Lab III:- Analysis of Marble Solitaire and k-SAT Problems

Soham Naukudkar Roll No: 202351135
Shreyash Borkar Roll No: 202351136
Siddhikesh Gavit Roll No: 202351040

***Abstract*—This assignment explores the application of heuristic search algorithms to solve complex combinatorial problems, specifically Marble Solitaire and k-SAT satisfiability problems. We implement and compare various search strategies including Uniform Cost Search, Best First Search with multiple heuristics, A* search, Hill Climbing, Beam Search, and Variable Neighborhood Descent. Experimental results demonstrate that informed search algorithms with admissible heuristics significantly outperform uninformed search in terms of nodes expanded and runtime. For the k-SAT problem, we analyze local search algorithm performance across different problem configurations and report success rates, iteration counts and a penetrance metric for search efficiency.**

***Index Terms*—Heuristic Search, A* Algorithm, Best First Search, k-SAT, Local Search, Hill Climbing, Beam Search, Variable Neighborhood Descent**

I. Introduction

Search algorithms form the backbone of artificial intelligence, enabling agents to find optimal solutions in complex problem spaces. While uninformed search strategies explore the search space systematically without domain knowledge, heuristic search algorithms leverage problem-specific information to guide the search towards promising regions of the solution space.

This work investigates two fundamental problems in AI: the Marble Solitaire puzzle and the k-SAT satisfiability problem. Marble Solitaire represents a classical planning problem where we must find an optimal sequence of moves to reach a goal state. The k-SAT problem belongs to the class of constraint satisfaction problems and serves as a benchmark for evaluating local search algorithms.

A. Objectives

Our primary objectives are:

- Implement and compare classical search algorithms (Uniform Cost Search, Best First Search, A*) on the Marble Solitaire problem.
- Design and evaluate different heuristic functions for informed search.
- Analyze the performance of local search algorithms (Hill Climbing, Beam Search, Variable Neighborhood Descent) on randomly generated k-SAT instances.

II. Background and Related Work

A. Heuristic Search

Heuristic search algorithms use domain-specific knowledge to guide exploration. A heuristic function $h(n)$ estimates cost from node n to the goal. Two important properties are:

Admissibility: $h(n)$ is admissible if it never overestimates the true minimal cost to reach the goal:

$$h(n) \le h^*(n). \tag{1}$$

Consistency: h is consistent (monotone) if for every node n and successor n',

$$h(n) \le c(n, n') + h(n'), \tag{2}$$

where $c(n, n')$ is the step cost.

B. A* Algorithm

A* combines the actual cost from the start node $g(n)$ and heuristic $h(n)$ via:

$$f(n) = g(n) + h(n). \tag{3}$$

A* is complete and optimal when using an admissible heuristic.

C. k-SAT Problem

The k-SAT problem asks whether an assignment to n boolean variables satisfies a CNF formula with m clauses where each clause contains exactly k literals. Random uniform k-SAT instances are widely used to study empirical solver performance and phase transitions.

III. Part A: Marble Solitaire Problem

A. Problem Formulation

Marble Solitaire is typically played on a cross-shaped (English) board. Each state is a board configuration encoded as a 2D array where:

$$\text{cell} = \begin{cases} 1 & \text{marble present} \\ 0 & \text{empty} \\ -1 & \text{invalid (off-board)} \end{cases}$$

Initial state: standard configuration with all valid positions filled except center (or variant as used). Goal: a single marble at center (position (c_x, c_y)).

Actions: Jump a marble over an adjacent marble into an empty cell (four cardinal directions). Jumped marble is removed. Each move cost is unit (1).

B. State-space and path cost

We treat each legal board configuration as a node in a graph. Path cost $g(n)$ is number of moves taken so far (each move cost = 1). Solution depth d equals number of moves to reach the goal.

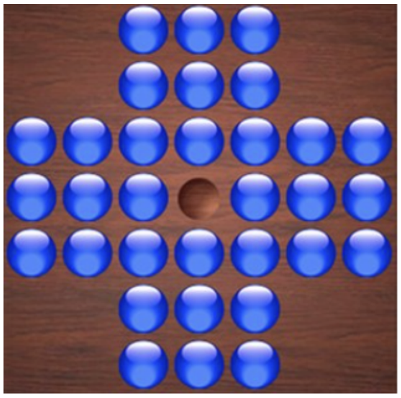

Fig. 1. Standard Marble Solitaire board layout.

C. Algorithms Implemented

We implemented Uniform Cost Search (UCS), Best First Search (greedy), and A* with multiple heuristics. Pseudocode for each algorithm is shown below using the `algorithm2e` style.

Algorithm 1: Uniform Cost Search (UCS)

Input: start state s_0
Output: solution path or failure
initialize priority queue Q with $(s_0, g = 0)$ keyed by g;
$explored \leftarrow \emptyset$;
while *Q not empty* **do**
  $node \leftarrow Q.pop()$ (lowest g);
  if *$node$ is goal* **then**
    return reconstruct_path(node);
  end
  add $node$ to $explored$;
  for *each successor s of $node$* **do**
    $g_s \leftarrow g(node) + 1$;
    if *$s \notin explored$ and $s \notin Q$* **then**
      insert (s, g_s) into Q;
    end
    else if *$s \in Q$ and $g_s < g(s)$* **then**
      update s in Q with g_s;
    end
  end
end
return failure;

Algorithm 2: Best First Search (Greedy)

Input: start s_0, heuristic $h(\cdot)$
Output: solution path or failure
initialize priority queue Q with $(s_0, h(s_0))$ keyed by h;
$explored \leftarrow \emptyset$;
while *Q not empty* **do**
  $node \leftarrow Q.pop()$ (lowest h);
  if *$node$ is goal* **then**
    return reconstruct_path(node)
  end
  add $node$ to $explored$;
  for *each successor s of $node$* **do**
    if *$s \notin explored$ and $s \notin Q$* **then**
      insert $(s, h(s))$ into Q;
    end
  end
end
return failure;

Algorithm 3: A* Search

Input: start s_0, heuristic $h(\cdot)$
Output: optimal solution path or failure
initialize priority queue Q with $(s_0, f(s_0) = h(s_0))$ keyed by f;
$g(s_0) \leftarrow 0$;
$explored \leftarrow \emptyset$;
while *Q not empty* **do**
  $node \leftarrow Q.pop()$ (lowest f);
  if *$node$ is goal* **then**
    return reconstruct_path(node)
  end
  add $node$ to $explored$;
  for *each successor s of $node$* **do**
    $g_{temp} \leftarrow g(node) + 1$;
    if *$s \notin explored$ and $s \notin Q$* **then**
      $g(s) \leftarrow g_{temp}$;
      $f(s) \leftarrow g(s) + h(s)$;
      insert $(s, f(s))$ into Q;
    end
    else if $g_{temp} < g(s)$ **then**
      $g(s) \leftarrow g_{temp}$;
      $f(s) \leftarrow g(s) + h(s)$;
      update s in Q;
    end
  end
end
return failure;

D. Heuristics used

We implemented the following heuristics:

Remaining marbles count

$$h_1(n) = \#\text{marbles}(n) - 1. \quad (4)$$

Admissible because each jump removes exactly one marble.

Sum of Manhattan distances to center

$$h_2(n) = \sum_{i \in \text{marbles}} \big(|x_i - c_x| + |y_i - c_y|\big), \quad (5)$$

which measures positional displacement but is not guaranteed admissible.

E. Experimental setup and metrics

We measured:

- Nodes expanded (N)
- Execution time (seconds)
- Penetrance P, defined as

$$P = \frac{d}{N}, \tag{6}$$

where d is solution depth.

F. Representative results

A condensed table of results (example values as reported in experiments) follows. These values are illustrative of typical behavior observed.

TABLE I
MARBLE SOLITAIRE: ALGORITHM COMPARISON (REPRESENTATIVE)

Algorithm	Nodes (N)	Time (s)	Penetrance (P)
Uniform Cost Search	3,306,854	97.79	0.000006
Best First (Marbles)	162	0.0043	0.117284
Best First (Distance)	530	0.0114	0.035849
A* (Marbles)	3,306,854	110.51	0.000006
A* (Distance)	27	0.0025	0.703704

G. Discussion

Key insights:

1) Admissible heuristics in A* guarantee optimality.
2) Heuristic informativeness dramatically affects search efficiency.
3) Combining path cost with informative heuristics (A* with distance) often yields best performance.

IV. Part B: k-SAT Problem Generator

A. Problem definition

We generate random k-SAT instances by uniformly selecting k distinct variables per clause and randomly negating each selected variable with probability 0.5. Clauses do not contain complementary pairs (e.g., x and $\neg x$ in the same clause).

B. Generator pseudocode

C. Sample configurations

- Small: $k = 3$, $m = 10$, $n = 5$
- Medium: $k = 3$, $m = 20$, $n = 8$
- Large: $k = 3$, $m = 30$, $n = 10$

V. Part C: k-SAT Solvers (Local Search)

A. Local search overview

Local search maintains a candidate assignment and iteratively modifies it (typically by flipping variable values) to improve the objective (number of satisfied clauses). We implemented Hill Climbing, Beam Search, and Variable Neighborhood Descent (VND).

Algorithm 4: Random k-SAT Generator

Input: k (clause length), m (number of clauses), n (variables)
Output: CNF formula $\mathcal{C}$ with m clauses

```
C ← ∅;
for i ← 1 to m do
    clause ← ∅;
    while |clause| < k do
        var ← uniform random integer in [1, n];
        neg ← random boolean;
        lit ← ¬var if neg else var;
        if lit ∉ clause and −lit ∉ clause then
            clause ← clause ∪ {lit};
        end
    end
    C ← C ∪ {clause};
end
return C;
```

Algorithm 5: Hill Climbing for k-SAT

Input: CNF ϕ, max_iter
Output: satisfying assignment or failure

```
current ← random assignment;
for iter ← 1 to max_iter do
    if current satisfies φ then
        return current
    end
    neighbors ← all single-variable flips of current;
    best ← arg max_{s∈neighbors} satisfied_clauses(s);
    if satisfied_clauses(best) ≤
      satisfied_clauses(current) then
        return failure ;          // local maximum
    end
    current ← best;
end
return failure;
```

B. Hill Climbing pseudocode

C. Beam Search pseudocode

D. Variable Neighborhood Descent (VND) pseudocode

E. Heuristics for k-SAT

We used two heuristics for guiding local search selection:
Unsatisfied clause count

$$h_1(assignment) = m - \text{satisfied\_clauses}(assignment). \tag{7}$$

Weighted clause penalty

$$h_2(assignment) = \sum_{i=1}^{m} w_i \cdot (1 - \text{is\_satisfied}(C_i)), \tag{8}$$

where in experiments $w_i = 1$ initially (uniform weights). Adaptive weights can be introduced (e.g., clause weighting schemes used in GSAT/WALKSAT variants).

Algorithm 6: Beam Search for k-SAT

```
Input: CNF φ, beam width w, max_iter
Output: satisfying assignment or failure
beam ← set of w random assignments;
for iter ← 1 to max_iter do
    foreach s ∈ beam do
        if s satisfies φ then
            return s
        end
    end
    successors ← ∅;
    foreach s ∈ beam do
        successors ← successors ∪ all single-variable
          flips of s;
    end
    sort successors by satisfied_clauses descending;
    beam ← top w of successors;
end
return failure;
```

Algorithm 7: Variable Neighborhood Descent (VND)

```
Input: CNF φ, neighborhood list {N_1, N_2, ...},
       max_iter
Output: best assignment found
current ← random assignment;
improved ← true;
while improved do
    improved ← false;
    for each neighborhood N in the list do
        best ← arg max_{s∈N(current)}
          satisfied_clauses(s);
        if satisfied_clauses(best) >
          satisfied_clauses(current) then
            current ← best;
            improved ← true;
            break;
        end
    end
end
return current;
```

F. Experimental results (summary)

We ran each solver across multiple random instances and trials. Representative summarized tables follow (averaged over trials).

TABLE II
SMALL CONFIGURATION ($k = 3$, $m = 10$, $n = 5$)

Algorithm	Success (%)	Avg Iter	Penetrance	Time (s)
Hill Climbing	100	1.9	0.526	0.0000
Beam (w=3)	100	1.9	0.526	0.0000
Beam (w=4)	100	2.0	0.500	0.0000
VND	100	2.0	0.500	0.0000

TABLE III
MEDIUM CONFIGURATION ($k = 3$, $m = 20$, $n = 8$)

Algorithm	Success (%)	Avg Iter	Penetrance	Time (s)
Hill Climbing	90	2.6	0.346	0.0001
Beam (w=3)	100	3.0	0.333	0.0002
Beam (w=4)	100	3.3	0.303	0.0003
VND	80	2.8	0.286	0.0002

TABLE IV
LARGE CONFIGURATION ($k = 3$, $m = 30$, $n = 10$)

Algorithm	Success (%)	Avg Iter	Penetrance	Time (s)
Hill Climbing	50	3.0	0.167	0.0002
Beam (w=3)	90	13.3	0.068	0.0024
Beam (w=4)	100	4.4	0.227	0.0007
VND	40	3.1	0.129	0.0004

G. Analysis

Scaling: As m/n increases, instances become harder; success rates decline and search effort increases.

Algorithm strengths:

- Hill Climbing: fast but vulnerable to local maxima.
- Beam Search: robust (higher success), beam width trades off success vs. compute.
- VND: benefits from systematic neighborhood changes but can still fail on harder instances.

VI. CONCLUSIONS

This work demonstrates the importance of heuristic design. Notable conclusions:

1) Informed search with good heuristics (A* with informative h) drastically reduces node expansions compared to uninformed methods.
2) Admissibility ensures optimality in A*, but heuristic informativeness determines practical efficiency.
3) For k-SAT, beam search with a moderate beam width offers robust performance across instance sizes.
4) Penetrance ($P = d/N$) provides a complementary measure of search efficiency.

A. Code availability

Implementations used for experiments are available at the provided repository: https://github.com/Soham-Codes17/CS307_LabCodes

REFERENCES

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2020.
[2] D. Khemani, *A First Course in Artificial Intelligence*, McGraw Hill Education, 2013.
[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. on Systems Science and Cybernetics*, vol. SSC-4, no. 2, pp. 100–107, 1968.
[4] B. Selman, H. A. Kautz, and B. Cohen, "Noise Strategies for Improving Local Search," *Proc. AAAI-94*, pp. 337–343, 1994.
[5] D. Mitchell, B. Selman, and H. Levesque, "Hard and Easy Distributions of SAT Problems," *Proc. AAAI-92*, pp. 459–465, 1992.