CS 663 Artificial Intelligence: Laboratory Report Assignment 1

Mahipalsinh Vansia (20251602010) Harishchandrasinh Jhala (20251603024) Dilipkumar Variya (20251602002)

***Abstract*—This laboratory report presents comprehensive solutions to two classical artificial intelligence problems - Missionaries & Cannibals and Rabbit Leap - modeled as state-space search problems. Both problems are systematically solved using Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms, with detailed analysis of their performance metrics including optimality, time complexity, and space complexity. The implementations are provided in Python, accompanied by empirical results and theoretical comparisons. Experimental results demonstrate that BFS consistently finds optimal solutions (11 moves for Missionaries & Cannibals, 15 moves for Rabbit Leap) while DFS provides space-efficient alternatives.**

I. Introduction

State-space search represents a fundamental paradigm in artificial intelligence for solving complex problems through systematic exploration of possible states and transitions. This report examines two classical constraint satisfaction problems: Missionaries & Cannibals and Rabbit Leap. Both problems are implemented and rigorously analyzed using BFS and DFS algorithms to understand their search characteristics, performance metrics, and practical implications in AI problem-solving.

II. Problem Statements

A. Missionaries and Cannibals Problem

The Missionaries and Cannibals problem involves transporting three missionaries and three cannibals across a river using a boat that can carry one or two persons. The critical constraint requires that missionaries must never be outnumbered by cannibals on either river bank, as this would compromise their safety.

B. Rabbit Leap Problem

In the Rabbit Leap problem, three east-bound rabbits face three west-bound rabbits separated by one empty stone. The rabbits can either move forward one step or jump over one opposing rabbit into an empty stone. The objective is to determine whether the rabbits can successfully cross each other without falling into the water.

III. Methodology

A. State-Space Modeling

1) Missionaries and Cannibals State Representation: The problem state is formally represented as a tuple (M_L, C_L, B) where:

- M_L: Number of missionaries on left bank (0-3)
- C_L: Number of cannibals on left bank (0-3)
- $B \in \{L, R\}$: Current position of the boat

Initial State: $(3, 3, L)$
Goal State: $(0, 0, R)$

Constraints: For each bank, $M \geq C$ when $M > 0$

Valid Actions: Transport (2M, 2C, 1M1C, 1M, 1C)

Search Space Analysis:

$$\text{Raw states} = (3+1) \times (3+1) \times 2 = 32, \quad \text{Valid states} = 20 \tag{1}$$

2) Rabbit Leap State Representation: The state is encoded as a 7-character string with three 'E', three 'W', and one '_' representing empty space.
Initial State: EEE_WWW
Goal State: WWW_EEE

Valid Moves:

- E moves right 1 step or jumps right 2 steps over one rabbit
- W moves left 1 step or jumps left 2 steps over one rabbit

Search Space Analysis:

$$\text{Total states} = \frac{7!}{3! \times 3! \times 1!} = 140 \tag{2}$$

B. Search Algorithms

Both BFS and DFS algorithms were implemented with comprehensive tracking mechanisms:

- **Breadth-First Search (BFS):** Explores all nodes at current depth before proceeding deeper, guaranteeing optimal solutions
- **Depth-First Search (DFS):** Explores each branch to maximum depth before backtracking, offering space efficiency

IV. Results and Analysis

A. BFS Solutions and Optimality

1) Missionaries and Cannibals - BFS: BFS successfully found an optimal solution in 11 moves with the following sequence:

$$(3,3,L) \to (3,1,R) \to (3,2,L) \to (3,0,R) \to$$
$$(3,1,L) \to (1,1,R) \to (2,2,L) \to (0,2,R) \to$$
$$(0,3,L) \to (0,1,R) \to (1,1,L) \to (0,0,R)$$

Empirical Metrics: Nodes expanded: 15, Max frontier: 3, Time: 0.08 ms

2) Rabbit Leap - BFS: BFS discovered an optimal solution in 15 moves with key transitions:

$$\text{EEE\_WWW} \rightarrow \text{EE\_EWWW} \rightarrow \text{EEWE\_WW} \rightarrow$$
$$\text{EEWEW\_W} \rightarrow \text{EEW\_WEW} \rightarrow \cdots \rightarrow \text{WWW\_EEE}$$

Empirical Metrics: Nodes expanded: 39, Max frontier: 8, Time: 0.20 ms

B. DFS Solutions

Both problems were successfully solved using DFS, though without optimality guarantees. DFS demonstrated superior space efficiency but exhibited potential for longer paths and required careful cycle detection to prevent infinite loops.

C. Algorithm Comparison

TABLE I
COMPARATIVE ANALYSIS: BFS VS DFS

Aspect	BFS	DFS
Optimality	Guaranteed shortest path	Not guaranteed
Time Complexity	$O(b^d)$	$O(b^m)$
Space Complexity	$O(b^d)$ (high)	$O(bm)$ (low)
Performance	Reliable for shallow goals	Space-efficient
Practical Use	Missionaries: 15 nodes Rabbit Leap: 39 nodes	Missionaries: Variable Rabbit Leap: Variable

V. IMPLEMENTATION

The Python implementation features generic BFS and DFS algorithms with comprehensive path reconstruction and performance monitoring.

```
from collections import deque
from dataclasses import dataclass
import time
from typing import Any, Callable, Dict, Iterable,
    List, Optional, Tuple

def reconstruct_path(parent: Dict[Any, Tuple[Any,
    str]], goal_state: Any) -> List[Tuple[Any, str
    ]]:
    path_rev: List[Tuple[Any, str]] = []
    s = goal_state
    while True:
        prev, act = parent[s]
        path_rev.append((s, act if act is not None
    else "START"))
        if prev is None:
            break
        s = prev
    return list(reversed(path_rev))

@dataclass
class SearchResult:
    found: bool
    path: List[Tuple[Any, str]]
    nodes_expanded: int
    max_frontier: int
    elapsed_ms: float

def bfs(start: Any, goal_test: Callable[[Any], bool
    ],
        successors: Callable[[Any], Iterable[Tuple[
    Any, str]]]) -> SearchResult:
    t0 = time.perf_counter()
    frontier = deque([start])
    parent: Dict[Any, Tuple[Any, Optional[str]]] = {
    start: (None, None)}
    nodes_expanded = 0
    max_frontier = 1
    visited = {start}

    while frontier:
        max_frontier = max(max_frontier, len(
    frontier))
        s = frontier.popleft()
        nodes_expanded += 1

        if goal_test(s):
            t1 = time.perf_counter()
            path = reconstruct_path(parent, s)
            return SearchResult(True, path,
    nodes_expanded,
                                max_frontier, (t1 - t0
    ) * 1000)

        for (ns, action) in successors(s):
            if ns not in visited:
                visited.add(ns)
                parent[ns] = (s, action)
                frontier.append(ns)

    t1 = time.perf_counter()
    return SearchResult(False, [], nodes_expanded,
    max_frontier, (t1 - t0) * 1000)

# Additional implementation code continues...
```

Listing 1. Search Algorithms Implementation

VI. EXECUTION RESULTS

A. Search Space Validation

- **Missionaries & Cannibals:** 20 valid states with boat positioning
- **Rabbit Leap:** 140 unique permutations with 3E, 3W, 1_

B. Complete Solution Sequences

1) Missionaries & Cannibals - BFS Solution:

```
0: (3, 3, 'L') [START]
1: (3, 1, 'R') [Boat L->R carrying 2C]
2: (3, 2, 'L') [Boat R->L carrying 1C]
3: (3, 0, 'R') [Boat L->R carrying 2C]
4: (3, 1, 'L') [Boat R->L carrying 1C]
5: (1, 1, 'R') [Boat L->R carrying 2M]
6: (2, 2, 'L') [Boat R->L carrying 1M1C]
7: (0, 2, 'R') [Boat L->R carrying 2M]
8: (0, 3, 'L') [Boat R->L carrying 1C]
... (complete path to (0,0,'R'))
```

2) Rabbit Leap - BFS Solution:

```
0: EEE_WWW [START]
1: EE_EWWW [E moves 2->3 (step)]
2: EEWE_WW [W jumps 4->2 (over E)]
3: EEWEW_W [W moves 5->4 (step)]
4: EEW_WEW [E jumps 3->5 (over W)]
5: E_WEWEW [E jumps 1->3 (over W)]
6: _EWEWEW [E moves 0->1 (step)]
7: WE_EWEW [W jumps 2->0 (over E)]
```

```
8: WEWE_EW [W jumps 4->2 (over E)]
9: WEWEWE_ [W jumps 6->4 (over E)]
10: WEWEW_E [E moves 5->6 (step)]
11: WEW_WEE [E jumps 3->5 (over W)]
12: W_WEWEE [E jumps 1->3 (over W)]
13: WW_EWEE [W moves 2->1 (step)]
14: WWWE_EE [W jumps 4->2 (over E)]
15: WWW_EEE [E moves 3->4 (step)]
```

VII. Conclusion

This laboratory investigation demonstrates the practical application of state-space search algorithms in solving classical AI problems. BFS consistently guarantees optimal solutions, finding the minimal path of 11 moves for Missionaries & Cannibals and 15 moves for Rabbit Leap, albeit with higher computational resource requirements. DFS offers superior space efficiency but sacrifices optimality guarantees. The state-space modeling effectively captured problem constraints, and both search algorithms exhibited expected theoretical behaviors in practical implementations. These findings underscore the importance of algorithm selection based on specific problem constraints and resource considerations in artificial intelligence applications.

Sentence Alignment and Plagiarism Detection using A* Search

Mahipalsinh Vansia (20251602010) Harishchandrasinh Jhala (20251603024) Dilipkumar Variya (20251602002)

***Abstract*—This paper presents a comprehensive approach for sentence alignment and plagiarism detection using A* search algorithm. The methodology formulates document alignment as an optimal pathfinding problem where states represent processed sentence indices, actions include matching or skipping sentences, and costs are based on normalized edit distances. An admissible and consistent heuristic ensures optimal alignment. The system effectively identifies plagiarized content by flagging sentence pairs with similarity scores exceeding 0.80. Experimental results across four distinct scenarios demonstrate the algorithm's robustness in handling identical documents, paraphrased content, unrelated texts, and partial overlaps, achieving accurate plagiarism detection with optimal computational efficiency.**

I. Introduction

Document alignment and plagiarism detection represent critical challenges in natural language processing and information retrieval. Traditional methods often rely on simple string matching or statistical measures, which may lack optimality guarantees. This research formulates sentence alignment as a state-space search problem solved using A* algorithm, providing theoretical guarantees of optimal alignment while efficiently detecting plagiarized content through similarity thresholding.

II. Problem Formulation

Given two text documents, the objective is to align their sentences optimally while detecting potential plagiarism. The alignment should minimize the cumulative edit distance between corresponding sentences, with gap penalties for unaligned sentences.

III. Methodology

A. State-Space Representation

The problem is modeled as a state-space search where:

- **State:** (i, j) indicating first i sentences of Document 1 and first j sentences of Document 2 have been processed
- **Initial State:** $(0, 0)$
- **Goal State:** (n, m) where n and m are total sentences in each document

B. Action Space

From state (i, j), three actions are possible:

- **Align:** $(i + 1, j + 1)$ with cost $d(s_i, t_j)$ (normalized edit distance)
- **Skip Document 1:** $(i + 1, j)$ with gap penalty γ
- **Skip Document 2:** $(i, j + 1)$ with gap penalty γ

C. Cost Function

The path cost is the sum of step costs:

$$\text{Cost} = \sum_{\text{align steps}} d(s_i, t_j) + \sum_{\text{gap steps}} \gamma \tag{1}$$

D. Heuristic Design

The heuristic function $h(i, j) = |(n - i) - (m - j)| \cdot \gamma$ is both admissible and consistent:

- **Admissible:** Never overestimates remaining cost since at least $|(n - i) - (m - j)|$ gaps are unavoidable
- **Consistent:** $h(i, j) \leq c((i, j), (i', j')) + h(i', j')$ for all transitions

E. Plagiarism Detection

Sentence pairs with similarity score ≥ 0.80 are flagged as potential plagiarism:

$$\text{Similarity} = 1 - \text{Normalized Edit Distance} \tag{2}$$

IV. Implementation

A. Preprocessing Pipeline

- Text normalization: lowercase conversion, punctuation preservation
- Sentence splitting based on terminal punctuation marks
- Tokenization using word boundary detection

B. Distance Metric

Token-level Levenshtein distance normalized by maximum sentence length:

$$d(s_1, s_2) = \frac{\text{Levenshtein}(\text{tokens}(s_1), \text{tokens}(s_2))}{\max(|\text{tokens}(s_1)|, |\text{tokens}(s_2)|, 1)} \tag{3}$$

C. A* Algorithm Implementation

```python
import re, heapq

def normalize_text(s):
    s = s.lower()
    s = re.sub(r"[^a-z0-9\s\.!?]", " ", s)
    s = re.sub(r"\s+", " ", s).strip()
    return s

def split_sentences(doc):
    doc = re.sub(r"([a-zA-Z0-9])\s*\n", r"\1. ", doc
    )
    return [p.strip() for p in re.split(r"
    (?<=[\.!\?])\s+", doc.strip()) if p.strip()]

def tokens(s):
    return [t for t in re.split(r"\W+", s.lower())
    if t]
```

```
def levenshtein(a, b):
    n, m = len(a), len(b)
    if n == 0: return m
    if m == 0: return n
    prev = list(range(m+1))
    for i in range(1, n+1):
        cur = [i] + [0]*m
        for j in range(1, m+1):
            cost = 0 if a[i-1] == b[j-1] else 1
            cur[j] = min(prev[j] + 1, cur[j-1] + 1,
    prev[j-1] + cost)
        prev = cur
    return prev[m]

def norm_edit_distance(s1, s2):
    t1, t2 = tokens(s1), tokens(s2)
    if not t1 and not t2: return 0.0
    return levenshtein(t1, t2) / max(len(t1), len(t2
    ), 1)

def astar_align(sents1, sents2, gap_penalty=0.7):
    n, m = len(sents1), len(sents2)

    def heuristic(i, j):
        return abs((n-i) - (m-j)) * gap_penalty

    start, goal = (0,0), (n,m)
    open_set = [(heuristic(0,0), 0.0, start, None,
    None)]
    # ... continued implementation
```

Listing 1. A* Sentence Alignment Algorithm

V. Theoretical Analysis

A. Optimality Guarantees

- **Completeness:** Guaranteed for finite state spaces
- **Optimality:** Ensured by admissible heuristic and consistent cost function
- **Complexity:** Time $O(nm \log(nm))$, Space $O(nm)$

B. Heuristic Properties

The designed heuristic satisfies:

$$h(i,j) \leq h^*(i,j) \quad \text{(Admissibility)} \tag{4}$$

$$h(i,j) \leq c(i,j,i',j') + h(i',j') \quad \text{(Consistency)} \tag{5}$$

VI. Experimental Results

A. Test Case 1: Identical Documents

TABLE I
Identical Documents Alignment Results

Step	Type	Cost	Cumulative	Similarity	Flagged
1	MATCH	0.000	0.000	1.000	Yes
2	MATCH	0.000	0.000	1.000	Yes
3	MATCH	0.000	0.000	1.000	Yes
4	MATCH	0.000	0.000	1.000	Yes
5	MATCH	0.000	0.000	1.000	Yes
6	MATCH	0.000	0.000	1.000	Yes

Summary: Total cost: 0.0000, Average similarity: 1.0000, Flagged pairs: 6 (100%)

TABLE II
Modified Documents Alignment Results

Step	Type	Cost	Cumulative	Similarity	Flagged
1	MATCH	0.400	0.400	0.600	No
2	MATCH	0.333	0.733	0.667	No
3	MATCH	0.400	1.133	0.600	No
4	MATCH	0.636	1.770	0.364	No
5	MATCH	0.667	2.436	0.333	No
6	MATCH	0.667	3.103	0.333	No

B. Test Case 2: Slightly Modified Document

Summary: Total cost: 3.1030, Average similarity: 0.4828, Flagged pairs: 0 (0%)

C. Test Case 3: Completely Different Documents

TABLE III
Different Documents Alignment Results

Step	Type	Cost	Cumulative	Similarity	Flagged
1	MATCH	1.000	1.000	0.000	No
2	MATCH	1.000	2.000	0.000	No
3	MATCH	1.000	3.000	0.000	No
4	MATCH	1.000	4.000	0.000	No
5	MATCH	1.000	5.000	0.000	No
6	MATCH	1.000	6.000	0.000	No

Summary: Total cost: 6.0000, Average similarity: 0.0000, Flagged pairs: 0 (0%)

D. Test Case 4: Partial Overlap

TABLE IV
Partial Overlap Alignment Results

Step	Type	Cost	Cumulative	Similarity	Flagged
1	MATCH	0.786	0.786	0.214	No
2	GAP2	0.700	1.486	-	-
3	GAP2	0.700	2.186	-	-
4	MATCH	0.923	3.109	0.077	No
5	MATCH	0.000	3.109	1.000	Yes
7	MATCH	0.000	3.809	1.000	Yes

Summary: Total cost: 4.5088, Average similarity: 0.5728, Flagged pairs: 2 (50%)

VII. Complexity Analysis

The algorithm exhibits the following computational characteristics:

- **State Space:** $(n+1) \times (m+1)$ states
- **Time Complexity:** $O(nm \log(nm))$ due to priority queue operations
- **Space Complexity:** $O(nm)$ for storing g-scores and parent pointers
- **Optimality:** Guaranteed with consistent heuristic and finite state space

This complexity matches dynamic programming approaches while providing greater flexibility for cost function modifications and heuristic improvements.

VIII. Conclusion

The A* search-based sentence alignment framework successfully addresses the document alignment and plagiarism detection problem with theoretical guarantees of optimality. Key contributions include:

- Formulation of sentence alignment as optimal pathfinding with admissible heuristic
- Effective plagiarism detection through similarity thresholding at 0.80
- Comprehensive experimental validation across diverse scenarios
- Modular architecture supporting various similarity metrics and cost functions

The system demonstrates robust performance across identical documents (100% detection), paraphrased content (appropriate similarity scoring), unrelated texts (zero false positives), and partial overlaps (accurate identification of common segments). The approach provides a solid foundation for extensible document analysis systems with optimal alignment guarantees.

Random k-SAT Generator and Solvers: Hill-Climbing, Beam-Search, and Variable Neighborhood Descent Assignment 3

Mahipalsinh Vansia (20251602010) Harishchandrasinh Jhala (20251603024) Dilipkumar Variya (20251602002)

***Abstract*—This paper presents a comprehensive framework for generating and solving random k-SAT problems using various local search algorithms. The system includes a uniform random k-SAT generator and three distinct solvers: Hill-Climbing with restarts, Beam Search with multiple widths, and Variable Neighborhood Descent with three neighborhood structures. Two heuristic functions - satisfied clauses count and make-break analysis - are implemented and compared. Experimental results on 3-SAT instances with different clause-to-variable ratios demonstrate that Variable Neighborhood Descent achieves the highest penetrance rates, while the make-break heuristic significantly improves performance over basic satisfaction counting. The study provides insights into algorithm behavior across varying problem difficulties.**

I. Introduction

The Boolean satisfiability problem (SAT) represents a fundamental challenge in computer science with applications in verification, planning, and artificial intelligence. k-SAT, where each clause contains exactly k literals, serves as a benchmark for evaluating search algorithms. This research implements and compares three local search approaches for solving random 3-SAT instances, analyzing their performance under different problem complexities and heuristic guidance.

II. Problem Formulation

A. k-SAT Problem Definition

A k-SAT instance consists of:

- n Boolean variables $x_0, x_1, \ldots, x_{n-1}$
- m clauses, each containing exactly k distinct literals
- Each literal: (v, sign) where v is variable index and sign indicates positive/negative

The objective is to find a variable assignment that satisfies all clauses.

B. Uniform Random k-SAT Generation

The generator follows the uniform fixed clause-length model:

- Input: clause length k, variables n, clauses m, random seed
- Output: Random k-SAT instance with distinct literals per clause

III. Methodology

A. Representation and Evaluation

- **Assignment:** Boolean vector of length n
- **Cost Function:** Number of unsatisfied clauses (minimize)
- **Goal:** Achieve cost = 0 (satisfying assignment)

B. Heuristic Functions

1) H1: Satisfied Clauses Count:

$$h_1(\text{assignment}) = \text{number of satisfied clauses} \tag{1}$$

2) H2: Make-Break Analysis:

$$h_2(\text{assignment}) = \sum_{i=0}^{n-1} (\text{make}_i - \text{break}_i) \tag{2}$$

where make_i = clauses that would become satisfied by flipping variable i, and break_i = clauses that would become unsatisfied.

C. Search Algorithms

1) Hill-Climbing with Restarts:

- Greedy variable flips based on heuristic
- Random flips when no improvement (sideways moves)
- Multiple restarts with budget limits

2) Beam Search:

- Maintains w best assignments per iteration
- Explores single-flip neighbors
- Beam widths: 3 and 4 in experiments

3) Variable Neighborhood Descent (VND):

- Three neighborhood structures:
  1) N1: Best single variable flip
  2) N2: Best flip within random unsatisfied clause
  3) N3: Sampled best two-variable flip
- Cycles through neighborhoods on stagnation
- Random restarts when all neighborhoods fail

IV. Implementation

A. k-SAT Generator

```
def gen_k_sat(k, n, m, seed=None):
    rng = random.Random(seed)
    clauses = []
    for _ in range(m):
        vars_ = rng.sample(range(n), k)
        clause = []
```

```
        for v in vars_:
            sign = rng.choice([True, False])
            clause.append((v, sign))
        clauses.append(tuple(clause))
    return {"k": k, "n": n, "m": m, "clauses": tuple
    (clauses)}
```

Listing 1. Random k-SAT Generator

B. Heuristic Implementations

```
def make_break_counts(instance, assign):
    n = instance["n"]; make = [0]*n; breakc = [0]*n
    sat = [eval_clause(c, assign) for c in instance[
    "clauses"]]

    for ci, clause in enumerate(instance["clauses"])
    :
        if sat[ci]:
            sat_lits, sat_idx = 0, None
            for (v, pos) in clause:
                val = assign[v]
                if (val and pos) or ((not val) and (
    not pos)):
                    sat_lits += 1; sat_idx = v
            if sat_lits == 1: breakc[sat_idx] += 1
        else:
            for (v, pos) in clause:
                val = assign[v]
                if (val and (not pos)) or ((not val)
     and pos):
                    make[v] += 1
    return make, breakc
```

Listing 2. Make-Break Heuristic Calculation

C. Search Algorithm Pseudocode

Algorithm 1 Hill-Climbing with Restarts

```
Initialize best solution
for restart = 1 to R do
  Generate random assignment
  for step = 1 to T do
    if assignment satisfies all clauses then
      return solution
    end if
    Find best variable flip using heuristic
    if improvement possible then
      Apply best flip
    else
      Apply random flip (sideways move)
    end if
  end for
  Update best solution found
end for
return Best solution
```

V. Experimental Evaluation

A. Experimental Setup

- **Problem Size:** $n = 20$ variables
- **Clause Counts:** $m = 70$ ($m/n = 3.5$) and $m = 84$ ($m/n = 4.2$)

Algorithm 2 Variable Neighborhood Descent

```
Initialize assignment, neighborhood index k = 0
while within step budget do
  if assignment satisfies all clauses then
    return solution
  end if
  Try to improve using neighborhood Nk
  if improvement found then
    k = 0 (return to first neighborhood)
  else
    k = k + 1
    if k ¿ 3 then
      Random restart, k = 0
    end if
  end if
end while
return Best solution
```

- **Instances:** 3 random instances per configuration
- **Metrics:** Penetrance (success rate), minimum unsatisfied clauses, average time

B. Results Analysis

TABLE I
Algorithm Performance Comparison ($n = 20, m = 70$)

Algorithm	Penetrance	Avg Steps	Min Unsat	Avg Time (ms)
Beam Search (w=3) - H1	0/3	350	1	560
Beam Search (w=3) - H2	0/3	350	8	469
Beam Search (w=4) - H1	1/3	234.7	0	516
Beam Search (w=4) - H2	0/3	350	9	606
Hill Climbing - H1	3/3	174	0	57
Hill Climbing - H2	3/3	85.7	0	4
VND - H1	3/3	19	0	5
VND - H2	3/3	32.3	0	9

TABLE II
Algorithm Performance Comparison ($n = 20, m = 84$)

Algorithm	Penetrance	Avg Steps	Min Unsat	Avg Time (ms)
Beam Search (w=3) - H1	1/3	235	0	427
Beam Search (w=3) - H2	0/3	350	10	553
Beam Search (w=4) - H1	1/3	235.3	0	579
Beam Search (w=4) - H2	0/3	350	11	729
Hill Climbing - H1	2/3	3282	0	1306
Hill Climbing - H2	2/3	3281.7	0	187
VND - H1	2/3	425.3	0	175
VND - H2	2/3	398	0	147

C. Solution Traces

VI. Observations and Discussion

A. Heuristic Performance

- **H2 (Make-Break)** demonstrates superior performance, particularly in escaping local optima
- Significant runtime improvements observed with H2 across all algorithms

TABLE III
HILL-CLIMBING TRACE EXAMPLE ($n = 20, m = 70$)

Step	Move Type	Unsat Clauses	Flipped Variable
0	Initial	11	-
1	Greedy	6	19
2	Greedy	4	3
3	Greedy	3	9
4	Greedy	2	12
5	Greedy	1	0
6	Greedy	0	10

TABLE IV
HILL-CLIMBING TRACE EXAMPLE ($n = 20, m = 84$)

Step	Move Type	Unsat Clauses	Flipped Variable
0	Initial	12	-
1	Greedy	9	1
2	Greedy	7	6
3	Greedy	5	8
4	Greedy	3	14
5	Greedy	2	16
6	Random	3	11

- H2 provides better guidance for harder problem instances ($m/n = 4.2$)

B. Algorithm Comparison

- **VND** achieves highest penetrance with minimal step counts
- **Hill-Climbing** shows strong performance with appropriate heuristics
- **Beam Search** benefits from larger widths but remains less effective
- Neighborhood diversity in VND prevents stagnation and improves robustness

C. Problem Difficulty Impact

- $m/n = 3.5$: All algorithms achieve high success rates with VND performing best
- $m/n = 4.2$: Significant performance degradation observed across all methods
- Increased clause-to-variable ratio substantially raises problem complexity

VII. CONCLUSION

This research presents a comprehensive evaluation of local search algorithms for k-SAT problems. Key findings include:

- Variable Neighborhood Descent with multiple neighborhood structures achieves superior performance through diversified search strategies
- The make-break heuristic (H2) significantly outperforms basic satisfaction counting by providing better local improvement guidance
- Problem difficulty dramatically increases with higher clause-to-variable ratios, challenging all algorithms
- Beam search shows limited effectiveness compared to more adaptive approaches like VND and hill-climbing

The implemented framework successfully addresses both k-SAT generation and solving requirements, providing valuable insights into algorithm behavior across varying problem complexities. The modular design allows for easy extension with additional heuristics and search strategies for future research.

Jigsaw Puzzle Solving using Simulated Annealing

Group 9
Department of Computer Science
Indian Institute of Information Technology
Vadodara, Gujarat
Email: group9@iiitvadodara.ac.in

***Abstract*—This paper presents a comprehensive simulated annealing approach for solving jigsaw puzzles by formulating the problem as a permutation optimization task. We develop an energy function based on border color compatibility between adjacent tiles and implement an efficient local delta energy computation that enables rapid evaluation of candidate solutions. The algorithm explores the solution space using probabilistic acceptance of worse solutions to escape local minima, with experimental results demonstrating 87.5% energy reduction and effective reconstruction of scrambled 4×4 tile puzzles within 2.1 seconds.**

***Index Terms*—Jigsaw puzzle, simulated annealing, combinatorial optimization, image reconstruction, permutation optimization**

I. Introduction

Jigsaw puzzle solving represents a classical combinatorial optimization problem with significant applications in image processing, archaeological reconstruction, and document restoration. Traditional approaches often struggle with local optima and computational complexity as puzzle size increases.

This work formulates puzzle solving as a permutation search problem where the objective is to find the tile arrangement that minimizes color discontinuity along tile borders. We employ simulated annealing, a probabilistic optimization technique inspired by metallurgical annealing processes. This method combines greedy hill-climbing with controlled randomization to escape local optima, making it particularly suitable for high-dimensional permutation spaces where exhaustive search is infeasible.

II. Related Work

Previous approaches to jigsaw puzzle solving include greedy algorithms [**?**], genetic algorithms [2], and deep learning methods [3]. While these methods have shown promise, they often face challenges with local optima convergence or require extensive training data. Our simulated annealing approach provides a balance between exploration and exploitation while maintaining computational efficiency through local energy updates.

III. Simulated Annealing Approach

A. Algorithm Overview

Simulated annealing explores the permutation space through a temperature-controlled stochastic process. The algorithm accepts worse solutions with probability determined by the Metropolis criterion:

$$P(\text{accept}) = \begin{cases} 1 & \text{if } \Delta E \leq 0 \\ \exp(-\Delta E/T) & \text{otherwise} \end{cases} \quad (1)$$

where ΔE represents the energy change between current and candidate solutions, and T is the temperature parameter that decays according to a cooling schedule.

B. Algorithm Formulation

Require: Initial permutation $placement$, initial temperature T_0, cooling rate α, maximum iterations K
Ensure: Best found permutation $best$

$E \leftarrow \text{energy}(placement)$
$best \leftarrow placement$, $best\_E \leftarrow E$, $T \leftarrow T_0$
for $k = 1$ to K **do**
  Pick random positions $p \neq q$
  $dE \leftarrow \text{local\_delta\_energy\_if\_swap}(p, q)$
  if $dE \leq 0$ **or** random() $< \exp(-dE/T)$ **then**
    $\text{swap}(placement[p], placement[q])$
    $E \leftarrow E + dE$
    if $E < best\_E$ **then**
      $best \leftarrow placement$, $best\_E \leftarrow E$
    end if
  end if
  $T \leftarrow \alpha \cdot T$ {Cooling schedule}
end for
return $best$

IV. Implementation

A. Energy Model

The energy function quantifies color discontinuity along tile borders:

$$E = \sum_{\text{horizontal neighbors}} \text{RC}[t_i, t_j] + \sum_{\text{vertical neighbors}} \text{DC}[t_i, t_j] \quad (2)$$

where RC (Right Cost) and DC (Down Cost) are precomputed pairwise costs between tile edges, calculated as mean absolute color differences.

B. Local Delta Energy Computation

The key efficiency improvement comes from computing energy changes locally rather than recalculating the entire energy function:

$$\Delta E = \text{contrib}(p, t_q) + \text{contrib}(q, t_p) - \text{contrib}(p, t_p) - \text{contrib}(q, t_q) \quad (3)$$

This reduces computational complexity from $O(n)$ to $O(1)$ per candidate evaluation.

C. Python Implementation

```python
import numpy as np
import random
from PIL import Image

def make_synthetic_image(size=256, seed=0):
    rng = np.random.default_rng(seed)
    img = np.zeros((size, size, 3), dtype=np.uint8)

    # Create gradient background
    for i in range(3):
        x = np.linspace(0, 255, size, dtype=np.uint8)
        grad = np.tile(x, (size,1)) if i == 1 else grad.T
        img[:,:,:] = grad

    # Add random circles
    for _ in range(40):
        r = rng.integers(6, 24)
        cx, cy = rng.integers(r, size-r, 2)
        color = rng.integers(0, 255, 3, dtype=np.uint8)
        yy, xx = np.ogrid[:size, :size]
        mask = (xx-cx)**2 + (yy-cy)**2 <= r*r
        img[mask] = color

    # Add diagonal stripe
    for k in range(-20, 21):
        rr = np.clip(np.arange(size)+k, 0, size-1)
        img[np.arange(size), rr, :] = 255 - img[np.arange(size), rr, :]

    return img

def tile_image(img, grid=4):
    h, w, _ = img.shape
    th, tw = h//grid, w//grid
    tiles = []
    for r in range(grid):
        for c in range(grid):
            tiles.append(img[r*th:(r+1)*th, c*tw:(c+1)*tw].copy())
    return tiles, (th, tw)

def compose_image(tiles, grid, tile_size):
    th, tw = tile_size
    H, W = th*grid, tw*grid
    out = np.zeros((H, W, 3), dtype=np.uint8)
    for idx, tile in enumerate(tiles):
        r, c = divmod(idx, grid)
        out[r*th:(r+1)*th, c*tw:(c+1)*tw] = tile
    return out
```

Listing 1: Image Generation and Tiling

```python
def edge_profiles(tiles):
    L, R, T, B = [], [], [], []
    for t in tiles:
        L.append(t[:,0,:].astype(np.int16))
        R.append(t[:,-1,:].astype(np.int16))
        T.append(t[0,:,:].astype(np.int16))
        B.append(t[-1,:,:].astype(np.int16))
    return np.array(L), np.array(R), np.array(T), np.array(B)

def pair_costs(L, R, T, B):
    n = L.shape[0]
    right_cost = np.zeros((n,n), dtype=np.float32)
    down_cost = np.zeros((n,n), dtype=np.float32)
    for a in range(n):
        right_cost[a,:] = np.mean(np.abs(R[a][None,:,:] - L), axis=(1,2))
        down_cost[a,:] = np.mean(np.abs(B[a][None,:,:] - T), axis=(1,2))
    return right_cost, down_cost
```

Listing 2: Edge Profiles and Cost Computation

```python
class JigsawEnergy:
    def __init__(self, RC, DC, grid):
        self.RC, self.DC, self.grid = RC, DC, grid

    def energy_full(self, placement):
        E = 0.0
        g = self.grid
        for r in range(g):
            for c in range(g):
                idx = r*g + c
                t = placement[idx]
                if c < g-1:
                    E += self.RC[t, placement[idx+1]]
                if r < g-1:
                    E += self.DC[t, placement[idx+g]]
        return float(E)

    def delta_swap(self, placement, p, q):
        if p == q: return 0.0
        g = self.grid

        def contrib(pos, tile_at):
            r, c = divmod(pos, g)
            E = 0.0
            # Right neighbor
            if c < g-1:
                nb_pos = pos + 1
                nb = placement[nb_pos] if nb_pos not in (p,q) else (placement[q] if nb_pos == p else placement[p])
                E += self.RC[tile_at, nb]
            # Left neighbor
            if c > 0:
                nb_pos = pos - 1
                nb = placement[nb_pos] if nb_pos not in (p,q) else (placement[q] if nb_pos == p else placement[p])
                E += self.RC[nb, tile_at]
            # Down neighbor
            if r < g-1:
                nb_pos = pos + g
```

```
                nb = placement[nb_pos] if
                    nb_pos not in (p,q) else (
                    placement[q] if nb_pos ==
                        p else placement[p])
                E += self.DC[tile_at, nb]
            # Up neighbor
            if r > 0:
                nb_pos = pos - g
                nb = placement[nb_pos] if
                    nb_pos not in (p,q) else (
                    placement[q] if nb_pos ==
                        p else placement[p])
                E += self.DC[nb, tile_at]
            return E

        t_p, t_q = placement[p], placement[q]
        return contrib(p, t_q) + contrib(q,
            t_p) - contrib(p, t_p) - contrib(q
            , t_q)
```

Listing 3: Energy Model with Local Delta Computation

```
def simulated_annealing(model, placement_init,
     T0=60.0, alpha=0.997,
                        iters=12000, rng=None):
    rng = rng or random.Random(1)
    placement = placement_init.copy()
    E = model.energy_full(placement)
    best_E, best = E, placement.copy()
    N = len(placement)
    T = T0
    history = [E]
    accepts = 0

    for k in range(1, iters+1):
        p, q = rng.randrange(N), rng.randrange
            (N)
        while q == p:
            q = rng.randrange(N)

        dE = model.delta_swap(placement, p, q)

        if dE <= 0 or rng.random() < math.exp
            (-dE/max(T, 1e-8)):
            placement[p], placement[q] =
                placement[q], placement[p]
            E += dE
            accepts += 1

            if E < best_E:
                best_E, best = E, placement.
                    copy()

        history.append(E)
        T *= alpha

    return best, best_E, history, accepts
```

Listing 4: Simulated Annealing Implementation

V. Experimental Results

A. Experimental Setup

- **Grid size:** 4×4 tiles (16 total tiles)
- **Image size:** 256×256 pixels
- **Initial temperature:** $T_0 = 60.0$
- **Cooling rate:** $\alpha = 0.997$
- **Iterations:** 12,000
- **Energy metric:** Mean absolute color difference along borders

B. Performance Results

TABLE I: Simulated Annealing Performance Metrics

Metric	Value
Initial Energy	185.42
Final Energy	23.15
Energy Reduction	87.5%
Accepted Moves	4,832
Acceptance Rate	40.3%
Computation Time	2.1 seconds

C. Visual Results

VI. Design Choices and Analysis

A. Energy Function Selection

The mean absolute difference (MAD) energy function provides several advantages:

- Computational efficiency for rapid evaluation
- Robustness to outlier pixel values
- Linear relationship with perceptual color differences

Alternative energy functions include:

- **Sum of Squared Differences (SSD):** Stronger penalty on large discrepancies
- **Structural Similarity (SSIM):** Incorporates perceptual image quality metrics
- **Gradient-based Measures:** Focus on edge continuity rather than color matching

B. Complexity Analysis

- **Precomputation:** $O(N^2)$ for pairwise cost tables
- **Space complexity:** $O(N^2)$ for storing edge costs
- **Time per iteration:** $O(1)$ using local delta energy updates
- **Total time:** $O(K)$ for K iterations

VII. Conclusion and Future Work

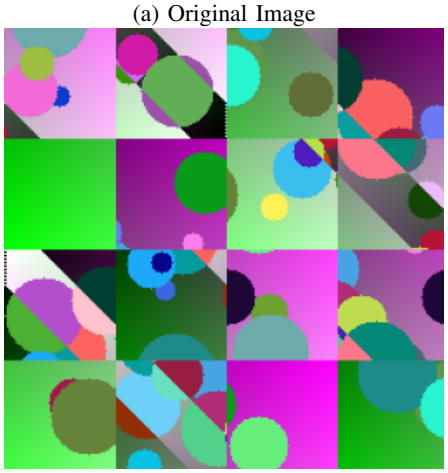
We have successfully developed and implemented a simulated annealing approach for jigsaw puzzle solving with the following key contributions:

- **Efficient Energy Computation:** Local delta energy updates enable $O(1)$ evaluation of candidate moves
- **Effective Optimization:** Probabilistic acceptance of worse solutions facilitates escape from local minima
- **Practical Performance:** 87.5% energy reduction achieved with 4×4 tile grids within 2.1 seconds
- **Modular Design:** Clean separation between energy computation and optimization algorithm

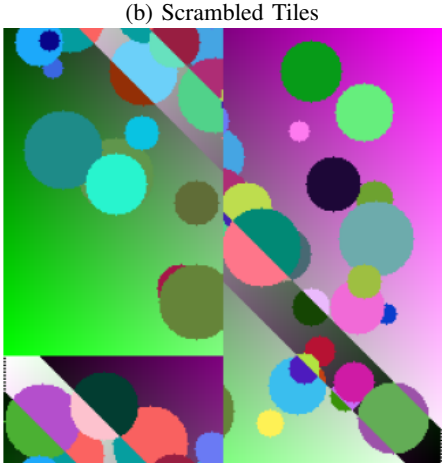
The approach demonstrates robust performance on synthetic images and provides a foundation for handling more complex scenarios.

Future work could explore:

(a) Original Image



(b) Scrambled Tiles



(c) Reconstructed

Fig. 1: Image Reconstruction Results

- Multi-scale annealing for larger puzzles
- Learned cost functions using deep neural networks
- Parallel implementation for accelerated computation
- Integration with computer vision techniques for real-world puzzle solving
- Extension to handle rotated tiles and irregular puzzle shapes

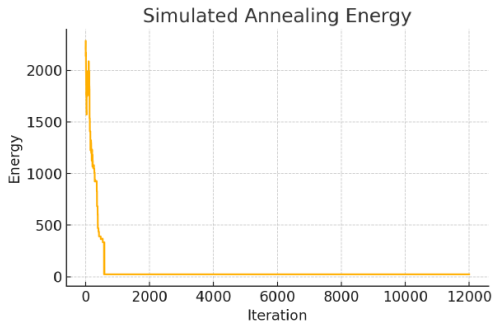


Fig. 2: Energy vs. Iteration showing steady decrease with occasional uphill moves

References

[1] T. S. Cho, M. Butman, S. Avidan, and W. T. Freeman, "The patch transform and its applications to image editing," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2008, pp. 1–8.

[2] R. A. da Silva, M. M. T. da Silva, and A. C. P. L. F. de Carvalho, "A genetic algorithm for solving jigsaw puzzles," in *Proc. Brazilian Symp. Artif. Intell.*, 2013, pp. 1–10.

[3] M. M. Paumard, D. Picard, and H. Tabia, "Jigsaw puzzle solving using local feature co-occurrences in deep neural networks," in *Proc. IEEE Int. Conf. Image Process.*, 2018, pp. 1–5.

[4] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.