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.

```
def encode_hashi(
    grid: Grid,
    pbenc: int = PBEncType.bdd,
    cardenc: int = CardEncType.mtotalizer,
    *,
    use_pysat : bool = False,
):
    # ...(previous codes)
    for idx, ,,degreeinislands :
        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_else1forcinclausesfor_inc)ifclauseselse1,
        )
    # ...(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:

```
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)assolver :
                while solver.solve():
                    model = solver.get_model()
                    num_clauses = solver.nof_clauses()
                    if not model or (num_clausesandnum_clauses > len(cnf.clauses)) :
                        break

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

                    solver.add_clause([−xforxinmodel])
```

6.2 A* Search with heuristic

A* search is a heuristic-driven algorithm that can be effectively applied to solving CNF problems by navigating the space of partial variable assignments. In this approach, each state represents a partial assignment of truth values to the CNF variables, and the algorithm uses a heuristic to estimate the cost from the current state to a complete, satisfying solution. Specifically, the heuristic can be defined as the number of clauses that are fully assigned but remain unsatisfied. This metric guides the search by prioritizing states that are closer to satisfying all clauses, thus reducing the exploration of paths that are likely to lead to dead ends. By systematically expanding these states and pruning those that immediately violate any clause, A* efficiently converges on a complete assignment that satisfies the entire CNF formula, if one exists.

Pseudocode

Algorithm 1 A* Search for Hashiwokakero (*grid*)

Input: *grid* (Hashiwokakero puzzle grid)
Output: Solution to the puzzle or failure
Encode the puzzle into CNF: *cnf, edge_vars, islands, variables* ← *encode_hashi(grid)*
if no islands exist **then**
    if *check_hashi([], [])* **then**
        return *generate_output(grid, [], [])*
    else
        return failure
    end if
end if
Initialize priority queue: *open_list* ← [(initial_state)]
Initialize *nodes_expanded* ← 0
while *open_list* is not empty **do**
    state ← dequeue(*open_list*)
    nodes_expanded ← *nodes_expanded* + 1
    if *state.assignment* is complete and satisfies all clauses **then**
        return *generate_output(grid, islands, solution)*
    end if
    for all neighbor states of *state* **do**
        if neighbor is valid (no violated clauses) **then**
            Compute heuristic: $h \leftarrow$ *compute_heuristic(neighbor)*
            Compute cost: $f \leftarrow g + h$
            Enqueue *neighbor* into *open_list*
        end if
    end for
    if *nodes_expanded* mod 100,000 = 0 **then**
        Print progress: "Expanded *nodes_expanded* nodes..."
    end if
end while
return failure

Implementation

- **compute_heuristic:** Computes a heuristic value by counting the number of clauses that are fully assigned but unsatisfied. This function guides the search towards more promising partial assignments.
- **is_clause_violated:** Checks whether a clause is violated by the current assignment by verifying that all its variables are assigned and that none of the literals satisfy the clause.
- **is_complete_assignment:** Determines if an assignment is complete by ensuring that every variable has been assigned a truth value.
- **check_full_assignment:** Verifies that a complete assignment satisfies all clauses in the CNF, ensuring the overall solution is valid.

- **expand_state:** Expands the current state by assigning the next variable with both possible truth values, while pruning any branch where a clause is already violated.
- **solve_with_astar:** Encodes the Hashi puzzle into a CNF and employs the A* search algorithm to explore assignments, validate complete solutions, and generate the final puzzle output.

Heuristics in A* Algorithm

The code implements A* search to solve a CNF-encoded Hashi puzzle. In this implementation, the total cost function is defined as:

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

where:

- $g(n)$: This is represented by the current level of the search, which corresponds to the number of variables that have been assigned values so far.
- $h(n)$: The heuristic is computed by the compute_heuristic function. It counts the number of clauses that are fully assigned yet remain unsatisfied. This estimate reflects how "far" the current partial assignment is from satisfying the overall CNF.

The algorithm starts with an empty assignment and iteratively expands states by assigning the next variable (selected in sorted order). It prunes any branch where a clause is already violated to avoid unnecessary exploration. Each new state is pushed into a priority queue (implemented with a heap) based on its $f(n)$ value, ensuring that states estimated to be closer to a solution are explored first.

Once a complete assignment is reached, the algorithm verifies that it satisfies all clauses, extracts a model, validates it against the puzzle's constraints, and finally generates the corresponding output if the solution is correct. Overall, the design carefully integrates the A* components $g(n)$, $h(n)$, and state expansion—to efficiently search for a valid solution.

Time and Space Complexity

Time Complexity: $O(b^d)$ in the worst case, where b is the branching factor and d is the solution depth. However, a well-designed heuristic significantly reduces the search space. If the heuristic is admissible and consistent, A* efficiently finds an optimal solution.

Space Complexity: $O(b^d)$, as the algorithm stores all generated nodes in memory. This can be a limiting factor for large problem instances but ensures completeness and optimality.

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.