Experiment 3 HEURISTIC AND NON-CLASSICAL SEARCH

Abstract—**This report delves non-deterministic search techniques and heuristic functions used to tackle complex problems, such as marble solitaire and the generation of random k-SAT problems. By implementing algorithms like Hill-Climbing, Beam Search, and Variable-Neighborhood Descent, we aim to find solutions for uniform random 3-SAT problems and marble solitaire. Heuristic functions play a crucial role in streamlining the search space and enhancing the algorithms' performance. Through a comparative analysis of various search strategies, we highlight their strengths and weaknesses, providing valuable insights into their effectiveness in solving these intricate challenges.**

I. Introduction

Tackling complex problems like marble solitaire and Boolean satisfiability (SAT) problems is a significant challenge in computer science. Marble solitaire involves removing marbles from a board until only one remains, while k-SAT problems focus on determining satisfying assignments for Boolean variables across multiple clauses. Both of these problems fall under the category of NP-hard problems, meaning they are notoriously difficult to solve efficiently for larger instances.

This experiment aims to explore non-deterministic search methods, augmented by heuristic functions, to find optimal or near-optimal solutions to these problems. The study of heuristic and non-classical search techniques is not only intellectually stimulating but also highly relevant in real-world applications. For example, the principles used to solve marble solitaire can inform various game AI algorithms, while k-SAT problems are foundational in optimizing network designs, scheduling tasks, and other computational challenges.

II. Non-deterministic Search

Non-deterministic search algorithms offer a flexible framework for exploring multiple potential solutions without following a predetermined path. Unlike deterministic algorithms, which yield consistent results for the same input, non-deterministic approaches introduce randomness at various stages of the search process. This is especially useful for navigating large search spaces where exhaustive searches become computationally expensive or impractical.

In the context of marble solitaire, non-deterministic search allows us to explore various sequences of moves that could lead to the desired configuration. Similarly, when tackling k-SAT problems, search algorithms can investigate different assignments of Boolean variables to find a combination that satisfies all clauses. The element of randomness in the search helps to prevent the algorithm from becoming trapped in local optima, enabling it to uncover more promising areas within the search space.

III. Marble Solitaire Problem

Marble solitaire is a classic puzzle where the objective is to remove all but one marble, ideally leaving the last marble in the center of the board. The game is played by jumping one marble over another into an empty hole, effectively removing the jumped marble from the board. This process continues until only one marble remains.

To solve marble solitaire, we employ the following strategies:

1) Implement a priority queue-based search considering path cost.
2) Propose two distinct heuristic functions, with justifications.
3) Implement the Best First Search algorithm.
4) Implement the A* algorithm.
5) Compare the results of these various search algorithms.

Algorithms for Marble Solitaire

1. Priority Queue-Based Search: **Input:** Initial board configuration and a set of valid moves.
Output: Sequence of moves leading to the goal state.
Algorithm:

Algorithm 1 Priority Queue Search

```
1: Initialize a priority queue
2: Add the initial state to the queue with priority 0
3: while the queue is not empty do
4:     Pop the state with the highest priority
5:     if the goal state is reached then
6:         Return the path to the goal
7:     end if
8:     for each valid move do
9:         Generate a new state
10:        Calculate the path cost
11:        Add the new state to the queue with the calculated priority
12:    end for
13: end while
14: Return failure
```

Pros: This method guarantees that the first solution found is optimal, as it explores the most promising nodes first.
Cons: The priority queue can grow large, leading to high memory usage and longer search times, especially as the state space expands.

2. Heuristic Functions: We propose two heuristic functions to enhance our search:

- **Manhattan Distance Heuristic:** This measures how far each marble is from the center, providing a straightforward metric for guiding the search. By summing the distances of all marbles from their target positions, we gain insight into the potential moves needed to reach the goal.
- **Exponential Distance Heuristic:**This is similar, but with a larger bias towards the center. If H and V are the horizontal and vertical distances from the center respectively, then the heuristic's value is $2\hat{(}$max(H, V)).

Pros of Heuristic Functions: Heuristics can significantly reduce the search space, leading to faster solutions.
Cons: Poorly designed heuristics may misguide the search, resulting in longer solution times.

3. Best First Search: **Input:** Initial board configuration and a set of valid moves.
Output: Optimal sequence of moves leading to the goal state.
Algorithm:

Algorithm 2 Best First Search

```
1: Initialize an empty priority queue
2: Add the initial state with cost using the heuristic
3: while the queue is not empty do
4:     Dequeue the state with the lowest cost
5:     if the goal state is reached then
6:         Return the path to the goal
7:     end if
8:     for each valid move do
9:         Generate a new state
10:        Calculate the cost using the heuristic
11:        Enqueue the new state with the updated cost
12:    end for
13: end while
14: Return failure
```

Pros: This approach efficiently finds solutions by employing an informed search strategy that takes the estimated cost into account.
Cons: It may be slower than other methods if the heuristic is poorly tuned, as it might explore suboptimal paths.

4. A Search Algorithm:* **Input:** Initial board configuration and a set of valid moves.
Output: Optimal sequence of moves leading to the goal state.
Algorithm:

Algorithm 3 A* Search Algorithm

```
1: Initialize an empty priority queue
2: Add the initial state with cost 0
3: while the queue is not empty do
4:     Dequeue the state with the lowest f(n) = g(n) + h(n)
5:     if the goal state is reached then
6:         Return the path to the goal
7:     end if
8:     for each valid move do
9:         Generate a new state
10:        Calculate g(n) and h(n)
11:        Enqueue the new state with f(n)
12:    end for
13: end while
14: Return failure
```

Pros: A* is highly efficient because it combines the cost to reach a node with the estimated cost to the goal, leading to a more directed search.
Cons: It requires more memory compared to simpler algorithms, as it keeps track of all nodes in the search space.

Results and Conclusion

After implementing these algorithms, we compared their performance based on several criteria: the number of moves required, time complexity, and overall efficiency in reaching the goal state. It's expected that the A* search algorithm will outperform the others due to its informed nature, while the best-first search offers a good balance between performance and simplicity. The priority queue search, while optimal, may lag in efficiency due to its higher memory requirements.

IV. K-SAT PROBLEM

The k-SAT problem is a well-known NP-complete problem that involves determining if a Boolean formula in conjunctive normal form (CNF) is satisfiable. In this experiment, we focus on the random generation of 3-SAT problems and solving them using various search algorithms.

A. Generating k-SAT Problems

To generate random k-SAT problems, we follow these steps:

1) Choose a number of variables n.
2) Choose a number of clauses m.
3) Randomly generate m clauses where each clause contains 3 literals, selecting from the set of variables and their negations.

Input: Number of variables n, number of clauses m.
Output: A random k-SAT problem in CNF.
Algorithm:

Algorithm 4 Generate Random 3-SAT Problem

```
1: Initialize an empty list of clauses
2: for i = 1 to m do
3:     Create a clause with 3 random literals
4:     Add the clause to the list
5: end for
6: Return the list of clauses
```

B. Solving k-SAT Problems

When it comes to solving k-SAT problems, we employ three algorithms:

1) Hill-Climbing Algorithm
2) Beam Search Algorithm
3) Variable-Neighborhood-Descent Algorithm

1. Hill-Climbing Algorithm: **Input:** A random 3-SAT problem.
Output: A satisfying assignment or failure.
Algorithm:

Algorithm 5 Hill-Climbing Algorithm

```
1: Initialize a random assignment of variables
2: while not satisfied do
3:     if current assignment satisfies the formula then
4:         Return current assignment
5:     end if
6:     Select a variable to flip
7:     Generate new assignment
8:     if new assignment is better then
9:         Update current assignment
10:    end if
11: end while
12: Return failure
```

Pros: The hill-climbing algorithm is straightforward and easy to grasp, making it a solid choice for those new to solving SAT problems.
Cons: However, it often gets stuck in local optima, which means it may miss the global solution.

2. Beam Search Algorithm: **Input:** A random 3-SAT problem.
Output: A satisfying assignment or failure.
Algorithm:

Algorithm 6 Beam Search Algorithm

```
1: Initialize beam with random assignments
2: while not satisfied do
3:     Expand each assignment in the beam
4:     Evaluate all new assignments
5:     Keep the best $w$ assignments in the beam
6: end while
7: Return best assignment or failure
```

Pros: Beam search effectively narrows down the search space by maintaining only the best candidates, which can enhance performance.
Cons: The effectiveness of the solution is highly dependent on the beam width; if the beam is too narrow, it may overlook optimal solutions.

3. Variable-Neighborhood-Descent Algorithm: **Input:** A random 3-SAT problem.
Output: A satisfying assignment or failure.
Algorithm:

Algorithm 7 Variable-Neighborhood-Descent Algorithm

```
1: Initialize random assignment of variables
2: while not satisfied do
3:     for each neighborhood function do
4:         Generate new assignment
5:         if new assignment satisfies the formula then
6:             Return new assignment
7:         end if
8:     end for
9:     Update current assignment based on the best found
10: end while
11: Return failure
```

Pros: This algorithm expands the search space systematically by exploring various neighborhoods, which helps escape local optima.
Cons: It may require more iterations than simpler methods, potentially affecting overall efficiency.

Comparative Analysis

After solving the uniform random 3-SAT problems using the three different algorithms, we compare their performance based on:

- The number of iterations to reach a solution.
- The quality of solutions found.
- Time complexity.

It's anticipated that the Beam Search and Variable-Neighborhood-Descent algorithms will outperform the Hill-Climbing algorithm in terms of solution quality, especially as the complexity of SAT instances increases. This comparative analysis highlights the practical effectiveness of various problem-solving approaches.

V. Conclusion

In summary, both marble solitaire and k-SAT problems pose substantial challenges that can be effectively addressed using non-deterministic search techniques and heuristics. Algorithms like A* and Beam Search significantly enhance search efficiency, enabling exploration of larger search spaces. Our findings illustrate that informed search strategies can greatly improve performance in solving complex problems, paving the way for further research into optimization and algorithm development. This exploration not only enriches our theoretical understanding of these challenges but also promotes practical applications across fields like artificial intelligence, operations research, and game development.