Ho Chi Minh University of Science
Department of Information Technology

Lab02 Report: Hashiwokakero

Course: Introduction to Artificial Intelligence

Instructor: Bui Tien Len
Le Nhut Nam
Vo Nhat Tan

23127296 - Nguyen Thanh Luan
23127302 - Tran Quang Phuc
23127539 - Nguyen Thanh Tien
23127543 - Vu Van Vu

August, 2025

Contents

1 Introduction

Table 1: Contribution table

Work	**Member**
Formulate CNF constraints	Nguyen Thanh Luan
Automate CNF generation	Nguyen Thanh Luan
Design input files	Tran Quang Phuc
Implement backtracking	Nguyen Thanh Tien
Implement brute-force	Vu Van Vu
Implement A*	Tran Quang Phuc
Solve using PySat	Nguyen Thanh Luan
Video demonstration	Vu Van Vu

Table 2: Requirement completion table

Requirements	**Completion level**
Design at least 10 input files	Yes
Formulate CNF constraints	Yes
Automate CNF generation	Yes
Use PySat to solve	Yes
Implement backtracking	Yes
Implement brute-force	Yes
Implement A*	Yes
Detailed explanation of algorithms	Yes
Results and evaluation	Yes
Report	Yes
Demonstration video	Yes

2 Problem Introduction

Hashiwokakero (also known as Bridges) is a Japanese puzzle. It is played on a rectangular grid. There are islands on the grid. The player can draw bridges between the islands in a straight line. The goal is to make all the islands form a single connected group. The puzzle follows the following rules:

- They must begin and end at distinct islands, travelling a straight line in between.
- They must not cross any other bridges or islands.
- They may only run orthogonally (i.e. they may not run diagonally).
- At most two bridges connect a pair of islands.
- The number of bridges connected to each island must match the number on that island.
- The bridges must connect the islands into a single connected group.

3 Encode Hashiwokakero to CNF

3.1 Logical Variable Definition

To obtain the CNF formula for the Hashiwokakero puzzle, first we introduce a logical variable

$$x_{i,j,i',j',k}$$

where

- (i, j) and (i', j') is the location of the 2 islands.
- k is the number of bridges between 2 islands.$(k = 1, 2)$

3.2 CNF constraints

The following constraints are converted to CNF in order to solve the Hashiwokakero puzzle:

3.2.1 Bridges must begin and end at distinct islands, travelling in a straight line

- The endpoints (i, j) and (i', j') must be different: $(i, j) \neq (i', j')$.
- The bridge must lie on the same row or the same column:

$$\forall i, j, i', j', k \quad (x_{i,j,i',j',k} \rightarrow [(i = i' \wedge j \neq j') \vee (j = j' \wedge i \neq i')])$$

3.2.2 Bridges must not cross any other bridges

Two bridges $x_{i_1,j_1,i'_1,j'_1,k_1}$ and $x_{i_2,j_2,i'_2,j'_2,k_2}$ intersect if:

- $i_1 = i'_1$, $j_2 = j'_2$ (one runs horizontal, one runs vertical),
- $i_2 < i_1 < i'_2$, and $j_1 < j_2 < j'_1$ (they cross each other),
- $\{i_1, j_1\}, \{i'_1, j'_1\} \neq \{i_2, j_2\}, \{i'_2, j'_2\}$ (they don't run from same islands).

3.2.3 At most two bridges connect a pair of islands

The number of bridges is equal or smaller than 2:

$$\forall i, j, i', j', k \quad (x_{i,j,i',j',k} \rightarrow 0 \leq k \leq 2)$$

3.2.4 The number of bridges connected to each island must match the number on that island

Let $B_{i,j}$ be the number of bridges connecting to the island located at (i, j).

$$\sum_{\substack{i',j',k \\ (i',j',k) \in D}} k \cdot x_{i,j,i',j',k} = B_{i,j}$$

where

- D is the set of all islands (i', j') that can go directly to (i, j).

3.3 Implementation

3.3.1 Generating bridge variables

Firstly, the logical variables are generated by looping through the input file. The idea is to consider 4 direction from an island (i, j). If an island (i', j') is met during a direction, then two variables $x_{i,j,i',j',1}$ and $x_{i,j,i',j',2}$ are added to the list of variables.

3.3.2 Adding constraints ensuring that $x_{i,j,i',j',1}$ and $x_{i,j,i',j',2}$ cannot coexist

For every pair $x_{i,j,i',j',1}$ and $x_{i,j,i',j',2}$, we add the following condition to ensure they cannot coexist.

$$\neg v_1 \vee \neg v_2$$

3.3.3 Adding constraints ensuring that the number of bridges to the island must match the number on that island

For every island, we do the following:

- Get all the logical variables representing bridges connect directly to that island.
- Find the combinations of those logical variables that satisfy the condition.
- We define a new variable c for every combination.
- For every combination v_1, v_2, ..., v_n, we add $\neg v_1 \vee \neg v_2 \vee \cdots \vee v_n \vee c$, and for every logical variable v_i in the combination, we add $\neg c \vee v_i$. These steps are the equivalence of adding $c \iff v_1 \wedge v_2 \wedge \cdots \wedge v_n$.
- For every combination, we also add the negative of all the logical variables in first step that are not in the combinations.
- For every pair of combinations c_1 and c_2, we add $\neg c_1 \vee \neg c_2$, to ensure at most one combination is valid.
- For all combinations c_1, c_2, ..., c_n, we add $c_1 \vee c_2 \vee \cdots \vee c_n$ to ensure at least one combination is correct.

3.3.4 Adding constraints ensuring no bridges cross each other

For every logical variable $x_{i_1,j_1,i'_1,j'_1,k_1}$ (or v_1), we do the following:

- For every logical $x_{i_2,j_2,i'_2,j'_2,k_2}$ (or v_2), check the following condition:
  - $i_1 = i'_1$, $j_2 = j'_2$ (one runs horizontal, one runs vertical),
  - $i_2 < i_1 < i'_2$, and $j_1 < j_2 < j'_1$ (they cross each other),
  - $\{i_1, j_1\}, \{i'_1, j'_1\} \neq \{i_2, j_2\}, \{i'_2, j'_2\}$ (they don't run from same islands).
- If all the conditions are satisfied we add $\neg v_1 \vee \neg v_2$ to ensure these bridges cannot coexist.

4 Algorithm Explanation

4.1 Backtracking

4.1.1 Intuition

Backtracking is a trial-and-error technique. If at some point the current path cannot lead to a valid solution, the algorithm backtracks.

4.1.2 Implementation

1. **Generate variables:** Assign Boolean variables to bridges between islands.
2. **Add logical constraints:**
    - Ensure no double connections
    - Enforce the degree constraints.
    - Prevent bridges from crossing each other.
3. **Initialize the assignment:** Create an assignment list to track which variables are currently set to `True` (bridge exists), `False` (bridge does not exist), or `None` (not yet assigned).
4. **Partial constraint checking:** During backtracking, after assigning a value to a variable, check whether all fully-assigned clauses are still satisfied. This prevents exploring invalid branches.
5. **Recursive backtracking:** Try assigning `True` or `False` to each variable in order.
    - If the partial assignment does not violate any constraints, proceed to assign the next variable.
    - If a violation is detected, undo the assignment and try the opposite value.
    - If all variables are assigned, check if the resulting graph is fully connected.
6. **Solution or failure:** If a complete valid assignment is found and the graph is connected, the solution is displayed. Otherwise, the algorithm concludes that no valid solution exists.

4.2 Brute-force

4.2.1 Intuition

The brute-force approach tries all possible combination of bridges to find the solution. It guarantees correctness, but consumes great amount of time.

4.2.2 Implementation

1. **Generate variables:** Define Boolean variables for all bridges.
2. **Add logical constraints:**
    - Prevent double bridges.

- Ensure that the total number of bridges connected to each island matches its degree.
- Forbid crossing bridges.

3. **Iterate through all assignments:** Try every possible combination of `True` or `False` for all variables.
4. **Constraint checking:** For each assignment, check whether all logical clauses are satisfied. If any clause is unsatisfied, skip to the next assignment.
5. **Check connectivity:** If the assignment satisfies all clauses, check whether the resulting graph is connected.
6. **Display result or fail:** If a valid combination is found, display the solution and stop. If after all assignments but there's no solution -> this puzzle is unsolvable.

4.3 A*

4.3.1 Intuition

A* is a best-first search algorithm that explores the most promising partial assignments first, guided by a cost function.

4.3.2 Heuristics

To guide the search efficiently, A* uses a heuristic function $h(n)$ that estimates the "cost to goal" from a given partial assignment. The total cost function is defined as:

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

where:

- $g(n)$ is the actual number of variables assigned so far (the path cost).
- $h(n)$ is the estimated number of unsatisfied clauses.

4.3.3 Implementation

1. **Generate variables:** Represent all possible bridges between islands using Boolean variables.
2. **Add constraints:** Encode logical conditions into CNF clauses:
   - No duplicate bridges.
   - Degree of each island must match its number.
   - No crossing bridges.
3. **Initialize priority queue:** Start with an empty assignment and compute its $f(n)$ value.
4. **Search loop:**
   (a) Pop the assignment with the smallest $f(n)$ from the priority queue.

(b) If all variables are assigned and all clauses are satisfied, and the resulting graph is connected, return the solution.

(c) Otherwise, pick one unassigned variable and expand the current state by assigning it both `True` and `False`.

(d) For each expanded state, compute $f(n)$ and push it into the priority queue.

(e) Skip states that already violate a clause.

5. **Fail case:** If the priority queue is exhausted without finding a solution, report failure.

5 Results

6 Test Cases and Results

This section details the 10 test cases used to evaluate the performance of the implemented algorithms. Each test case includes the initial puzzle grid and the corresponding solution found by the program. The execution time for the PySAT solver is also provided for each case. Note that the solution displayed is not the program expected output formats, this version has been passed to a drawing function to make the result readable

6.1 Test Case 1 (4x5)

- **Input Puzzle:**

```
1 0 2 0 0
0 0 0 0 0
0 0 0 0 0
1 0 4 0 2
```

- **Output Solution:**

```
1 - 2
    |
    |
1 - 4 = 2
```

6.2 Test Case 2 (4x4)

- **Input Puzzle:**

```
3 0 0 2
0 0 0 0
0 0 0 1
3 0 0 1
```

- **Output Solution:**

```
3 - - 2
$     |
$     1
3 - - 1
```

6.3 Test Case 3 (4x4)

- **Input Puzzle:**

```
2 0 4 0 1
0 0 0 0 0
0 0 0 0 0
1 0 2 0 0
```

- **Output Solution:**

```
2 - 4 - 1
|   $
|   $
1   2
```

6.4 Test Case 4 (5x4)

- **Input Puzzle:**

```
0 1 0 2
0 0 0 0
0 0 0 3
0 0 0 0
2 0 0 2
```

- **Output Solution:**

```
  1     2
  |     $
  |     3
  |     |
  2 - - 2
```

6.5 Test Case 5 (7x7)

- **Input Puzzle:**

```
0 2 0 0 0 2 0
0 0 0 0 0 0 0
0 2 0 0 1 0 1
3 0 0 0 0 3 0
0 0 0 0 0 0 0
0 2 0 0 0 3 0
3 0 0 0 0 0 2
```

- **Output Solution:**

```
  2 - - - 2
  |       |
  2 - - 1 | 1
3 - - - - 3 |
$         | |
$ 2 = = = 3 |
3 - - - - - 2
```

6.6 Test Case 6 (8x8)

- **Input Puzzle:**

```
1 0 0 0 3 0 2 0
0 3 0 2 0 0 0 2
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 2 0 0 3 0 2 0
0 0 0 0 0 0 0 0
0 2 0 0 4 0 0 4
```

- **Output Solution:**

```
1 - - - 3 - 2
  3 = 2 |   |   2
  |     |   1   $
  |     |       $
  |     |       $
  2     3 - 2   $
  |     |   |   $
  |     |   1   $
  2 - - 4 = = = 4
```

6.7 Test Case 7 (9x9)

- **Input Puzzle:**

```
2 0 2 0 0 3 0 0 3 0 4
0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0
0 0 3 0 0 4 0 0 0 3 0
0 0 0 0 0 0 0 0 0 0 0
0 0 2 0 0 0 0 0 0 0 0
1 0 0 0 0 1 0 0 0 2 0
0 0 0 1 0 0 0 0 0 0 4
0 0 0 0 0 0 0 0 0 0 0
1 0 0 2 0 0 0 0 0 0 2
```

- **Output Solution:**

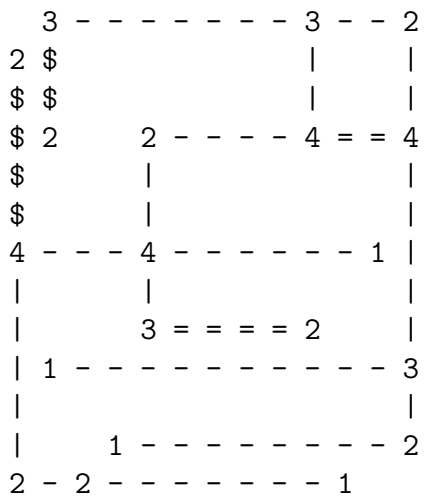
```
2 - 2 - - 3 - - 3 = 4
|         |         $
|         |         $
|   3 - - 4 - - - 3 $
|   $     |       $ $
|   2     |       $ $
1         1       2 $
    1 - - - - - - - 4
                    |
1 - - 2 - - - - - - 2
```

6.8 Test Case 8 (12x12)

- **Input Puzzle:**

```
0 3 0 0 0 0 0 0 0 3 0 0 2
2 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 2 0 0 2 0 0 0 0 4 0 0 4
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
4 0 0 0 4 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 3 0 0 0 0 2 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 3
0 0 0 0 0 1 0 0 0 0 0 0 2
2 0 2 0 0 0 0 0 0 0 0 1 0
```

- **Output Solution:**

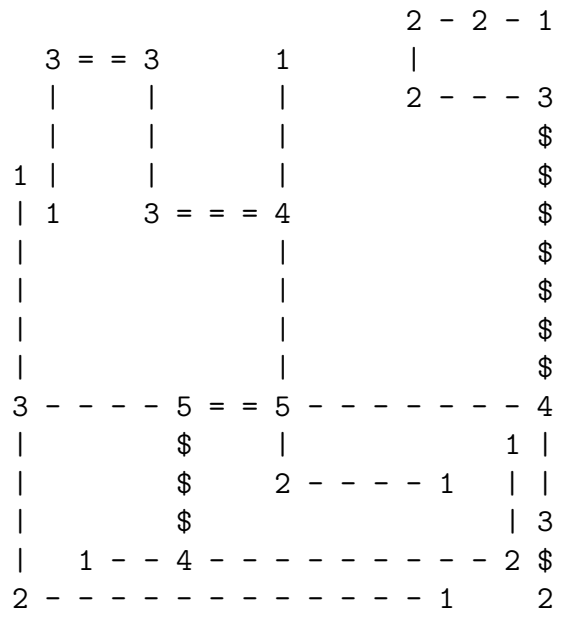


6.9 Test Case 9 (13x13)

- **Input Puzzle:**

```
0 0 0 0 0 0 0 0 0 0 2 0 2 0 1
0 3 0 0 3 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 2 0 0 0 3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 3 0 0 0 4 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 5 0 0 5 0 0 0 0 0 0 4
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 2 0 0 0 0 1 0 0 3
0 0 0 1 0 0 4 0 0 0 0 0 0 0 0 2
2 0 0 0 0 0 0 0 0 0 0 0 1 0 2
```

- **Output Solution:**

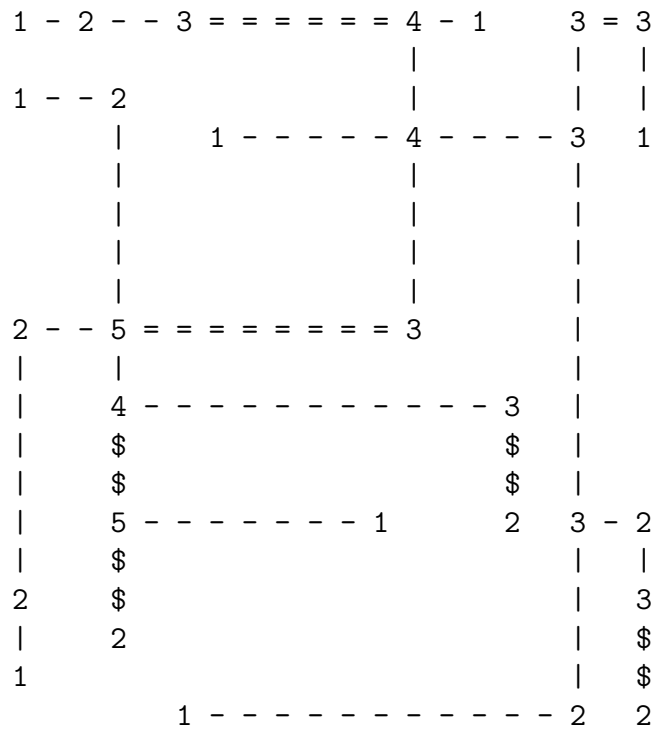


6.10 Test Case 10 (17x17)

- **Input Puzzle:**

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 2 0 0 3 0 0 0 0 0 0 4 0 1 0 0 3 0 3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 4 0 0 0 0 3 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 5 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 5 0 0 0 0 0 0 0 1 0 0 0 2 0 3 0 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3
0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 2 0 2
```

- **Output Solution:**



6.11 Results and evaluation

Table 3: Runtime of PySat

Input	Grid size	Runtime (seconds)
1	5×5	0.002015
2	5×5	0.000835
3	5×5	0.000967
4	5×5	0.000763
5	7×7	0.001444
6	9×9	0.002692
7	11×11	0.003437
8	13×13	0.004388
9	17×17	0.005211
10	20×20	0.007632

Table 4: Runtime of A*

Input	Grid size	Runtime (seconds)
1	5×5	0.254938
2	5×5	0.052099
3	5×5	0.459319
4	5×5	0.231828
5	7×7	881.163072

Table 5: Runtime of brute-force

Input	Grid size	Runtime (seconds)
1	5×5	53.396381
2	5×5	0.262703
3	5×5	60.120098
4	5×5	78.364327

Table 6: Runtime of Backtracking

Input	Grid size	Runtime (seconds)
1	5×5	0.028819
2	5×5	0.003294
3	5×5	0.015089
4	5×5	0.086659
5	7×7	2.903167

Note: For input with grid size larger than 7×7, algorithms such as backtracking, brute-force and A* have large execution time (larger than 1800 seconds).

Table 7: Memory Complexity

Algorithms	Memory Complexity
Brute-force	$O(n)$
Backtracking	$O(n)$
A*	$O(b^d)$

where

- n is the number of literals.
- b is the branching factor of A*.
- d is the depth of A*.

6.11.1 Evaluation

- Backtracking has best runtime compared to A* and brute-force.
- Theoretically A* should has best runtime but A* performance depends on heuristics functions, poor heuristics may led to poor runtime.
- Brute-force has much larger runtime than A* and backtracking. For example, for input 1, brute-force runtime is 2500 times that of backtracking and 250 times that of A*.
- For input with grid size larger than 7×7, runtime of algorithms like A*, backtracking and brute-force is very large. (larger than 1800 seconds)

6.12 Algorithm Comparisons

Among 3 algorithms:

- Backtracking is the best choice for this problem since it always produces good results in short time.
- A-Star is acceptable because time taken is not so large, despite its big memory requirement.
- Brute-force is the worst one, because it doesn't have any optimization, that's why it consumes huge amount of time by iterating all possible combinations.

7 Demonstration Video

Link to video: https://drive.google.com/file/d/1eZ6wVR_Vcj3PoFwhdMB9LIKuB9HB9x80/view?fbclid=IwZXh0bgNhZW0CMTAAYnJpZBExczFOU2VEcDREQlVISnI0NgEeG-edljuA2Jx4RY8eMviGaldlaem_1ejOiUFpvh6YpKs0qzHsAQ