end_of_list.
```

5. Crypto-Arithmetic

```
    set(arithmetic).
assign(domain_size, 10).
assign(max_models, -1).

list(distinct).
    [M,A,T,H,B,I].
end_of_list.

formulas(assumptions).
   M!=0.
   H!=0.
   M*1000+A*100+T*10+H+M*1000+A*100+T*10+H=
   H*10000+A*1000+B*100+I*10+T.
end_of_list.
```