Vietnam National University,
Ho Chi Minh City

University of Science

Faculty of Information Technology

Project 02: Hashiwokakero

CS14003 – Introduction to Artificial Intelligence

Ngo Nguyen The Khoa 23127065
Bui Minh Duy 23127040
Nguyen Le Ho Anh Khoa 23127211

April 5, 2025

Contents

1 Group Information

- **Subject:** Introduction to Artificial Intelligence.
- **Class:** 23CLC09.
- **Lecturer:** Bui Duy Dang, Le Nhut Nam.
- **Team members:**

No.	Fullname	Student ID	Email
1	Ngo Nguyen The Khoa	23127065	nntkhoa23@clc.fitus.edu.vn
2	Bui Minh Duy	23127040	bmduy23@clc.fitus.edu.vn
3	Nguyen Le Ho Anh Khoa	23127211	nlhakhoa23@clc.fitus.edu.vn

2 Project Information

- **Name:** Hashiwokakero.
- **Developing Environment:** Visual Studio Code (Windows, WSL).
- **Programming Language:** Python.
- **Libraries and Tools:**
  - **Libraries:**
    - **uv:** An extremely fast Python package and project manager, written in Rust.
    - **PySAT:** The power of SAT technology in Python
      - `pysat.formula.CNF`: class for manipulating CNF formulas.
      - `pysat.solvers.Glucose42`: Glucose 4.2.1 SAT solver.
      - `pysat.pb.PBEnc`: used for creating a CNF encoding of a weighted EqualsK constraint, i.e. of $\sum_{i=1}^{n}(a_i \cdot x_i) = k$
  - **Tools:**
    - **Git**, **GitHub**: Source code version control.
    - **ChatGPT**, **DeepSeek**: Scaffolding and debugging.
    - **Visual Studio Code**: Code editor for Python, Latex.
- **Demo video:** **Google Drive**.

3 Work assignment table

No.	Task Description	Assigned to	Rate
1	Solution description: Describe the correct logical principles for generating CNFs.	The Khoa	100%
2	Generate CNFs automatically.	The Khoa	100%
3	Use the PySAT library to solve CNFs correctly.	The Khoa	100%
4	Implement A* to solve CNFs without using a library.	Anh Khoa	100%
5	Implement Brute-force algorithm to compare with A* (speed).	Anh Khoa	100%
5	Implement Backtracking algorithm to compare with A* (speed).	Minh Duy	100%
6	Write a detailed report on formulating and generating CNF.	The Khoa	100%
7	Thoroughness in analysis and experimentation.	All	100%
8	Provide at least 10 test cases with different sizes (7×7, 9×9, 11×11, 13×13, 17×17, 20×20) to verify the solution.	Anh Khoa, Minh Duy	100%
9	Compare results and performance.	The Khoa, Minh Duy	100%

4 Self-evaluation

No.	Task Description	Rate
1	Describe the correct logical principles for generating CNFs.	100%
2	Generate CNFs automatically.	100%
3	Use the PySAT library to solve CNFs correctly.	100%
4	Implement A* to solve CNFs without using a library.	100%
5	Implement Brute-force algorithm.	100%
5	Implement Backtracking algorithm.	100%
6	Detailed report on formulating and generating CNF.	100%
7	Thoroughness in analysis and experimentation.	100%
8	Provide at least 10 test cases with different sizes to verify the solution.	100%
9	Compare results and performance.	100%

5 CNF

5.1 Logical principles for generating CNFs

- **Variables**

  For every pair of adjacent islands A and B, define two Boolean variables:

  - $x^1_{A,B}$: A single bridge exists between islands A and B.
  - $x^2_{A,B}$: A double bridge exists between islands A and B.

- **Constraints**

  - At most 2 bridges between two islands, that means there can be 0, 1, or 2 bridges between two islands.
  - The number of bridges connected to an island must equal the island's number.
  - No crossing bridges

5.2 Formulate CNF Constraints

- **Mutual Exclusion**

  Ensure that two variables cannot be true at the same time:

$$\neg x^1_{A,B} \vee \neg x^2_{A,B}$$

- **Island Degree Constraints**

  For each island A with number n, the sum of bridges connected to it must equal n Let $Adj(A)$ be the set of islands adjacent to A. For example, if A connects to B, C, D, then:

$$\sum_{X \in Adj(A)} x^1_{A,X} + 2 \cdot x^2_{A,X} = n$$

  This can be encoded using **pseudo-Boolean constraints** (e.g., `PBEnc.equals` in PySAT).

- **No crossing bridges**

  For every horizontal bridge (A, B) and vertical bridge (C, D) that cross, add clauses to block coexistence:

$$\neg x^1_{A,B} \vee \neg x^1_{C,D}$$
$$\neg x^1_{A,B} \vee \neg x^2_{C,D}$$
$$\neg x^2_{A,B} \vee \neg x^1_{C,D}$$
$$\neg x^2_{A,B} \vee \neg x^2_{C,D}$$

6 Algorithms' Implementations

6.1 Solve using PySAT

For *'Island degree constrants'*, we use `PBEnc.equals` to generate the clauses.

```python
def encode_hashi(
    grid: Grid,
    pbenc: int = PBEncType.bdd,
    cardenc: int = CardEncType.mtotalizer,
    *,
    use_pysat: bool = False,
):
    # ...(previous codes)
    for idx, _, _, degree in islands:
        lits = island_incident[idx]
        weights = [[2, 1][x & 1] for x in lits]
        if not lits and degree:
            print(f"[unsat]: no edges for island {idx}")
            cnf.append([])
            continue

        if use_pysat:
            clauses = PBEnc.equals(
                lits,
                weights,
                degree,
                var_counter,
                encoding=pbenc,
            ).clauses
        else:
            clauses = encode_pbequal(lits, weights, degree, var_counter)

        cnf.extend(clauses)
        var_counter = max(
            var_counter,
            max(abs(_) if _ else 1 for c in clauses for _ in c) if clauses else 1,
        )
    # ...(remaining codes)
```

(the `PBEnc.equals` function is used to generate the clauses for the island degree constraints.)
(the `encode_pbequal` function is used to generate the clauses for the island degree constraints without using PySAT.)

The solver using PySAT looks like this:

```python
for pbenc in range(7):
    for cardenc in range(10):
        try:
            cnf, edge_vars, islands, _ = encode_hashi(grid, pbenc, cardenc,
            ↪  use_pysat=True)
            with Glucose42(bootstrap_with=cnf) as solver:
                while solver.solve():
                    model = solver.get_model()
                    num_clauses = solver.nof_clauses()
                    if not model or (num_clauses and num_clauses > len(cnf.clauses)):
                        break

                    if validate_solution(islands, edge_vars, model):
                        return extract_solution(model, edge_vars), islands

                    solver.add_clause([-x for x in model])
```

6.2 A* Search with heuristic

The A* algorithm is a widely used technique for pathfinding and graph traversal. It relies heavily on heuristics to estimate the future cost of reaching the goal. Essentially, A* extends uniform cost search by incorporating a heuristic function to guide its path selection, making it more efficient in many cases. The choice of heuristic depends on the specific problem domain, highlighting the importance of domain knowledge. For A* to be consistent, the modified cost function must always remain greater than zero.

Pseudocode

Algorithm 1 A* Search (*start, goal, heuristic*)

priority queue ← [(start, cost = 0, estimated total cost = heuristic(start))]
while priority queue is not empty **do**
    (node, cost) ← dequeue(priority queue)
    if node = goal **then**
        return path
    end if
    for all neighbor in valid moves **do**
        new cost ← cost + move cost
        estimated total cost ← new cost + heuristic(neighbor)
        if neighbor not visited or new cost < previous cost **then**
            mark neighbor as visited
            enqueue(priority queue, (neighbor, new cost, estimated total cost))
        end if
    end for
end while
return failure

Implementation

-

Heuristics in A* Algorithm

A* search uses a heuristic function $h(n)$ to estimate the cost from a given state to the goal. The total cost function is defined as:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$ is the actual cost from the start state to the current state n.
- $h(n)$ is the estimated cost from n to the goal state.

For Sokoban, common heuristic choices include:

- **Manhattan Distance:** The sum of the absolute differences between the x and y coordinates of each stone and its nearest goal position.
- **Hungarian Algorithm:** Computes an optimal assignment of stones to switches, minimizing the total weighted Manhattan distance.

A* is optimal if $h(n)$ is *admissible* (never overestimates the true cost) and *consistent* (satisfies the triangle inequality).

Time and Space Complexity

Time Complexity: $O(b^d)$ in the worst case, but with a good heuristic, it can be significantly reduced. If the heuristic is admissible and consistent, A* is optimal and complete.

Space Complexity: $O(b^d)$, as it keeps all generated nodes in memory.

7 Algorithms Benchmark

7.1 Tests and Results

- **Test cases description**
  - Each map size in (7 × 7), (9 × 9), (11 × 11), (13 × 13), (17 × 17), (20 × 20) has at least 5 test cases with different difficulty levels.
  - The test cases are generated randomly and the difficulty levels are classified as easy, medium, hard, and very hard. The results of the tests are shown in the table below. The time is measured in milliseconds (ms) and the peak memory is measured in kilobytes (KB).
- **Experiment results**
  - Solving Time (ms)
  - Peak Memory (KB)

7.2 Challenges

- **Challenges in the implementation**
  - The algorithm is not optimal for large maps, which leads to a long solving time and high peak memory usage.
  - The algorithm is not able to solve some test cases with a very high difficulty level.
- **Challenges in**

7.3 Comparison

-

8 References

1.