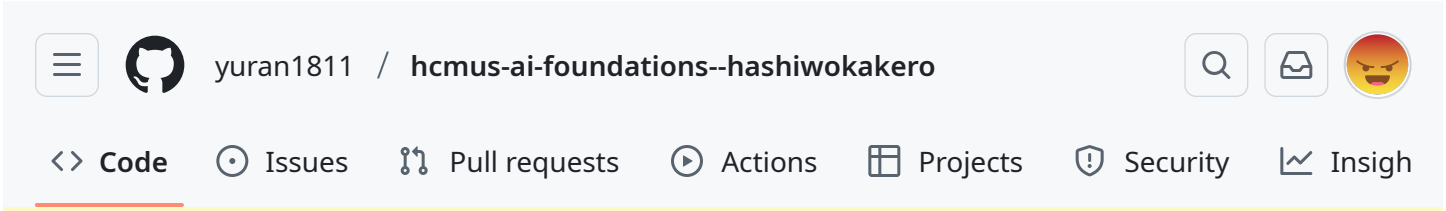
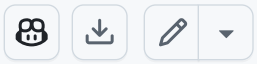


This repository was archived by the owner on Apr 6, 2025. It is now read-only.

hcmus-ai-foundations--hashiwokakero / Report / **Report.pdf**

464 KB



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

- **Special cases**

  - **One way connection:** If an island has only one neighbor in a cardinal direction (north, south, east, or west), it must connect to that neighbor—using either a single or double bridge based on its required bridge count.
  - **Do not connect islands of 1 between themselves:** If a 1-bridge island has multiple connection options but only one leads to an island with ≥ 2 bridges, that connection must be made. Otherwise, two 1-bridge islands might link to each other and become isolated, breaking the rule that all islands must be part of a single connected group.
  - **Between two islands of 2, no double bridge can be placed between them:** If two islands both have the degree of 2, they can only connect with a single bridge. If a double bridge is placed between them, it would create a situation where the whole map is separated into at least 2 components, and that situation violates the game's rule.
  - **Six bridges with special neighbors:** An island with 6 bridges can sometimes follow the same logic as those with 7 or 8. If it has neighbors in only 3 directions, all bridges must be drawn and doubled to reach 6. If one neighbor is a 1, then only 5 bridges remain for the other 3 directions. Since two directions doubled give just 4 bridges, each direction must have at least one bridge, allowing one bridge to be drawn in all three directions immediately.
  - **Number 7:** An island labeled 7 means one of the 8 possible bridges is missing—so every direction must have at least one bridge. It's safe to draw one

bridge in all four directions, then later decide which direction needs a second bridge.

- **Number 8:** An island can have at most 8 bridges—2 in each of the 4 directions (north, south, east, west). So, if an island is labeled with 8, all its possible

bridges must be drawn immediately, as they are guaranteed to be part of the solution.

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).
  We've also implemented manual encoding versions for this constraint in two diffrent approaches ('Tseytin Transformation' and 'Dynamic Programming'), but it produces a large amount of clauses (larger than PySAT's encoding does), so we've used the `PBEnc` instead of our own implementations (they still remains in the source, but not be used as default).

- **No crossing bridges**

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

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

- **Special cases**

  - **Do not connect islands of 1 between themselves:** Mark the variable of both single and double bridges as negative.
  - **Between two islands of 2, no double bridge can be placed between them:** Just mark the variable of double bridge as negative

then: just mark the variable of double bridge as negative.

- **Remain special cases:** Just follow the rules each case and mark the variables as positive or negative.

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)
    if use_pysat:
        for idx, _, _, degree in islands:
            lits = [v for _, v in 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

            clauses = (
                PBEnc.equals(
                    lits,
                    weights,
                    degree,
                    var_counter,
                    encoding=pbenc,
                ).clauses
                if not use_self_pbenc
                else encode_pbequal(
                    lits,
                    weights,
                    degree,
                    var_counter,
                )
            )
            cnf.extend(clauses)
            var_counter = update_var_counter(var_counter, clauses)
```

```
    # ...(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, but it's still not stable because of the large amount of generated clauses.)

The solver using PySAT looks like this:

```python
    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

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
    return failure
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 ← compute_heuristic(neighbor)
            Compute cost: f ← g + h
            Enqueue neighbor into open_list
        end if
    end for
end while
return failure
```

Implementation

- **compute_heuristic:** Compute the heuristic value for a given partial assignment. The heuristic is the number of clauses that are fully assigned but unsatisfied.
- **is_clause_violated:** Checks if a clause is violated by confirming that all its variables are assigned and none of its literals evaluate to True. It is used to prune branches that cannot lead to a valid solution.
- **is_complete_assignment:** Determines whether every variable in the problem has been assigned a value. It simply checks if the assignment covers all variables.

- **check_full_assignment:** Verifies that a complete assignment satisfies every clause in the CNF. It confirms the validity of the overall solution by ensuring no clause is left unsatisfied.
- **expand_state:** Expands the current state by assigning the next unassigned variable with both True and False, generating new partial assignments. It prunes any branch immediately if a clause is violated by the new assignment.
- **solve_with_astar:** Encodes the Hashi puzzle into a CNF and employs an A* search to explore the space of assignments. It validates complete solutions and generates the final puzzle output upon finding a valid assignment.

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.

Strengths

Efficient Search: A*

- **Efficient Search:** A* narrows down the search space using a heuristic function, prioritizing more promising configurations. This makes it much faster than brute-force search, particularly for puzzles of moderate size.
- **Optimality:** When using an admissible heuristic (one that does not overestimate the cost), A* guarantees that the solution found will be optimal, providing the best possible configuration.
- **Scalability:** The use of heuristics helps A* scale better than brute-force algorithms, as it avoids exploring irrelevant or unlikely configurations. It can handle puzzles of increasing size more efficiently.