Universiteit Utrecht

The NP-completeness of some lesser known logic puzzles

Author: Mieke Maarse - 5750032

Supervisor: Dr. Benjamin Rin
Second evaluator: Prof. dr. Michael Moortgat

7.5 ECTS
Bachelor Kunstmatige Intelligentie
Utrecht University

June 28, 2019

Abstract

Logic puzzles have been gaining popularity and many puzzles have been proved to be NP-complete. When a puzzle is NP-complete, it is not feasible to try to compute a solution. In this case algorithms that approximate the solution, or programs that are limited in input size can be used. In this paper we use the Hamiltonian path and cycle problems in grid graphs, and the Latin square completion problem to show that the puzzles *unequal* and *adjacent*, *towers*, *chains* and *linesweeper* are NP-complete.

Contents

1 Introduction

Logic puzzles are puzzles that can be solved through deductive reasoning, like *sudoku* or *kakuro*. These puzzles are satisfying to solve because, when finished, it is immediately clear if a solution is correct, but solving one is a challenge. This type of puzzle is gaining popularity, and so more research on the topic of these puzzles is being done [1][2]. Puzzles are interesting because the instructions are usually quite simple and easy to understand. The difficulty is in the structure of these puzzles. What is even more interesting is that if two puzzles are both in a certain complexity class, one can be solved by solving another.

Not just puzzles have complexity classes. A lot of real world problems are in the same complexity class as these puzzles. The time complexity of a problem is expressed as the amount of steps it takes to solve the problem as a function of the size of the input. This way we can organise problems into different classes of complexity. Two important classes of time complexity are the classes P and NP. P is the class of problems that can be solved in polynomial time by a deterministic Turing machine. NP is the class of problems of which a solution can be checked in polynomial time by a deterministic Turing machine, or, in other words, the class of problems which can be solved by a non-deterministic Turing machine. It is not known if P and NP are the same class, or if P is a strict subset of NP. A problem is NP-hard if all problems in NP are polynomial time mapping reducible[1] to it. A problem is NP-complete if it is both NP-hard and in NP. It is assumed from now on that readers of this paper are familiar with the contents of *Introduction to the Theory of Computation* by Michael Sipser [3]. The beauty of NP-complete problems is that they are all reducible to each other. If one NP-complete problem is found to be solvable in polynomial time, all problems in NP are, and P = NP. The more NP-complete problems we know, the more we have to use in NP-hardness proofs, and try to find polynomial time solutions for, if you believe P = NP.

According to Cobham's thesis, something can only be feasibly computed if it can be computed in polynomial time [4]. This means that finding the time complexity class of a problem tells us if it is feasible to create a solver. If it is not, it can be better to approximate the solution, restrict the input size for a solver, or find another efficient way to find a solution. If a problem is NP-complete, finding a polynomial time solution to it is impossible if P $\neq$ NP, and if P = NP in this time, it is still highly unlikely you'll find one. If you find a polynomial time solution to an NP-complete problem, you prove that P = NP. Many people have tried this, but nobody has succeeded. This is why knowing

[1] In *Introduction to the Theory of Computation*, on page 300, definition 7.29, Michael Sipser gives the following definition of polynomial time mapping reducible:
"Language A is ***polynomial time mapping reducible***, or simply ***polynomial time reducible***, to language B, written A $\leq_P$ B, if a polynomial time computable function $f : \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \Longleftrightarrow f(w) \in B.$$

The function f is called the ***polynomial time reduction*** of A to B." [3]

a problem is NP-complete is useful. It saves a lot of time to know beforehand that finding a polynomial time solver is not going to happen. This is also why we want to prove the NP-completeness of these puzzles. It is a stepping stone to finding another way to solve these puzzles.
Sudoku and *kakuro* have been proved to be NP-complete [5][6], among many other puzzles, such as *pearl* [7] and *pipelink* [8]. You can find many more NP-complete problems in *A survey of NP-complete puzzles* [9]. The intention of the authors of the survey was to motivate further research in this area. Even though puzzles have been an interest in artificial intelligence research before [9][10], there are still many puzzles that have not been researched yet. Four puzzles that have not been researched before to our knowledge, are *unequal/adjacent*, *towers*[2], *chains*(also known as *link-a-pix*), and *linesweeper*. *Unequal/adjacent* are two variants of the same puzzle. In this paper we will prove these puzzles to be NP-complete. We will do this by proving them to be in NP, and proving them to be NP-hard. The puzzles *unequal/adjacent* and *towers*, are Latin square completion problems, and because of that, the NP-hardness proofs are trivial. To prove *chains* NP-hard, we will show a reduction from the Hamiltonian path problem in grid graphs. To prove *linesweeper* NP-hard, we will show a reduction from the Hamiltonian cycle problem in grid graphs.
For the NP-hardness proofs for *chains* and *linesweeper* we formulate these puzzles as the following decision problem: is this specific *chains/linesweeper* puzzle solvable?

In the next two subsections we explain the Hamiltonian path and cycle problem in grid graphs and the Latin square completion problem.

1.1 The Hamiltonian path and cycle problem in grid graphs

In this section we will elaborate on the problem of finding a Hamiltonian path or Hamiltonian cycle in a grid graph. A Hamiltonian path is a path through a graph from a start node s to an end node t. This path goes through every node exactly once. A Hamiltonian cycle is a cycle through a graph that goes through every node exactly once.
A grid graph is a node-induced finite subgraph of the infinite grid graph. The infinite grid graph is a graph of which the nodes are connected by an edge if and only if the distance between them is 1. In figure 1 a part of the infinite grid graph can be seen. A node-induced finite subgraph of the infinite grid graph exists of a finite

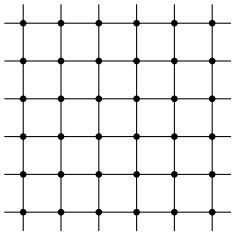


Figure 1: A part of the infinite grid graph

[2]Unfortunately we found a proof for the NP-completeness of *towers* under the name *building puzzle* [11] only right before the deadline of this thesis. This name was not known to us before and we came across it by coincidence. The proof for *towers* in this thesis was written before we found this paper. This paper has had no influence whatsoever in the process of writing this thesis.

number of nodes from the infinite grid graph, with an
edge between nodes if and only if the distance is 1. Figure 2 is an example of a grid graph with both a Hamiltonian path and cycle.
Finding a Hamiltonian path or cycle in a grid graph is NP-complete, and so is answering the question whether there is a Hamiltonian cycle or path in a certain grid graph [12]. The decision problem here is: is there a Hamiltonian path/cycle in this grid graph?

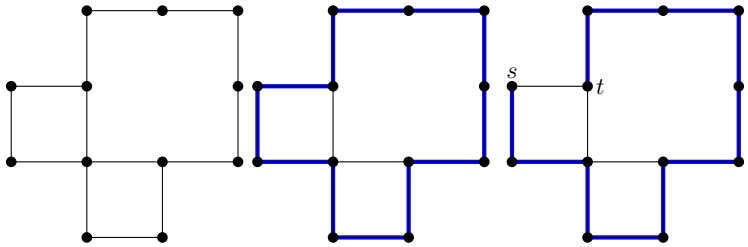


Figure 2: A grid graph (left) with a Hamiltonian cycle (middle) and a Hamiltonian path (right)

1.2 The Latin square completion problem

Another NP-complete problem we will come across in this paper is the Latin square completion problem. A Latin square is an n by n grid with the numbers 1 to n occurring exactly once in every row and every column. The Latin square completion problem is the problem of completing an incomplete Latin square. This problem has been proven to be NP-complete [13], even when at most three unfilled squares exist in the grid [14]. The decision problem is to tell whether it is possible to complete an incomplete Latin square. Figure 3 shows an example of a Latin square, and an incomplete square with at most three unfilled cells.

4	1	2	3
3	4	1	2
2	3	4	1
1	2	3	4

	1		3
	4		
2		4	1
1	2		4

Figure 3: A Latin square (left) and an incomplete Latin square (right)

1.3 Structure of this paper

The following sections will each be about *unequal/adjacent*, *towers*, *chains*, and *linesweeper* respectively. This way the puzzles are ordered from the simplest reduction to the most complicated reduction. For each puzzle we will do three things: we explain the rules of the puzzle, we show the puzzle to be in NP,

and we show the puzzle to be NP-hard. Finally, we make some suggestions for interesting future research.

2 Trivial proofs

2.1 *unequal* and *adjacent*

Unequal and *adjacent* are two variants of a Latin square completion puzzle with the added constraint of signs inside the grid.
Unequal and *adjacent* puzzles both consist of an n by n grid that is an incomplete Latin square, with some additions. In an *unequal* puzzle, there can be a smaller-than ($<$) or greater-than ($>$) sign in between two squares. These signs indicate that the number in one square needs to be respectively smaller than, or greater than the number in the square next to it. An example of an *unequal* puzzle can be seen in figure 4.
In the *adjacent* puzzle there can be a line in between two squares. The number in one square needs to be exactly one higher than the number in the other square. Figure 5 shows an example of an *adjacent* puzzle.

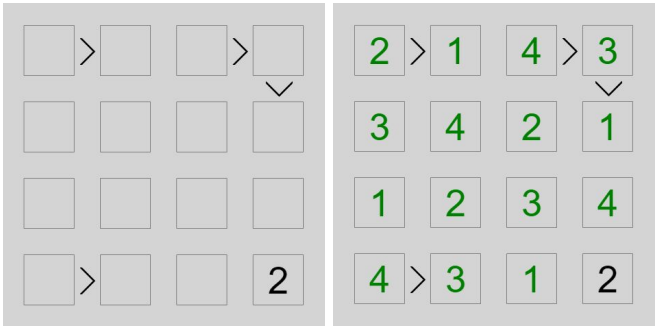


Figure 4: An *unequal* puzzle (left) and its solution (right)[3]

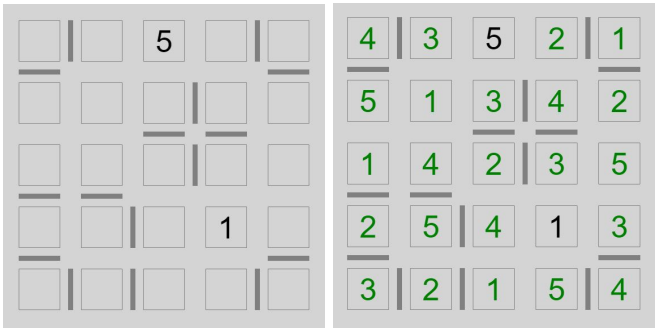


Figure 5: An *adjacent* puzzle (left) and its solution (right)[3]

2.1.1 NP

To check these puzzles, two aspects need to be checked: the Latin square aspect and the *unequal* or *adjacent* signs. To check if the Latin square aspect is completed, for every row and every column it needs to be checked if every number occurs exactly once. This takes $\mathcal{O}(n^2)$ time for an n by n puzzle. Next, the *unequal* or *adjacent* signs have to be checked. There can be a maximum of $2n^2 - 2n$ of signs in an n by n puzzle. For every sign, one thing need to be checked, so the time for checking signs is $\mathcal{O}(n^2)$.
In total the time it can take to check the solution of an n by n *unequal* or *adjacent* puzzle is $\mathcal{O}(n^2)$, so *unequal* and *adjacent* are both in NP.

2.1.2 NP-hardness

As mentioned in the explanation of these two puzzles, the smaller-than or greater-than signs and the *adjacent* signs do not always appear. This means that there is a *unequal* puzzle and there is an *adjacent* puzzle that is just an incomplete Latin square without any additional signs.
Because completing an incomplete Latin square is NP-complete [13], so are these variants.

2.2 *towers*

Each *towers* puzzle consist of an n by n grid with an incomplete Latin square. All squares of the Latin square grid represent a tower. The number in a square represents the height of a tower. There can be numbers on the outside of the grid. These numbers represent how many *towers* can be seen looking at the grid from that point. The objective is to fill in the Latin square keeping to the constraints of any numbers that may be on the sides as can be seen in figure 6.

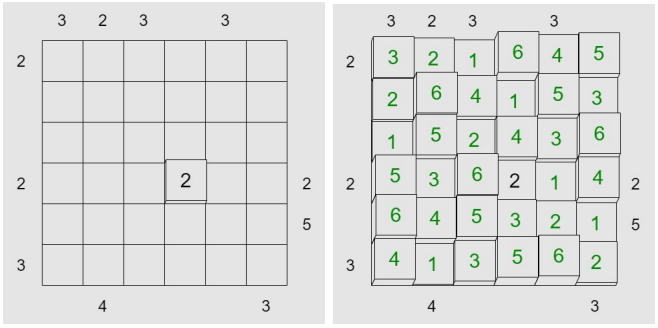


Figure 6: A *towers* puzzle (left) and its solution (right)[4]

[3]Puzzles from: https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/unequal.html

[4]Puzzle from: https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/towers.html

2.2.1 NP

To check this puzzle, two aspects need to be checked: the Latin square aspect and the number aspect. We check the Latin square in the same way as with *unequal* puzzle. This takes $\mathcal{O}(n^2)$ time for an n by n puzzle. There can be up to $4n$ numbers on the sides of an n by n puzzle. For every number it will need to check a row or column of n squares, so checking the numbers takes $\mathcal{O}(n^2)$ time.
In total, checking if the solution to an n by n *towers* puzzle is correct takes $\mathcal{O}(n^2)$, so *towers* is in NP.

2.2.2 NP-hardness

As with *unequal* and *adjacent*, the numbers at the edge of the *towers* puzzle are optional. Without these numbers, solving a *towers* puzzle, is completing an incomplete Latin square, which is NP-complete [13].

3 *chains*

Each *chains* puzzle consists of an n by n grid with with numbers in it. The objective is to connect pairs of the same numbers to each other with a path that is the length of the numbers it connects. This is why numbers always appear in pairs except from the number one. This number comes in singles and connect to themselves. The puzzle is completed when all numbers are connected, as can be seen in figure 7.

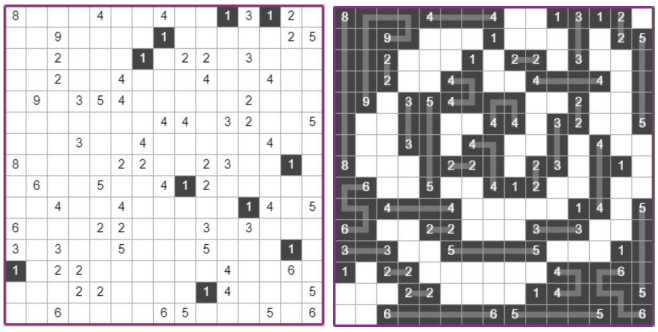


Figure 7: A *chains* puzzle (left) and its solution (right)[5]

[5]puzzle from: https://www.janko.at/Raetsel/Ketten/007.a.htm

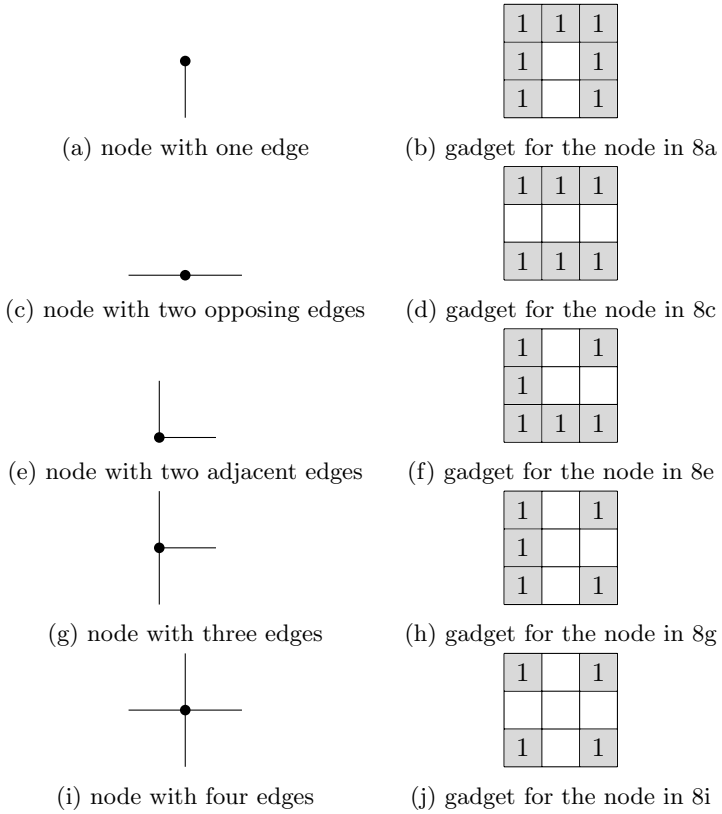
3.1 NP-completeness

Theorem 3.1. *Chains is NP-complete*

Proof idea. Showing that chains is in NP is easy, and we will do so shortly. What is more complicated, is to prove the NP-hardness. To do so, we reduce the Hamiltonian path problem in grid graphs to *chains*. We reconstruct the grid graph in a *chains* puzzle by making 'walls' made of 1's. Then we put one pair of numbers on the grid. One in the middle of the start node, and one in the middle of the end node. This number will be specified in such a way that the path has to go through every 'node'.

Proof. First we show that *chains* is in NP. If the solution of a *chains* puzzle is correct, every number connects to another number with the same value, and the length of the path by which they connect is the same as the value of the numbers.
The maximum amount of numbers in an n by n *chains* puzzle is n^2. Checking if a path is a certain length and if two numbers are the same takes 2 steps per number. This means checking if the solution is correct takes $\mathcal{O}(n^2)$ time, so *chains* in in NP.

Next, we show the NP-hardness of *chains* by a reduction from the Hamiltonian path problem in grid graphs.
For this reduction we categorise nodes in the grid graph by all combinations of edges and show gadgets for each type as can be seen in figure 8. In these gadgets pathways are made with 1's, so that any path passing through a gadget can pass through it the same way it would through a node in a grid graph.
We can piece all these gadgets together so that every node in the grid graph is represented by a gadget. The gadgets are placed in the same position as the nodes in the grid graph so that the gadgets in the chains puzzle connect to each other in the same way the nodes connect in the grid graph connect to each other. This way the possible paths in the chains puzzle are the same as the possible paths in the grid graph as can be seen in figure 9a and b.
Now that we have a representation of the graph in a *chains* puzzle, we need to add the element of finding a Hamiltonian path. To do this we add a pair of numbers, of which one will be in the middle of the start node, and one will be in the middle of the end node. We need to make sure that there is no way that the path visits one node representation more than once and we need to make sure that every node has to be visited.
Because every gadget has one square in the middle where every path that goes through the gadget has to go through, it is not possible to visit a node representation more than once.
To make sure the path has to visit every node, the pair of numbers will have the value of $3m - 2$ for a graph with m nodes. If the path goes through a gadget of a node, this will add a length of three to the path. In the start and end node, it will only add a length of two, because the number is in the middle of the node gadget. This means that if there is a Hamiltonian path from the beginning to



(a) node with one edge (b) gadget for the node in 8a

(c) node with two opposing edges (d) gadget for the node in 8c

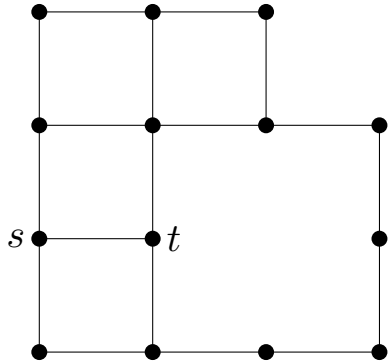
(e) node with two adjacent edges (f) gadget for the node in 8e

(g) node with three edges (h) gadget for the node in 8g

(i) node with four edges (j) gadget for the node in 8i

Figure 8: gadgets for all different types of nodes

the end node in the grid graph, there will be a path of length $3m - 2$ from the representation of the beginning node to the representation of the end node in the *chains* puzzle. This makes the puzzle solvable. An example of a grid graph with a Hamiltonian path, and it's reduction to a chains puzzle can be seen in figure 9a and c. Note that there is a Hamiltonian path from node s to t in 9a, and that the chains puzzle in 9c is solvable.



(a) A grid graph with start node s and end node t for a Hamiltonian path

1	1	1	1	1	1	1	1	1	1	1	1
1								1	1	1	1
1		1	1		1	1		1	1	1	1
1		1	1		1	1		1	1	1	1
1											1
1		1	1		1	1	1	1	1		1
1		1	1		1	1	1	1	1		1
1					1	1	1	1	1		1
1		1	1		1	1	1	1	1		1
1		1	1		1	1	1	1	1		1
1											1
1	1	1	1	1	1	1	1	1	1	1	1

(b) The nodes of the grid graph in 8a are replaced by their gadgets

1	1	1	1	1	1	1	1	1	1	1	1
1								1	1	1	1
1		1	1		1	1		1	1	1	1
1		1	1		1	1		1	1	1	1
1											1
1		1	1		1	1	1	1	1		1
1		1	1		1	1	1	1	1		1
1	40			40	1	1	1	1	1		1
1		1	1		1	1	1	1	1		1
1		1	1		1	1	1	1	1		1
1											1
1	1	1	1	1	1	1	1	1	1	1	1

(c) The reduction of the grid graph in 8a to a *chains* puzzle

Figure 9: The reduction from grid graph to chains puzzle

If a grid graph has no Hamiltonian path from the beginning node to the end node in the grid graph, it will also not be possible to find a path of length $3m-2$ from the representation of the beginning node to the representation of the end node in the *chains* puzzle. This means the puzzle is not solvable.

This is a polynomial time reduction because going from a grid graph with a start and end node to its representation in a *chains* puzzle only takes $\mathcal{O}(m)$ time for a graph with m nodes.
This means *chains* is NP-hard. □

4 *linesweeper*

Each *linesweeper* puzzle is made up of an n by n grid with numbers on it. The objective is to find a loop through the grid that takes the numbers into account. The numbers indicate how many squares adjacent to it (both horizontally, vertically and diagonally) are covered by the loop. An example of a *linesweeper* puzzle can be seen in figure 10.

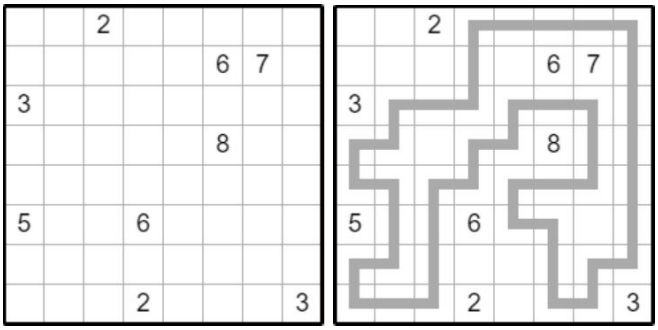


Figure 10: A *linesweeper* puzzle (left) and its solution (right)[6]

4.1 NP-completeness

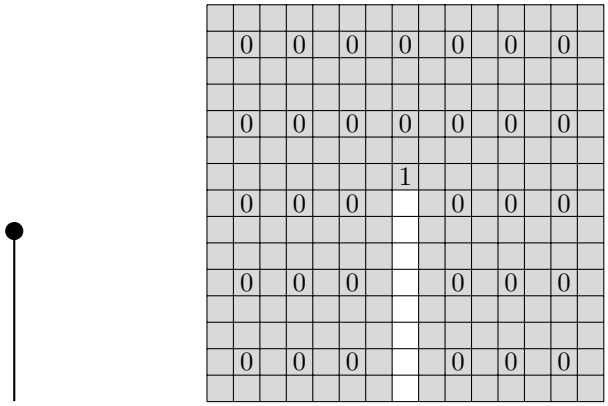
Theorem 4.1. *Linesweeper is NP-complete*

Proof idea. First we show *linesweeper* is in NP. This is quite simple. The hard part is showing *linesweeper* is NP-hard. To do so we show a reduction from the Hamiltonian cycle problem to *linesweeper*. The idea of this reduction is similar to the idea of the *chains* reduction. We categorise the nodes of the graph in the same way and again build 'walls' to represent the grid graph in a *linesweeper* puzzle. Because *linesweeper* already has the element of the loop, we just need to make sure it goes through every node representation exactly once. We do this by using numbers and creating structures of pathways.

Proof. First we show that *linesweeper* is in NP. To check if a solution to a *linesweeper* is correct, for every number on the grid the adjacent squares have to be checked for lines, and it has to be checked if the path of the lines form a loop. A loop covers a minimum of four squares, so the maximum amount of numbers in an n by n *linesweeper* puzzle is $n^2 - 4$. This means checking the squares around every number will take $\mathcal{O}(n^2)$ time.
To check if the path of lines form a loop, the path must be followed and if it ends where it started and all the pieces of path are checked, we know it is a loop. If there are no numbers in an n by n puzzle, the loops maximum length is n^2, so checking the loop can be done in $\mathcal{O}(n^2)$ time. This means *linesweeper* is in NP.

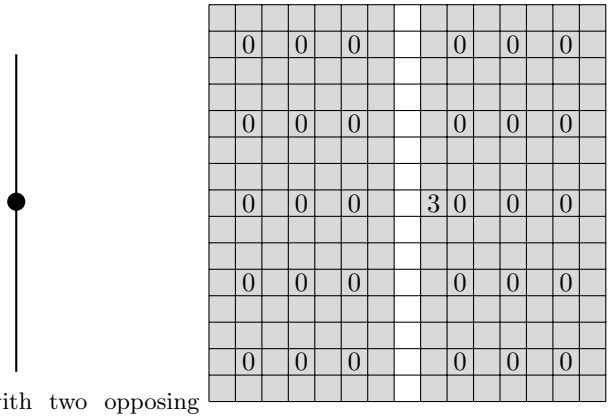
[6]puzzle from: https://www.janko.at/Raetsel/linesweeper/082.a.htm

Now we show the NP-hardness of *chains* by a reduction from the Hamiltonian cycle problem in grid graphs.
The gadgets for nodes with different combinations of edges are made by creating pathways over which the path is able to go. These pathways are made using 'walls' from 0's. 0's create a 3 by 3 block of squares over which the path is not able to go. The gadgets can be seen in figures 11, 12, 13, 14, and 15. In these figures all squares over which the path cannot go are grey. This is purely a visual aid and not actually a part of the gadgets.



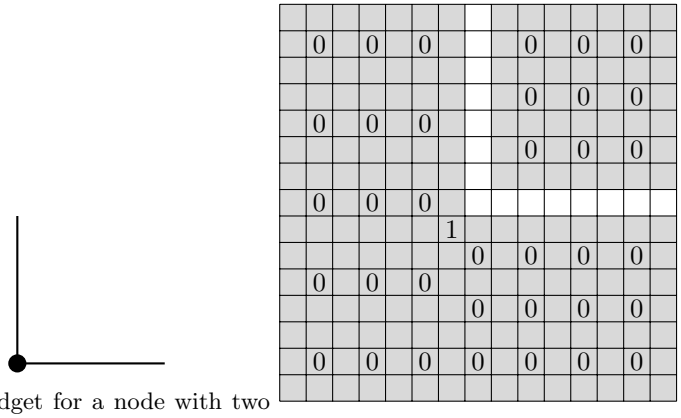
(a) a node with one edge (b) A gadget for the node in 11a

Figure 11: A gadget for a node with one edge



(a) A node with two opposing edges (b) A gadget for the node in 12a

Figure 12: A gadget for a node with two opposing edges



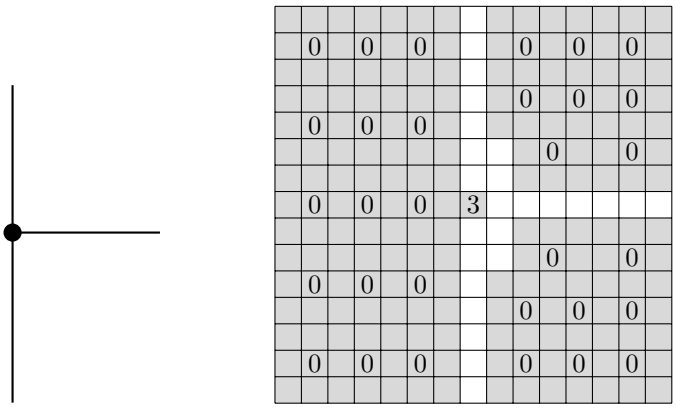
(a) A gadget for a node with two adjacent edges

(b) A gadget for the node in 13a

Figure 13: A gadget for a node with two adjacent edges

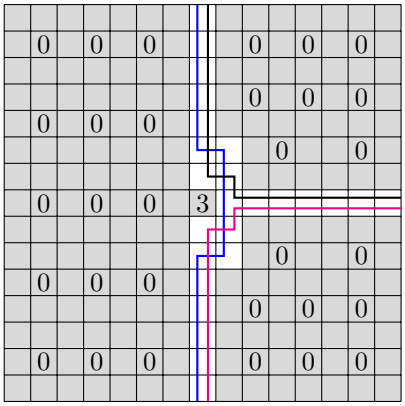
There is only one way to get to and go from a node with two edges, and only one way to get to a node with one edge. So, in the gadgets for nodes with one or two edges, there is only one path possible for a line. This makes it possible to just have single square wide pathways with a number along the middle of the pathway that indicates how many squares of the pathway next to it have to be visited. This adds the constraint that the line must go to or through this node. This way these nodes have to be visited exactly once as can be seen in figures 11, 12, and 13.

When a node has three or four edges, there are multiple ways to go to and from the node, so we don't know how the paths through these gadgets will go beforehand. Because of this, we cannot have the same single block wide pathways as in the gadgets for the nodes with one or two edges. As figure 14 shows, in the gadget for a node with three edges, there are three possible paths through the gadget, with the number three in the middle giving the constraint that the node has to be visited. Figure 14c shows which squares can get visited by the path to meet this constraint. Because all pathways are only one square wide at the outside of the gadget, this gadget can only be visited once as there are only three ways to get in or out of this gadget.



(a) A node with three edges

(b) A gadget for the node in 14a

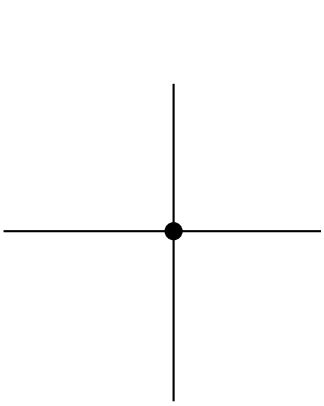


(c) Possible paths through the gadget in 14b

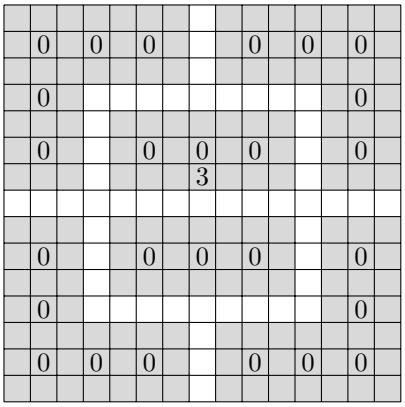
Figure 14: A gadget for a node with three edges

For a gadget with four nodes, there are four ways in or out of the gadget. A way to stop the path visiting the node twice would be to make a single square wide crossing. This way the path has to go through the single square in the middle of the crossing so the middle would be blocked for any other path going through. The problem with this is that it is not possible to make visiting this node compulsory by adding a number. Adding a number in any square in this construction would indicate a specific path through the node, which we do not want. To solve this problem, we add a line going straight through the gadget as can be seen in figure 15. This way, any path through the gadget has to cross the pathway in the middle of this node, with the 3 along it. This three gives the constraint that the path must go through the pathway in the middle of the gadget. When the path goes through this pathway, it visits the middles of the two single square wide crossings at the ends of the pathway. Once these crossings are visited, there is no other way through this gadget anymore. So

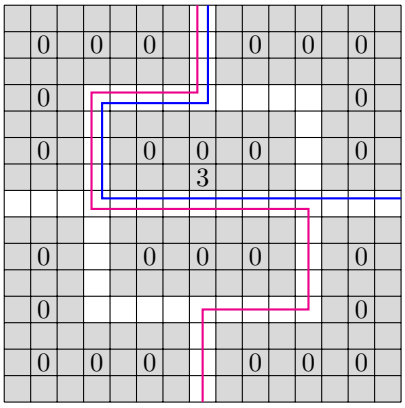
this gadget has to be visited exactly once. In figure 15c a path from north to east and a path from north to south show how this works. A path from west to east would go straight through the middle, and other paths are a rotated or mirrored version of the path from north to east.



(a) A node with four edges



(b) A gadget for the node in 15a



(c) A north to south path and a north to east path through the gadget in 15b

Figure 15: Gadget for a nodes with four edges

When these gadgets are pieced together, the representation of a graph is formed. Because the goal of the puzzle is to find a loop that meets the constraints of every number, and the numbers are now placed in such a way that the pathways represent the edges and nodes of the grid graph, the reduction is complete. Note that the edges in a grid graph are the same as the paths between gadgets in the linesweeper puzzle that is the reduction of this grid graph. This means that if the grid graph has a Hamiltonian cycle, the linesweeper puzzle that is

the reduction of this grid graph is solvable,, and that if there is no Hamiltonian cycle in the grid graph, the linesweeper puzzle that is the reduction of this grid graph is not solvable.

This reduction takes $\mathcal{O}(m)$ time for a graph with m nodes, so it is a polynomial time reduction. This means *linesweeper* is NP-hard. □

5 Conclusion

We have shown the NP-completeness of *unequal* and *adjacent*, *towers*, *chains* and *linesweeper*. While writing this paper, we have tried to prove the NP-completeness of some other puzzles, but we have not found a reduction yet. Two puzzles that would be very interesting to see an NP-hardness proof for are *signpost* from Simon Tatham's portable puzzle collection[7] and *koburin* from nikoli[8]. We have tried to find a reduction from various versions of the Hamiltonian path/cycle problem, but have not succeeded.
A *signpost* puzzle consists of an n by n grid where every square but one is marked with an arrow pointing north, north east, east, south east, south, south west, west, or north west. One square is also marked with the number 1 and one square is marked with the number n^2. This last square is not marked with an arrow. The goal is to find a path from 1 to n^2 following the directions of the arrows and going through every node. A reduction from the Hamiltonian path problem was attempted because the puzzle is a Hamiltonian path problem. The problem in the reduction was that once an arrow in square A faces a certain direction, all squares in that direction can be reached by square A. This is a problem because when a large grid is needed for a large graph, it could not be avoided that paths were possible in the grid that were not possible in the graph, possibly causing a puzzle to be solvable when a graph had no Hamiltonian path.

In this paper we proved *unequal* and *adjacent*, *towers*, *chains* and *linesweeper* NP-complete by asking the question whether a puzzle is solvable, in other words, if it has at least one solution. It would also be interesting to know what the complexity of these puzzles would be if we were asking the question whether there is exactly one solution. Asking this question would probably place these puzzles in the complexity class US (Unique Polynomial-Time) in stead of in NP. This class consists of problems that can be solved by a non-deterministic Turing machine in polynomial time such that the answer is yes if and only if there exists exactly one computation path [15].
It would also be interesting to see what existing algorithms can be used to approximate solutions to these puzzles in polynomial time.

[7] signpost can be played here: https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/signpost.html

[8] koburin can be played here: https://www.janko.at/Raetsel/Koburin/index.htm

References

[1] T. Mantere and J. Koljonen, "Solving, rating and generating sudoku puzzles with ga," in *2007 IEEE congress on evolutionary computation*, pp. 1382–1389, IEEE, 2007.

[2] A. M. Smith, E. Butler, and Z. Popovic, "Quantifying over play: Constraining undesirable solutions in puzzle design.," in *FDG*, pp. 221–228, 2013.

[3] M. Sipser, *Introduction to the theory of computation*. Cengagae Learning, 2013.

[4] A. Cobham, "The intrinsic computational difficulty of functions," *The Journal of Symbolic Logic*, vol. 34, no. 04, p. 24–30, 1969.

[5] I. Lynce and J. Ouaknine, "Sudoku as a sat problem.," in *ISAIM*, 2006.

[6] T. Seta, "The complexities of puzzles, cross sum, and their another solution problems (asp)," 2002.

[7] E. Friedman, "Pearl puzzles are np-complete," *Unpublished manuscript, August*, 2002.

[8] A. Uejima, H. Suzuki, and A. Okada, "The complexity of generalized pipe link puzzles," *Journal of Information Processing*, vol. 25, pp. 724–729, 2017.

[9] G. Kendall, A. Parkes, and K. Spoerer, "A survey of np-complete puzzles," *ICGA Journal*, vol. 31, no. 1, pp. 13–34, 2008.

[10] J. Schaeffer and H. J. Van den Herik, "Games, computers, and artificial intelligence," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 1–7, 2002.

[11] C. Iwamoto and Y. Matsui, "Computational complexity of building puzzles," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 99, no. 6, pp. 1145–1148, 2016.

[12] A. Itai, C. H. Papadimitriou, and J. L. Szwarcfiter, "Hamilton paths in grid graphs," *SIAM Journal on Computing*, vol. 11, no. 4, pp. 676–686, 1982.

[13] C. J. Colbourn, "The complexity of completing partial latin squares," *Discrete Applied Mathematics*, vol. 8, no. 1, p. 25–30, 1984.

[14] T. Easton and R. G. Parker, "On completing latin squares," *Discrete Applied Mathematics*, vol. 113, no. 2, pp. 167 – 181, 2001.

[15] A. Blass and Y. Gurevich, "On the unique satisfiability problem," *Information and Control*, vol. 55, no. 1-3, pp. 80–88, 1982.