INF 421 PROGRAMMING PROJECT HAMILTONIAN PATHS AND RIKUDO SOLVER

FEDRIGO BARCIK Gabriel
OLIVEIRA FRANCA Matheus

Objective:

The objective of the following paper is to describe and to solve the tasks proposed by professor Vincent Pilaud in the project Hamiltonian Paths and Rikudo Solver.

Introduction:

We are supposed to create algorithms and build an architecture that will solve a game named Rikudo and also create new instances of this game. First, we will describe the game, then we will verify that solving one instance of this game is equivalent to finding a Hamiltonian path in a directed graph. As indicated on the paper, two solutions of finding a Hamiltonian path were implemented, using a SAT Solver and using a backtracking algorithm. After comparing these algorithms and adapting them to solve the Rikudo game we describe our architecture that creates new instances of the game from binary images.

Rikudo Game:

The game was invented in 2015 in the physics department of École Polytechnique. Giving a tiling on a plane with n hexagons tiles find a path from the vertex marked as 1 to the vertex marked as n, by visiting a vertex I at the step I, if this vertex was marked on the tiling before the games has started, and by constructing the path only by passing through adjacent vertices and not repeating any vertex on the path. There is also another constrain different from the one that at the beginning of the game some vertices would be already labeled by a number and that you should pass by them at the step corresponding to its label. The other constrain is the presence of edges called diamond edges. Such edge implies that you must at some moment use this edge, but it does not matter the direction that you do it, as the graph is directed, it suffices to contain this edge in only one direction.

Hamiltonian Path

If we consider a directed graph G = (V, A) with vertex set V and oriented set A, where each vertex in V represents a tile in the tiling plane and the edges set contains only the possible connection between two tiles. Indeed, the problem of finding a solution of the Rikudo game is equivalent to find a Hamiltonian path on the graph G. In the next section we'll describe the first algorithm to find a Hamiltonian path, the SAT Solver.

SAT Solver

Task 1:

We have created a method that translates a Hamiltonian path problem to a SAT problem and that uses the library SAT4j to solve it. For this, we have created 4 functions that implement the following ideas:

1. Each node j must appear in the path:
- x_{ij} v x_{2j} v … v x_{nj} for each j
2. No node j appears twice in the path:
- -x_{ij} v -x_{kj} for all I, j, k with i ≠ j

3. Every position i on the path must be occupied:
- x_{i1} v x_{i2} v … v $x_{i,n}$ for each i
4. No two nodes j and k occupy the same position in the path:
- $-x_{ij}$ v $-x_{ik}$ for all I, j, k with j ≠ k
5. Nonadjacent nodes I and j cannot be adjacent in the path:
- $-x_{ki}$ v $x_{k+1,j}$ for all (i, j) not in G and k=1,2… , n-1

We have implemented these ideas by constructing a matrix X of integers. Each row represents the possible positions of one vertex, and each column represents all the vertices that could be in the possition associated with this column. The computation runtime for the SAT Solver algorithm can be find in table 1.

TABLE 1:

Type of the graph	Number of vertex	Computation Time (ms)
Complete graph	80	84
Complete graph	160	1060
Cycle graph	1000	5264
Cycle graph	2000	29657
Grid graph	25 (9x9)	1060
Grid graph	121 (11x11)	43153

Remark 1 (ii) To adapt the SAT problem to find a Hamiltonian cycle, it suffices to try to find a a Hamiltonian Path from a source to a neighbor.

Backtracking Algorithm

Task 2:
We did in total five backtracking algorithms, all of them implemented in the class Backtracking. The first algorithm is the backtracking function which solves the Task 2. The main idea is to start from the source vertex, parameter source, and add a new vertex to the path (vertex v) if doing such we can find a Hamiltonian path starting from vertex v and of course not using the vertices already visited, that means the vertices there are already in the vector path. The computation runtime for the backtracking algorithm can be find in table 2.

TABLE 2:

Type of the graph	Number of vertices	Computation Time (ms)
Complete graph	10	157
Complete graph	12	8851
Cycle graph	1000	74
Cycle graph	2000	240
Grid graph	25 (5x5)	88
Grid graph	36 (6x6)	1845

Remark 2:
(I) Using algorithm backtracking2 we can count the number of paths, for that we used a static variable number_of_paths. There is no path in a grid NxN if N is even, otherwise the number of paths for N=3 is 2, for N=5 is 104, for N=7 is 111712. Actually, we have found in the book by

Karen L. Collins and Lucia B. Krompart [1], that exist formulas for a grid with m vertices in each column and n vertices in each row. For m=1, the answer is 1 for every value of n. For m=2, the answer is 0 if n is even and 1 if n is odd. For m=3, the answer is 2^{n-2}. Finally, for m=4 and m=5, they provide a generating function which is the quotient of two polynomials.

Now we explain the phenomenon that when N is even there isn't a Hamiltonian path between the two opposite vertices, lying in the diagonal of the grid. Imagine the grid colored like a chess board. Say that the upper left vertex is white, then the right lower vertex has the same color, white. Suppose by contradiction that exists such a path, then the Hamiltonian path takes N^2-1 transitions to reach the target vertex from the source vertex. As N^2-1 is always odd when N is even, in order to reach the target vertex we need to make an odd number of transitions, so we will arrive in a black cell, and not in a white one, as supposed in the beginning. By contradiction, there is not a Hamiltonian path for N even.

(II) We simply create a function backtracking_k that calls the function backtracking3, one function similar to backtracking2, but that saves the paths in a linked list. And print the expected result, depending on the value of k.

Task 3:
We have adapted the SAT solver algorithm by adding new constrains, the ones related to the lambda function are clauses that must be true and the ones related to the collection of diamond edges we have added clauses that ensures that at least in one position the vertices on the edge must be visited adjacently.
In order to solve this task we have used the algorithm backtracking2 as base code. The constrains were added just by rejecting undesired vertices during the construction of a path. More precisely, we reject a vertex if in the position that we will add it has a different label, in other words, the vertex to be added should match the lambda map vertex. With respect to the diamond edges constrains, we reject a vertex if the last vertex added needs to connect with another vertex by the diamond edge constrain. So, we check for every neighbor of the last added vertex if there is a diamond edge between then, we check if the neighbor is different than the current vertex to be added and if it is not in the path, these conditions implies that neighbor necessarily will follow the last added vertex and we need to refuse the vertex v.

Remark 3:

The additional constrain that we have created is the Star Edge. When there is a start edge between two vertices we can pass from one to the other even if they are not adjacent in the graph. However, by construction, we will only create a star edge between vertices in the boarder of the graph, as there was a connection between them below the board of the game. For this new version of the game it is not necessary to change the backtracking nor is to change the sat solver algorithm. We compare the two algorithms in table 3.

TABLE 3:

Type of the graph	Number of vertices	Algorithm	Computation Time (ms)
Complete graph	12	Backtracking	8851
Complete graph	12	SAT Solver	3
Complete graph	80	Backtracking	infinity
Complete graph	80	SAT Solver	58
Cycle graph	1000	Backtracking	74
Cycle graph	1000	SAT Solver	5264
Cycle graph	2000	Backtracking	240
Cycle graph	2000	SAT Solver	29657
Grid graph	25 (5x5)	Backtracking	496
Grid graph	25 (5x5)	SAT Solver	134
Grid graph	81 (9x9)	Backtracking	infinity
Grid graph	81 (9x9)	SAT Solver	1060

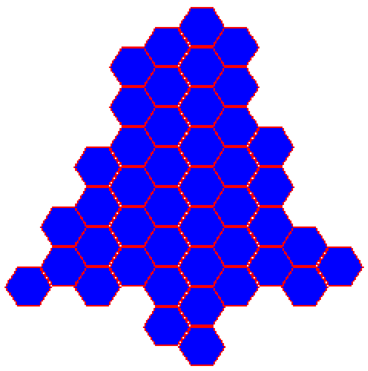
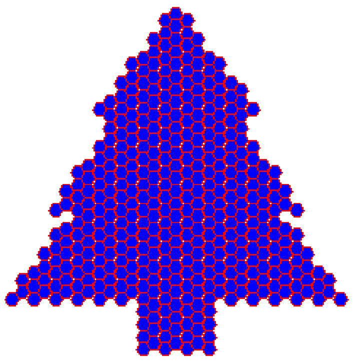
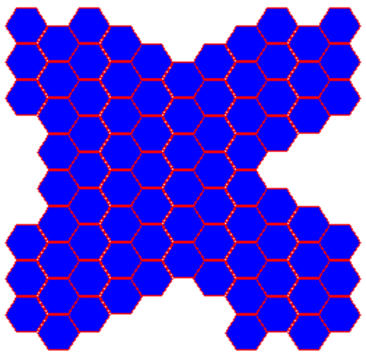
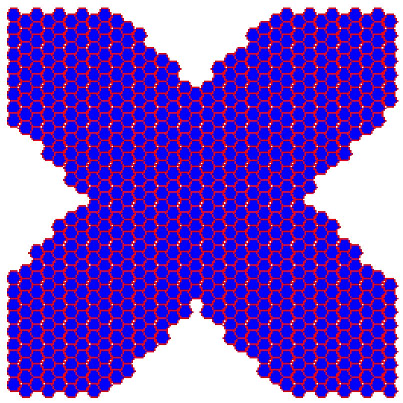
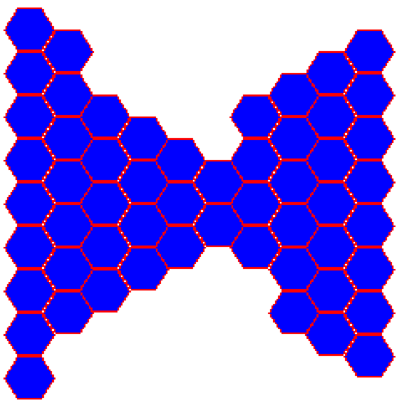
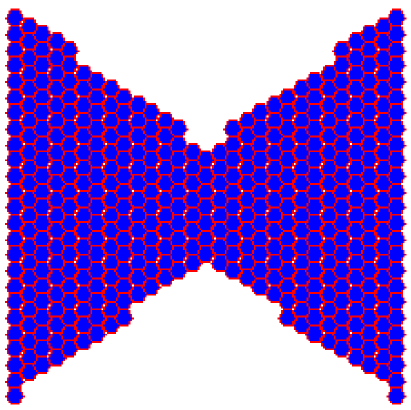
Comparing the performance of the algorithms we can conclude that the SAT Solver is better than the backtracking algorithm for either the graphs grid and the complete one. In the case where the graph is a cycle, the backtracking algorithm outperformed the SAT Solver algorithm. This happens because the backtracking algorithm will only try the obvious solution whereas the SAT Solver solution will create many unnecessary clauses, as it did not exploit the structure of the graph.
Considering that the game will not have a similar shape of a cycle graph, which is a reasonable assumption, we are going to use the SAT Solver to implement the others tasks of the project.

Task 4:
In order to decide if a given partial labeling and a given collection of diamond edges determines an unique solution, we used the function hasUnicSolution and used the class SingleSolutionDetector from SAT4j.

Now, to decide if this information is minimal for this property, we adapted two functions of the SAT problem of questions 3: defineVerticesInTheBeginning and addDiamondEdges.
For each vertex or diamond added, we calculate the number of solutions using the class SolutionCounter from SAT4j. If the number of solutions does not decrease with one more restriction, then it is not minimal.

Task 5:
For the task: Given a graph G with a source vertex s and a target vertex t, design one or more strategy to create rikudo instances on G with precisely the minimal information to admit a unique solution, we had two ideias: first we could just use Isolver.model from SAT4j, that will return one of many solutions. In the following, we start to add vertices and edges using minimal and single solutions provided by question 4 until it is the only valid one.
Another idea is to use a random variable from 0 to 1. If it is less than 0.5, we draw a vertex until it is possible to decrease the number of solution and there is still at least one solution. On the other hand, if it is more then 0.5, we do the same with an edge.
In our program, we tried to implement the second kind of solution. The addition of an aleatory edge is well implemented, but there is still an unsolved bug while trying to trying to find an aleatory vertex to add to Lambda.

Task 6:
We have created a class BinarytoGraph to solve this task. Given the filename of a binary image and the size of the hexagon, which represents the resolution, we can construct a grid of hexagons that it's similar to the image and its associated graph. The discretization function gives the relation between the label of the hexagons, I and J, and its center is x and y coordinates. Finally, we loop through the binary image and we create a hexagon if its center belongs to the interior of the image. At the same time, we are able to construct the graph by adding a new vertex every time we find a compatible center and for the construction of the edges, we store for each hexagon label the name of the vertex and after, for each vertex, we try to add all its six possible neighbors, verifying this possibility using the map map_coordinate_vertices. The images created based on the binary images provided can be find below, using the size of the hexagon as 15 on the left and as 5 on the right.

Certainly, as we are dealing with a game the number of vertices will not be so high, and the figures of the left would be selected to be a part of a possible website containing the Rikudo game.

Having the graph it is easy to create a puzzle with a unique solution, because it suffices to pass the graph as a parameter to the function created in task 5.

Bibliography

[1] “The number of Hamiltonian paths in a rectangular grid”; Karen L. Collins, Lucia B. Krompart; Elsevier.