Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Savut Dan Cosmin and Tocan Robert-Alexandru

Group:30432

Email:cosminsavut@gmail.com and roberttocan@yahoo.com

Teaching Assistant: Marius Adrian Stoica
stoica.ma.marius@student.utcluj.ro

Contents

Chapter 1

A1: Search

1.1 Introduction

In the realm of Artificial Intelligence (AI), search algorithms are essential for solving problems by exploring the problem space to find solutions. This chapter focuses on agents that solve problems by searching, comparing uninformed search strategies (such as depth-first search, breadth-first search, and uniform cost search) against informed search strategies (like the A* algorithm). The strengths of the A* algorithm are highlighted through various heuristics. These search strategies are illustrated through a game-based framework, specifically the Pacman agent. The Pacman game also provides a platform for practicing adversarial search.

1.2 Search Algorithms in AI

Search algorithms in AI are designed to help agents find the optimal solution by exploring different scenarios and alternatives. A search problem typically includes a search space, a start state, and a goal state. By simulating various scenarios, search algorithms enable AI agents to identify the optimal path or solution for a given task.

AI agents use logic to process the initial state and try to reach the goal state, making the AI systems heavily reliant on the efficiency of the search algorithms that ate used. These agents perform tasks without requiring software literacy from users, aiming to achieve goals by developing action plans. Completion of these plans involves evaluating all possible alternatives and choosing the best path.

1.3 Uninformed Search in AI

Uninformed search algorithms, also known as blind search algorithms, do not use additional information about the goal state other than what is provided in the problem definition. The strategies to reach the goal differ only by the order and length of actions. Uninformed search algorithms explore the search space systematically without considering the cost of reaching the goal.

1.3.1 Depth-First Search (DFS)

Depth-First Search (DFS) explores the deepest nodes in the search tree first using a stack to manage the frontier. It is a systematic search method, but it's still blind and that can be inefficient for large search spaces. For the implementation a stack that is part of the utils of the project as used.

1.3.2 Breadth-First Search (BFS)

Breadth-First Search (BFS) explores the shallowest nodes in the search tree first using a queue. It is complete and optimal for uniform cost problems but can be memory-intensive. For the BFS implementation we used a queue.

1.3.3 Uniform Cost Search (UCS)

Uniform Cost Search (UCS) explores the node with the least total cost first using a priority queue based on the distance from the start node to the node to be explored. We used problem.getCostOfActions(pathToNode) since the cost of each action is one and it can be deduced from the path. It guarantees an optimal solution but can be slow for large search spaces.

1.4 Informed Search in AI

Informed search algorithms use additional information about the goal state to guide the search process more efficiently. This information is typically provided by a heuristic function that estimates how close a state is to the goal state. Informed search algorithms include A* search, Best-First search, and Greedy search.

1.4.1 A* Search

A* search combines the cost to reach a node and a heuristic to estimate the cost to the goal. It uses a priority queue to manage the frontier and is both complete and optimal if the heuristic is admissible and consistent. For the implementation, we used the priority queue defined in util and the nullHeuristic already defined.

1.5 Comparison of Search Strategies

Uninformed search strategies, such as DFS, BFS, and UCS, explore the search space systematically without additional information. They are simple to implement but can be inefficient for complex problems. In contrast, informed search strategies like A* use heuristics to guide the search process, making them more efficient and goal-directed.

1.6 Conclusion

Search algorithms are fundamental in AI for finding optimal solutions to problems. Uninformed search algorithms provide a basic framework for exploring the search space, while informed search algorithms enhance efficiency through heuristics. The Pacman game serves as an excellent platform for practicing and understanding these concepts, demonstrating the practical applications of search strategies in AI.

1.7 Eat all Food and Avoid Ghosts problem

For this problem a layout with food and at least one ghost will be used. The solution we found is not optimal and it is based on one that finds a reasonable path quickly (when there are no ghosts).

It is based on a greedy strategy that uses a modified BFS to find the closest dot. The agent then performs the first action in the path to this closest path if it is far enough from a ghost. If a ghost is too close to the square to which our agent would advance, it continues the modified BFS searching for another dot. If no dot is found, towards which the agent can advance, it will perform the first action available that keeps a safe distance from the ghosts.

The implementation differs from the BFS in the following ways:

- The algorithm has a different goal state. It calls another isGoalState function which checks if the state contains a dot.
- The algorithm is called after each action the agent performs.
- The algorithm returns only the first step of the path to the closest dot
- Extra checks at each found goal state, to check if the action gets the agent too close to a ghost
- Extra logic at the end of the algorithm, in case no available path was found towards any dot.

Chapter 2

A2: Logics

2.1 Introduction

In the world of Artificial Intelligence (AI), logic plays a crucial role in helping machines process information and make decisions. Logic is the foundation that allows AI systems to understand relationships, apply reasoning, and solve problems systematically. In this chapter, we explore different types of logic used in AI, from basic propositional logic to more advanced systems like Bayesian reasoning.

2.2 Logic and Logical Reasoning

At its core, logic is the study of principles for valid reasoning. In AI, logical reasoning refers to the process of drawing conclusions based on available information. Think of it as the thought process of an AI system trying to figure out the best course of action or solution to a problem. This reasoning helps AI systems to understand their environment, make informed decisions, and solve complex problems.

Logic provides a structured way to handle knowledge, define rules, and make decisions. It's like a roadmap for AI systems to follow when solving problems, ensuring that they can evaluate different possibilities and find the best solution.

2.3 Types of Logic in AI

There are several types of logic used in AI to handle different problem-solving scenarios. Each type of logic has its strengths and uses depending on the complexity and nature of the problem. Here are the main types of logic used in AI:

2.3.1 Propositional Logic

Propositional logic, also known as sentential logic, is the simplest form of logic. It deals with propositions, which are statements that can either be true or false. In this system, logical reasoning is based on combining propositions with logical operators like AND, OR, and NOT. Propositional logic is useful for basic decision-making, where the problem can be broken down into simple true/false statements.

2.3.2 First-Order Logic

First-Order Logic (FOL) is an extension of propositional logic that allows for more complex reasoning. While propositional logic only deals with simple true or false statements, FOL enables reasoning about objects, their properties, and the relationships between them.

In FOL, predicates are used to describe the properties of objects or the relationships between objects. A predicate can represent something like "isHuman" or "isTall" and can be applied to different objects, such as "John is human" or "Alice is tall."

FOL also introduces quantifiers, which allow us to make statements about objects in a given domain. There are two main types of quantifiers: - The universal quantifier is used to express that something is true for all objects in a domain. For example, we can say "all humans are mortal." - The existential quantifier is used to express that something is true for at least one object in a domain. For example, we can say "there exists at least one human who is a doctor."

With these features, First-Order Logic provides a way to represent more complex relationships and reasoning processes compared to propositional logic. It is a powerful tool in AI for representing knowledge and reasoning about entities and their relationships.

2.3.3 Fuzzy Logic

Fuzzy logic is different from classical logic in that it doesn't just deal with true or false values. Instead, it deals with degrees of truth, allowing for a more nuanced approach to reasoning. For example, fuzzy logic can represent a situation where something is partially true, like "The weather is somewhat sunny." Fuzzy logic is useful in handling uncertainty and imprecision, especially in real-world scenarios where exact values are hard to define.

2.3.4 Modal Logic

Modal logic introduces modalities, such as necessity and possibility, into classical logic. It allows AI systems to reason about different scenarios, such as what must be true, what could be true, or what is believed to be true. Modal logic is useful in scenarios where there are multiple possible outcomes or when reasoning about future or hypothetical events.

2.3.5 Bayesian Logic

Bayesian logic is based on probability theory and is used to handle uncertainty. It allows AI systems to make decisions based on the likelihood of different outcomes. By continuously updating probabilities as new information is received, Bayesian logic helps AI systems refine their decisions over time. It is particularly useful in fields like machine learning, where models improve as more data becomes available.

2.4 Knowledge Representation and Reasoning

Knowledge Representation and Reasoning (KRR) is a crucial part of AI that focuses on how to represent and reason about knowledge. It involves the creation of knowledge bases that store information about the world, and reasoning techniques to draw conclusions from that knowledge.

2.4.1 Components of KRR

KRR involves several key components:

- **Ontologies**: Structured frameworks that define concepts and categories in a domain, helping to organize knowledge.
- **Rules**: Logical frameworks that guide the reasoning process by defining how knowledge can be applied.
- **Semantics**: The meaning behind the data, helping to interpret and relate different pieces of information.

2.4.2 Techniques of Knowledge Representation

There are several techniques for representing knowledge in AI:

- **Logical Representation**: Using formal logic to define relationships and enable deductive reasoning.
- **Semantic Networks**: Graphical representations of relationships between concepts, making it easier to visualize knowledge.
- **Frames and Scripts**: Structures used to represent typical situations and allow the system to anticipate and respond to events.
- **Rule-Based Systems**: Using sets of rules to guide decision making, simulating expert-level reasoning.

2.4.3 Reasoning Techniques

Reasoning is the process by which AI applies logic to derive new knowledge or make decisions. There are different types of reasoning:

- **Deductive Reasoning**: Drawing conclusions from known facts or premises, ensuring logical certainty in specific contexts.
- **Inductive Reasoning**: Making generalizations from specific observations, useful in situations where exact answers are not available.
- **Abductive Reasoning**: Forming hypotheses to explain observed phenomena, especially useful in diagnostic systems.

2.5 Demonstrating First Order Logic Capabilities

This section explores the capabilities of first-order logic (FOL) in modeling and solving logical problems. We will demonstrate how real-world puzzles and scenarios can be translated into logical formulations, using automated tools such as Prover9 and Mace4, allowing systematic analysis and solutions.

2.5.1 Overview of Tools

Prover9 is an automated theorem prover for first-order and equational logic. It checks the validity of logical statements and excels at proving theorems, verifying logical deductions, and validating relationships defined by axioms and assumptions.

Mace4, on the other hand, complements Prover9, searches for finite models and counterexamples to validate logical formulations.

Together, these tools provide a comprehensive framework for solving problems that require reasoning about logical relationships, verifying consistency, and deducing valid conclusions from a set of assumptions.

2.5.2 Using First-Order Logic for Problem Solving

We approach each problem by:

- **Formalizing the Problem:**
  Translating the scenario into a set of logical axioms and rules.
  Clearly defining the domain, relations, and functions involved.

- **Defining Goals:**
  Specifying the logical question we aim to answer (e.g., "Who is guilty?" or "What is the correct solution?").

- **Applying Automated Reasoning:**
  Using Prover9 to prove or disprove the specified logical goals.
  Employing Mace4 to explore possible models and identify valid configurations.

2.5.3 Example Problems

Who robbed the bank?

The angry chief of police did roar,
"Who robbed the bank out of you four?"
When Al was asked, he said right then,
"The thief was Ben! The thief was Ben!"
Ben said to the policin' man,
"The thief was Dan! The thief was Dan!"
When Carl was questioned, he exclaimed,
"I'm not the one who should be blamed!"
When Dan was questioned, he replied,
"When Ben said it was me, he lied!"

If three of four suspects speak true,
And one speaks false, then who oh who?
And what if only one was frank?
But most of all, who robbed the bank?

Einstein's Riddle

Einstein wrote the following riddle and claimed that 98% of the world could not solve it. However, several NIEHS scientists were able to solve it and remarked that it is not overly difficult if approached with attention and patience. Try solving it yourself:

There are 5 houses in a row, each painted a different color. In each house lives a person of a different nationality. The 5 owners drink a specific type of beverage, smoke a specific brand of cigar, and keep a specific pet. No two owners share the same pet, smoke the same brand of cigar, or drink the same beverage. Here are the clues:

1. The Brit lives in the red house.

2. The Swede keeps dogs as pets.

3. The Dane drinks tea.
4. The green house is immediately to the left of the white house.
5. The owner of the green house drinks coffee.
6. The person who smokes Pall Mall rears birds.
7. The owner of the yellow house smokes Dunhill.
8. The person living in the center house drinks milk.
9. The Norwegian lives in the first house.
10. The person who smokes Blends lives next to the one who keeps cats.
11. The person who keeps horses lives next to the one who smokes Dunhill.
12. The person who smokes Bluemasters drinks beer.
13. The German smokes Prince.
14. The Norwegian lives next to the blue house.
15. The person who smokes Blends lives next to the one who drinks water.

Who owns the fish?

Turners and Lerners

Quinn, Ralph, and Steve each belong either to the Turner family, who always tells the truth, or the Lerner family, who always lies. Quinn says, "Either I belong or Ralph belongs to a different family than the other two." Whose family can be determined?

Solution: We can represent the statements logically.

- Quinn's statement: "Either I belong or Ralph belongs to a different family than the other two."

If Quinn is telling the truth (Turner family), then one of the following must be true: 1. Quinn belongs to a different family than Ralph and Steve, or 2. Ralph belongs to a different family than Quinn and Steve.

If Quinn is lying (Lerner family), then both of the above statements must be false.

We can analyze this further by considering the family assignments for Ralph and Steve and checking which combination of truth-tellers and liars holds consistently.

Murder Mystery Problem

In the small town of Dreadbury, a tragic incident has occurred. Aunt Agatha has been found dead in her mansion, and a murder investigation is underway. The suspects are Agatha, the butler, and Charles. The following facts are known:

1. Someone who lives in Dreadbury Mansion killed Aunt Agatha. This means that the killer must be one of the people who reside in the mansion.
2. Agatha, the butler, and Charles live in Dreadbury Mansion, and they are the only people who live there.

3. A killer always hates his victim, and is never richer than his victim. The murderer must have hatred for the victim and cannot be wealthier than the victim.
4. Charles hates no one that Aunt Agatha hates. Charles does not hate any person whom Aunt Agatha hates.
5. Agatha hates everyone except the butler. Agatha holds hatred for everyone, with the exception of the butler.
6. The butler hates everyone not richer than Aunt Agatha. The butler has a strong hatred for everyone who is not wealthier than Aunt Agatha.
7. The butler hates everyone Aunt Agatha hates. If Aunt Agatha hates someone, the butler also shares that hatred.
8. No one hates everyone. It is impossible for anyone to hate every single person.
9. Agatha is not the butler. Agatha and the butler are two distinct individuals.

- $Hates(X, Y)$ represents that person X hates person Y.
- $Wealth(X) > Wealth(Y)$ means that X is wealthier than Y.
- $Killer(X)$ represents that person X is the killer.
- $X \neq Y$ indicates that X and Y are distinct individuals.

We can express the given facts in a logical format to identify the murderer:

2.6 Conclusion

Logic plays an essential role in AI by providing a structured way to represent knowledge, make decisions, and solve problems. By using different types of logic—such as propositional logic, first-order logic, fuzzy logic, modal logic, and Bayesian logic—AI systems can handle a wide range of tasks, from simple decision-making to complex reasoning under uncertainty. Understanding and applying logic is key to building intelligent systems that can reason effectively in various real-world situations.

Bibliography

[1] https://www.javatpoint.com/search-algorithms-in-ai

[2] Berkeley Pacman Documentation https://inst.eecs.berkeley.edu/~cs188/fa24/projects/proj1/#q7-4-pts-eating-all-the-dots

[3] Themiscodes Github https://github.com/Themiscodes/Berkeley-Pacman-Projects/tree/main

[4] AI_Labs documentation

Appendix A

Your original code

Below are the implementations of various search algorithms used in the Pacman AI projects. Each algorithm solves a search problem using a different strategy.

A.1 Depth First Search (DFS)

DFS explores the deepest nodes in the search tree first. It uses a stack to manage the frontier.

```
def depthFirstSearch(problem: SearchProblem) -> List[Directions]:
    "*** YOUR CODE HERE ***"
    stack = util.Stack()
    visited = []
    # we use a stack for LIFO strategy
    if problem.isGoalState(problem.getStartState()):
        return []

    stack.push((problem.getStartState(), []))
    # the initial state is pushed and popped from the stack
    while not stack.isEmpty():
        current_state, path = stack.pop()
        if problem.isGoalState(current_state):
            return path

        visited.append(current_state)
        successors = problem.getSuccessors(current_state)
        for succ in successors:
            # succ example ((5, 4), 'South', 1)
            if succ[0] not in visited:
                stack.push((succ[0], path + [succ[1]]))
                # path is updated with the direction
```

A.2 Breadth First Search (BFS)

BFS explores the shallowest nodes in the search tree first. It uses a queue to manage the frontier.

```
def breadthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """Search the shallowest nodes in the search tree first."""
    "* YOUR CODE HERE *"
    queue = util.Queue()
    visited = []
    #we are using a queue for FIFO strategy
    if problem.isGoalState(problem.getStartState()):
        return []
    queue.push((problem.getStartState(),[]))
    while not queue.isEmpty():
        current_state, path = queue.pop()
        if problem.isGoalState(current_state):
            return path

        visited.append(current_state)
        successors = problem.getSuccessors(current_state)
        for succ in successors:
            if succ[0] not in visited:
                queue.push((succ[0], path + [succ[1]]))
                visited.append(succ[0])
                #the difference from DFS is that we mark all the
                successors as visited
    return []
```

A.3 Uniform Cost Search (UCS)

UCS explores the node of least total cost first. It uses a priority queue to manage the frontier based on the cost.

```
def uniformCostSearch(problem: SearchProblem) -> List[Directions]:
    """Search the node of least total cost first using Uniform
    Cost Search (UCS)."""
    priority_queue = util.PriorityQueue()
    start_state = problem.getStartState()
    priority_queue.push((start_state, []), 0)
    visited = []

    while not priority_queue.isEmpty():
        current_state, path = priority_queue.pop()
        if problem.isGoalState(current_state):
            return path

        if current_state not in visited:
            visited.append(current_state)
            for succ in problem.getSuccessors(current_state):
                succ_state, succ_action, succ_cost = succ
                if succ not in visited:
                    priority_queue.push((succ_state, path + [succ_action]),
                    problem.getCostOfActions(path + [succ_action]))
                    # works because cost is one at each step

    return []
```

A.4 A* Search

A* search uses both the cost to reach the node and a heuristic to estimate the cost to the goal. It uses a priority queue to manage the frontier.

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic) ->
List[Directions]:
    """Search the node that has the lowest combined cost and heuristic
    first."""
    "*** YOUR CODE HERE ***"

    priority_queue = util.PriorityQueue()
    visited = {}
    start_state = problem.getStartState()
    if problem.isGoalState(start_state):
        return []
    priority_queue.push((start_state, [], 0),
    heuristic(start_state, problem))
    # here we set the priority of the start state as the heuristic

    while not priority_queue.isEmpty():
        current_state, path, current_cost = priority_queue.pop()
        if problem.isGoalState(current_state):
            return path
        # if the current state is already visited, if our current
        cost is <= than the previous one
        # it means that we found a cheaper way
        if current_state in visited and visited[current_state] <=
        current_cost:
            continue
        visited[current_state] = current_cost

        successors = problem.getSuccessors(current_state)
        for succ in successors:
            succ_state, succ_action, succ_cost = succ
            cost_until_succ = current_cost + succ_cost
            # if the state has not been visited or it can be reached
            # at a lower cost than previously recorded
            # proceed to calculate the total cost and push the state onto
            the priority queue.
            if succ[0] not in visited or visited[succ_state] >
            cost_until_succ:
                total_cost = cost_until_succ +
                heuristic(succ_state, problem)
                priority_queue.push((succ_state, path + [succ_action],
                cost_until_succ), total_cost)
    util.raiseNotDefined()
```

A.5 Eat all Dots and Avoid Ghosts

The implementations started from the Eat all Dots problem using the ClosestDotSearchAgent as showed in the documentation of the Pacman project, Q8: Suboptimal search. We modified this class as follows:

```
class ClosestDotSearchAgent(SearchAgent):
...
    def findPathToClosestDot(self, gameState: pacman.GameState):
        """
        Returns a path (a list of actions) to the closest dot,
        starting from gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)
        "*** YOUR CODE HERE ***"

        return search.breadthFirstSearchAvoidGhosts(problem, gameState)
```

We also modified the isGoalState method of the AnyFoodSearchProblem to check if certain state contains food.

```
class AnyFoodSearchProblem(PositionSearchProblem):
...
    def isGoalState(self, state: Tuple[int, int]):
        """
        The state is Pacman's position. Fill this in with a goal test
        that will complete the problem definition.
        """
        x, y = state
        "*** YOUR CODE HERE ***"

        if state in self.food.asList():
            return True
        else:
            return False
```

The next method that needed to be implemented is the modified breadth first search.

```
def breadthFirstSearchAvoidGhosts(problem: SearchProblem,
gameState: pacman.GameState) -> List[Directions]:
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    queue = util.Queue()
    visited = []

    ghostPositions = gameState.getGhostPositions()
    print("GHOST" + str(ghostPositions))
    ghostAvoidanceDistance = 2
    startState = problem.getStartState()
    if problem.isGoalState(problem.getStartState()):
        return []
    queue.push((problem.getStartState(),[]))
    while not queue.isEmpty():
        currentState, path = queue.pop()
        if problem.isGoalState(currentState):
            # if a dot is found, check if you can make a move towards it
            nextState = getNextSqaure(startState, path[0])
            tooCloseToGhost = False
            for ghostPositon in ghostPositions:
                if util.manhattanDistance(nextState, ghostPositon) <
                ghostAvoidanceDistance:
                    tooCloseToGhost = True
            if tooCloseToGhost:
                # get search for the next dot that is closest
                continue
            return [path[0]]

        visited.append(currentState)
        successors = problem.getSuccessors(currentState)
        for succ in successors:
            if succ[0] not in visited:
                queue.push((succ[0], path + [succ[1]]))
                visited.append(succ[0])

    # if there is no available action towards a dot do the first
    available move
    successors = problem.getSuccessors(startState)
    for succ in successors:
        tooCloseToGhost = False
        for ghostPosition in ghostPositions:
            if util.manhattanDistance(succ[0], ghostPosition) <
            ghostAvoidanceDistance:
                print(succ[0])
                tooCloseToGhost = True
        if tooCloseToGhost:
            continue
        else:
            return [succ[1]]
    return []
```

Appendix B

Code for Logics Assignment

B.1 Who is the Robber?

```
set(arithmetic).

list(distinct).
   [Al, Ben, Carl, Dan].
end_of_list.

formulas(assumptions).

   Truth(Al) <-> Thief(Ben).
   Truth(Ben) <-> Thief(Dan).
   Truth(Carl) <-> -Thief(Carl).
   Truth(Dan) <-> -Truth(Ben).

   exists x (Thief(x) & all y (Thief(y) -> y = x)).

   (Truth(Al) & Truth(Ben) & Truth(Carl) & -Truth(Dan)) |
   (Truth(Al) & Truth(Ben) & -Truth(Carl) & Truth(Dan)) |
   (Truth(Al) & -Truth(Ben) & Truth(Carl) & Truth(Dan)) |
   (-Truth(Al) & Truth(Ben) & Truth(Carl) & Truth(Dan)).
end_of_list.
```

B.2 Einstein Puzzle

```
set(arithmetic).

list(distinct).
    [Dane, German, Brit, Swede, Norwegian].
    [Red, Green, Yellow, White, Blue].
    [Tea, Milk, Coffee, Beer, Water].
    [Dunhill, Bluemasters, Prince, PallMall, Blends].
    [Fish, Dog, Horse, Cat, Bird].
end_of_list.

formulas(utils).
```

```
    RightOf(x,y) <-> x + 1 = y.
    LeftOf(x,y)  <-> x = y + 1.
    NextTo(x,y) <-> RightOf(x,y) | LeftOf(x,y).
end_of_list.

formulas(assumptions).

    Brit = Red.
    Swede = Dog.
    Dane = Tea.
    LeftOf(White, Green).
    Green = Coffee.
    PallMall = Birds.
    Yellow = Dunhill.
    Milk = 2.
    Norwegian = 0.
    NextTo(Blends, Cat).
    NextTo(Horse, Dunhill).
    Bluemasters = Beer.
    German = Prince.
    NextTo(Norwegian, Blue).
    NextTo(Blends, Water).

end_of_list.
```

B.3 Turners and Lerners

```
formulas(assumptions).

    Truth(Quinn) <-> Family(Quinn, Turner).
    Truth(Ralph) <-> Family(Ralph, Turner).
    Truth(Steve) <-> Family(Steve, Turner).

    Family(Quinn, Lerner) <-> -Truth(Quinn).
    Family(Ralph, Lerner) <-> -Truth(Ralph).
    Family(Steve, Lerner) <-> -Truth(Steve).

    ((Family(Quinn, Turner) & Family(Ralph, Lerner) & Family(Steve, Lerner))
        | (Family(Quinn, Turner) & Family(Ralph, Lerner) & Family(Steve, Turn
    ((Family(Quinn, Lerner) & Family(Ralph, Lerner) & Family(Steve, Lerner))
        (Family(Quinn, Lerner) & Family(Ralph, Lerner) & Family(Steve, Turner

end_of_list.

formulas(goals).
    Family(Ralph, Lerner).
end_of_list.
```

B.4 Murder Mystery

Intelligent Systems Group