Theory of Computation

Roberto Palmieri, Alan Copa, Diell Kryeziu, Daniel S. L. Costa

Homework 5

SAT encoding (100 points)

Due Tuesday, 30 May 2023, 11:55 PM

1 Introduction

The Hamiltonian cycle problem is a well-known computational problem in graph theory. Given an undirected graph, the objective is to determine whether a path exists that traverses all the nodes of the graph exactly once and returns to the starting node.

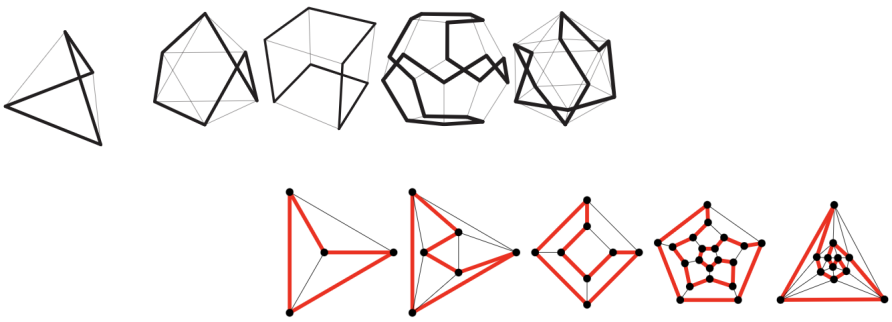

Figure 1: Examples of Hamiltonian cycles

This problem finds applications in various domains, including computer graphics, electronic circuit design, genome mapping, and operations research. In computer graphics, for instance, finding a Hamiltonian cycle can help optimize the rendering of complex scenes by efficiently determining the order in which objects should be processed. In electronic circuit design, the problem can aid in identifying the most efficient paths for electrical signals to traverse. Furthermore, in genome mapping, the Hamiltonian cycle problem assists in analyzing and sequencing genetic information accurately. Lastly, in operations research, it helps optimize logistical planning and transportation routes.

The Hamiltonian cycle problem is classified as an NP-Problem, indicating that there is no known polynomial-time algorithm that can solve it for large graphs. The brute-force approach involves testing all possible paths, which can be computationally expensive. One common method is to employ a recursive algorithm that systematically explores different paths until a Hamiltonian cycle is found. Another approach is to encode the problem as a boolean problem and utilize a SAT solver, which is a specialized tool for solving boolean satisfiability problems.
In this report, we will present a solution employing the latter strategy. By formulating the problem as a boolean instance and leveraging the capabilities of a SAT solver, we aim to efficiently find Hamiltonian cycles in graphs, thereby addressing the computational challenges associated with the problem.

2 Method

2.1 Logic Encode and Decode

To process the graph file, encode the logical sentences, and save the results, Python was utilized. The rules were then passed to Z3, a cutting-edge theorem prover developed by Microsoft Research. Z3 is renowned for its capability to verify the satisfiability of logical formulas across one or multiple theories.
The program takes as input a path to a TXT file that defines the graph. The first line of the file specifies the number of nodes and edges, followed by the edges defined by pairs of nodes. The graph file format is:

```
N E
n1 n2
n2 n1
n2 n3
n3 n4
n4 n5
...
```

A matrix of size NxN is generated to map all the edges. If there exists an edge between node i and node j, the corresponding positions (i, j) and (j, i) in the matrix are marked as true. Conversely, all the relations without an edge are marked as false.
To encode the problem, we formulate variables to represent the position in the path and the node itself, aiming to find a sequence of nodes that form a Hamiltonian cycle. Therefore, we create a variable for each node in each position, denoted as **inPosition_i_j**, where i represents the position in the path and j denotes the node number. Ultimately, we seek to identify the set of variables that are true and satisfy the problem's constraints.

One of the constraints stipulates that in each position, only one node from the graph is permissible. This constraint can be expressed as follows:

$$(\neg inPosition\_i\_j \vee \neg inPosition\_i\_k)$$

Here, the clause represents the conjunction of every pair of nodes, and this clause is repeated for each position.
Similarly, we need to ensure that each node occupies only one position. This requirement leads to the following rule:

$$(\neg inPosition\_i\_j \vee \neg inPosition\_m\_j)$$

We also need to guarantee that every position is occupied. To achieve this, a logical disjunction of all the variables in that position is used to ensure that at least one of them is true:

$$(inPosition\_i\_0 \vee inPosition\_i\_1 \vee ... \vee inPosition\_i\_n)$$

This rule is repeated for each position. Likewise, we must ensure that every node occupies at least one position, since a Hamiltonian cycle must traverse all the nodes. To accomplish this, we employ the following rule:

$$(inPosition\_0\_j \vee inPosition\_1\_j \vee ... \vee inPosition\_n\_j)$$

Finally, the last restriction we need to satisfy is that two nodes in consecutive positions must have an edge between them to form a cycle. Thus, if the matrix mapping the edges indicates that node i does not have an edge connecting it to node j, j can neither precede nor follow i. To enforce this, we utilize the following rule:

$$(\neg inPosition\_m\_i \vee \neg inPosition\_m + 1\_j)$$

Consequently, j cannot occupy the position immediately following node i because they lack an edge connection.
With these restrictions structured according to Conjunctive Normal Form (CNF), we pass the satisfiability problem to Z3. If a solution exists, Z3 will print the set of clauses that are true, and the resulting path will be saved in a file.
In the documentation we have a toy example what can be helpful to understand the logical encoding for this problem.

2.2 Interface

We have successfully implemented a Graphical User Interface (GUI) utilizing VUE, a powerful JavaScript framework. This GUI provides users with the ability to input

a text file containing the graph information and receive a visual representation of the corresponding answer. Alternatively, users can choose to utilize a graph generation script. In this case, they can specify the desired number of nodes and edges, as well as indicate whether they prefer a random graph or a graph with a pre-defined Hamiltonian Cycle.
To facilitate the communication between the frontend and the backend, we employed Flask, a Python-based web framework. This technology allowed us to seamlessly handle incoming user requests and efficiently deliver the appropriate responses. For detailed information on the setup and execution of the system, please refer to the comprehensive documentation available along with the code.

3 Results

To validate the effectiveness of our method, we conducted a comprehensive comparative analysis, pitting our approach against a recursive method for solving the problem. Our experiments involved executing the functions multiple times across various graph sizes, and the summarized results can be found in the table below:

Type	Nodes	Edges	Graph	Time(s)	STD
Logic	5	5	Satisfiable	0.01471577	0.00349578
Recursive	5	5	Satisfiable	0.00001478	0.00001917
Logic	5	5	Unsatisfiable	0.01394639	0.00023176
Recursive	5	5	Unsatisfiable	0.00001287	0.00000916
Logic	10	30	Satisfiable	0.01879218	0.00061698
Recursive	10	30	Satisfiable	0.00005007	0.00001099
Logic	10	9	Unsatisfiable	0.02003145	0.00189383
Recursive	10	9	Unsatisfiable	0.00004971	0.00001881
Logic	20	60	Satisfiable	0.06106699	0.01288237
Recursive	20	60	Satisfiable	0.03188779	0.00258284
Logic	20	19	Unsatisfiable	0.07875459	0.01472240
Recursive	20	19	Unsatisfiable	0.00077970	0.00018679
Logic	50	500	Satisfiable	0.74559853	0.16780346
Recursive	50	500	Satisfiable	0.08432350	0.00219526
Logic	50	49	Unsatisfiable	0.87202044	0.03156883
Recursive	50	49	Unsatisfiable	0.00007253	0.00003321

For each graph, we performed the operation ten times, calculating both the mean time and standard deviation of these executions. Satisfiable graphs were defined as those containing a Hamiltonian Cycle, while unsatisfiable graphs lacked this specific path.

The comparative analysis revealed that the recursive approach exhibited superior performance. This is expected, as the worst-case complexity of the recursive method is O(N!), where N represents the number of nodes. This approach exhaustively explores all possible paths. On the other hand, the logical approach utilizing Z3 involves N^2 variables, corresponding to N positions and N nodes. Considering each variable can have true or false values, the theoretical complexity becomes $O(2^(N^2))$, significantly worse than $O(N!)$.
An interesting observation is that, despite the theoretical complexity being $O(2^{N^2})$, the actual execution time does not strictly adhere to this theoretical rule when increasing the number of nodes. For instance, let's consider the graphs with 5 and 50 nodes. According to theoretical calculations, a graph with ten times more nodes should have a complexity of $t_{10n} = 2^{(10n)^2}$.

$$t_{10n} = 2^{(10n)^2} = 2^{100n^2} = 2^{99n^2+n^2} = 2^{99n^2} \cdot 2^{n^2} = 2^{99n^2} t_n$$

Consequently, the graph with 50 nodes should have a time of $2^(99(5)^2) = 2^2475$ times more than the graph with 5 nodes, suggesting a computation time exceeding 1000 years. However, in practice, the algorithm only takes approximately 80 times longer. This showcases the remarkable optimization capabilities of Z3, which effectively reduces the complexity of the problem to something very close to a polynomial algorithm.
It is essential to note that this analysis provides a simplified overview, and other factors, such as the number of edges and their distribution throughout the graph, can directly impact the algorithm's complexity. Nevertheless, it offers valuable empirical insights into the behavior and performance of our solution.

4 Conclusion

It can be concluded that our experiment has yielded successful results. We have successfully developed an encoder capable of receiving a graph file as input and determining the existence of a Hamiltonian Cycle. In the case of a positive outcome, the encoder also generates the sequence of the cycle. Additionally, an intuitive interface has been created to allow users to easily utilize the encoder, obtain the graph with the identified path, and explore this intriguing problem across various scenarios.
As discussed in our previous class, while satisfiability remains an exponential problem in general, there are specific scenarios where we observe behavior closely resembling a polynomial algorithm. This observation has been validated through thorough data analysis, providing empirical evidence that suggests this problem belongs to the NP class. What is aligned with the theoretical proof of complexity for CNF discussed in the lesson.

As next steps, it is imperative to conduct an in-depth analysis in order to optimize the logical algorithm and achieve a performance level comparable to that of the recursive implementation. Exploring alternative encoding strategies that involve a reduced number of variables may prove to be a fruitful avenue for optimization. Additionally, enhancing the versatility of our system by enabling users to upload graphs in various formats and representations should be a priority. Currently, node representation is limited to a sequence of positive integers, but expanding this capability will enhance the usability and flexibility of the application. These steps will contribute to the overall improvement of our solution and ensure its effectiveness in a broader range of scenarios.