Parks Puzzle is NP-Complete

K . Aditya Karan
akamir16@earlham.edu
Earlham College
Richmond, Indiana

ABSTRACT

Parks Puzzle is a popular puzzle game that is played on a square grid. A Parks Puzzle consists of an $n \times n$ grid with n contiguous regions known as parks. The aim of the puzzle is to place trees within parks such that every row, column, and park contains one tree, and no two trees are on squares that border one another. In this paper, we prove that deciding the solvability of a Parks Puzzle is NP-Complete.

KEYWORDS

NP-completeness, Parks Puzzle, puzzle games

1 INTRODUCTION

The P versus NP problem is a major open problem in computer science, and one the Millennium Problems stated by the Clay Mathematical Institute in 2000 with a $1,000,000 prize for a solution. Despite decades of research into computational complexity, the current state of the theoretical understanding of complexity leaves much to be wanted, and the P versus NP problem potentially holds the key to further our understanding of this sphere of theoretical computer science. For a detailed exposition on the problem itself, as well as its place in computer science research, see [4].

The problem of determining the complexity class of some problem is an important aspect of the study of complexity classes, of which P and NP have taken center stage due to the practical outcomes associated with their study. The study of the P and NP complexity classes is a very mature field with vast swathes of published literature regarding the complexity classes and problems contained within them. As a subset of this field, the study of puzzles in NP is also quite mature with several decades worth of work related to identifying NP-complete puzzles, developing strategies for proving reductions, and advancing algorithmic techniques to create practical solvers for these puzzles. However, the Parks Puzzle itself is a largely unknown problem without any corresponding, or closely related academic literature.

A parks puzzle consists of an $n \times n$ grid with n contiguous regions known as parks, each marked with a different colour on the grid. Each square may be marked by a tree, represented by T, or an X, which is used to indicate that a square does not contain a tree, or may simply be left empty. A solution to a parks puzzle is a configuration such that,

- Each row contains a tree
- Each column contains a tree
- Each park contains a tree
- No two trees are on squares that border one-another (even diagonally).

Note that any park that consists of a single square must contain a tree. Also, due to the arrangement of puzzle, it is possible that some squares cannot contain a park in any situation (such as any square sharing a row or column with a unary park). In the examples we will see, we will mark out these basic moves so that the relationship between various gadgets, and parks within those gadgets become more evident.

The general *Parks Puzzle* decision problem is to determine, given a board with some marked trees, if there is some configuration of the board that is a valid solution to the puzzle. In our paper we show that the subset of the decision problem without any marked trees, *Parks*, is NP-Complete, proving a lower bound for the general decision problem. Given a polynomial time oracle for the decision problem, the function problem of finding an explicit solution to a Parks Puzzle would be trivially solvable in polynomial time by checking every square, each in polynomial time. In this paper we prove that Parks is NP-Complete, which suggests that it is very unlikely that there exist any efficient algorithms for the decision as well as function problems.

2 RELATED WORK

At this point in time there is no academic literature that is directly concerned with the Parks Puzzle. Thus there have neither been any published attempts to determine the complexity class of Parks, nor have there been any formal algorithmic approaches to Parks. The works most closely related to the proposed problem of showing that Parks is NP-complete fall into the general category of computer science articles that prove that some problem is NP-complete, and more specifically, the category of research concerned with proving that various puzzles are NP-complete. A large number of common puzzles such as Minesweeper and Sudoku, as well as popular pencil puzzles such as Katakuri and Yosenabe, among several others have already been shown to be NP-Complete [10] [11] [12] .

3 BACKGROUND

Complexity Theory abstracts away the specific demands of particular models of computation operating in their particular contexts, to instead study abstracted models of computation, and the behaviour of algorithms with respect to these abstract models. For the sake of this discussion it is sufficient to understand that we often seek

Earlham College, Computer Science Senior Capstone,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to determine an upper-bound on the computational time taken by an algorithm, and we express this upper bound in terms the time complexity of an algorithm.

Recall that the Turing Machine is model of computation that consists of an infinitely long tape, a "control", and some set of symbols that constitute a language such that the machine can read and write from the tape. The time complexity of a function, or algorithm, expressed with reference to the Turing Machine, a standard model of computation, is defined as follows,

Definition (**Time Complexity**). *[15] Let M be a Turing machine that halts on all inputs. The time complexity of M is the function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the maximum number of steps that M uses on any input of length n.*

In general, all of the standard models of computation offer polynomial time simulation in each other, which is to say that every standard model can simulate any other model within a polynomial time factor. Thus it is meaningful, and often much more productive, to speak of algorithms and their behavior without reference to any model of computation at all. We simply measure the time complexity of algorithms based on the number of elementary operations, or steps, that are performed.

Since the running time of an algorithm can be a fairly complex expression, asymptotic analysis is used to estimate the running time, especially since we are usually interested in an algorithm's behavior for large inputs. This notion of complexity allows for the classification of problem based on the best known algorithms used to solve them.

The complexity class P and NP arise naturally from this line of reasoning. P consists of all decision problems that have polynomial time algorithms, which is to say that there exists some $O(n^k)$ algorithm that decides the problem. P has been shown to contain several common problems such as recognizing whether a string is a palindrome, and more recently, PRIMES, the decision problem of determining whether a given number is a prime.

The complexity class NP consists of all decision problems that can be verified in polynomial time, which means that given a polynomial size certificate (a solution or proof) to a problem in NP, there exists a polynomial time algorithm to check whether the solution is valid or not. NP includes a number of problems that have a practical application in many problem areas, such as the knapsack problem and the general boolean satisfiability problem. While every problem in P is trivially also in NP, the *P vs NP* problem is the open problem of determining whether every problem in NP is also in P.

There are also problems that are much harder than those in NP, which cannot even be verified in polynomial time, such as the clique optimization problem. The complexity class NP-Hard consists of problems that are at least as hard as every problem in NP, which includes every problem harder than NP, but also, curiously enough, a large number of problems in NP. One of the early advances on the P versus NP problem came in the form of the Cook-Levin theorem, which states that the boolean satisfiability problem is both in NP as well as NP-Hard, that lead to the definition of a class of problems called NP-Compete problems [5]. Since NP-Complete problems are in NP and at least as hard as any other problem in NP, a polynomial time algorithm for any NP-complete problem would imply that every problem in NP is solvable in polynomial time.

Given an arbitrary decision problem $A \in$ NP, it is sufficient to find a polynomial time reduction from any NP-Complete problem B to A, to show that A is NP-Complete. That is, given an NP-Complete problem B, if we were to provide an algorithm that transforms any instance of problem B to an instance of problem A in polynomial time, such that the solutions to both problems coincide exactly, then A must also be NP-Complete. In this paper we first prove that Parks is in NP, and then show that Parks is NP-Complete by describing such a reduction from 3SAT, an NP-Complete variant of the boolean satisfiability problem, to Parks.

4 PARKS IS IN NP

For a given instance of Parks π, let the certificate σ be a list of indices marking the position of each tree, and let π^* be the parks puzzle π with the trees marked according to σ. Then σ is trivially of polynomial size, and the validity of σ can be checked by traversing every row, column, and park in π^* exactly once, each in approximately $n + 8\sqrt{n}$ steps, to ensure that they each contain exactly one tree and that no two trees are on squares that border one-another. Therefore $\langle \pi, \sigma \rangle$ can be verified in polynomial time, and Parks is in NP.

5 NON-CONTIGUOUS PARKS IS NP COMPLETE

We will build up to the main result by proving the result for a version of parks with more relaxed conditions. An **NCPark** puzzle is a Parks puzzle without the requirement that the parks must be contiguous, which means that individual parks may be disconnected, but still behave in the same manner as in a Parks puzzle. It is quite straightforward to verify that NCParks $\in$ NP using the same certificate as the proof for Parks. We will show that NCParks is NP-complete by sketching a polynomial time reduction from 3SAT.

Definition (Consistent). *A puzzle is consistent if the position of every tree on the Park is consistent with some solution to the puzzle*

Our aim is to provide a scheme for representing a boolean expression in 3CNF form as a NCPark Puzzle such that an assignment of variables in the puzzle is consistent IFF the same assignment of variables satisfies the boolean expression. Since a 3SAT expression is a conjunction of ternary disjunctions, we will require a ternary OR gadget, to represent each disjunction, and an IFF gadget that allows variables variables to appear in multiple clauses. Let us first begin by defining a variable park, which is a binary park that we will be reading and writing to, to represent whether a variable is True or False.

Definition (Variable park). *A variable park is a park with two squares, corresponding to the value of the variable, and its negation. In general, as a matter of convention, the value of the variable will refer to the state of the topmost, and leftmost square, such that if the that square contains a tree, the variable is said to be True (in which case it's negation must be False) and vice-versa.*

The variable parks will represent each variable in a 3SAT expression. The relationships between these variables will be determined using gadgets, which are some set of parks in a particular configuration that impose a relationship between some set of variable

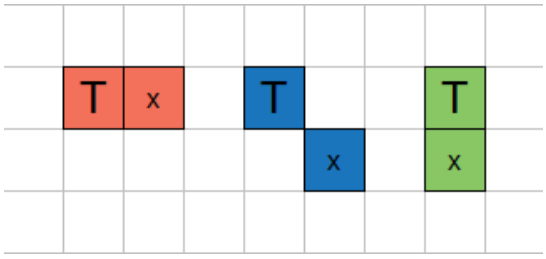
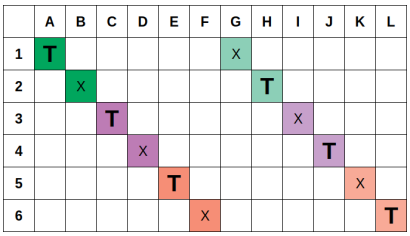


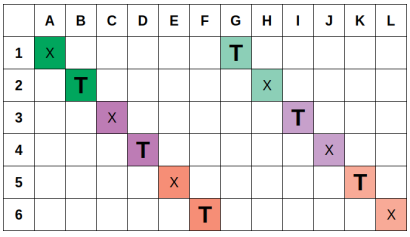
Figure 1: Three variable parks set to True

parks, such that the entire configuration is consistent if and only if the variables share that relationship.

The first gadget we will see is the general IFF gadget shown in figure 2. The gadget consists of two chains of variable parks laid side by side, such that each variable park is inversely related to the park to its side. For example, in 2 the park $\{A1, B2\}$ is the negation of $\{G1, H2\}$. By convention we will only set the values of the top-most, and leftmost park, in this case $\{A1, B2\}$. Note that there are only two valid configurations of this gadget, when the setter park is set to True, and when it is set to False, shown in figure 2 (a) and figure 2 (b) respectively. In general, any time that we would like to use the negation of a variable (in some gadget) we simply use one of the variable parks on the right chain.



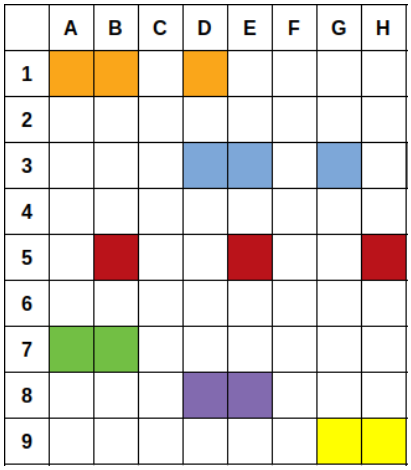
(a) IFF gadget set to True



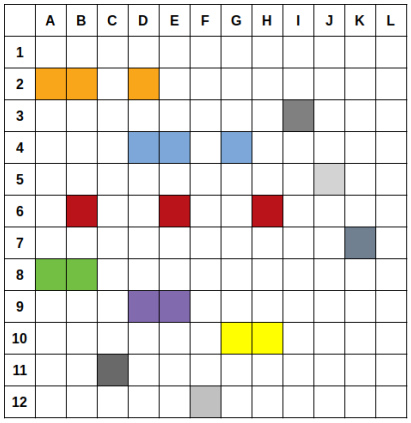
(b) IFF gadget set to False

Figure 2: IFF Gadget for non-contiguous parks

Figure 3 (a) demonstrates the ternary OR gadget which is equivalent to the binary expression $X \vee Y \vee Z$ where the green, purple and yellow squares represent the values of X, Y and Z respectively. It is quite straightforward to verify that there are no solutions when all three variables are set to False due to the placement of the red park, and that every other permutation produces a unique and consistent configuration. Figure 3 (b) demonstrates how the OR gadget may be embedded into a valid parks puzzle. The white park serves as the background park, which is set up so that $L1$ is always True. The other unary parks, which we will color in greyscale by convention, are placed to occupy in-between rows and columns to ensure that we have a valid parks puzzle.



(a) Ternary OR Gadget



(b) Ternary OR gadget embedded in a park

Figure 3: Ternary OR gadget for non-contiguous parks

We are now able to represent any ternary disjunction in the form of a parks puzzle using the gadgets we have developed thus far. The key insight that allows us to represent arbitrary conjunctions of ternary disjunctions is the fact that, given some number of disjunctions embedded in a parks puzzle, every one of those disjunctions must be satisfied in any valid solution to the puzzle. Therefore, as long as we place the OR gadgets in a manner that they do not interfere with one another, we may represent any 3SAT formula as a non-contiguous parks puzzle. In general, this placement may be achieved by reserving 6 rows per OR gadget, and placing them diagonally so that no two OR gadgets share any rows or columns, and do not border one-another.

For example, the expression $(X \vee X \vee X) \wedge (!X \vee !X \vee !X)$ may be represented as a non-contiguous parks puzzle as shown in figure 4. The figure shows the configuration of the puzzle after all of the basic moves have been completed, which is to say that a tree has been placed wherever it is mandatory to place a tree, such as in all the unary parks, and every square that is blocked by one of these trees has been crossed off. One may observe that the park consists of an OR gadget for every disjunction, and a single IFF gadget, since we are using a single variable (and its negation). Every 3SAT expression follows the same pattern, with additional IFF and OR gadgets for each new variable, and additional disjunctive clause.

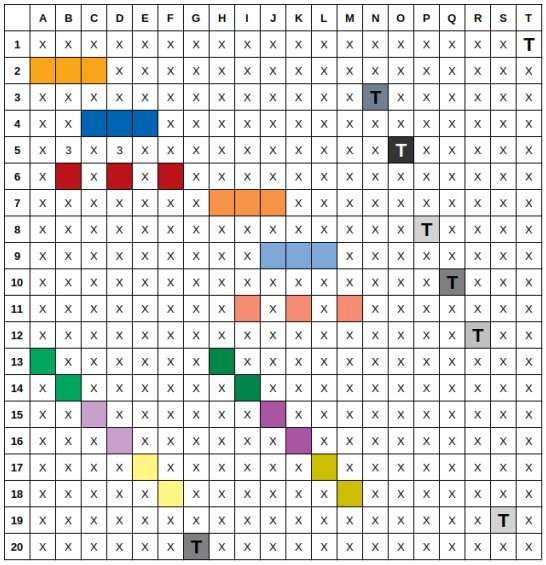


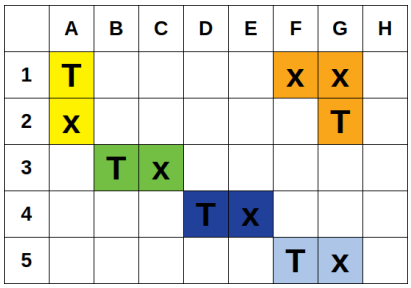
Figure 4: $(X \vee X \vee X) \wedge (!X \vee !X \vee !X)$

6 PARKS IS NP-COMPLETE

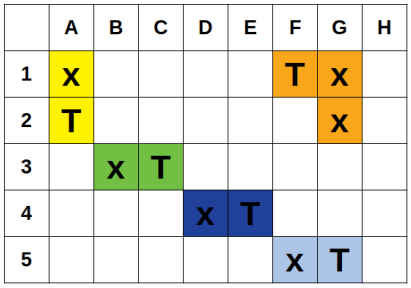
Now that we have seen that the non-contiguous Parks puzzle is NP complete, we will prove that the regular Parks Puzzle is NP-Complete following the same line of reasoning. The key insight to this proof, as with the proof for NCParks, is noting that if we can represent all of the disjunctions independently in a parks puzzle, then any assignment of variables must satisfy every one of these disjunctions for the puzzle to be consistent, and thus there is no need for an explicit AND gadget. Thus, we will develop gadgets for the IFF and OR operators, and show that Parks is NP complete by showing that there exists a polynomial time reduction from 3SAT to Parks. As with NCParks, our aim is to provide a scheme for representing a boolean expression in 3CNF form as a Park Puzzle such that an assignment of variables in the puzzle is consistent IFF the same assignment of variables satisfies the boolean expression.

Figure 5 illustrates a binary (over two variables) version of the N-Ary IFF gadget that will be used to address repeated variables across disjunctive clauses. In this instance, the **green** and **navy blue** parks are variable parks. The state of the **yellow** park, the setter, also sets the state of the green and blue parks (this relation is symmetric, and it doesn't actually matter which park is set first). As with it's NCParks equivalent, the basic IFF gadget has only two possible states, with 5 (a) and 5 (b) corresponding to when the setter park is set to True and False respectively.

This general structure can be extended to accommodate any additional number of variable parks by introducing an additional row and two columns, moving the corner piece ($\{F1, G1, G2\}$) two columns to the right, the light blue park ($\{F5, G5\}$) two columns to the right and one row down, and placing the new park next to the navy blue park ($\{D4, E4\}$ in the same way that the navy blue park is placed next to the green park ($\{B3, C3\}$).



(a) IFF gadget set to True

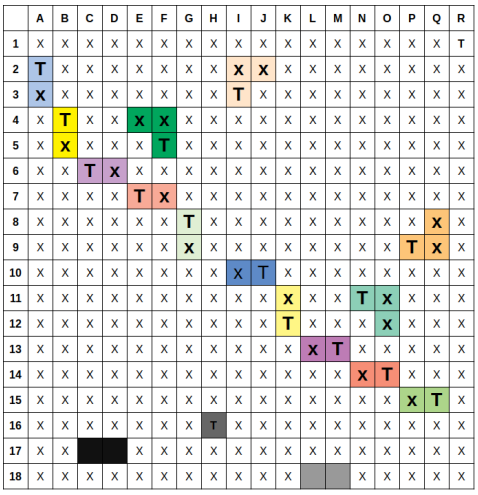


(b) IFF gadget set to False

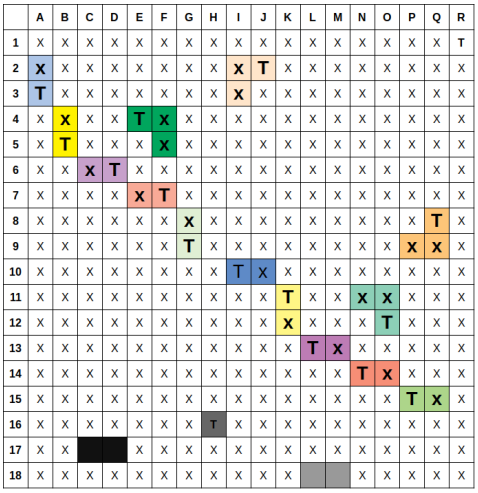
Figure 5: IFF gadget

While it is possible to complete the proof using the OR and IFF gadgets by suitably modifying the OR gadget when the negation of a variable is required, the proof may be simplified by introducing the IFF-NOT gadget, which is analogous to the IFF gadget for NCParks. The IFF-NOT gadget consists of two IFF gadgets connected such that each of the sub-gadgets has its variable parks set to the inverse of the other gadget. The two possible configurations of the IFF-NOT gadget are shown in 6 (a) and 6 (b), corresponding to the setter park being set to True and False respectively. As with the IFF gadget, the IFF-NOT gadget can be extended to include an arbitrary number of variable parks, whereby the number of variable parks in each sub-gadget does not have to be the same.

Figure 7 illustrates the OR gadget, the equivalent of a ternary OR expression with the **purple**, **yellow** and **green** parks as variable parks. As with the corresponding NCParks gadget, the only inconsistent configuration arises when all three variable parks are set to false, and each of the remaining 7 possible variable assignments corresponds to a unique consistent state of gadget.



(a) IFF-NOT gadget set to True



(b) IFF-NOT gadget set to False

Figure 6: IFF-NOT gadget

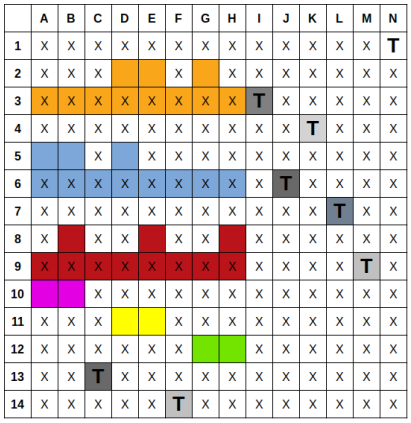
The three gadgets developed so far are sufficient to express any 3SAT expression as a Parks puzzle, and we conclude that Parks is NP-Complete. See appendix A for a detailed proof.

7 EXAMPLES

The following are a couple of simple examples that depict how the gadgets are used in conjunction with one another to represent any statement in 3CNF form. Figure 8 (a) depicts $X \vee X \vee X$, a simple parks instance using one ternary IFF gadget, and one ternary OR gadget. The value of X may be set to true by placing a tree in *B10*, and to false by placing a tree in *B11*. It is quite straightforward to verify, using figure 8 (b), that only one of these placements leads to a solution to the puzzle, corresponding to when X is True.

As the expressions involve more variables and clauses, the equivalent instance of parks also becomes significantly larger. Figure 9

(a) Ternary OR



(b) Basic Moves Completed

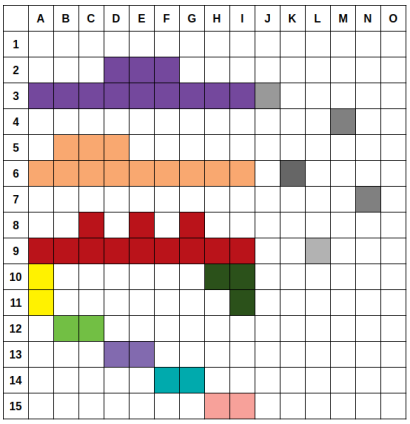
Figure 7: Ternary OR gadget

depicts $X \vee Y \vee Z$, a larger puzzle with three unary IFF gadgets and one OR gadget. The value of X, Y and Z may be set using the corresponding parks $\{B10, B11\}$, $\{H14, H15\}$ and $\{N18, N19\}$. The solutions to this puzzle are identical to the solutions to the ternary OR gadget shown in figure 7.

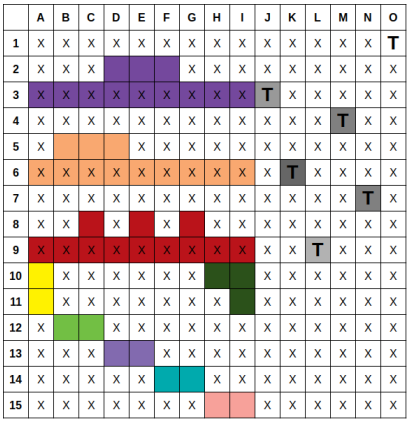
The final example we will see is the reduction for the expression $(X \vee X \vee X) \wedge (!X \vee !X \vee !X)$, shown in figure 10, which uses two ternary OR gadgets and an IFF-NOT gadget. While the resulting park is very large, it is straightforward to verify that there are no solutions to the puzzle, which is consistent with the fact that the original expression is not satisfiable.

8 CONCLUSION

Parks is NP-Complete. We know almost certainly that there are no efficient algorithms to solve parks. In the future, we may explore some algorithmic approaches to NP-complete puzzles using techniques such as genetic algorithms, and heuristic techniques to attempt to solve some subset of the problem efficiently.



(a) $X \vee X \vee X$



(b) Basic Moves Completed

Figure 8: $X \vee X \vee X$

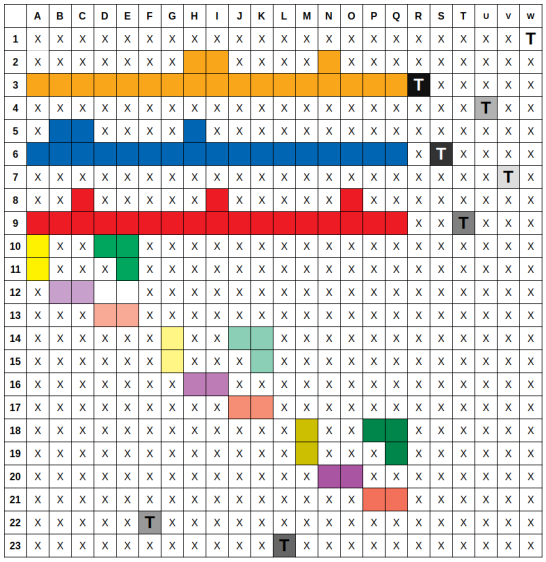


Figure 9: $X \vee Y \vee Z$ **with basic moves completed**

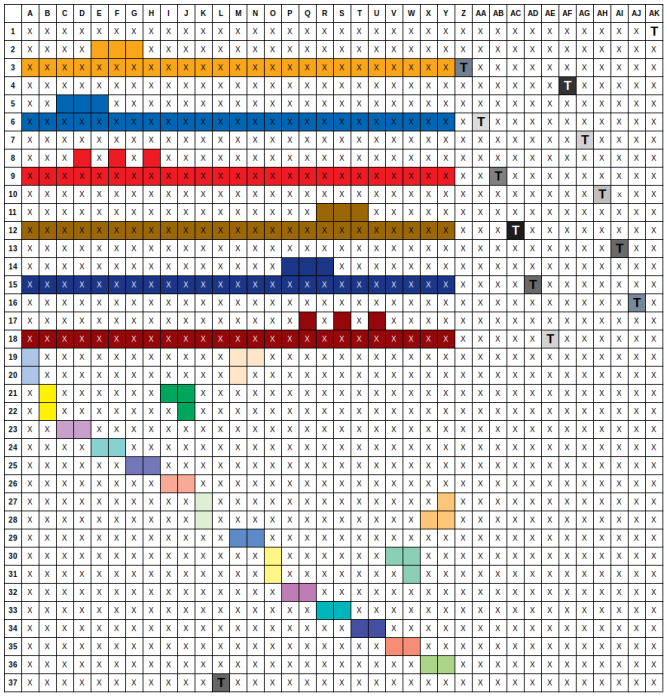


Figure 10: $(X \vee X \vee X) \wedge (!X \vee !X \vee !X)$ **with basic moves completed**

9 ACKNOWLEDGEMENTS

I would like to thank Doctor Igor Minevich for his insightful comments and support, without which I could not have completed the proof. I would also like to thank the Department of Computer Science, and Doctor Charlie Peck and Doctor Xunfei Jiang in particular, for providing valuable feedback while I was working on this project.

REFERENCES

[1] Martyn Amos, Matthew Crossley, and Huw Lloyd. 2019. Solving Nurikabe with Ant Colony Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19)*. ACM, New York, NY, USA, 129–130. https://doi.org/10.1145/3319619.3338470

[2] Amit Benbassat. 2019. Genetic Algorithms Are Very Good Solved Sudoku Generators. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19)*. ACM, New York, NY, USA, 49–50. https://doi.org/10.1145/3319619.3326793

[3] David W. Binkley and Bradly M. Kuhn. 1997. Crozzle: An NP-complete Problem. In *Proceedings of the 1997 ACM Symposium on Applied Computing (SAC '97)*. ACM, New York, NY, USA, 30–34. https://doi.org/10.1145/331697.331705

[4] Stephen Cook. 2003. The Importance of the P Versus NP Question. *J. ACM* 50, 1 (Jan. 2003), 27–29. https://doi.org/10.1145/602382.602398

[5] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC '71)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/800157.805047

[6] Pierluigi Crescenzi, Emma Enström, and Viggo Kann. 2013. From Theory to Practice: NP-completeness for Every CS Student. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*. ACM, New York, NY, USA, 16–21. https://doi.org/10.1145/2462476.2465582

[7] Achiya Elyasaf, Ami Hauptman, and Moshe Sipper. 2011. GA-FreeCell: Evolving Solvers for the Game of FreeCell. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO '11)*. ACM, New York, NY, USA, 1931–1938. https://doi.org/10.1145/2001576.2001836

[8] Lance Fortnow and Steven Homer. 2003. A Short History of Computational Complexity. *Bulletin of the EATCS* 80 (01 2003), 95–133.

[9] Robin Houston, Joseph White, and Martyn Amos. 2012. Zen Puzzle Garden is NP-complete. *Inf. Process. Lett.* 112, 3 (Jan. 2012), 106–108. https://doi.org/10.1016/j.ipl.2011.10.016

[10] Chuzo Iwamoto. 2014. Yosenabe is NP-complete. *Journal of Information Processing* 22, 1 (2014), 40–43. https://doi.org/10.2197/ipsjjip.22.40

[11] Richard Kaye. 2003. Minesweeper is NP-complete. *The Mathematical Intelligencer* 22 (March 2003), 9–15.

[12] Graham Kendall, Andrew Parkes, and Kristian Spoerer. 2008. A Survey of NP-Complete Puzzles. In *ICGA Journal*.

[13] Oliver Ruepp, Markus Holzer, Paolo Boldi, and Luisa Gargano. 2010. The Computational Complexity of the Kakuro Puzzle, Revisited. In *Fun with Algorithms. FUN 2010. Lecture Notes in Computer Science, vol 6099.*

[14] Jorge A. Ruiz-Vanoye, Joaquín Pérez-Ortega, Rodolfo A. Pazos R., Ocotlán Díaz-Parra, Juan Frausto-Solís, Hector J. Fraire Huacuja, Laura Cruz-Reyes, and José A. Martínez F. 2011. Survey of Polynomial Transformations Between NP-complete Problems. *J. Comput. Appl. Math.* 235, 16 (June 2011), 4851–4865. https://doi.org/10.1016/j.cam.2011.02.018

[15] Michael Sipser. 1996. *Introduction to the Theory of Computation* (1st ed.). International Thomson Publishing.

[16] Niklas Sörensson and Niklas Eén. [n.d.]. "Introduction to MiniSat". http://minisat.se/ Accessed: 2010-09-30.

[17] Gerhard J. Woeginger. 2003. Combinatorial Optimization - Eureka, You Shrink! Springer-Verlag New York, Inc., New York, NY, USA, Chapter Exact Algorithms for NP-hard Problems: A Survey, 185–207. http://dl.acm.org/citation.cfm?id=885909.885927

A DETAILED PROOF

Theorem. ***Parks Puzzle is NP-complete***

Proof. We have already seen that Parks is in NP, so it is sufficient to prove that Parks is NP-Hard. Let a 3SAT statement be represented as a list of lists, with each inner list consisting of three elements, and representing a single disjunctive clause. Every 3SAT statement can be reduced to an instance of Parks in the following manner.

(1) Traverse the expression and create a table with each variable, and the number of times it, as well as its negation appear in the expression. Additionally, create a column to hold the number of instances of a variable that have appeared so far, and the starting coordinate of the IFF gadget holding the corresponding variable parks, and initialize them to 0 and (0,0) respectively.
(2) Initialize an $n \times n$ array with

$$n = 9d + \sum_{i=1}^{r}(3 + k_i) + \sum_{j=1}^{s}(12 + k_j + l_j)$$

where,
- d is the number of disjunctive clauses
- r is the number of variables appearing without their negation, with each k_i equal to the number of time each variable appears in the expression
- s is the number of variables appearing with their negation, with each l_j representing the number of times the negation appears in the expression

(3) Reserve $9d$ rows for the OR gadgets and save the coordinates $1, 9d + 1$ which locates the first IFF, or IFF-NOT gadget.
(4) For each variable x without a negation that appears k times, allocate a k-ary IFF gadget. Every successive IFF gadget starts at $(a + 3 + k, b + 3 + 2k)$ where (a, b) is the starting point of the previous gadget. Store the starting coordinate of the gadget in the table.
(5) For each remaining variable y that appears k times, with its negation appearing l times, allocate the corresponding IFF-NOT gadget. The first IFF-NOT gadget starts at $(a + 3 + k, b + 3 + 2k)$ where (a, b) is the starting point of the previous IFF gadget. Every successive IFF-NOT gadget starts at $(a + 12 + k + l, b + 12 + 2k + 2l)$ relative to the previous IFF-NOT gadget. Store the starting coordinate of the gadget in the table.
(6) Note the coordinate $(\bar{x}, \bar{y})$ of the rightmost, and bottom-most square of the final IFF, or IFF-NOT gadget, which determines the width of every OR gadget. Allocate a park of height 1 and width $\bar{x} - 2$, starting from the second column, to ever third row in the space left open for the OR gadgets
(7) For each disjunctive clause, identify the corresponding variable parks by calculating an offset from the start position in the table. Then allocate the 2nd row of the corresponding OR gadget, which consists of the three open squares, over the three variable parks.
(8) Finally, place the surrounding blocks. Starting from $(\bar{x}+1, 3)$, place a unary park every three rows down and one column to the right to block every third row of the OR gadget. Then, starting from $(\bar{x} + 1 + 3(d - 1) + 2, 4)$, follow the same pattern to block the row between each horizontal section of the OR gadgets. If there is only one IFF gadget (one variable without a negation) we are done. Since we have noted down the starting point of every gadget, starting from the $\bar{y}$ row, for every additional variable k without a negation, find the starting point of its corresponding gadget (a, b) and place a unary park at $(\bar{x} + 1, \bar{y} + k)$.
If there is only one IFF-NOT gadget with m and n appearances of the variable and its negation, starting at (a, b), place a unary park at $(a + 5 + 2m, \bar{y})$. For every additional variable corresponding to a IFF-NOT gadget, with m and n appearances of the variable and its negation, and k unary parks prior to it, place a unary park at $(\bar{x} - 1, \bar{y} + k)$ and $(\bar{x} + 5 + 2m, \bar{y} + k)$.

We then have a complete and valid Park Puzzle that corresponds to the initial expression.
Since each step of the above process may be completed in polynomial time with respect to the size of the original expression it is a polynomial time reduction. □