

# Graph-Based Analysis of Solution Spaces in the Game *Diplomatico*

Filippo Garagnani  
Dipartimento di Ingegneria "Enzo Ferrari"  
Università degli Studi di Modena e Reggio Emilia  
298707@studenti.unimore.it

**Abstract**—The game of *Diplomatico* (also known as 'The 100 Squares Puzzle' or 'Hopido' [1]) has been studied during the last decades. The few results found are limited to determining whether the game is solvable or not for certain given constraints. However, to the best of our knowledge, no one has ever tried to analyze different approaches to explore the solution space in its entirety. In this work, we model the game as a graph where nodes represent squares and edges represent valid moves between these cells. We apply various graph analytics techniques to uncover structural properties of the solution space, identify key states, and explore potential strategies for solving the game.

## I. INTRODUCTION

The game of *Diplomatico* is a combinatorial puzzle played on a rectangular grid. The objective is to fill the grid with consecutive numbers, subject to the following rules:

- 1) The number 1 may be placed on any available square.
- 2) For  $i > 1$ , the number  $i$  must be placed either at a distance of 3 horizontally or vertically, or at a distance of 2 diagonally from the square containing  $i - 1$ .
- 3) If no valid square is available for placing  $i$ , the game ends.

An example of valid moves is shown in Figure 1. Here, the number 2 is placed diagonally with respect to 1, while 3 follows by a horizontal move from 2.

3			2
	6		
	1		
4			5

**Fig. 1:** A  $4 \times 4$  grid following the rules of *Diplomatico*.

Although the game is often introduced on square grids, this restriction is not essential: *Diplomatico* can also be played on rectangular grids of arbitrary size, possibly with some cells blocked (i.e., unavailable for placement).

A few preliminary results are immediate. In particular, the game is unsolvable for square grids of size  $4 \times 4$  or smaller, except for the trivial case of a  $1 \times 1$  grid (see Figure 2).



**Fig. 2:** The trivial case of a  $1 \times 1$  grid.

The rules of the game admit a natural graph-theoretic formulation. Each cell of the grid is represented as a node, and a valid move (as defined above) corresponds to a directed edge between two nodes. Thus, constructing a valid play corresponds to finding a directed Hamiltonian path in this graph. Since determining whether a Hamiltonian path exists is an NP-complete problem [2], the complexity of solving *Diplomatico* grows rapidly with the grid size. This observation motivates the need for efficient heuristics and search strategies. Some classical approaches (such as Warnsdorff's Rule [3]) will be revisited in this context, along with new strategies inspired by graph-theoretic centrality measures. In particular, we will investigate the relationship between different centrality indices (betweenness, closeness, degree, eigenvector) and the number of Hamiltonian paths available on a given board configuration. This correlation analysis provides insight into the structural properties of the game graph and their potential role in guiding play.

## II. PRELIMINARIES

The following definitions are used throughout this work. The main case being considered is the one in which the game grid is rectangular and doesn't have any cell blocked.

### Definition II.1. Board Graph

Given a board of size  $n \times m$  (with  $n, m \in \mathbb{N}$ ), the **Board Graph** of size  $n \times m$  is an undirected graph  $\mathfrak{G}_{n \times m} = (V, E)$ , where:

$$V = \{v_{i,j}, i = 1, \dots, n \wedge j = 1, \dots, m\}$$

and where the following condition holds:

$$\begin{aligned} \forall e = \{v_{i,j}, v_{k,\ell}\} \in E, \quad & (|i - k| = 3 \wedge j = \ell) \\ & \vee (|j - \ell| = 3 \wedge i = k) \\ & \vee (|i - k| = |j - \ell| = 2) \end{aligned}$$

Of course, the node  $v_{i,j}$  will represent the square located at row  $i$  and column  $j$  of the original grid of the game. The

condition required for the edge set  $E$  ensures that only legal moves are present as undirected arcs in a Board Graph. The edges are undirected because, of course, the movement rules are perfectly symmetrical (i.e., being able to move from cell  $A$  to cell  $B$  means that it's also possible to move from cell  $B$  to cell  $A$ ).

**Definition II.2.** *Winning Path* Given a Board Graph  $\mathfrak{G}_{n \times m} = (V, E)$ , a **Winning Path** is an Hamiltonian Path in  $\mathfrak{G}_{n \times m}$ . Specifically, an Hamiltonian Path is a sequence of vertices  $\mathfrak{h} = (v^1, v^2, \dots, v^{n \times m})$  such that:

$$\forall \ell = 1, \dots, k-1, \{v^\ell, v^{\ell+1}\} \in E$$

and such that:

$$\forall v \in V, \exists! v^\ell \in \mathfrak{h} : v^\ell = v$$

An Hamiltonian Path is simply, as the Definition II.2 gives, a path in the graph that visits every node exactly once.

**Definition II.3.** *Solvability*

Given a Board Graph  $\mathfrak{G}_{n \times m}$ , it is said to be **solvable** if:

$$\exists \mathfrak{h} = (v^1, v^2, \dots, v^{n \times m}) :$$

$\mathfrak{h}$  is a Winning Path in  $\mathfrak{G}_{n \times m}$ .

Definition II.3 simply states that a Board Graph is solvable if it contains at least one Hamiltonian Path; that is, if there is a way to start from a node (i.e., a cell in the game of *Diplomatico*), and to be able to reach every other node - exactly once - by following the game rules.

Simply from the Definitions II.1-II.3, it is possible to prove the following Proposition.

$v_{1,1}$	$v_{1,2}$	$v_{1,3}$	$v_{1,4}$
$v_{2,1}$	$v_{2,2}$	$v_{2,3}$	$v_{2,4}$
$v_{3,1}$	$v_{3,2}$	$v_{3,3}$	$v_{3,4}$
$v_{4,1}$	$v_{4,2}$	$v_{4,3}$	$v_{4,4}$

**Fig. 3:** The board representing the graph  $\mathfrak{G}_{4 \times 4}$ . In corresponding colors, the only possible edges from each central node.

**Proposition II.1.** *Unsolvability of  $\mathfrak{G}_{4 \times 4}$*   
The Board Graph  $\mathfrak{G}_{4 \times 4}$  is unsolvable.

*Proof.* We can prove the unsolvability of  $\mathfrak{G}_{4 \times 4}$  by contradiction. Assume that there exists a Winning Path  $\mathfrak{h} = (v^1, v^2, \dots, v^{16})$  in  $\mathfrak{G}_{4 \times 4}$ .

Since  $\mathfrak{h}$  is a Hamiltonian Path, it must visit every node exactly once. Consider the nodes in the center of the board - specifically, vertices  $v_{2,2}, v_{2,3}, v_{3,2}, v_{3,3}$  (see Figure 3). Being  $\mathfrak{G}_{4 \times 4}$  a Board Graph, the conditions for edges linking those vertices can be explicitly determined (focusing, for the time being, on  $v_{2,2}$ ):

$$\forall e = \{v_{2,2}, v_{i,j}\} \in E,$$

$$(|2-i|=3 \wedge 2=j) \implies v_{i,j} \in \{v_{5,2}, v_{-1,2}\}$$

$$(|2-j|=3 \wedge 2=i) \implies v_{i,j} \in \{v_{2,5}, v_{2,-1}\}$$

$$(|2-i|=|2-j|=2) \implies v_{i,j} \in \{v_{0,0}, v_{0,4}, v_{4,0}, v_{4,4}\}$$

However, of the 'suggested' nodes above, only  $v_{4,4}$  is actually present in  $\mathfrak{G}_{4 \times 4}$  (the others are to consider 'out of bounds'). This means that the node  $v_{2,2}$  can only be connected to  $v_{4,4}$ , implying that it must be either the first or the last node in the Winning Path  $\mathfrak{h}$  (since it has degree 1). The same reasoning can be applied to the other three 'central' nodes; this means that four nodes must either be first or last in the path. This is not possible, hence the contradiction.  $\square$

The following Definition and Proposition will give a sense to the complexity of the game, and to the main focus of the work.

**Definition II.4.** *Numbers of Solutions*

Given a Board Graph  $\mathfrak{G}_{n \times m}$ , the **Number of Solutions**  $N(\mathfrak{G}_{n \times m})$  is defined as:

$$N(\mathfrak{G}_{n \times m}) = |\{\mathfrak{h} : \mathfrak{h} \text{ is a Winning Path in } \mathfrak{G}_{n \times m}\}|$$

**Proposition II.2.** *Bounds on  $N(\mathfrak{G}_{n \times m})$*

Given a Board Graph  $\mathfrak{G}_{n \times m}$ , with  $n, m \geq 5$ :

$$2^{(n-4)(m-4)/25} \leq N(\mathfrak{G}_{n \times m}) < 7^{nm-1}$$

The proof of Proposition II.2 is given in Appendix A.

The derived bounds demonstrate that the number of solutions increases exponentially with the board's size. This exponential growth renders exhaustive search methods infeasible for larger boards, emphasizing the necessity for more efficient algorithms and heuristics to navigate the solution space effectively. Furthermore, identifying even a single solution is a challenging task. The subsequent sections will delve into different search strategies and evaluate their performance.

### III. IMPLEMENTATION

1) *Finding solutions:* In order to efficiently analyze the solution space in the game of *Diplomatico*, a few implementations have been developed. Three of them were done in Neo4J [4], leveraging its graph database capabilities to model the Board Graphs and perform various queries. To make the

comparison valuable, another implementation in Python has been tested along the other. Before any query is run, the Board Graph is generated and stored in the database offered by Neo4J through a simple script that simply requires the number of rows and of columns of the board. Moreover, each query can be edited by command line arguments, by specifying - if needed - the start node of the path, the end node of the path and the total number of paths required.

The four main implementations of solution searching are briefly described below.

a) *The Raw Implementation:* The Raw implementation is a straightforward query in Neo4J, which tries to expand from a starting node to find one or more Winning Paths.

```
MATCH p = (start:Node)
  -[:MOVE*{parameters["pathLength"]}]->
    (end:Node)
WHERE ALL (
  n IN nodes(p) WHERE
  single(m IN nodes(p) WHERE m = n)
)
RETURN p
```

The query above simply matches a path  $p$  starting from any node labeled as Node, and tries to expand it by the number of edges specified in the parameter `pathLength` (which should be equal to  $n \times m - 1$  for a board of size  $n \times m$ ). The WHERE clause ensures that the path is Hamiltonian, by checking that every node in the path appears once and exactly once.

The starting node can be specified by preappending a MATCH (start:Node {row: r, col: c}) clause before the main MATCH clause, where  $r$  and  $c$  are the row and column of the desired starting node. The same applies to the end node. Moreover, the query could be modified to return only a few solutions (by adding a LIMIT clause at the end of the query). Thanks to the Python API of Neo4J, the element inside the parameters dictionary is modified at runtime.

b) *The Constructive Implementation:* The Raw query, albeit simple, shows a few issues. The main problem being that it first finds all the possible paths of length  $n \times m - 1$ , and then filters out the non-Hamiltonian ones. This means that, for larger boards, the query will take a very long time to complete - if it completes at all. In order to address this issue, a more 'constructive' approach has been implemented. The idea is to start from a node, and to try to expand the path step by step, ensuring each time that the Hamiltonian property is preserved. The query is built in Python in the following way:

```
MATCH (n0:Node) -[:MOVE]-> (n1:Node)
WHERE id(n1) NOT IN {id(n0)}
MATCH (n1) -[:MOVE]-> (n2:Node)
WHERE id(n2) NOT IN {id(n0), id(n1)}
...
```

The query then contains every full path at the end, and these are guaranteed by construction to be Hamiltonian. The starting and ending nodes can be specified in the same way as in the Raw implementation, as well as limiting to only a few solutions.

c) *The APOC Implementation:* The APOC library [5] for Neo4J contains the `apoc.path.expandConfig` procedure. By leveraging its parameters, it's possible to convert it into an Hamiltonian-finding algorithm.

```
MATCH (start:Node)
CALL apoc.path.expandConfig(
  startNode, {
    relationshipFilter: "MOVE>",
    minLevel: {parameters["pathLength"]},
    maxLevel: {parameters["pathLength"]},
    uniqueness: "NODE_GLOBAL",
    labelFilter: 'Node'
  }
) YIELD path
```

If needed, the starting node can be specified in the same way as in the previous implementations; the ending node can be passed as an additional parameter to the APOC function itself, just like the number of paths to return.

d) *The Python Implementation:* The last implementation is a pure Python one. It makes use of a backtracking algorithm to find one or more Winning Paths in a Board Graph. The algorithm is implemented in a recursive way, and it can be summarized as follows:

```
find_path(current_path, board):
  if current_path covers all nodes:
    return current_path;
  for each move:
    add move to current_path;
    find_path(current_path, board);
    remove move from current_path;
```

Another known problem, similar to *Diplomatico*, is the knight's tour problem. A well known heuristics for that problem is the Warnsdorff's rule [3]: this says to first visit nodes with fewest moves available. This heuristics has been implemented in the Python solution as well, and the effects of this rule over the speed of resolution has been analyzed in Section IV.

2) *Centrality Measures:* The aim of this work also included the analysis of some graph-theoretic centrality measures to provide specific heuristics for solving a Board Graph. For analysis purposes, these can be directly computed inside a Neo4J server through the apposite `gds` library [6]. The centrality measures employed are listed below.

a) *Degree Centrality:* Given an undirected graph  $G = (V, E)$ , the **degree centrality** of a node  $v \in V$  is exactly the

number of edges this node has:

$$C_D(v) = |\{e \in E | v \in e\}|$$

This value can be normalized by dividing each node's value by the maximum centrality found within the graph.

*b) Betweenness Centrality:* Given a graph  $G = (V, E)$ , the **betweenness centrality** of a node  $v \in V$  is defined as the number of shortest paths between pairs of nodes that pass through  $v$ . Formally, let  $\sigma_{st}$  be the total number of shortest paths between nodes  $s, t \in V$ , and let  $\sigma_{st}(v)$  be the number of those paths that pass through  $v$ . Then

$$C_B(v) = \sum_{\substack{s, t \in V \\ s \neq v \neq t}} \frac{\sigma_{st}(v)}{\sigma_{st}}.$$

Nodes with high betweenness act as bridges, controlling the flow of information between different regions of the graph.

*c) Closeness Centrality:* The **closeness centrality** of a node  $v \in V$  measures how close  $v$  is to all other nodes in the graph. It is defined as the reciprocal of the sum of the shortest-path distances  $d(v, u)$  from  $v$  to every other node  $u \in V$ :

$$C_C(v) = \frac{1}{\sum_{u \in V \setminus \{v\}} d(v, u)}.$$

Nodes with high closeness centrality can reach other nodes quickly and are often well-positioned to spread information.

*d) Eigenvector Centrality:* The **eigenvector centrality** of a node  $v \in V$  assigns relative scores to nodes such that connections to high-scoring nodes contribute more than connections to low-scoring ones. Formally, it is defined as the  $v$ -th component of the principal eigenvector of the adjacency matrix  $A$  of the graph:

$$C_E(v) = \frac{1}{\lambda} \sum_{u \in V} A_{vu} C_E(u),$$

where  $\lambda$  is the largest eigenvalue of  $A$ . This centrality highlights nodes that are influential not just by their degree, but by being connected to other important nodes.

#### IV. RESULTS AND DISCUSSION

In this section, the results of the different approaches in finding Hamiltonian paths for the *Diplomatico* graphs are analyzed, along with some improved versions enhanced by the results of the centrality measures. The experiments were conducted on Neo4J version 2025.07.01, the APOC library version 2025.07.1core and the gds library version 2.20.0. The softwares were given maximum 16GB of RAM to execute the code.

*1) Searching the Solution Space:* The results shown in Table I have been obtained by running each query - or algorithm - 5 times, and by averaging the time taken to complete the task. The starting node and the ending node were not specified, and only one solution was required.

While, instead, the results shown in Tabel II have been obtained by running each query at the same conditions as above, but by also specifying the starting node as  $v_{1,1}$  and the

Board Size	Raw	Constructive	APOC	Python
$4 \times 5$	0.4797s	1.3337s	0.1615s	<b>0.0085s</b>
$4 \times 6$	>30s	2.3749s	0.2981s	<b>0.0180s</b>
$4 \times 7$	>30s	4.4709s	0.6444s	<b>0.1897s</b>
$5 \times 5$	>30s	5.7675s	0.7468s	<b>0.0040s</b>

**TABLE I:** Time results in finding only one solution for a certain board graph.

Board Size	Raw	Constructive	APOC	Python
$4 \times 5$	0.7142s	0.0417s	0.0256s	<b>0.0032s</b>
$4 \times 6$	>30s	0.0659s	<b>0.0250s</b>	0.0514s
$4 \times 7$	>30s	3.2629s	<b>0.5479s</b>	0.5982s
$5 \times 5$	>30s	0.6837s	0.4557s	<b>0.0147s</b>

**TABLE II:** Time results in finding only one solution, with constraints on the starting and ending nodes.

ending node as  $v_{n,m}$  - the top-left and bottom-right corners of the board, respectively.

In Table III, instead, not only the starting and ending nodes were specified, but also every solution was required.

Board Size	Raw	Constructive	APOC	Python
$4 \times 5$	1.1492s	0.0331s	0.0143s	<b>0.0034s</b>
$4 \times 6$	>30s	0.6907s	<b>0.0307s</b>	0.0503s
$4 \times 7$	>30s	16.6208s	<b>0.6338s</b>	0.6547s
$5 \times 5$	>30s	3.1501s	2.9022s	<b>0.4530s</b>

**TABLE III:** Time results in finding every solution, with constraints on the starting and ending nodes.

As it can be easily seen, the two most competitive implementations are the APOC one and the pure Python one. In order to compare them, a few more tests have been conducted on larger boards - excluding the too slow Raw and Constructive implementations. More specifically, each of the time results shown in Table IV has been obtained as a mean over 20 runs; only one solution was required and no starting and ending node were specified.

Board Size	APOC	Python
$5 \times 6$	<b>0.0578s</b>	0.4754s
$4 \times 8$	<b>3.9755s</b>	6.3072s
$5 \times 7$	0.1442s	<b>0.0007s</b>
$6 \times 6$	<b>0.2700s</b>	0.5473s

**TABLE IV:** Time results in finding only one solution for larger boards.

As expected, as the board size increases, the time taken to find a solution has an exponential growth. Curiously enough, it seems that - in general - solutions are easier to be found for more 'regular' board sizes, rather than very rectangular ones. Solutions also seem easier to find if the starting and ending node are specified - probably because this reduces the overall search space. This could be due to the fact that, in more rectangular boards, there are less possible moves available from each node - hence, the search space is smaller. Overall, the APOC implementation seems to be more stable in terms of time taken for runs, while the Python one starts going slower as the complexity rises. The board  $5 \times 7$  is solved very fast - probably due Warnsdorff's heuristics working accurately well in this particular case.

In Table V, it has been compared the usage of the Warnsdorff's rule with a random choice for the nodes; the results have been obtained by averaging over 10 runs of the algorithms. Except a few cases, as a rule of thumb, it could be said that employing Warnsdorff's rule for choosing the next node is an efficient strategy, improving by a bit the results in time complexity.

4 × 5	4 × 6	4 × 7	5 × 5	5 × 6	4 × 8	5 × 7	6 × 6
0.0085s	<b>0.0180s</b>	<b>0.1897s</b>	<b>0.0040s</b>	0.4754s	6.3072s	<b>0.0007s</b>	<b>0.5473s</b>
<b>0.0055s</b>	0.0241s	0.2776s	0.0330s	<b>0.0462s</b>	<b>4.5177s</b>	>30s	0.8116s

**TABLE V:** Comparison between employing Warnsdorff's rule (first row) or random nodes (second row).

Table VI shows a collection of the number of solutions for boards that have been completely explored.

Board Size	N° of Solutions
4 × 5	144
4 × 6	128
4 × 7	72
5 × 5	12400

**TABLE VI:** Number of solutions for each board graph thoroughly analyzed.

2) *Starting Nodes:* The usage of centrality measures, as it will be seen, can be exploited to see how it relates to starting nodes. More specifically, it has been examined the relationships between the number of paths starting from a certain node  $v_{i,j}$  and its centrality measures. In order to do so, Pearson's correlation has been analyzed - to find trends between these two variables. In order to make the analysis statistically significant, the relative  $p$ -values are also analyzed to understand the likelihood of the findings. The results of the experiments can be found on Table VII.

Board	Metric	Centrality Measure			
		Betweenness	Closeness	Degree	Eigenvector
4 × 5	Correlation	<b>-0.8228***</b>	-0.5899**	-0.7845***	-0.1728
	$p$ -value	0.0000	0.0062	0.0000	0.4664
4 × 6	Correlation	<b>-0.7744***</b>	-0.6316***	-0.7197***	-0.5179**
	$p$ -value	0.0000	0.0009	0.0001	0.0095
5 × 5	Correlation	<b>-0.6820***</b>	-0.2543	-0.4037*	-0.1460
	$p$ -value	0.0002	0.2199	0.0454	0.4861
4 × 7	Correlation	<b>-0.9098***</b>	-0.8204***	-0.8596***	-0.5842**
	$p$ -value	0.0000	0.0000	0.0000	0.0011
5 × 6	Correlation	-0.7353*	-0.4310	<b>-0.7729*</b>	-0.5072
	$p$ -value	0.0240	0.2468	0.0146	0.1634

**TABLE VII:** Correlation coefficients ( $r$ ) and  $p$ -values between Hamiltonian paths and centrality measures across different board sizes.

First of all, to comment these results, it can be easily seen that most of them - except a few - can be declared conclusive, thanks to the very low  $p$ -values. Then, these results may seem counter-intuitive at first: they indicate that the higher the centrality values of a certain node, the lesser the number of Winning Paths will span starting from it. This, however, may

be justified by the fact that starting from a more central node - for the betweenness centrality, a *bottleneck* - may occlude more paths rather than starting from a more unreachable node.

## V. CONCLUSION AND FUTURE WORK

This work has presented a comprehensive graph-based analysis of the solution space in the game of *Diplomatico*. By modeling the puzzle as a Board Graph and leveraging both Neo4J and Python implementations, we have demonstrated the exponential complexity inherent in enumerating Hamiltonian paths and provided empirical benchmarks for several search strategies. The results highlight the effectiveness of advanced graph database procedures (such as APOC) and heuristic-driven algorithms (notably Warnsdorff's rule) in efficiently navigating large solution spaces.

Key findings include the derivation of theoretical bounds on the number of solutions, the identification of structural properties that impact solvability, and the comparative performance of different computational approaches. The experiments confirm that exhaustive search quickly becomes infeasible as board size increases, underscoring the need for scalable heuristics and optimized queries.

Future work may focus on extending these methods to boards with blocked cells, exploring parallelization strategies, and integrating machine learning techniques to predict solvability or guide search. Additionally, further theoretical investigation into tighter bounds and the combinatorial structure of solution spaces could yield deeper insights. The approaches and tools developed here provide just a solid foundation for still unexplored game of *Diplomatico*.

## REFERENCES

- [1] R. C. contributors, "Solve a hopido puzzle," [https://rosettacode.org/wiki/Solve\\_a\\_Hopido\\_puzzle](https://rosettacode.org/wiki/Solve_a_Hopido_puzzle), 2025, accessed: 2025-09-17.
- [2] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. Cengage Learning, 2013, pp. 292–314.
- [3] H. C. von Warnsdorff, "Des rösselsprunges einfachste und allgemeinste lösung." [Online]. Available: [https://zs.thulb.uni-jena.de/receive/jportal\\_jparticle\\_00189099](https://zs.thulb.uni-jena.de/receive/jportal_jparticle_00189099)
- [4] Neo4j, Inc., *Neo4j Graph Database*, 2025, version 2025.07.01. [Online]. Available: <https://neo4j.com>
- [5] Neo4j Labs, "Apoc: Awesome procedures on cypher," 2025, version 2025.07.1core. [Online]. Available: <https://neo4j.com/labs/apoc>
- [6] Neo4j, Inc., *Neo4j Graph Data Science Library Manual*, 2025, online documentation and software plugin. [Online]. Available: <https://neo4j.com/docs/graph-data-science/current/>

## APPENDIX A

### OVER THE BOUNDS OF THE NUMBER OF SOLUTIONS

Here is the detailed proof of the lower bound referenced in the main text, specifically for Proposition II.2. The first part - the upper bound - of the proof can be easily given, while a set of intermediate results are later needed for the second part of it - the lower bound.

**Proof. Upper Bound.** Any winning path  $h$  must pass from a 'corner' node (i.e.,  $v_{1,1}$ ,  $v_{1,m}$ ,  $v_{n,1}$ ,  $v_{n,m}$ ). By evaluating the possibilities, it can be easily determined that only three edges are connected to any corner node.

$$\begin{aligned} \forall e = \{v_{1,1}, v_{i,j}\} \in E, \\ (|1-i|=3 \wedge 1=j) &\implies v_{i,j} \in \{v_{4,1}, v_{-2,1}\} \\ (|1-j|=3 \wedge 1=i) &\implies v_{i,j} \in \{v_{1,4}, v_{1,-2}\} \\ (|1-i|=|1-j|=2) &\implies \\ v_{i,j} &\in \{v_{3,3}, v_{3,-1}, v_{-1,3}, v_{-1,-1}\} \end{aligned}$$

In this specific case, the only possible adjacent nodes are  $v_{4,1}$ ,  $v_{1,4}$ ,  $v_{3,3}$ . The same reasoning applies to the other corner nodes.

Moreover, the start node - in the general scenario - will have at most 8 edges connected to it (assuming, from the previous verification, that each node is valid). Every other node, however, will have at most 7: surely, the edge chosen to reach it should not be considered, otherwise we would obtain a path that couldn't be Hamiltonian - reaching an already visited node. This means that, combinatorially speaking:

$$N(\mathfrak{G}_{n \times m}) \leq 8 \cdot 3^4 \cdot 7^{nm-5} < 7^{nm-1}$$

□

To reach the second part of the proof, some handful lemmas are presented.

**Lemma A.1.** *For any Board Graph  $\mathfrak{G}_{n \times m}$  with  $9 \geq n, m \geq 5$ , there exist at least 2 distinct Winning Path  $h$  such that:*

$$h_1 = v_{3,1} \wedge h_{nm} = v_{3,m-2}$$

The Lemma A.1 can be proved, to the best of my knowledge, only by enumerating all the possible couples  $(n, m)$  - with, without loss of generality,  $n \geq m$ , otherwise the indices of the nodes could be easily swapped - and by explicitly finding the 2 distinct Winning Paths for each of them. What the Lemma states is that there is a guarantee of 2 different solutions which 'start' from the border node  $v_{n-2,1}$  and that end with the node displaced by 2 both horizontally and vertically from the outer edges. This constraint guarantees the possibility to create a tilization of the board for every  $n, m \geq 5$  - hence, guaranteeing a lower bound.

**Lemma A.2.** *Given the Board Graph  $\mathfrak{G}_{(n+k) \times m}$  with  $9 \geq n, m, k \geq 5$ , then  $\mathfrak{G}_{(n+k) \times m}$  is solvable.*

17	9	2	16	8	42	34	27	41	33
4	14	19	5	13	29	39	44	30	38
1	22	25	10	21	26	47	50	35	46
18	6	3	15	7	43	31	28	40	32
24	11	20	23	12	49	36	45	48	37

**Fig. 4:** A  $5 \times 10$  grid solved by tiling. The two halves have the same path connecting  $v_{1,3}$  to  $v_{3,3}$ . In blue, the 'link' between two otherwise disjoint paths.

**Proof.** Consider the two Board Graphs  $\mathfrak{G}_{n \times m}$  and  $\mathfrak{G}_{k \times m}$ . By Lemma A.1, both of them are solvable - since  $9 \geq n, m, k \geq 5$ . Let  $h^1$  be one of the Winning Paths in  $\mathfrak{G}_{n \times m}$  guaranteed to exists; and let  $h^2$  be one of the Winning Paths in  $\mathfrak{G}_{n \times k}$  guaranteed to exists. Defining  $\hat{h}^2$  in the following way:

$$\hat{h}^2 = (v_{i,j+m} : v_{i,j} \in h^2)$$

So: the new defined path  $\hat{h}^2$  is simply the path  $h^2$  'shifted' horizontally by  $m$  columns. It can be proven that the following  $h^3$  is indeed a Winning Path:

$$h^3 = (h^1, \hat{h}^2)$$

First of all, the two paths  $h^1$  and  $\hat{h}^2$  are disjoint, since they refer to different nodes of the graph; moreover, they completely 'touch' each node of the partitions, being Winning Paths. The only thing left to prove is that the last node of  $h^1$  is connected to the first node of  $\hat{h}^2$ . By Lemma A.1, the last node of  $h^1$  is  $v_{3,m-2}$ ; while the first node of  $\hat{h}^2$  is  $v_{3,m+1}$ . It can be easily verified that:

$$(|m+1-m-2|=3 \wedge 3=3)$$

hence, the edge  $\{v_{3,m-2}, v_{3,m+1}\}$  exists, giving a complete solution for the Board Graph  $\mathfrak{G}_{(n+k) \times m}$ . □

The result exposed in the Lemma A.2 is graphically represented in Figure 4. It's to note that the above Lemma A.2 can be easily adapted to increasing values of columns for Board Graphs.

The following Lemmas, instead, focus on stacking Board Graphs vertically.

**Lemma A.3.** *For any Board Graph  $\mathfrak{G}_{n \times m}$  with  $9 \geq n, m \geq 5$ , there exist at least 2 distinct Winning Path  $h$  such that:*

$$h_1 = v_{3,1} \wedge h_{nm} = v_{n,m}$$

Once again, a proof of the Lemma would require an enumeration of all the possible couples  $(n, m)$ , and their respective solved distinct Hamiltonian.

**Lemma A.4.** *Given the Board Graph  $\mathfrak{G}_{n \times (m+k)}$  with  $9 \geq n, m, k \geq 5$ , then  $\mathfrak{G}_{n \times (m+k)}$  is solvable.*



*Proof.* Consider the two Board Graphs  $\mathfrak{G}_{n \times m}$  and  $\mathfrak{G}_{n \times k}$ . Without loss of generality, assume  $m \geq k$ . Then, by knowing Lemma A.3, both of them are solvable - since  $9 \geq n, m, k \geq 5$ . Let  $\mathfrak{h}^1$  be one of the Winning Paths in  $\mathfrak{G}_{n \times m}$  guaranteed to exist; and let  $\mathfrak{h}^2$  be one of the Winning Paths in  $\mathfrak{G}_{n \times k}$  guaranteed to exist, either by Lemma A.1 or by Lemma A.3. Then, define  $\hat{\mathfrak{h}}^2$  in the following way:

$$\hat{\mathfrak{h}}^2 = (v_{i+n, m-j+1} : v_{i,j} \in \mathfrak{h}^2)$$

This path  $\hat{\mathfrak{h}}^2$  is simply the path  $\mathfrak{h}^2$  swapped over the columns and 'shifted' vertically by  $n$  rows. It can be proven that the following  $\mathfrak{h}^3$  is indeed a Winning Path:

$$\mathfrak{h}^3 = (\mathfrak{h}^1, \hat{\mathfrak{h}}^2)$$

First of all, the two paths  $\mathfrak{h}^1$  and  $\hat{\mathfrak{h}}^2$  are disjoint, since they refer to different nodes of the graph; moreover, they completely 'touch' each node of the partitions, being Winning Paths. The only thing left to prove is that the last node of  $\mathfrak{h}^1$  is connected to the first node of  $\hat{\mathfrak{h}}^2$ . By Lemma A.3, the last node of  $\mathfrak{h}^1$  is  $v_{n,m}$  while the first node of  $\hat{\mathfrak{h}}^2$  is  $v_{n+3, m}$ . It can be easily verified that:

$$(|n+3-n| = 3 \wedge m = m)$$

Hence, the edge  $\{v_{n,m}, v_{n+3, m}\}$  exists, giving a complete solution for the Board Graph  $\mathfrak{G}_{n \times (m+k)}$ .  $\square$

*Proof. Lower Bound.* We will show that for any  $n, m \geq 5$ :

$$N(\mathfrak{G}_{n \times m}) \geq 2^{(n-4)(m-4)/25}$$

To establish this, we can use a combinatorial argument based on the structure of the board graph. More specifically, given a generic Board Graph  $\mathfrak{G}_{n \times m}$ , we can partition it into smaller sub-graphs of size  $5 \times 5$  plus - in case of  $n, m$  not divisible by 5 - some additional sub-graphs of size  $k_i \times \ell_i$ , with  $9 \geq k_i, \ell_i \geq 5$ . Thanks to Lemma A.2 and Lemma A.4, we know that all these sub-graphs are solvable, and particularly that the whole Board Graph  $\mathfrak{G}_{n \times m}$  is also solvable, by tiling the sub-graphs accordingly.

The total number of sub-graphs in which to partition the original Board Graph is  $\lfloor \frac{n}{5} \rfloor \cdot \lfloor \frac{m}{5} \rfloor$ . This can be proven by simply considering that each sub-graph has a size of  $5 \times 5$ ; hence, the number of sub-graphs that can fit in the original Board Graph is simply the integer division of the dimensions of the original graph by 5. Each of these sub-graphs has at least 2 distinct Winning Paths, as guaranteed by Lemma A.1 and Lemma A.3. This means that, combinatorially speaking, the total number of distinct Winning Paths in the original Board Graph is at least:

$$N(\mathfrak{G}_{n \times m}) \geq 2^{\lfloor n/5 \rfloor \cdot \lfloor m/5 \rfloor}$$

In order to show the original, exponential in  $n, m$ , bound, it's only necessary to consider that:

$$\lfloor n/5 \rfloor \geq (n-4)/5$$

This is because, in the worst case scenario,  $n = 5 \times k + 4$ ; hence,  $\lfloor n/5 \rfloor = k$  and  $(n-4)/5 = k$ . The same reasoning applies to  $m$ ; hence, the original bound is proven.  $\square$

10	5	2	26	8	3
22	15	12	21	18	13
1	27	9	4	28	25
11	6	19	14	7	20
23	16	29	24	17	30
44	49	56	43	48	55
52	35	46	51	34	41
38	58	32	37	57	31
45	50	53	42	47	54
60	36	39	59	33	40

**Fig. 5:** A  $5 \times 6$  grid solved by tiling. The two halves have the same path, albeit rotated over the columns, connecting  $v_{3,1}$  to  $v_{5,6}$  and then  $v_{8,6}$  and  $v_{10,1}$ . In blue, the 'link' between two otherwise disjoint paths.

## APPENDIX B

### OVER THE CREATION OF A BOARD GRAPH IN NEO4J

In order to provide a common ground for retrieving a solution in Neo4J [4] (see Section III), a variable-dependent query was designed to create a Board Graph of desired size. The structure of the graph is organized in the following way:

- Each node is created with the label `Node`, and has - as properties - `row` and `col`, respectively the indices of row and of column in the original board-like structure.
- Each link between two nodes (present only if a legal move allows the movement between those two nodes) is created, with label `MOVE`.

The Python function implementing the creation of this query is described in the current Appendix in pseudocode.

```
create_graph_query(rows, cols):  
    for i = 0, ..., rows - 1:  
        for j = 0, ..., cols - 1:  
            query += create_query(i, j)
```

This snippet regulates the creation of all the necessary nodes, each with its own specific properties. The code loops through all the possible row indices (from 0 to the input parameter `rows-1`) and through all the possible column indices. For every possible pair of row and column indices, then, it stores a line created by the method `create_query(i, j)`. This returns the following query:

```
"CREATE (n_i_j:Node {row: i, col: j})\n"
```

of course, substituting the two parameters  $i, j$  with the passed values. For instance, for the node at position (2,3), the line will ask to create a node with label `Node`, called `n_1_2`, and with properties `row=1` and `col=2` (because values start from 0).