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.
  - **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:

```
    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* 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.

- **Flexibility:** The heuristic can be adapted to different puzzle constraints, allowing for more customization based on specific requirements of the puzzle or variations of Hashiwokakero.

Limitations

- **Complex Heuristic Design:** Designing an effective heuristic that accurately reflects the puzzle's complexity and constraints can be challenging. A poorly designed heuristic may lead to inefficient searches or suboptimal solutions.
- **Memory Usage:** A* requires storing a large number of states in memory as it explores the search space. This can lead to high memory consumption, especially for larger puzzles with many possible configurations.
- **Computational Cost:** Despite its more efficient search compared to brute-force, A* can still be computationally expensive, particularly for large puzzles. The evaluation of multiple potential configurations and heuristic calculations can lead to high computational overhead.
- **Dependence on Heuristic Quality:** The effectiveness of A* is directly tied to the quality of the heuristic function. If the heuristic is poorly chosen, the algorithm may perform poorly or even fail to find an optimal solution within a reasonable time frame.

Conclusion

The A* algorithm, when applied to solving the Hashiwokakero puzzle using a well-designed heuristic, provides a significantly more efficient and scalable approach compared to brute-force methods. By utilizing heuristics such as remaining bridges and constraint satisfaction, A* focuses the search on promising configurations, reducing unnecessary exploration of infeasible states. The algorithm guarantees optimality if an admissible heuristic is used and effectively handles the constraints of the puzzle. However, its success depends heavily on the quality of the heuristic and the complexity of the puzzle. While it offers a powerful solution for medium-sized puzzles, larger instances may still pose challenges due to high memory and computation costs.

6.3 Backtracking with DPLL-based SAT Solver

The backtracking algorithm used for solving Hashiwokakero puzzles is based on a simplified DPLL (Davis–Putnam–Logemann–Loveland) SAT solver. The puzzle is first encoded into CNF form, and then a recursive backtracking search is employed to find a satisfying assignment of Boolean variables. This approach systematically explores the space of assignments and applies inference rules like unit propagation and pure literal elimination to prune inconsistent paths early, improving efficiency over naïve brute force methods.

Implementation Details

- **unit_propagate():**

  This function iteratively applies unit clause inference.

When a clause has only one unassigned literal, it forces the corresponding variable to satisfy the clause, reducing the search space.

- **dpll():**

  The core recursive backtracking function. It applies unit propagation and pure literal elimination before branching.
  For each unassigned variable, it tries both True and False values recursively until it finds a satisfying assignment or concludes unsatisfiability.

- **solve_with_backtracking():**

  Encodes the puzzle into CNF, calls the DPLL solver, and if a model is found, it validates and extracts the solution using utility functions like *validate_solution* and *generate_output*.

Inference Techniques

The algorithm integrates classical SAT-solving inference techniques to enhance performance:

- **Unit Propagation:** Forces assignments based on clauses with only one remaining unassigned literal.
- **Pure Literal Elimination:** Assigns values to literals that always appear with the same polarity (positive or negative) across all clauses.

Time and Space Complexity

Time Complexity: In the worst case, the algorithm explores all possible assignments, resulting in $O(2^n)$ time complexity, where n is the number of variables. However, unit propagation and pure literal elimination help prune large portions of the search tree.

Space Complexity: $O(n+m)$, where n is the number of variables and m is the number of clauses. Additional space is used to store the recursive call stack and intermediate assignments during backtracking.

Strengths

- **High Performance with DPLL:** The backtracking solver, now integrated with the DPLL-based approach, can solve puzzles with impressive speed and efficiency, even for large instances, significantly improving performance over the base version.
- **Scalability in Size:** The DPLL-enhanced solver can handle large puzzle sizes effectively, maintaining high performance across a wide range of problem dimensions.
- **Simplicity of the Base Algorithm:** The core backtracking logic is simple to implement, providing an easy starting point for logic puzzle solving, though the performance of this base approach is not optimal.

Limitations

- **Implementation Complexity with DPLL:** The DPLL-based approach, while offering superior performance, is more advanced to implement and debug. It requires careful handling of SAT-solving techniques like unit propagation and pure literal elimination, which can be error-prone.
- **Base Algorithm Performance:** While the base backtracking algorithm is easy to implement, its performance is poor compared to the DPLL-enhanced version, making it unsuitable for large or complex puzzles.
- **Debugging Difficulty in DPLL:** Debugging issues in the DPLL-based version is more challenging due to its abstract nature, making it harder to trace failures compared to more traditional, step-by-step solvers.

Conclusion

The backtracking solver, when enhanced with the DPLL algorithm, represents a significant improvement in terms of performance and scalability for solving logic puzzles. The DPLL-based approach enables the solver to handle larger problem instances efficiently, achieving impressive speed compared to the base backtracking method. While the base algorithm remains easy to implement and serves as a simple foundation for puzzle solving, its performance is insufficient for more complex problems. On the other hand, the DPLL version, though more advanced and challenging to implement, offers a much more powerful solution.

6.4 Brute-Force

Brute-force search is a straightforward but computationally intensive method for solving CNF problems. In this approach, the algorithm systematically explores all possible truth assignments for the CNF variables, checking each complete assignment to determine whether it satisfies all clauses. While this guarantees that a solution will be found if one exists, it becomes impractical as the number of variables increases, due to the exponential growth of the search space. Brute-force does not leverage any heuristics or early pruning, making it inefficient for large or complex instances. However, for small puzzles with limited variables, brute-force can still serve as a simple and reliable baseline for correctness verification.

Pseudocode

Algorithm 2 Brute-Force Search for Hashiwokakero (*grid*)

Input: *grid* (Hashiwokakero puzzle grid)
Output: Valid solution or failure
$(cnf, edge\_vars, islands, variables) \leftarrow encode\_hashi(grid)$
if no islands or *cnf.clauses* is empty **then**
    return generate_output(grid, {}, {})
end if
Build mapping *var_to_index* from *variables*
for all possible boolean assignments of *variables* **do**
    Evaluate all clauses in *cnf.clauses*
    if every clause is satisfied **then**
        model ← build model from the assignment
        if validate_solution(islands, edge_vars, model) **and** check_hashi(islands, model) **then**
            return generate_output(grid, islands, model)
        end if
    end if
end for
return failure (no valid assignment found)

Implementation

- **encode_hashi:** Transforms the Hashiwokakero grid into a CNF formula, along with a list of islands, variables, and edge constraints. This encoding allows the puzzle to be interpreted as a SAT problem.
- **solve_with_bruteforce:** Iterates over all possible truth assignments to the CNF variables. For each assignment, it checks whether all clauses are satisfied and whether the resulting solution forms a valid Hashi configuration.
- **validate_solution:** Ensures that a satisfying assignment from the SAT check also satisfies puzzle-specific constraints such as island degree and bridge rules.
- **extract_solution:** Translates a valid variable assignment into a bridge configuration representing connections between islands.
- **check_hashi:** Final validation step that confirms the connectivity and correctness of the bridge layout based on the Hashiwokakero rules.
- **generate_output:** Converts the verified bridge solution into the desired output format (e.g., a solved puzzle grid).

Time and Space Complexity

Time Complexity: In the worst case, the brute-force search examines all possible assignments, leading to a time complexity of $O(2^{|variables|})$, where $|variables|$ is the number of CNF variables. This exponential growth makes brute-force impractical for large instances.

Space Complexity: The space complexity is $O(|variables|)$, since at any given time the algorithm only needs to store the current assignment (a tuple of booleans) and minimal additional bookkeeping. This ensures that while time requirements may be prohibitive, the memory footprint remains relatively small.

Strengths

- **Conceptual Simplicity:** The brute-force method is straightforward and easy to implement, requiring minimal algorithmic complexity or heuristics.
- **Guaranteed Completeness:** It explores all possible assignments, ensuring that if a valid solution exists, it will eventually be found.
- **Useful for Small Instances:** For small puzzles with limited variables, brute-force performs adequately and can serve as a correctness benchmark for more advanced solvers.

Limitations

- **Exponential Time Growth:** The algorithm suffers from poor scalability due to its $O(2^n)$ time complexity, making it infeasible for puzzles with more than a few dozen variables.
- **No Pruning or Heuristics:** Unlike more sophisticated methods, brute-force lacks any optimization strategy to reduce the search space or prioritize promising paths.
- **Inefficient for Real-Time Use:** Due to its exhaustive nature, the algorithm is too slow for practical use in interactive or large-scale puzzle solving.

Conclusion

The brute-force solver for Hashiwokakero, while theoretically complete, is limited in practical usability due to exponential runtime. It is best suited for validating small puzzle instances or serving as a baseline for correctness. Its simplicity makes it easy to understand and implement, but the lack of efficiency and scalability necessitates the adoption of smarter strategies like heuristic search or SAT-based optimizations for larger or more complex puzzles.

7 Algorithms Benchmark

7.1 About Test Cases

- **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. (There are also test cases for (3 × 3) and (5 × 5) for brute force algorithm.)
  - The test cases are generated randomly and the difficulty levels are classified as easy, medium, hard, and very hard with the increasing of crossing edges, the number of islands with high degrees, etc. The results of the tests are shown in the table and figures below. The time is measured in milliseconds (ms) and the peak memory is measured in kilobytes (KB).
- **Challenges**
  - **Generating CNF**
    - Generating CNF clauses for constraints like 'degree constraints' is quite complex and time-consuming. The algorithm needs to consider all possible combinations of edges and nodes to generate the CNF clauses.
  - **Algorithm implementations**
    - 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.

7.2 Experiment results

- **Brute Force**
  - The brute force algorithm is able to solve all test cases with a map size of (3 × 3) and (5 × 5) slowly. It is not able to solve larger maps due to the exponential growth of the search space.

Map Size	Time (s)	Memory
3 × 3	0.02	11704 B (≈ 11.38 KB)
5 × 5	413.68	75176 B (≈ 73.41 KB)
7 × 7	879.78	125816 B (≈ 122.86 KB)

Table 1: Comparison of Time and Memory for Different Map Sizes

- **Backtracking**
  - The backtracking algorithm is able to solve almost test cases with all map size in a reasonable time.
- **A***

  - The A* algorithm is able to solve all test cases with almost map size in a reasonable time. However, it is not able to solve some test cases with a very high difficulty level due to the large search space.

- **PySAT**

  - The PySAT algorithm is able to solve all test cases with all map size in nearly instant time. It is the fastest algorithm among all algorithms.

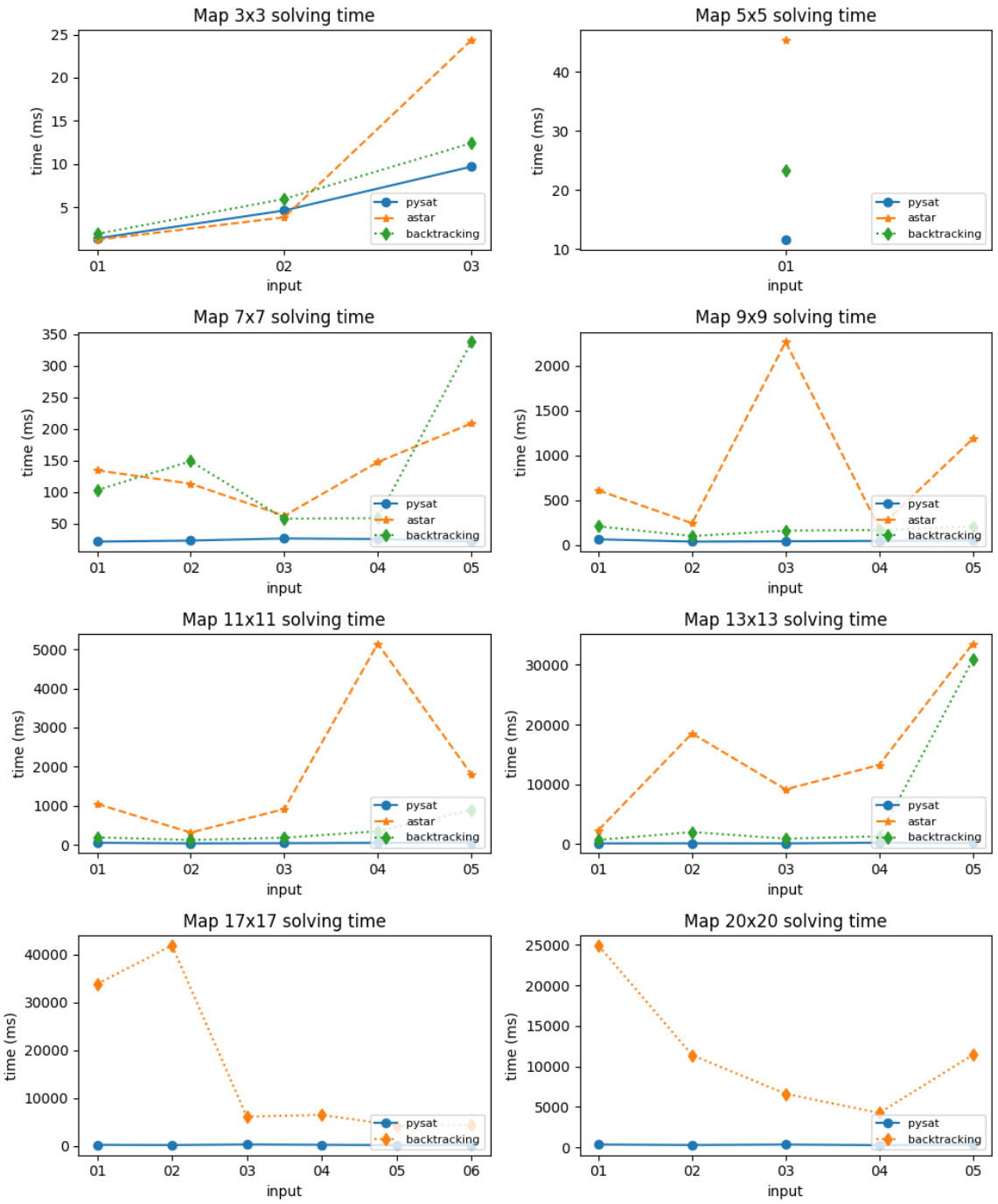

Figure 1: Solving time Benchmark

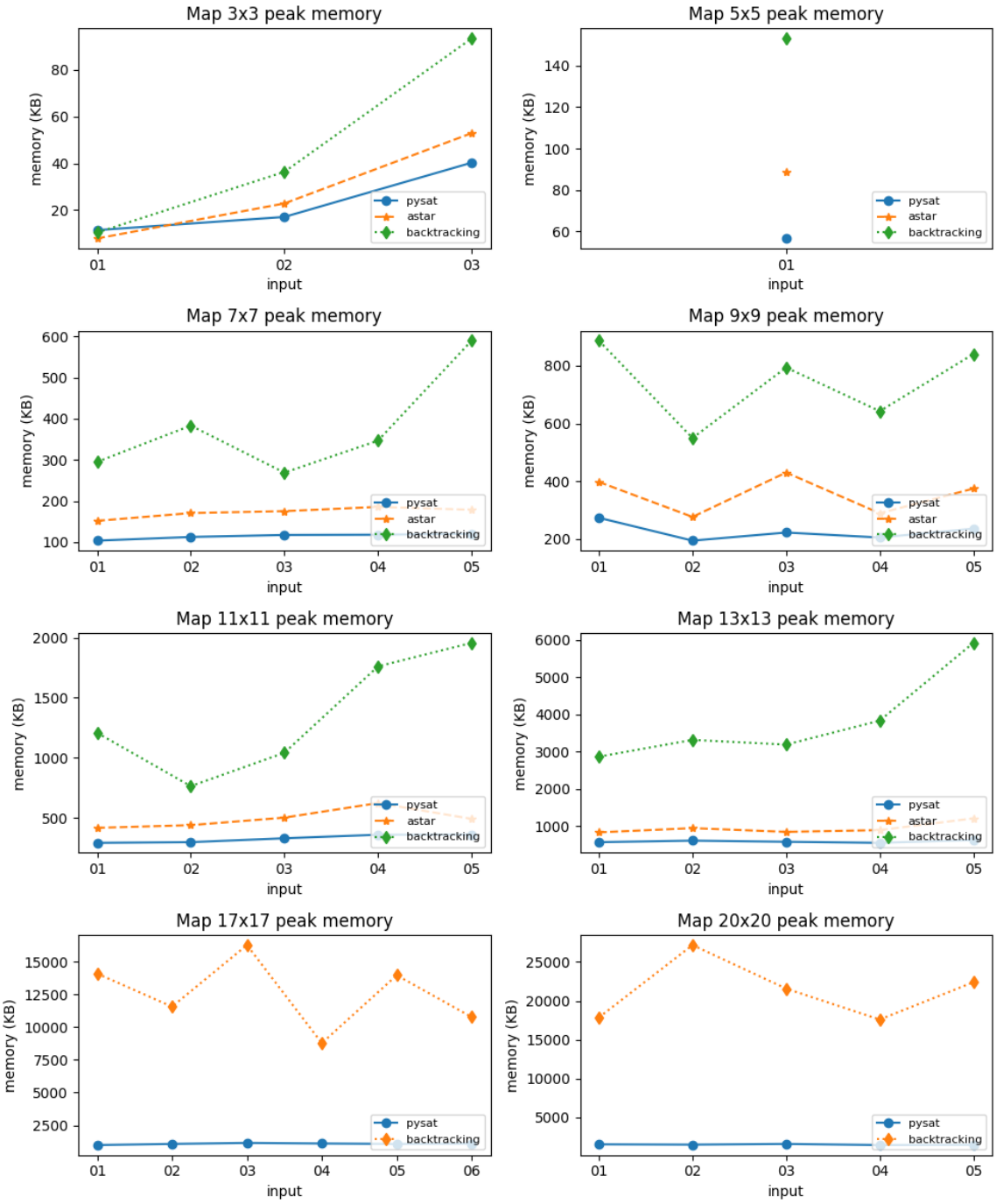

Figure 2: Solving time Benchmark

8 References

1. Hashi Info: How to solve Hashiwokakero puzzles successfully (for human players)
2. Tseytin Transformation: About generating CNF clauses from DNF