Mathematical Games

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Joshua Hoak

May 2009

Acknowledgements

Thanks to Jamie for being a terrific advisor – I've really enjoyed seeing this thing come together. Thanks also to Jim for his many great comments and insights. Thanks to my parents for being there through these five crazy years. Thanks to Mike and Rachael for being consistently awesome. Gregsan wa also totally awesome desu. Thanks to Jeremy and Dan for the many great Go games and to Aaron for his words of wisdom. Thanks to Slava and John for their awesome comments and advice on this thesis. It really wouldn't be nearly as awesome without their help. Last but not least, thanks to Erica for holding me together.

Table of Contents

List of Figures

Abstract

This thesis investigates the complexity of three puzzles, Peg Solitaire, Grid and Power Grid, by using Constraint Logic, a reduction tool developed by Hearn and Demaine. For that end, the basics of complexity theory and Constraint Logic are developed and then applied to these examples. While the games are not proved to be NP-complete in their natural forms, reduced forms are shown to be NP-complete, which points toward a relatively high level of difficulty in making algorithms for the original puzzles.

Chapter 1

Complexity Theory

How long does it take for a computer to find the answer to a problem? This is a fundamental problem in computer science. To answer this question, this chapter will be devoted to developing the basic ideas of *complexity theory*, which will allow us more ease in addressing the difficulty or hardness of problems. Specifically, we will be working toward generating tools that will apply to games. Unfortunately, a complete discussion of complexity theory is beyond the scope of this thesis, so a general knowledge of Turing machines and nondeterminism is assumed. The material that follows is primarily taken from Sipser, with additional information from Papadimitriou and Rich [16, 12, 14].

1.1 Basic Definitions and Theorems

Definitions: An **alphabet** Σ, is a set of symbols; Σ^* is the set of all combinations or **strings** of symbols in Σ. A **language** $\mathcal{A}$ is a set of strings; i.e., $\mathcal{A} \subseteq \Sigma^*$. We say that M **decides** the language $\mathcal{A}$ if there is a Turing machine M that accepts an input $w \in \Sigma^*$ whenever $w \in \mathcal{A}$ and rejects whenever $w \notin \mathcal{A}$.

Whether a language can be decided is the subject of **computability theory**, whereas determining how hard it is to decide a language is the subject of **complexity theory** – the subject of this thesis.

A fundamental idea in computer science is that we can encode computational problems as a problem of deciding some language with a Turing machine. Assuming that it is possible (and sometimes it is not) to decide a language, we can ask how long a machine takes to run or how much space it uses to compute the answer. Analyzing either of these two characteristics, *time* and *space*, is how we determine complexity.

First, we need to define the unit of time - the step. Since the single-tape Turing machine is one of the simplest and most well known models for computation, it shall be used as the model in this discussion. There are many varied and curious models, but all are more or less equivalent by the Church-Turing thesis [16]. For a given input size n, we think of a step as the movement of the read-write head from one cell to another. To measure time complexity, we will associate a function f that corresponds to time taken by a machine M.

Definition: Let M be a deterministic Turing machine that accepts or rejects on all inputs.

The **running time** or **time complexity** is the function $f : \mathbb{N} \rightarrow \mathbb{N}$, where $f(n)$ is the maximum number of steps that M takes to halt on a given input of size n.

As a hypothetical example, let $\mathcal{A}$ be a language, and let M be a Turing machine that decides whether a string w is in $\mathcal{A}$. By analyzing the machine, we might be able to determine that for w with length n the algorithm will at most have a running time of

$$f(n) = 2n^3 + 3n^2 + 5, \tag{1.1}$$

which is to say that for a given input n, the machine will always halt in fewer than $2n^3 + 3n^2 + 5$ steps.

However, because our particular choice of computing model can change the exact running time for a machine deciding a particular problem, we are generally only interested in estimating the time complexity of an algorithm. To do so, we usually consider how the time complexity $f(n)$ behaves for large values of n. In our particular example, for large values of n, $f(n)$ is asymptotically equivalent to n^3, which we notate with $f(n) = O(n^3)$.

Definition: Let f and g be functions $f, g : \mathbb{N} \longrightarrow \mathbb{N}^+$. We write $f(n) = O(g(n))$ and call $g(n)$ an upper bound for $f(n)$ if there are $c, k \in \mathbb{N}$ such that for all $n \geq k$,

$$f(n) \leq cg(n)$$

Since the g in the definition is an upper bound (or an **asymptotic upper bound**) on f, we can use g to help generalize our discussion of the time complexity of a problem. The definition conveniently allows us to disregard constant coefficients. As an example, if $f(n) = 2n^3 + 3n^2 + 5$, we can let c = 3 and $g(n) = n^3$, and then $f(n) \leq cg(n)$. However, notice that $f(n) = O(n^4)$ as well, but practically, we are interested in the best upper bound on $f(n)$ as a means or relating two different algorithms.

Definition: Let $t : \mathbb{N} \longrightarrow \mathbb{N}$. The **time complexity class**, TIME$(t(n))$, is then,

$$\text{TIME}(t(n)) = \{\mathcal{L} | \mathcal{L} \text{ is a language decided by an } O(t(n)) \text{ time-bounded } \textit{deterministic} \text{ Turing machine}\}.$$

Definition: Similarly, Let $t : \mathbb{N} \longrightarrow \mathbb{N}$. The **nondeterministic time complexity class**, NTIME$(t(n))$, is then,

$$\text{NTIME}(t(n)) = \{\mathcal{L} | \mathcal{L} \text{ is a language decided by an } O(t(n)) \text{ time-bounded } \textit{nondeterministic} \text{ Turing machine}\}.$$

If machine M decides language $\mathcal{A}$ in $f(n) = 3n^2 + 5$ steps, and machine N decides language $\mathcal{B}$ in $g(n) = n^2 + n$ steps then both $\mathcal{A}$, $\mathcal{B} \in$ TIME(n^2). Of course, it is also true that $\mathcal{A}$, $\mathcal{B} \in$ TIME(n^3), since TIME$(n^2) \subset$ TIME(n^3), which is illustrated in Figure 1.1. Naturally[1], we will generalize these concepts even further.

[1]since this is mathematics

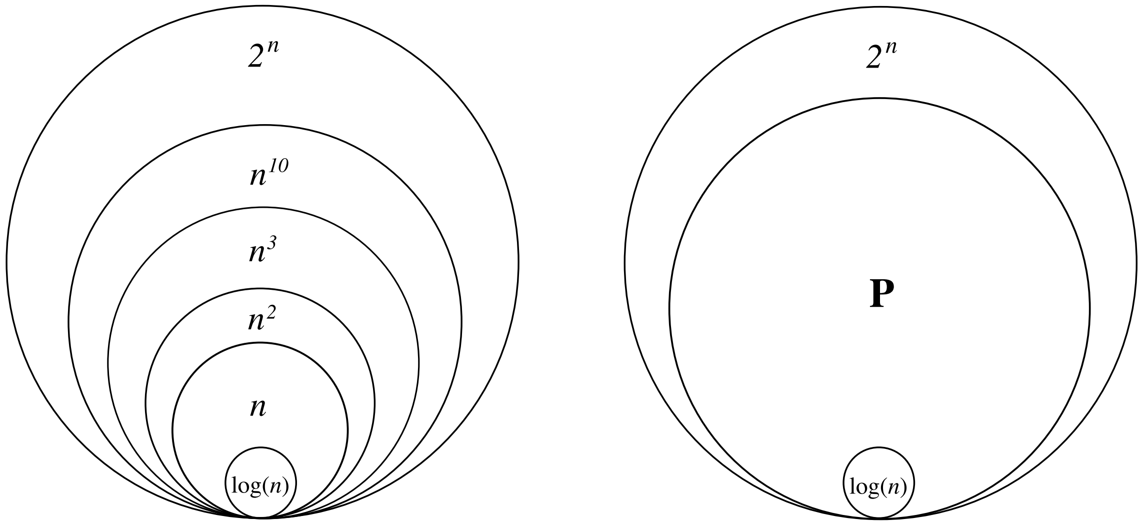


Figure 1.1: A diagram of the relationship between different complexity classes TIME$(t(n))$, for various $t(n)$.

Definitions: We define the set P as,

$$\text{P} = \bigcup_k \text{TIME}(n^k);$$

similarly, we define the set NP as,

$$\text{NP} = \bigcup_k \text{NTIME}(n^k);$$

and finally, we define the set EXPTIME as,

$$\text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k}).$$

It is generally assumed that nondeterministic machines are significantly more powerful than deterministic machines, although computer scientists are generally unsure by how much. Thus P $\subseteq$ NP, although it may be the case that P = NP – a fundamental unresolved problem in computer science. At the present, the best known methods for solving certain problems in NP use exponential time, so we can construct the following *complexity hierarchy*: P $\subseteq$ NP $\subseteq$ EXPTIME.[16, 12]

Now that we have described time-complexity, we shall describe *space-complexity* for a single-tape Turing machine.

Definition: Let M be a deterministic Turing machine that either accepts or rejects on all inputs. The **space complexity** is the the function $f : \mathbb{N} \longrightarrow \mathbb{N}$, where $f(n)$ is the maximum number of cells that M *scans* on input of length n.

We can think of scanning as reading the contents of a cell. As an example, if we have an algorithm that reads each cell in a string w of length n and then halts, then the space complexity is $f(n) = n$. If the algorithm scans each cell twice and then marks each cell, we still have $f(n) = n$. As one would expect, we can then define a space complexity class like the time complexity class.

Definition: Let $f : \mathbb{N} \longrightarrow \mathbb{N}$. The **space complexity class** SPACE(f(n)) is defined to be,

$$\text{SPACE}(f(n)) = \{\mathcal{L} | \mathcal{L} \text{ is a language decided by an } O(f(n)) \text{ space Turing machine}\},$$

with $O(f(n))$ space defined as one would expect.

We can similarly define the nondeterministic space complexity class NSPACE, except that instead it is the set of languages decided by a $O(f(n))$ bounded nondeterministic Turing machine. Similar to our earlier definitions for time-complexity, we arrive at the useful generalizations PSPACE and NPSPACE.

Definitions: We define PSPACE to be,

$$PSPACE = \bigcup_k \text{SPACE}(n^k),$$

and we define NPSPACE to be,

$$NPSPACE = \bigcup_k \text{NSPACE}(n^k).$$

We would like to place PSPACE somewhere within the complexity hierarchy created above.

Lemma 1.1.1. *P $\subseteq$ PSPACE, NP $\subseteq$ PSPACE.*

Proof. First, we show that P $\subseteq$ PSPACE. Let $t(n)$ be the time-complexity for machine M. Given that $t(n) \geq n$, any machine that halts in $t(n)$ time can use at most $t(n)$ space. At each step the machine can explore a maximum of one new cell, so a $t(n)$ bounded Turing machine can explore at most $t(n)$ new cells. Thus P $\subseteq$ PSPACE. Similarly, NP $\subseteq$ NPSPACE.

Later, Savich's Theorem will show us that NSPACE($f(n)$) $\subseteq$ SPACE($f^2(n)$), which will imply that NPSPACE = PSPACE and so NP $\subseteq$ PSPACE. □

Thus, we can modify the complexity hierarchy to be P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME. This latter containment, that PSPACE $\subseteq$ EXPTIME, I leave unproved, citing Sipser [16]. Although it could be that P = NP or that NP = PSPACE, most believe that these containments are proper [16, 12]. Moreover, these classes provide useful generalizations about the difficult of problems. If a problem lies in P – meaning that there exists a polynomial-time algorithm for deciding the problem, then we consider it to be much easier (to compute) than if a problem lies within NP or PSPACE.

1.2 Mapping Reductions and Completeness

To aid in the task of determining the complexity of an algorithm, it would be convenient (if possible) to reduce it to a similar and already solved problem. For that end, we introduce the ideas of computable functions and mapping reductions.

Definitions: A function $f : \Sigma^* \longrightarrow \Sigma'^*$ is **computable** if for all inputs w there exists a Turing machine that always halts with $f(w)$ on its tape.

If there is a computable function $f : \Sigma^* \longrightarrow \Sigma^*$, such that

$$w \in \mathcal{A} \Longleftrightarrow f(w) \in \mathcal{B},$$

then we say $\mathcal{A}$ is **mapping reducible** to $\mathcal{B}$ and f is called the **reduction** from A to B. We write $\mathcal{A} \leq_m \mathcal{B}$.

This definition gives us a tool to relate different languages. Moreover, mapping reductions will allow us more adroit use the generalizations of P, NP, PSPACE and EXPTIME. Let us make a refined definition of mapping reduction.

Definition: Language $\mathcal{A}$ is **polynomial time mapping reducible** to $\mathcal{B}$ if there exists a polynomial time computable function f such that,

$$w \in \mathcal{A} \ \Longleftrightarrow f(w) \in \mathcal{B},$$

notated $\mathcal{A} \leq_p \mathcal{B}$.

Now, using this definition we will show that the following theorem shows why this restriction is so useful, for in some sense, P is closed under $\leq_p$.

Theorem 1.2.1. *If $\mathcal{B}$ is in the complexity class P and $\mathcal{A} \leq_p \mathcal{B}$, then $\mathcal{A} \in$ P.*

Proof. Let M be a polynomial time Turing machine that decides $\mathcal{B}$. Let f be a polynomial time computable function from $\mathcal{A}$ to $\mathcal{B}$. Let N be the polynomial time Turing machine that decides $\mathcal{A}$. We construct N informally as follows: On input w, compute $f(w)$ and then run M on input $f(w)$ and let output$_N$ = output$_M$. Computing $f(w)$ will take polynomial time on the size of w and running M on $f(w)$ will also take polynomial time on the size of $f(w)$, so N will also take polynomial time, since polynomials are closed under composition. $\square$

Also, since P is contained within NP, PSPACE, and EXPTIME, analogous theorems are also true for these complexity classes. Now, using the these theorems, we create definitions for the hardest problems in a particular complexity class. These definitions are essential for the discussions that occur later.

Definitions: A language $\mathcal{B}$ is **NP-complete** if it satisfies the following:

1. $\mathcal{B}$ is in NP,
2. every $\mathcal{A}$ in NP is polynomial time reducible to $\mathcal{B}$.

A language $\mathcal{B}$ is **PSPACE-complete** if it satisfies the following:

1. $\mathcal{B}$ is in PSPACE,
2. every $\mathcal{A}$ in PSPACE is polynomial time reducible to $\mathcal{B}$.

If a language $\mathcal{B}$ satisfies (2) but not (1), we say that $\mathcal{B}$ is NP/PSPACE-hard.

We can create definition of P-completeness, but we cannot use polynomial-time reductions since this would give us the entire complexity class of P. Instead, we need a log space reduction. P-completeness is presented here, only to show that such a definition is possible. Also, one could design

Definition: A language $\mathcal{B}$ is **P-complete** if it satisfies the following:

1. $\mathcal{B}$ is in P
2. every $\mathcal{A}$ in P is log space reducible to $\mathcal{B}$.

1.3 Fundamental Complexity Theorems

The following theorems are central to any proofs involving NP or PSPACE completeness. Later, we want to show that particular games are hard by showing that they are NP or PSPACE-complete. However, before we can use mapping reductions we need problems in NP and PSPACE! For that end we define the problem SAT:

Decision Problem: The Satisfiable Problem (SAT)
Instance: ψ, a Boolean formula[2].
Question: Is ψ satisfiable[3]?

Cook-Levin Theorem. *SAT is NP-complete.*

Proof. Building an algorithm to show SAT is in NP is easy. We can test any variable assignment nondeterministically in polynomial time so SAT is in NP.

Now, we need to show that any language A in NP reduces to SAT. Since A is in NP, there is a non-deterministic Turing machine M that decides A. We will construct a function ϕ that produces a boolean formula which simulates M. Then, asking whether M accepts an input w is the same as asking whether ϕ is satisfiable.

First we define a tableau for M on w, which is an $n^k \times n^k$ table in which each row represents a valid configuration, and each tableau represents one branch of a computation. Thus, it is an $n^k \times n^k$ table since this is the time complexity of M. So, certainly, when we run M on w, it is quite likely that numerous tableaus will be created in the process (as there will be numerous branches), but all will be created (or run) in parallel. Thus, asking

[2]For example, $\phi = (x \vee y) \wedge \overline{z} \vee x$.

[3]A formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1 (or true)

#	q_0	w_1	w_2	...	w_n	⊔	...	⊔	#	start configuration
#									#	second configuration
#									#	
#									#	n^kth configuration

Table 1.1: An $n^k \times n^k$ tableau. Image reproduced from Sipser [16].

whether M accepts w is equivalent to asking whether there is an *accepting tableau* – that is, a tableau with an accepting configuration. An illustration of such a tableau is given in Table 1.1.

I use the notation of Sipser. Let Q and Γ be the state set and the tape alphabet of M, and let let $C = Q \cup \Gamma \cup \{\#\}$. Let i and j be row and column indices respectively between 1 and n^k, and let $s \in C$. Each element of of the $n^k \times n^k$ tableau is a cell specifically identified as $cell[i,j]$, which contains a symbol s from C If variable $x_{i,j,s} = 1$, then $cell[i,j]$ contains an s.

Let us now construct ϕ. The function ϕ is the logical AND of four boolean parts: $\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$. Each part simulates one aspect of the running machine M. Presented here are the four parts:

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} [(\bigvee_{s \in C} x_{i,j,s}) \wedge (\bigwedge_{s,t \in C, s \neq t} \neg(x_{i,j,s} \wedge x_{i,j,t})].$$

In ϕ_{cell}, we make sure that each cell only has one variable. Parsing the symbols, ϕ_{cell} says that for element in the table, there must be at least one symbol assigned to it ($\bigvee_{s \in C} x_{i,j,s}$) and there can be no two symbols assigned to the same variable. However, we have yet to assign any values for the variables.

$$\begin{aligned} \phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{2,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}. \end{aligned}$$

As can be seen, ϕ_{start} constructs the first row of the configuration.

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}.$$

The component ϕ_{accept} ensures that for ϕ to be true, at least one configuration must contain a cell that accepts (that is, with the accept symbol).

$$\phi_{\text{move}} = \bigwedge_{1 \leq i \leq n^k, 1 \leq j \leq n^k} (\text{the } (i,j) \text{ window is legal}).$$

Lastly, ϕ_{move} ensures that the head of our Turing machine (that is q_0), moves in a fashion as we would wish it. Some details have been left out, but the presentation should be enough to convince the reader that SAT is NP-complete. □

Now we need an example of a problem in PSPACE, but first, to show that we need only talk about PSPACE, and not PSPACE and NSPACE separately.

Savitch's Theorem. *For any function $f : \mathbb{N} \longrightarrow \mathbb{N}$, where $f(n) \geq n$*

$$\textit{NSPACE}(f(n)) \subseteq \textit{SPACE}(f^2(n)),$$

Proof. Recall that SPACE and NSPACE refer to the deterministic and nondeterministic space complexity classes. We wish to simulate the running of a machine for deciding a language in NSPACE with a machine deciding a language in SPACE. The outline of this argument runs as follows: First we create an algorithm using a nondeterministic machine N, then we modify N so that a deterministic machine M simulates it, and finally, we analyze the algorithm to show that M runs in $O(f^2(n))$ space.

For that end we construct the algorithm CANYIELD(c_1, c_2, t), which takes configurations c_1, c_2, and t and outputs *accept* if c_1 results in c_2 in t steps (and *reject* otherwise).

```
CANYIELD($c_1, c_2, t$) = "On inputs $c_1, c_2$ and $t$
1. If $t = 1$: Test whether $c_1 = c_2$ or whether $c_1$ results in $c_2$ (with nondeterministic machine
      $N$) in one step. Accept if either are yes, and Reject otherwise.
2. If $t > 1$: Then, for each $c_m$ of $N$ on $w$ using space $f(n)$
3.     Recursively run CANYIELD($c_1, c_m, \lceil \frac{t}{2} \rceil$).
4.     Recursively run CANYIELD($c_m, c_2, \lceil \frac{t}{2} \rceil$).
5.     If steps 3 and 4 accept then accept for step 2.
6. Otherwise, reject."
```

Now, we create a machine M that simulates N. First, call the start configuration $c_{\text{start}} = N$ running on w. Let t' be the constant $2^{df(n)}$. We choose this t' since we know that for given space $f(n)$, we can have at most $k^{f(n)}$ configurations, for some k (depending on alphabet size), and of course, $k^{f(n)} = 2^{df(n)}$ for some d. Thus t' is an upper bound on the running time for N on w. Let c_{accept} be the accepting configuration of N. Then, we define M to be the machine that on input $w = c_{\text{start}}$, runs CANYIELD($c_{\text{start}}, c_{\text{accept}}, 2^{df(n)}$).

Upon each recursion, CANYIELD must store c_1, c_2 and t, which means that it will require $2 * f(n) + c$ space, which, using asymptotic notation, is just $O(f(n))$. Also, the depth of the recursion is $O(f(n))$ since $t = 2^{df(n)}$, and each level divides the time by two. Thus, the depth is $O(\log 2^d f(n)) = O(f(n))$, and so the total space used is $O(f^2(n))$. □

Now we are ready to define an example and prove that the problem is in PSPACE (or equivalently NPSPACE by Savitch's theorem).

Decision Problem: The Quantified Boolean Formula Satisfiable Problem (QBF)
Instance: ψ, a quantified Boolean formula[4].
Question: Is ψ satisfiable?

Theorem 1.3.1. *QBF is PSPACE complete.*

Proof. Using a method similar to that used in showing SAT is NP-complete, we are going to create a Boolean formula ϕ which is true if and only if Turing machine M running on an input w halts using polynomial space. Thus formula ϕ can be thought of as simulating M. However, our method must be somewhat different. Whereas our tableau in the proof of SAT was an $n^k \times n^k$ table, PSPACE machines can run in exponential time, which means that a tableau for M could have an exponential number of rows. A polynomial time reduction using the method in the proof of SAT cannot produce an exponential size result, so this fails to show $A \leq_p$ QBF.

First we need to show that the problem of satisfying a quantified boolean formula is in PSPACE. Let ϕ be a fully quantified boolean formula. Let T_ϕ be the following algorithm containing 3 parts:

1. If ϕ contains no quantifiers, then the ϕ contains only boolean operators and constants $\in \{0, 1\}$, so evaluate ϕ. If ϕ evaluates to true, then accept ϕ. If ϕ evaluates to false, then reject.

2. If $\phi = \exists x\ \psi$ then recursively take T on ψ (that is T_ψ), with 0 substituted for x and then take T with 1 substituted for x. If either result is accepted, then accept for ϕ. Otherwise, reject.

3. Similarly, If $\phi = \forall x\ \psi$ then recursively take T on ψ with 0 substituted for x and then with 1 substituted for x. If both are accepted, then accept ψ. Otherwise, reject.

We will only ever need to store n quantified variables, so the complexity will be $O(n)$, or linear space. Thus TQBF satisfiability is in PSPACE.

To show that ϕ is PSPACE-hard, we need to give a polynomial-time reduction from M to QBF. As mentioned, in the worst case scenario for M running on w, the tableau will be an $n^k \times 2^{df(n)}$ table. The goal is to create an algorithm of moving between rows that requires only a polynomial amount of space. Let c_1, c_2 be two different (not necessary subsequent) configurations in the tableau. For a number $t > 0$, let $\phi_{c_1,c_2,t}$ be a boolean formula that is true if M can go from c_1 to c_2 in at most t steps. Ultimately, we will want ϕ to be $\phi_{c_{\text{start}},c_{\text{accept}},h}$, where $h = 2^{df(n)}$, for some constant d.

As in the proof of the Cook-Levin theorem, Let the formula ϕ encode the cells of the tableau and transition function. For ϕ, we consider two cases: for $t = 1$ and for $t > 1$. If $t = 1$, then we have $\phi_{c_1,c_2,t}$, which means that the on the tableau for M, we travel between configurations c_1 and c_2 in one step.

[4]Boolean formula with quantifiers $\forall, \exists$. For example, $\phi = \forall x \exists y[(x \vee y) \wedge \overline{z} \wedge (x \vee z)]$.

For $t > 1$, we construct the function

$$\phi_{c_1,c_2,t} = \exists m_1 \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\}[\phi_{c_3,c_4,\lceil \frac{t}{2} \rceil}]\}$$

Let us deconstruct this formula. We can write $\forall(c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\}[\phi \ldots]$ by first order logic since this can be equivalently rewritten $\forall x[(x = y \vee x = z) \rightarrow \phi \ldots]$; $(\rightarrow)$ and $(=)$ also can be rewritten in first order logic. [5] So while the reader may be convinced that this is a valid quantified boolean formula, we still need to argue its purpose. The formula is saying that for either we can take a pair of configurations (c_3, c_4) to be either (c_1, m_1) or (m_1, c_2). Then, we recursively call the formula on both of these possibilities. Now we need to show that our formula uses polynomial space. For the formula $\phi_{c_{\text{start}},c_{\text{accept}},h}$, recall that h refers to the total possible of configurations or $2^{df(n)}$. We need to store the size of each configuration, so we need size $O(f(n))$ and because of the recursion, the number of levels of recursion will be $log(2^{df(n)} = O(f(n))$. Thus the total space required is $O(f^2(n))$. $\square$

[5] $P \rightarrow Q \Leftrightarrow \neg P \vee Q$

Chapter 2

Constraint Logic

Mapping reductions are essential for determining the complexity of a problem. Once we know one problem is in a particular hardness class, e.g. QBF or SAT, we can use the problem as a tool for showing other problems are similarly hard. Then to show completeness, we only need to show that it lies in a particular complexity class. **Constraint Logic** is a set of rules on a directed graph, and is developed by Hearn and Demaine [5, 7, 6], who in turn are generalizing work done by Flake and Baum on the logic of the puzzle Rush-Hour [3]. The subject matter in this chapter comes directly from Hearn and Demaine. Certain formulations of Constraint Logic are better suited for different types of games, and since puzzles (1-player games) will be the main focus, parts of Constraint Logic will be omitted from this thesis.

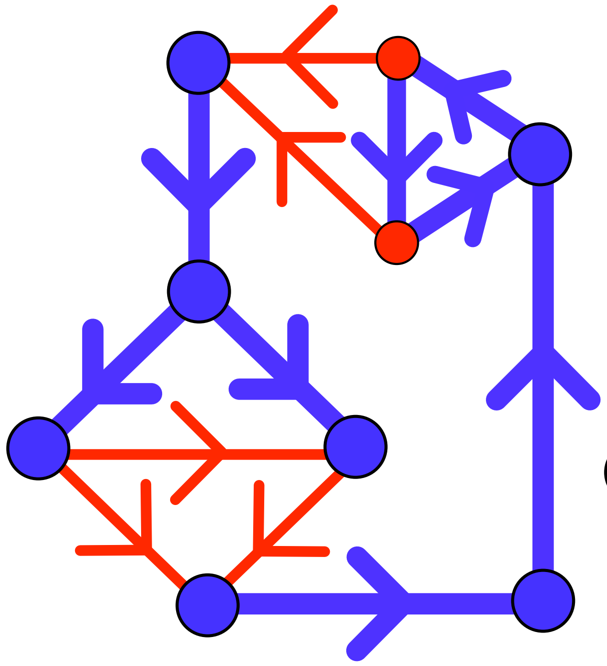
Games are often represented as graphs – some of the most famous being Geography and the Shannon-switching game (see [16] and [12] for examples); many games are also planar, such as checkers, chess, and go. Conveniently, Constraint Logic is a set of rules on a graph, and later we will show that we can also make equivalent planar graphs. These characteristics will allow us to use Constraint Logic as a reduction tool for games.

2.1 An Introduction to Constraint Logic

Definitions: A **Constraint Graph** is a directed graph with edge weights either 1 or 2, referred to as red or blue, respectively. The **inflow** at each vertex is the sum of the edge weights directed toward the vertex. Each vertex has a minimum inflow of either 1 or 2, which are called the **constraints**.

In a **legal Constraint Graph** in Constraint Logic, the inflow into a vertex must meet or exceed a vertex's constraints (minimum inflow), in which case we say the constraint is met. Also, every edge must be adjoined to two different vertices, i.e., there cannot be any dangling edges. This additional rule also prevents loops. In some graphs, it may be possible to reverse the direction of an edge and still meet the constraints. Such a reversal is called a **move**. From here on, it will be assumed that all Constraint Graphs are legal, except where specified.

Since we are working with graphs, diagrams will be a crucial part of the following discussion, so some note should be made about them. Edges with weight 2 will be colored blue and are generally thicker; those with weight 1 will be colored red and are thinner. Vertices are similarly colored according to their constraint. Arrows will indicate the direction of the edges. Often the directions of the edges will be omitted in an illustration of a Constraint Graph, with the expectation that the reader can determine the possible directions. Figure 2.1 gives an illustration of a legal Constraint Graph and its generalization.


(a) A legal Constraint Graph.

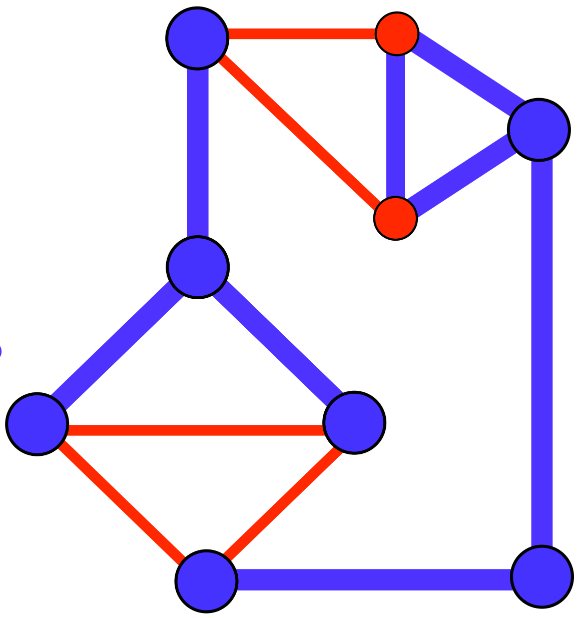
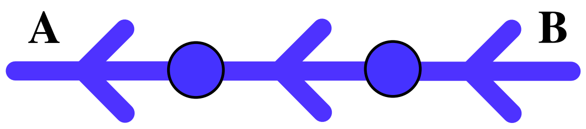
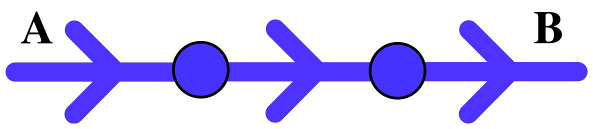

(b) An abstration.

Figure 2.1: Some examples.

2.2 Decision Problems



(a) Before sending a signal.



(b) After sending a signal.

Figure 2.2: An example of a wire in Constraint Logic.

Constraint Logic naturally lends itself to a circuit-like description. If we have a series of constraint 2 vertices each with two edges of weight 2, then we can create a wire gadget that can send information, as in Figure 2.2. If edge A is reversed, then the adjacent edge can reverse, which then allows edge B to reverse. In some sense, a signal has been transfered from edge A to edge B. Later, when we talking about sending a **signal**, this is the sense in which we mean it.

Definitions: An **AND-vertex** has a constraint of 2 and three connecting edges with weights 1, 1, and 2. An **OR-vertex** also has a constraint of 2, with three connecting edges of weights 2. An **AND/OR constraint graph** is a constraint graph with only AND or OR-vertices (Figure 2.3)

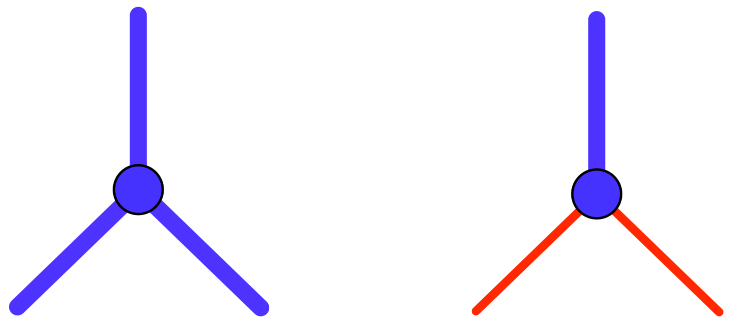

(a) An OR vertex. (b) An AND vertex.

Figure 2.3: The AND and OR vertices.

By these definitions, the Constraint Graph in Figure 2.1 is not an AND/OR Constraint Graph, and neither is the wire in Figure 2.2. However, later it will be clear that we can create wires just from OR and AND vertices. The AND-vertex acts like a logical AND in that the edge of weight 2 may only be directed outward (away from the vertex) if both edges of weight 1 are directed inward (toward the vertex). For the OR-vertex, at least one edge must be directed inward for any edge to be directed outward.

In some sense, the AND and OR-vertices are more robust then their logical counterparts. For the AND-vertex, if the edge of weight 2 is directed inward, then both edges of weight 1 can be directed outward, allowing the signal to fan out. When oriented so, we say that the AND-vertex is instead a FANOUT-vertex. This interpretation will be essential in later proofs.

Computational questions also naturally arise in Constraint Logic. For our purposes, we will be asking: Is there a sequence of moves on an arbitrary AND/OR Constraint Graph that reverses a designated edge? Recalling that Constraint Logic is the set of rules on a Constraint Graph, different formulations of the rules lead to problems that are harder or easier to answer computationally. There are several ways to look at this question. Informally, if the problem is *deterministic*, it means that we require that every move be forced; if the problem is *nondeterministic*, it means that at every turn we are able to choose which edge we wish to reverse, as long as the constraints are still met. If we make problem *bounded*, it means that we may only reverse each edge in the Constraint Graph once; if graphs are *unbounded*, each edge is reversible. The decision problems are stated formally below for bounded and unbounded non-deterministic Constraint Logic. This thesis focuses on bounded and unbounded nondeterministic Constraint Logic, and so these decision problems are stated formally here.

Decision Problem: Bounded Nondeterministic Constraint Logic
Instance: AND/OR Constraint Graph G with edge $e \in G$.
Question: Is there a sequence of moves that reverses e given that no edge may be reversed more than once?

Decision Problem: Unbounded Nondeterministic Constraint Logic
Instance: AND/OR Constraint Graph G with edge $e \in G$.
Question: Is there a sequence of moves that reverses e?

The reason why these particular problems are so useful is that it is often very natural to mapping reduce from Constraint Logic to many different games. Deterministic Constraint Logic is a useful tool for reducing to simulations or *zero-player games*, like the game of Life [1], whereas nondeterministic Constraint Logic seems to reduce well to *one-player* games or puzzles like Tipover and Sliding blocks [5]. We can also add different colored edges to arrive at a formulaion that can be used to reduce to 2-player games. However the construction of two player Constraint Logic is more involved and is mentioned here only for completeness. A summary of Hearn's Constriant Logic proofs are given below.

The Constraint Logic Theorems. *Summarizing Hearn's theorems [5],*

1. *Bounded deterministic Constraint Logic (DCL) is P-complete,*
2. *Unbounded DCL is PSPACE-complete,*
3. *Bounded nondeterministic Constraint Logic (NCL) is NP-complete,*
4. *Unbounded NCL is PSPACE-complete,*
5. *Bounded 2-Player Constraint Logic (2CL) is PSPACE-complete,*
6. *Unbounded 2CL is EXPTIME-complete.*

	Zero Player (Deterministic)	One Player (Non-deterministic)	Two Player
Unbounded	PSPACE	PSPACE	EXPTIME
Bounded	P	NP	PSPACE

2.3 Constraint Logic Construction

Before we prove that bounded and unbounded nondeterministic Constraint Logics are NP-complete and PSPACE-complete respectively, it will be useful to develop some notation and definitions in Constraint Logic. Although only AND and OR-vertices will be necessary in the construction of the Constraint Graphs in the proofs to come, various abstractions constructed out of multiple AND and OR vertices will be quite useful – what Hearn calls **gadgets**. With the following four gadgets, it will be easier (especially visually) to create the necessary Constraint Graphs.

2.3.1 CHOICE-Vertex Gadget

It will be necessary to choose between reversing two different edges, and so we need to make a CHOICE-vertex gadget. A CHOICE-vertex has constraint 2 and three red edges. Clearly, if one edge is directed away from the vertex, both remaining edges must be directed toward the vertex. If all three edges are facing toward the vertex, we have the option of *choosing* whether to reverse any one of the edges. We can create an equivalent construction with three AND-vertices, as illustrated in Figure 2.5.

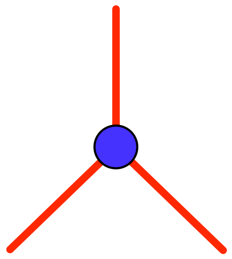


Figure 2.4: A Choice Vertex.

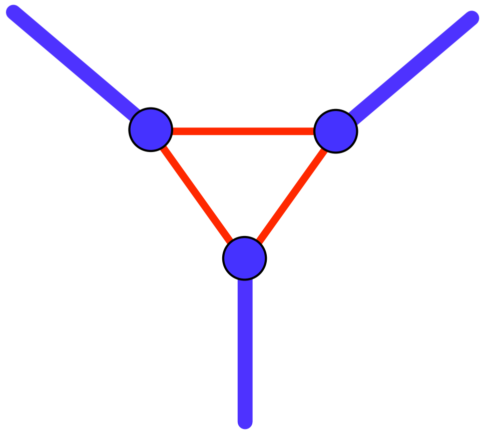


Figure 2.5: An equivalent set of vertices using only ANDs.

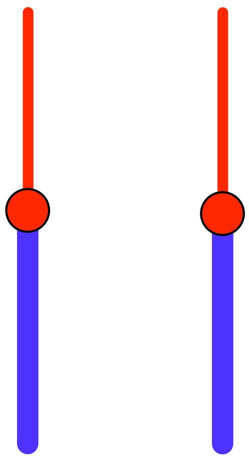


Figure 2.6: A Red-Blue Conversion

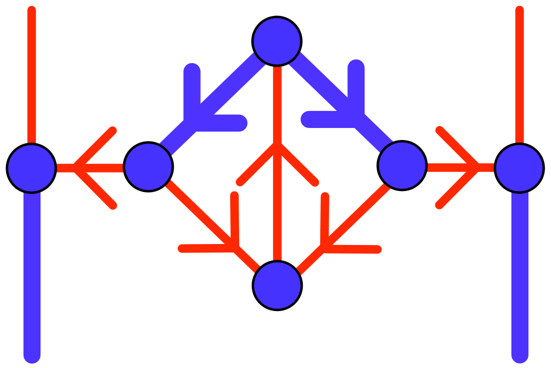


Figure 2.7: An equivalent set of vertices using AND-vertices and a CHOICE-vertex

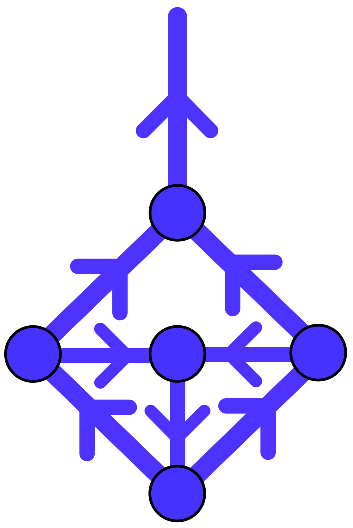


Figure 2.8: Free Terminal Edge

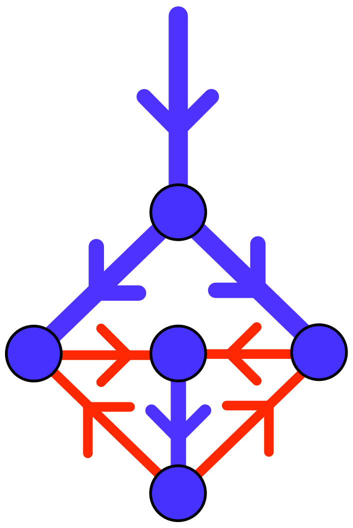


Figure 2.9: Constrained Terminal Edge

2.3.2 Red-Blue Conversion Gadget

When using the Choice gadget and when using AND-vertices, it will sometimes be useful to connect blue edges to red edges. To do so, we construct a red-blue conversion gadget. We can achieve an equivalent AND/OR-gadget if we have two instances in which we want to connect blue edges to red edges (see Figure 2.6). Since we always need red-blue conversions to come in pairs, so one might worry that in our AND/OR graph, the pairs will be too few. However, this cannot happen (cf. Hearn's *Games, Puzzles and Computation* [7]).

2.3.3 Terminal Edge Gadgets

In a legal Constraint Graph, all edges must be connected to two vertices – that is, we can never have loose edges. If there are loose edges in a Constraint Graph construction, we can construct legal Constraint Graph by attaching a Terminal Edge Gadget. There are two flavors of this gadget: we can either allow the one loose edge to be reversed, as in Figure 2.8, or we can prevent the loose edge from reversing, as in Figure 2.9. If we have loose red edges, we can convert these to blue edges using the previous red-blue conversion gadget.

2.3.4 Crossover Gadget

Later, we will want to be able to create planar Constraint Graphs, so that we can reduce from Constraint Logic to planar games. For that end, we can create a Crossover gadget (Figure 2.10) that converts overlapping edges into into non-overlapping edges that retain the same properties. To show how this is manifest, consider a pair of opposing edges in the Crossover gadget. This gadget has the property that if one of these edges faces outward, then the opposing edge is forced to face inward. Thus, if one of these edges switches orientation, the opposing edge must also switch orientation. The proof of these facts is not given here, but one could convince oneself of these facts (cf. Hearn [5]). [1] Note that the red-red-red-red vertex in the Crossover gadget is a mnemonic for the Half-crossover gadget illustrated in Figure 2.11. Also, when using the Crossover gadget, it might be necessary to use a red-blue conversion to retain the properties of the overlapping edges.

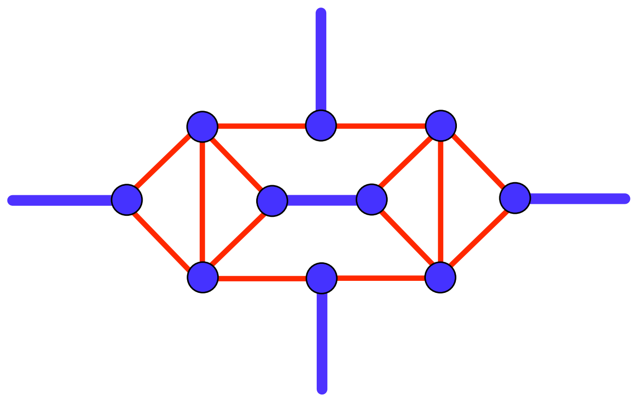
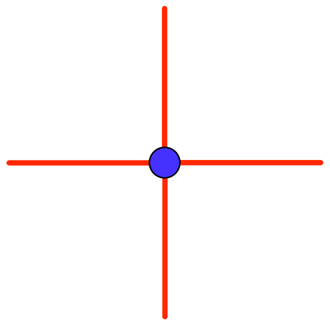
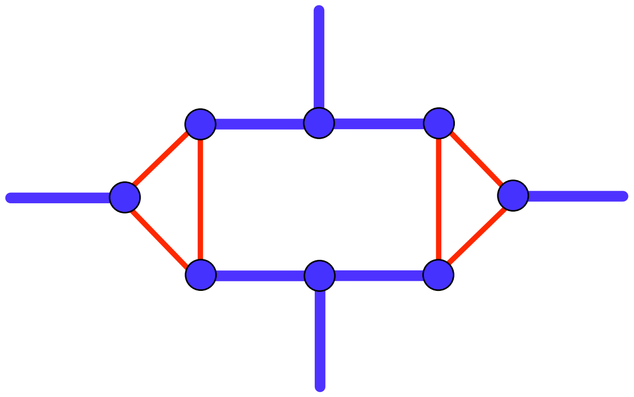


Figure 2.10: The Crossover Gadget



(a) The Reduced Half-crossover



(b) The Half-crossover in full

Figure 2.11: The half-crossover subgadget used in the Crossover gadget.

2.4 Hardness Proofs

2.4.1 Bounded NCL

Theorem 2.4.1. *Bounded NCL is NP-complete (cf. Hearn [5]).*

Proof. The follow is a sketch of the proof given by Hearn. We will first show that bounded nondeterministic Constraint Logic is NP-hard. To do so, we will preform a reduction from

[1]And, the proof is not very illuminating.

3SAT to bounded NCL. As a side note, the decision problem 3SAT is the similar to SAT in that we are trying to satisfy a Boolean formula that is a conjunction of clauses, where each clause in 3SAT is the disjunction of 3 literals. This problem is NP-complete [16].

We perform the reduction by taking an arbitrary (satisfiable) formula in 3-conjunctive normal form (3CNF) and representing it as an AND/OR Constraint Graph. However, since we have identified various mnemonics that are equivalent to a construction with AND/OR vertices, we will only be constructing the Constraint Graph with the gadgets, leaving the reader to convert the graph to an exclusively AND/OR Constraint Graph.

Let Φ be a formula in 3CNF and let G' be the corresponding Constraint Graph. Then, the following are the steps we need to perform:

1. For each variable in Φ, we create a CHOICE-vertex in G'.

2. For each logical OR in Φ, we create an OR-vertex in G'.

3. For each logical AND in Φ, we create an AND-vertex oriented appropriately in G'.

4. For each distinct literal, designate one branch of the the CHOICE-vertex as x_i and the other branch as $\overline{x_i}$, and connect accordingly to the AND and OR-vertices, using reversed AND vertices (FANOUTs) where necessary.

5. Designate an edge as the satisfied-out edge e, so that e reverses precisely when the formula is true.

This process is shown in by Figure 2.12. Informally, we are creating a circuit out of Constraint Logic, using CHOICE-vertices to simulate inverters. Following this method of reduction from 3SAT, Bounded NCL is in NP-Hard, since the designated satisfied-out edge e will be reversed if and only if the formula is satisfiable, which follows directly from the properties of each gadget. Since we are working under nondeterministic Constraint Logic, we can choose a solution and verify it in polynomial time. NCL is also clearly in NP since given (n) edges, we can verify a solution in polynomial (and in fact linear) time.

Thus, bounded NCL is NP-complete $\square$

2.4.2 Unbounded NCL

To show that unbounded nondeterministic Constraint Logic (UNCL) is PSPACE-complete, we need to show that UNCL is in PSPACE and also PSPACE-hard. First we shall show that UNCL is PSPACE-hard, which Hearn proves by way of a reduction from QBF to UNCL (see Theorem 1.3.1). In showing hardness, our goal is to algorithmically build a Constraint Graph in which we are able to reverse edge e exactly when an arbitrary formula in QBF is satisfiable. Recall that in the proof that QBF is PSPACE-complete, we gave an algorithm that decided whether a quantified Boolean formula F was true by recursively called itself with ones and zeros substituted for particular variables – on each recursion, removing an additional quantifier. This is exactly what we are going to do in Constraint Logic.

An arbitrary formula in QBF (e.g. $\exists y \forall w \forall x \cdots \exists z[(x \vee \overline{y}) \wedge \cdots \wedge (\overline{w} \vee x \vee \overline{y})]$) is composed of two distinct parts – a formula in CNF ($[(x \vee \overline{y}) \wedge \cdots \wedge (\overline{w} \vee x \vee \overline{y})]$) and a

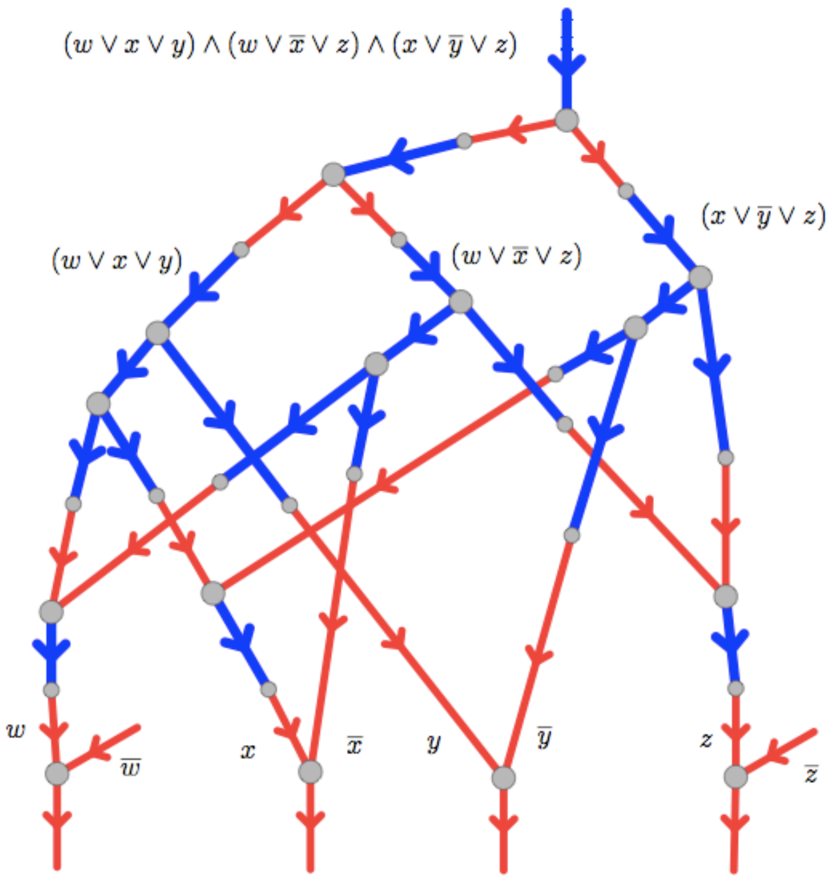

$(w \vee x \vee y) \wedge (w \vee \overline{x} \vee z) \wedge (x \vee \overline{y} \vee z)$
$(w \vee x \vee y)$
$(w \vee \overline{x} \vee z)$
$(x \vee \overline{y} \vee z)$
w
$\overline{w}$
x
$\overline{x}$
y
$\overline{y}$
z
$\overline{z}$

Figure 2.12: An example given by Hearn of a Constraint Graph that is a reduction from a Boolean formula $(w \vee x \vee y) \wedge (w \vee \overline{x} \vee z) \wedge (x \vee \overline{y} \vee z)$ to a Constraint Graph.

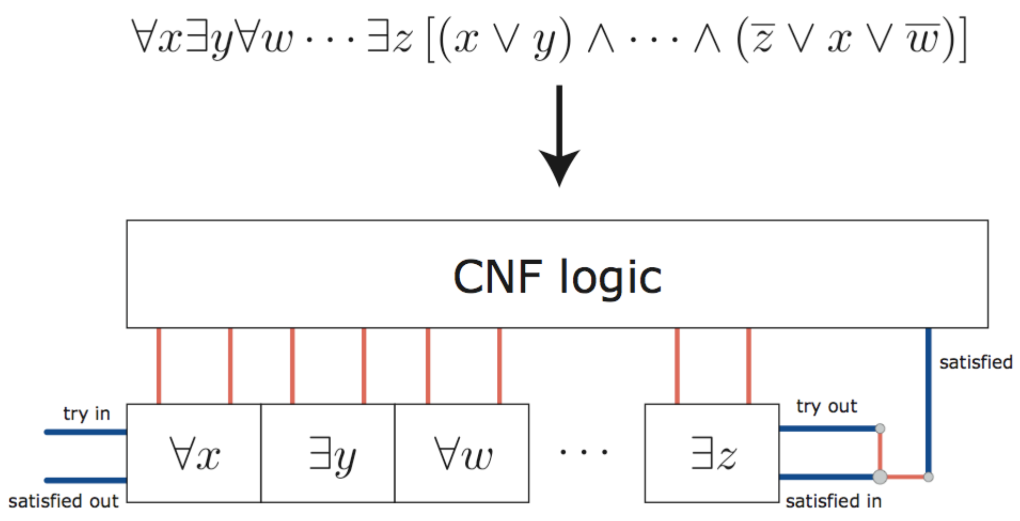
sequence of quantifiers ($\exists y \forall w \forall x \cdots \exists z$). From the proof that bounded NCL is PSPACE complete, we already know that how construct a Constraint Graph with an edge e that is reversed when a formula in CNF is satisfiable. Thus, our task will be to construct gadgets that function like universal and existential quantifiers and which connect in such a way to mimic recursion. Hearn gives an elegant illustration of this goal in Figure 2.13.

2.4.3 Latch Gadget

Although not a quantifier gadget, the Latch gadget (Figure 2.14) plays an important role in both the universal and existential quantifier gadgets. It has the property that if edge a is facing left, only one of edges b or c can be facing right and these edges are forced to face these ways. Then, if a is directed right, then b and c can reverse their orientations, but upon a being directed left again, the orientations of b and c are 'locked.'

Lemma 2.4.2. *For the Latch gadget, as given in Figure 2.14,*

- *If edge a is facing left, either edge b or c must face left.*
- *If edge a is facing right, b or c may have any orientation.*



$\forall x \exists y \forall w \cdots \exists z \, [(x \vee y) \wedge \cdots \wedge (\overline{z} \vee x \vee \overline{w})]$
CNF logic
satisfied
try in
try out
satisfied out
satisfied in
$\forall x$
$\exists y$
$\forall w$
$\cdots$
$\exists z$

Figure 2.13: A schematic given by Hearn of the construction of the proof that Unbounded NCL is PSPACE complete. We will build quantifier gadgets that interact with a Constraint Graph representation of a formula in SAT (as in Figure 2.12). The signal starts at try in, and as it passes through the quantifiers, it assigns values to the variables, locking the assignment by using the latch gadget. Then, after the signal reaches try out, it proceeds again through the quantifiers, checking the other variable assignment for the universal quantifiers (∀).

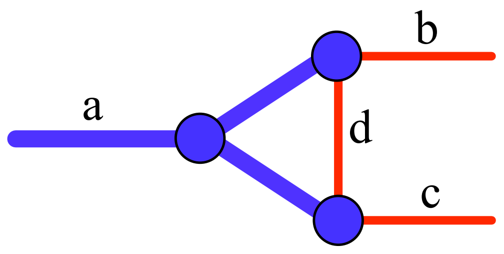


Figure 2.14: The Latch gadget.

2.4.4 The Existential Quantifier

The existential quantifier gadget is the easier of the quantifier gadgets to construct (See Figure 2.15). It is assumed that the initial configuration of an existential quantifier gadget is with Try In and Out directed left and Satisfied In/Out directed Right, although these follow from Lemma 2.4.3 and the assumption that the initial Try In is directed left in the construction (Figure 2.13).

Lemma 2.4.3. *(cf. Hearn [5]) For the Existential gadget given in Figure 2.15,*

1. *If Try In is directed to the left, then Try Out must be directed to the left. Similarly, if Try Out is directed right, Try In must be directed right.*

2. *If Satisfied In is directed left (right), then Satisfied Out, must be directed to the left (right).*

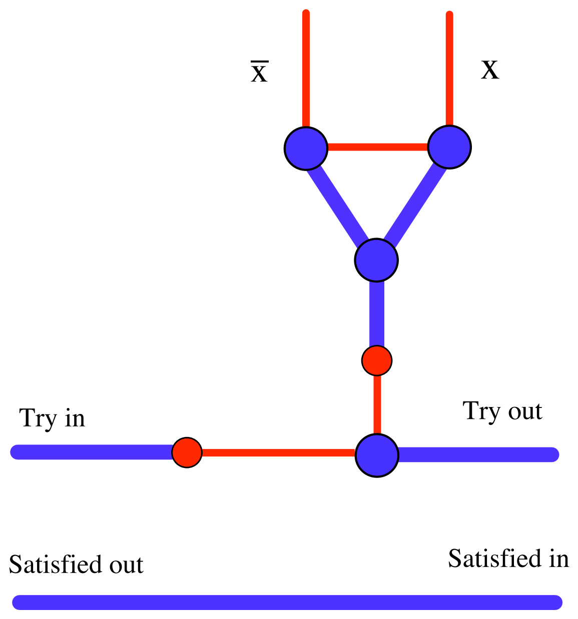


Figure 2.15: The existential quantifier gadget ($\exists x$)

3. *Once Try Out is directed right, then only one of x or $\overline{x}$ can be directed outward, which is fixed as long as Try Out is directed right.*

Proof. (1) follows since if Try In is directed left, this forces the adjoined red edge to face left. Then, to meet the AND vertex constraint, the Try Out vertex must be directed left. if Try Out is directed right, this forces both red edges to be directed toward the adjacent AND vertex, which forces Try In to be directed right. (2) is clear since Satisfied In/Out is one edge. (3) If Try Out is directed right, then the AND vertex to which it is adjoined must have its constraints met by both red edges. Then, by properties of the Latch (Figure 2.14), only one of x or $\overline{x}$ can be directed out, which is locked as long as Try Out is directed right. □

Since there are two possibilities for the orientation of the Latch gadget, we rely on the nondeterminism of the Constraint Logic to "choose" the correct orientation, which allows whichever x or $\overline{x}$ to be selected.

2.4.5 The Universal Quantifier

The Universal Quantifier is significantly more difficult to construct. Since the Try In - Try Out portion of the graph is similar to the Existential Quantifier structure, it can still be deduced that If Try In is directed left then the Try Out must be directed left. Similarly, if Try Out is directed right, then Try In must be directed right. Then, we only need to address when the Satisfied Out edge can be directed outward. Note again that we assume in the starting position that Try In is directed left and Satisfied Out is directed right.

Lemma 2.4.4. *(cf. Hearn [5]) For the Universal Quantifier gadget, given in 2.16, the Satisfied Out edge may be directed left if and only if:*

1. *the Satisfied In edge is directed left while variable state is locked in the false* $(\overline{x})$ *assignment and then,*
2. *the Satisfied In edge is directed left while its variable state is locked in the true* (x) *assignment with Try In directed right.*

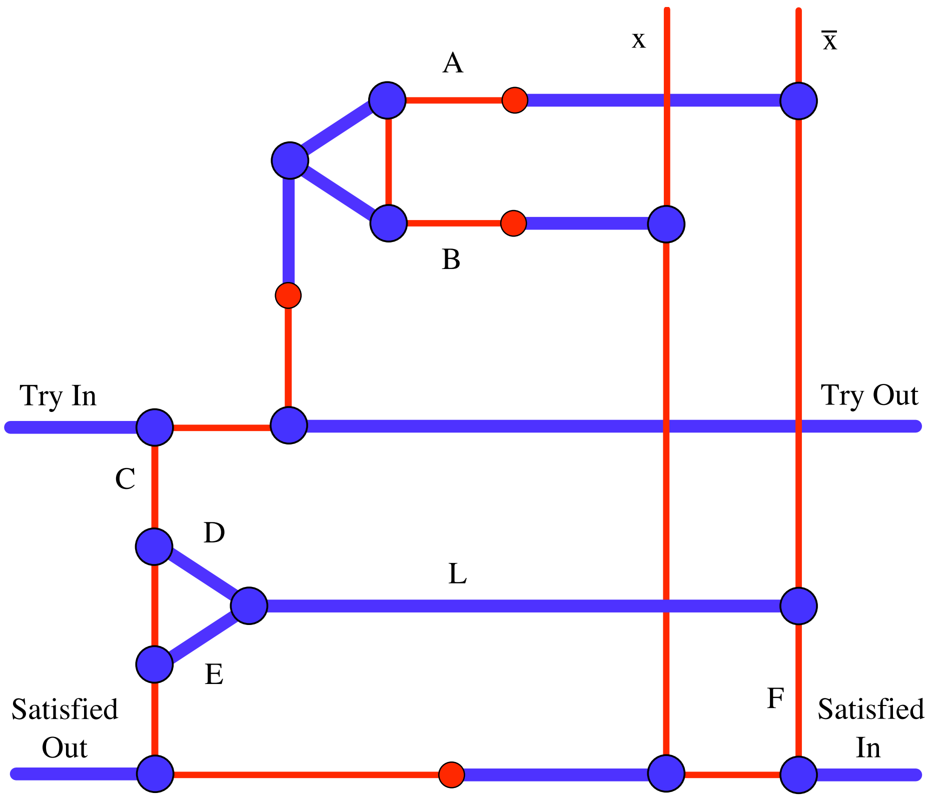


Figure 2.16: The universal quantifier gadget $(\forall x)$

Proof. First note that in the initial position, edge E is forced to face right. This is a direct consequence of the Universal quantifier initially having Try In facing left and Satisfied In facing right. If Try Out is directed right, then one variable edge will be locked facing down and the other will be allowed to face up. So, assume that the variable locked up is the false variable $(\overline{x})$. Then, if the Satisfied In edge faces left, the conditions for (1) are satisfied. This allows edge E to face left (since edge L can be directed left), and so the Satisfied In can reverse with edge E still facing left (with edge L reversing). Importantly, this allows edge C to face up. Note that edge L could not have been directed left if we originally chose to select the true variable as the one that would get locked up.

With edge E reversed, we send the signal back through the Quantifier Gadgets, until Try Out can face left. Without reversing Try In, this allows for the variable latch to be unlocked and locked with $\overline{x}$ in the down position. Now, we can travel back through the sequence and reverse Satisfied In, which satisfies condition (2). Finally, the signal can travel right and we can reverse Satisfied Out. □

2.4.6 The Unbounded NCL is PSPACE-complete

Now that we have constructed the necessary quantifier gadgets, we can finally prove two crucial facts about the quantifier gadgets. For reference, Figure 2.13 shows how to adjoin the Quantifier gadgets.

Lemma 2.4.5. *(cf. Hearn [5]) For the existential and universal quantifier gadgets (Figures 2.15, 2.16),*

1. *The Satisfied In edge can not be directed left if Try Out is directed left.*

2. *The Satisfied Out edge can be directed left if and only if Try In is directed right, and the formula represented by the CNF Logic-graph is true given variable assignments fixed by the quantifier gadgets.*

Proof. (1.) follows by by the UNCL construction (Figure 2.13) and by induction. The final Try Out connects by way of an AND vertex to the initial Satisfied In Vertex. And by construction, if the final Try Out is directed right, then each of the connecting Try In and Try Out edges for each of the adjoining quantifier gadgets must be directed right. (2.) is also shown by induction. Try In must be directed right by (1.), and the construction of the existential and universal quantifier gadgets ensures that The CNF Logic-graph must be satisfied. □

Now that we are finally able to prove the UNCL is PSPACE-Complete.

Theorem 2.4.6. *(cf. Hearn [5]) UNCL is PSPACE-complete.*

Proof. The final Satisfied Out edge of Figure 2.13 can be directed outward if and only if Satisfied Out edge of the leftmost quantifier gadget is directed leftward. This is only possible if the formula is true under the variable assignments fixed by the quantifier gadgets, by Lemma 2.4.5. Thus, the final Satisfied Out edge can be reversed precisely when the formula in TQBF is true. Thus, UNCL is PSPACE-hard.

Also, UNCL is in PSPACE because the state of a Constraint Graph can be described in linear space. We can then nondeterministically travel through the state space, choosing the correct move and recording only the current state. UNCL is then in NSPACE, and so by Savich's Theorem, UNCL is in PSPACE □

This proof only applies for non-planar graphs, due to overlapping edges in the Quantifier gadgets and the CNF-graph construction, but it turns out that:

Theorem 2.4.7. *UNCL is PSPACE-complete for planar graphs.*

The Crossover gadget constructed in Figure 2.10 is precisely constructed so that any any pair of overlapping edges in a Constraint Graph can be replaced with the Crossover Gadget to generate an equivalent planar Constraint Graph.

Chapter 3

Applications

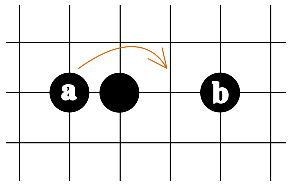
Since we know bounded nondeterministic Constraint Logic is NP-Complete from Chapter 2, we can use this problem as a reduction tool. An instance of Bounded NCL is composed of AND, OR, CHOICE, and FANOUT vertices, and so to show a problem is NP-Hard, we only need to construct gadgets with precisely the same properties. Similarly, we know Unbounded NCL is PSPACE-complete, so to prove a game is PSPACE-hard, we only need to construct gadgets with the same properties as reversible AND and OR Constraint vertices. Although several of the games I looked at seemed to have reversible properties, I was only able to show that modified versions of the games were NP-complete, and so we will rely primarily on the proof that Bounded NCL is NP-complete.

All methods for showing that a problem is NP-complete rely on analyzing what happens in a general instance of a game – for example, checkers on an $n \times n$ board. There are hundreds of different problems that have been shown to be NP-complete, and usually, one chooses selectively from one of the many different reductions, to find one that works (For numerous examples, see [4]). In contrast to this method, we have chosen our reduction tool – Constraint Logic – and now we will be searching for instances in which Constraint Logic works as a reduction technique. As such, we will often have to modify the games slightly, in order for Constraint Logic to give a valid reduction.

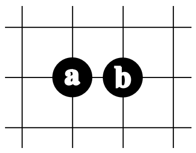
3.1 Peg Solitaire

Peg Solitaire (sometimes known as Hi-Q) is a popular one-player bounded puzzle, both in its online and physical forms [2]. It is played on an $n \times n$ grid with *pegs* located at some intersections, and not at others. At each turn, a player may choose to jump a peg over one (and only one) other peg, either horizontally, vertically, or diagonally. Then the jumped peg is removed (see Figure 3.1). The goal of Peg Solitaire is make a sequence of jumps in such a fashion that just one peg is left, its resting place on a designated intersection.

Uehara and Iwata [17] showed that Peg Solitaire is NP-complete, through a reduction from the Hamiltonian Path problem (cf. [16]). I wanted to see if it was possible to reduce from Constraint Logic. There are several characteristics that seem promising: Peg Solitaire is a one player puzzle played on a grid, it is already known to be NP-complete, and Hearn showed that Konane, a two-player Peg Solitaire variant, is PSPACE-complete using



(a) Before a move (jump).



(b) After the move (jump).

Figure 3.1: An example of a move in Peg Solitaire

Constraint Logic [5].

However, it seems quite difficult to build CHOICE and OR gates in normal Peg Solitaire, with the rule that we can have only one peg remaining to win. The reason for this is that, if we examine the CHOICE vertices in Constraint Logic, we must choose one of two paths. However, in Peg Solitaire, every peg but one must be eliminated, and so *both* paths of pegs from a CHOICE gadget in Peg Solitaire must be eliminated. There does not seem an easy way to accomplish this task. So, instead let's ask: Is it possible to jump over a particular peg? Thus, we have:

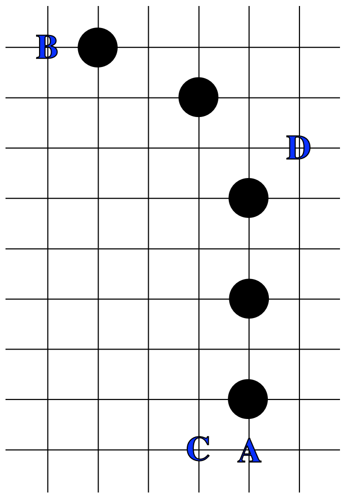
Decision Problem: Modified Peg Solitaire
Instance: Grid with pegs and empty intersections, with designated peg e .
Question: Is there a sequence of moves that allow us to jump over peg e?

Since we would like to reduce from Bounded NCL to Peg Solitaire, we need to construct AND, OR, CHOICE and FANOUT gadgets. First, let's construct a connecting gadget – that is, a wire. Examples of these are given in Figure 3.2. These allow us to connect the gadgets that follow. In the **two-way wire**, if there is a peg at intersection A, then that peg, through a sequence of jumps, will arrive at intersection B. A peg at intersection B can similarly reach the intersection A.

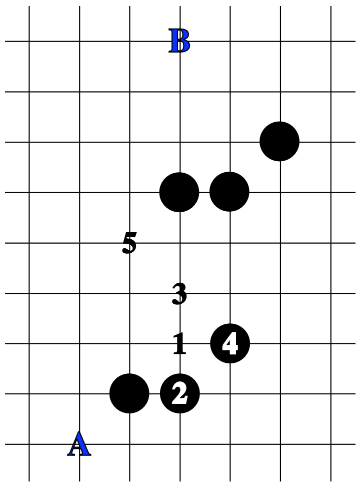
Unfortunately, one can also transfer a signal diagonally along the two-way wire. In the diagram, if there is a peg at C, then, through a series of jumps, the peg can reach D. This turns out to be somewhat problematic but can be avoided through the placement of turns; a diagonally-traveling peg that starts at C will never jump the peg adjacent to B. If the turn is directed right instead of left (not shown), then a diagonally traveling signal would still not be able to travel around the turn. However, being forced to include turns everywhere is clunky, and later, we will be able to avoid this situation more elegantly.

In the **one-way wire**, a peg at A can reach B through a sequence of jumps, A to 1, 2 to 3, and 4 to 5; this pattern is repeated until the peg reaches B. However, if there is a peg at B, no sequence of jumps will ever result in a peg at A. It is left to the reader to envision how the wire might be elongated. In the reduction that follows, these two wires will be essential to transferring a peg-signal.

The OR/CHOICE and AND gadgets: Hearn details how to construct an OR/CHOICE gadget for Konane [5], which will be used as both the OR and CHOICE gadgets here, as illustrated in Figure 3.3 (a). The AND gadget, in Figure 3.3 (b), is not much harder to



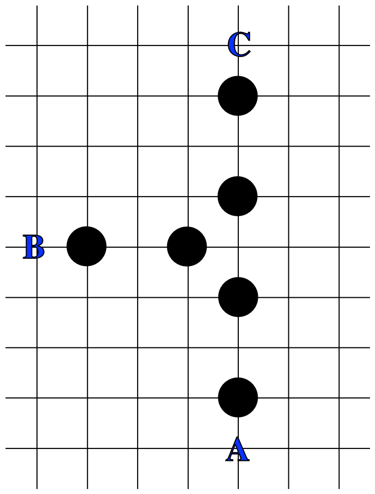
(a) A two-way wire with a turn.



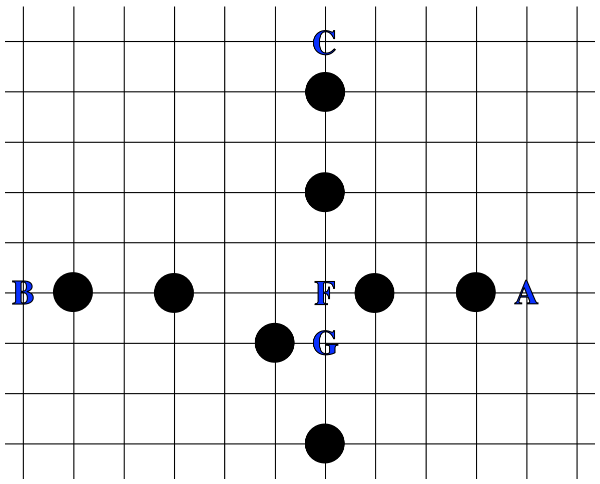
(b) A one-way wire.

Figure 3.2: These wires represent two ways to transfer a signal in Modified Peg Solitaire.

construct.



(a) The OR/CHOICE gadget.



(b) The AND gadget.

Figure 3.3: Two gadgets in Modified Peg Solitaire.

Lemma 3.1.1. *Referring to the gadgets in Figure 3.3,*

1. *In (a), if A and B correspond to inputs and C corresponds to an output, then the gadget functions exactly like a OR vertex in Bounded NCL.*
2. *In (a), if A corresponds to an input, and B and C correspond to outputs, then the gadget functions exaclty like an CHOICE vertex in Bounded NCL.*
3. *In (b), if A and B correspond to inputs and C corresponds to the output, the gadget functions exactly as an AND vertex in Bounded NCL.*

Proof. For (1.), note that a peg may reach an output C if and only if there is either a peg at B or a peg at A. It is possible that with a peg arriving at A, the peg could travel down the B-input wire. To prevent the OR gadget from functioning as a CHOICE gadget, we can attach 1-way wires to the inputs A and B. For (2.), if there is a peg at A, with B and C designated as outputs, then only one of B or C can be reached. Note also, that the OR/CHOICE gadget has three pegs that are touching in the center, which allows a diagonally-traveling signal to proceed at any time. This can be prevented with the attachment of several turns at A, B, and C.

For (3.), note that for there to be a peg at intersection C, we need a peg at intersection F and G. If there is a peg at A, there is a sequence of forced jumps starting at A that results in a peg at F. Similarly, If there is a peg at B, there is a sequence of forced jumps that result in a peg at G. One might be worried that if there is a peg at F, the peg to the left of the G-intersection could jump F diagonally, and then proceed down the wire. If we want to prevent this outcome, we can include several turns after the gadget which will prevent a diagonally-traveling signal from continuing (See Figure 3.2). □

The FANOUT gadget: All that now remains for us to do is to create a FANOUT gadget. However, I did not find a way to construct a FANOUT in Modified Peg Solitaire, and it seems like the creation of a FANOUT gadget (if possible) could be very difficult. Recall that a FANOUT vertex in Constraint Logic is equivalent to a reversed AND, and has the properties that when the blue edge faces in, the two red edges face out. Following the circuit interpretation, we need to create a forking gadget in Modified Peg Solitaire.

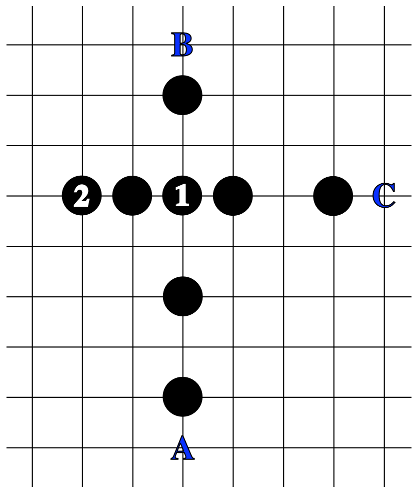
When a peg travels down a wire the jumped pegs are removed, so it might seem reasonable to try to create a FANOUT that works by using the pegs in a wire to block a signal. Using this idea, it might seem that the illustration in Figure 3.4 (a) works as a FANOUT, since the peg labeled 2 cannot reach intersection C until peg 1 is jumped. However, if peg 1 jumps the adjacent peg, peg 2 can jump its adjacent peg, and so the C wire can send a signal without a peg ever reaching A.

In Figure 3.5, another possible FANOUT is illustrated, using the 1-way wire. Some pegs are discolored to show the way the 1-way wires are adjoined. However, in this example as well, after peg 1 jumps to 2, 3 jumps to 1, and 4 jumps to five, we arrive at the diagram in Figure 3.5 (b).

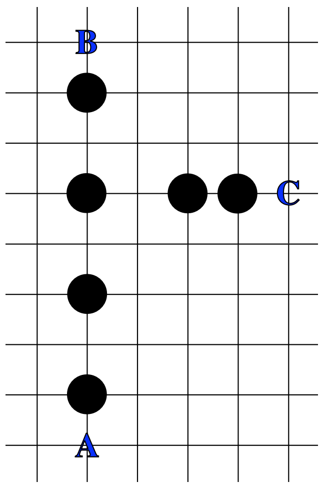
We can prove an even more restricted version of this problem. Let us add the rule that we can designate **immobile pegs** – that is, pegs that cannot jump. Although immobile pegs cannot jump, they otherwise function exactly as normal pegs; they can be jumped, and when they are jumped, they are removed. We can construct FANOUT gadgets by restricting peg 1 in Figure 3.4 to be an immoblie peg. Adding immobile pegs also allows us to create much more elegant gadgets that avoid the problem of diagonally traveling signals. To distinguish this version of Peg Solitaire, we will call it **Variant Peg Solitaire**. Variant gadgets are given in Figure 3.6, with empty circles indicating immobile pegs.

Lemma 3.1.2. *Referring to the gadget in Figure 3.6 (d), A is designated as the input and B and C are designated as outputs, then the gadget functions as a FANOUT vertex in Bounded NCL.*

Proof. Follows from the discussion above. □

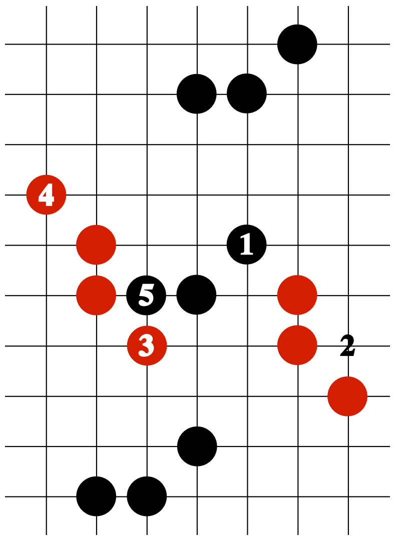


(a) An initial setup of a prospective FANOUT gadget



(b) After moving 1 and 2.

Figure 3.4: A plausible FANOUT gadget that fails.



(a) An initial setup of a prospective FANOUT gadget

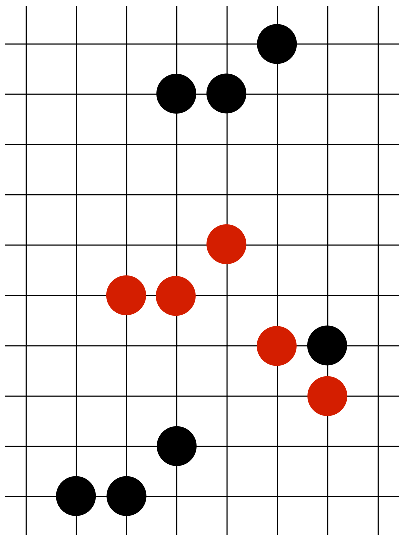
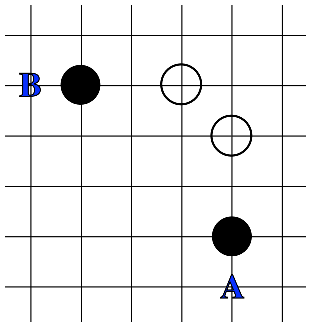
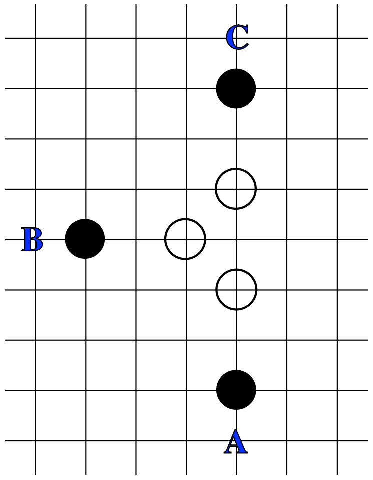

(b) After moving 1, 3 and 4 in (a).

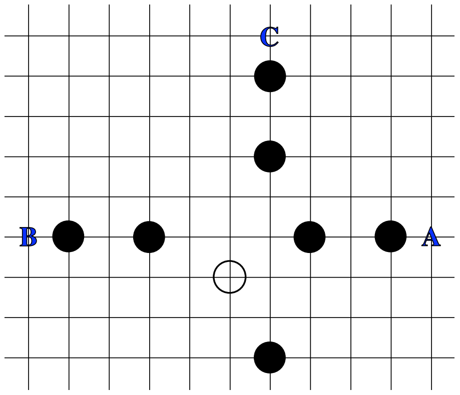
Figure 3.5: Another possible FANOUT gadget using 1-way wires that fails in Modified Peg Solitaire.



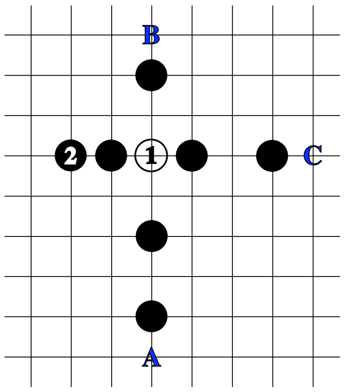
(a) A turn gadget.



(b) The OR/CHOICE gadget.



(c) The AND gadget.



(d) The FANOUT gadget.

Figure 3.6: The vertex gadgets in Variant Peg Solitaire.

Decision Problem: Variant Peg Solitaire
Instance: Grid with some pegs able to jump, others unable to jump, and empty intersections, with designated peg e.
Question: Is there a sequence of moves that allow us to jump over peg e?

Theorem 3.1.3. *Variant Peg Solitaire is NP-complete.*

Proof. Given a bounded planar Constraint Graph made of AND, OR, FANOUT and CHOICE vertices, we can construct a corresponding Variant Peg Solitaire puzzle. For each vertex in the Constraint Graph, we can now create a vertex-gadget in Variant Peg Solitaire, then we can connect the gadgets with the wiring indicated in Figure 3.2. Since each gadget has the same properties as each of the vertices, a designated peg can be jumped precisely when a designated edge in the Constraint Graph can be reversed. Thus, by a reduction from Constraint Bounded NCL to Variant Peg Solitaire, Variant Peg solitaire is NP-Hard.

Variant Peg Solitaire is in NP since there are only a linear number of pegs. Thus a solution can be verified in polynomial time, which is the definition of NP. Thus, Variant Peg Solitaire is NP-complete. □

3.2 Grid

Grid is a relatively new game, introduced as a Flash application by Mark James [9]. Grid is deterministic and so Grid is more a simulation than a game. Surprisingly, it has features similar to the Rotor Router model developed by Propp, as well as Sandpile models [13, 11]. Given this context, deterministic Grid probably warrants more study than shall be given here.

Deterministic Grid is played on an $n \times n$ grid, where each cell in the grid looks like one quarter of a circle. One starting position for a game of Grid is illustrated in Figure 3.7. Each cell has the property that when selected, it will rotate 90 degrees clockwise. If the cell's edges are touching another cell's edges, then the adjacent cells will rotate 90 degrees clockwise. A player accumulates points for each cell that rotates as a result of the initial rotation, and so the longer the chain reaction, the greater the score.

Grid seems to be a sort of reversible simulation. Conveniently (although not covered in Chapter 2), Hearn develops an unbounded deterministic Constraint Logic [5] tailored specifically toward simulations like Grid, so a logical question might be: Can we reduce from unbounded DCL to Grid? I can't find a way to create completely reversible wires in Grid, and so this approach stalls before it gets started.

Let's instead consider Grid with additional rules. We can make Grid into a 1-player game, which will be called **Puzzle-Grid**, by allowing some cells in the grid to be rotatable elements and others to be empty cells. In Puzzle-Grid, a player is allowed to rotate *any* cell 90 degrees clockwise with edges in contact with another cell's edges (see Figure 3.9), under the restriction that each cell can rotate at most two times. Then we can ask: Is it possible to rotate a particular element?

These rules require some explanation. It seems more natural to allow the entire grid to be filled with rotatable elements, like in Figure 3.7. In practice, since we allow a player to

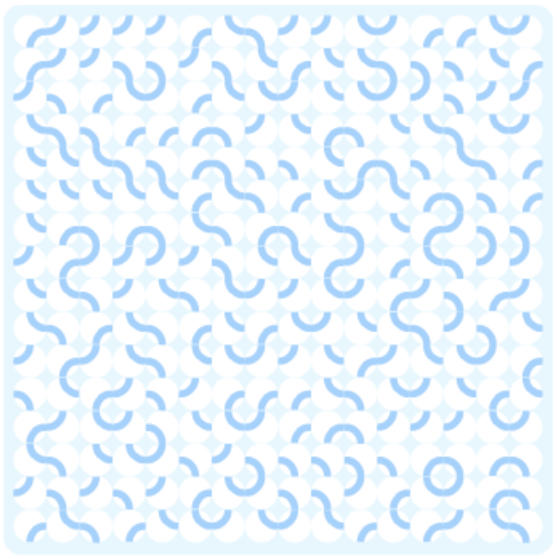


Figure 3.7: An example of a starting position in Grid

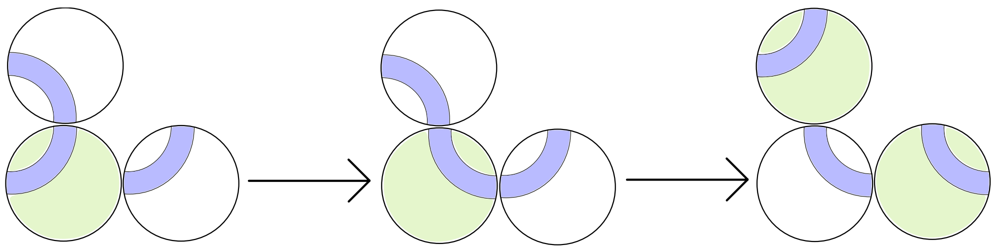


Figure 3.8: A example of a series of forced moves in the deterministic variant of Grid.

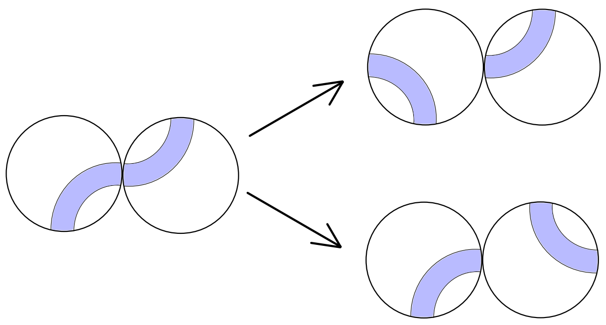


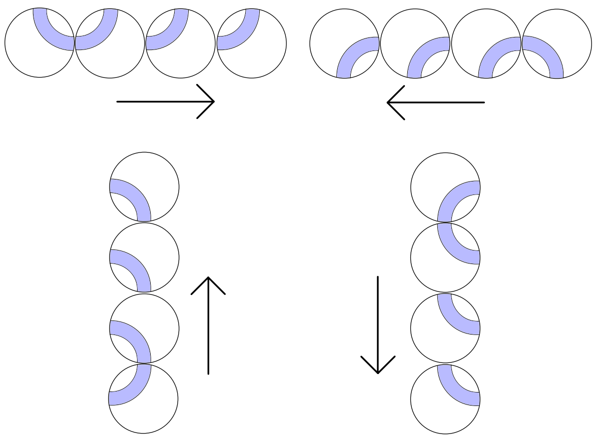
Figure 3.9: A example of moves in Puzzle-Grid.

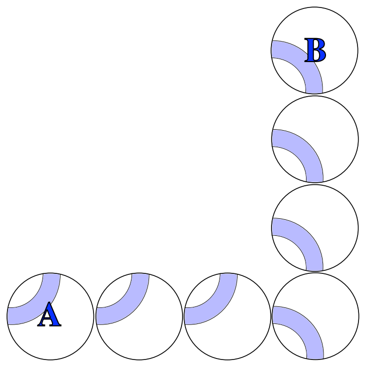
rotate any connecting cells, it is impossible to fill up the a grid with rotatable cells in such a way that cells do not interact with the wires and gadgets that we create.

It also seems more natural to consider the game without restrictions on the number of times we can rotate – in which case, we might be looking for a reduction from Unbounded NCL. Unfortunately, this makes our game too reversible! With unlimited rotations, a wire can send a signal in both directions simultaneously. For example, in Figure 3.10 (b), if cell A rotates, then the cell in the corner of the turn can rotate. Then, the cells can rotate back, until cell A is at its initial position. However, the corner cell can remain in its once-rotated position, eventually allowing B to rotate. This problem plagues attempts at creating gadgets as well. As such, to use a reduction from Constraint Logic, the most assured route is to restrict rotations; I leave the complexity of these other variants as open problems.

Decision Problem: Puzzle-Grid
Instance: A planar grid of cells with rotatable cells and empty cells, and a designated rotatable cell e.
Question: Is there a sequence of moves that allows us to rotate cell e?

As with Peg Solitaire, we will try to create wires and vertex gadgets that have precisely the same properties as the vertices in Constraint Logic. Directional wires are fairly easy to construct in Puzzle Grid, as are turns (Figure 3.10). In fact, the reader can check that these (perhaps slightly modified) also work as 1-way wires in deterministic Grid. The turn has the property that the cell labeled B can only turn if A turns first. The constructions for left and right turns are slightly different; for an example of a right turn, see the CHOICE gadget (Figure 3.12

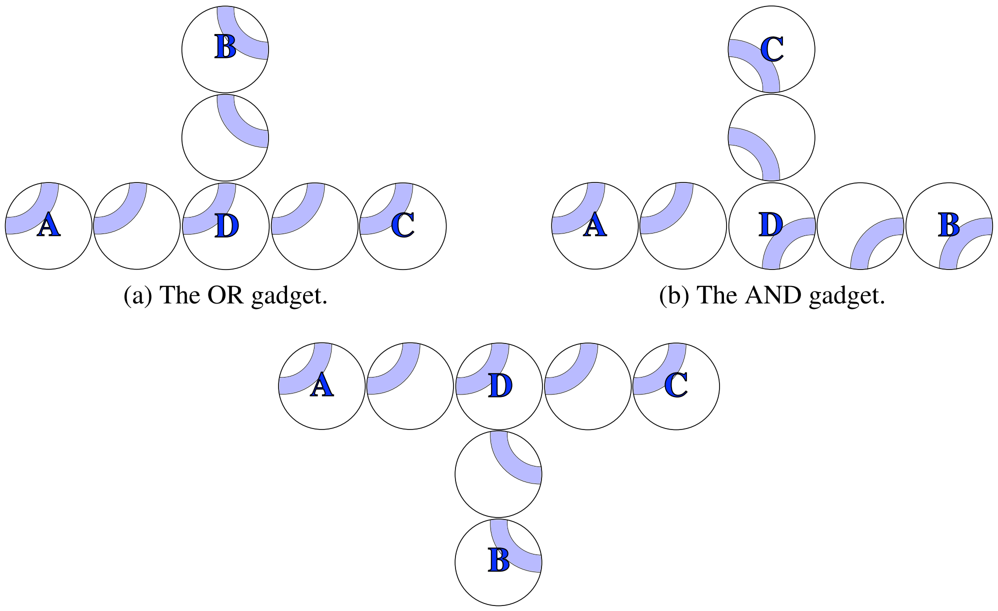

(a) Four directional wires.



(b) A left turn in a wire.

Figure 3.10: An illustration demonstrating how to construct wires in Puzzle-Grid

The OR, AND, and FANOUT gadgets: The OR, AND, and FANOUT gadget are variations on a common them. In each, we have a central cell labeled D that embodies the relevant properties of the gadgets.



(a) The OR gadget.

(b) The AND gadget.

(c) The FANOUT gadget.

Figure 3.11: Various gadgets in Puzzle-Grid

Lemma 3.2.1. *Referring to the gadgets in Figure 3.11,*

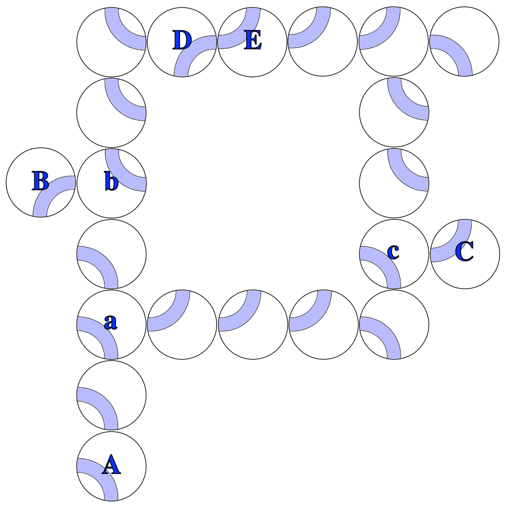
1. *For the OR gadget (a), if A and B correspond to inputs and C corresponds to the output, the gadget functions as an AND vertex in Bounded NCL.*
2. *For the AND gadget (b), if A and B correspond to inputs and C corresponds to the output, the gadget functions as an OR vertex in Bounded NCL.*
3. *For the FANOUT gadget (c), if A corresponds to the input and B and C correspond to the outputs, the gadget functions as a FANOUT vertex in Bounded NCL.*

Proof. For (1), the rotatable cell D has the property that if either A or B rotates, then C has the possibility of rotating. For (2), if B rotates, then D can rotate, and for D to rotate again, A must first rotate. For (3), when A rotates, D can rotate. But since D is then in contact with an adjacent cell, D can rotate again, allowing both B and C to rotate. □

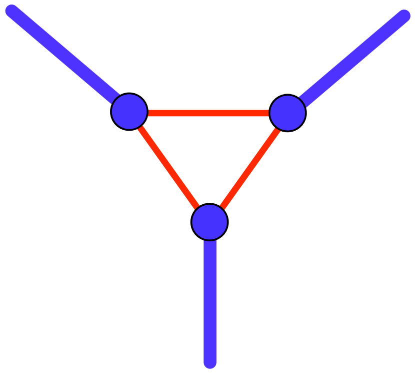
The CHOICE gadget: While the CHOICE gadget is more complicated than the other gadgets presented, it is essentially the same design s that of the NCL CHOICE gadget, constructed from AND vertices.

Lemma 3.2.2. *Referring to the gadget in Figure 3.12 (a), if A corresponds to the input and B and C correspond to the outputs, the gadget functions precisely as an CHOICE vertex in Bounded NCL.*

Proof. If cell A rotates, then cell a can rotate twice, acting as a FANOUT. Then, only one of B or C can rotate since only one of D or E can rotate. Once D or E rotates, the other can never rotate since each cell can only rotate twice. □



(a) The CHOICE gadget.



(b) The model for the CHOICE gadget in Constraint Logic.

Figure 3.12: The CHOICE gadget in Puzzle-Grid is construct from two AND and one FANOUT gadget, as in Constraint Logic.

Theorem 3.2.3. *Puzzle-Grid is NP-complete.*

Proof. This proof is essentially the same as the one for Variant Peg Solitaire. Given a bounded planar Constraint Graph, we can construct a corresponding instance of Puzzle-Grid with the same properties, by Lemmas 3.2.1 and 3.2.2. Since each gadget has the same properties as each of the Constraint vertices, then a designated cell can rotate precisely when a designated edge in the Constraint Graph can be reversed. Although we may not be able to construct arbitrary Constraint Graphs, by a similar reduction from 3SAT to Puzzle Grid, Puzzle-Grid is NP-Hard.

Puzzle Grid is in NP since there are only a linear number of cells that can be rotated. As such, a solution can be verified in polynomial time. Thus, Puzzle-Grid is NP-complete. □

3.3 Power Grid

Grid is also the name of circuit-like game developed by Chris Sheeler and Jeremiah Lapointe [15]. To distinguish this game from the Grid above, this Grid will be called Power Grid. Power Grid is essentially a modified version of the game KPlumber, popular in several computerized versions [8, 18]. KPlumber has been shown to be NP-complete [10], and so it might be fair to suppose that Power Grid is also NP-complete.

In an instance of Power Grid (Figure 3.13), the player is given a grid of cells with one designated as the *power source*. The rest of the grid is populated with either empty or rotatable cells, where the rotatable cells are one of four types – a straight piece, a T piece, an L turn, and a dead-end (Figure 3.14). Occasionally a fifth piece, the cross, is included in the game, but it will not be necessary for the discussion here. The player is allowed to rotate these cells 90 degrees clockwise or counter-clockwise only if the cells are connected to the power source. The goal of Power Grid is to connect each cell in the grid to the power source *and* to ensure that every edge of each piece is connected to another piece (i.e. no loose edges).

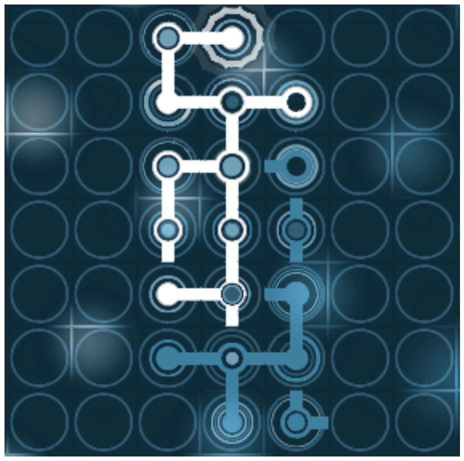


Figure 3.13: An instance of Power Grid.

Although KPlumber is NP-complete, it seems like Power Grid should be in PSPACE since every move is reversible. Yet, there does not seem to be an clear way to construct AND and OR gadgets that are totally reversible in Power Grid. Moreover, since the problem is to connect every cell, a reduction from Constraint Logic seems quite difficult. Even if it were possible to create CHOICE vertices, always having to choose between two paths implies that one path will never be powered. As such, there doesn't seem to be a clear way to connect the entire graph if we use a Constraint Logic reduction, which is a similar problem to the one we had in Peg Solitaire.

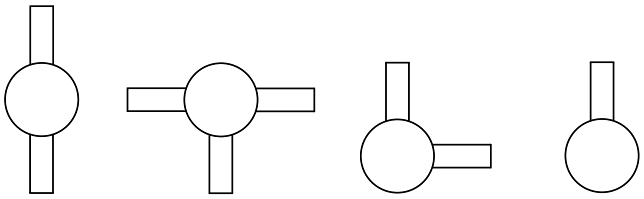


Figure 3.14: The four pieces of Power Grid.

Instead, let's ask the restricted decision problem: Can a particular cell be powered? Also, we'll add the rule that each cell can be rotated only once. With each cell being able to rotate as many times as we want, it turns out that the Constraint Logic gadgets are much more difficult to construct since with the restriction we don't have to worry about reversibility.

Decision Problem: Restricted Power Grid
Instance: Grid with empty and rotateable and empty cells, with power cell and designated cell e .
Question: Is there a sequence of moves that allow us to power cell e?

Lemma 3.3.1. *Referring to the gadgets in Figure 3.15,*

1. *For the OR gadget (a), if A and B correspond to inputs and C corresponds to the output, the gadget functions as an AND vertex in Bounded NCL.*
2. *For the AND gadget (b), if A and B correspond to inputs and C corresponds to the output, the gadget functions as an OR vertex in Bounded NCL.*
3. *For the FANOUT gadget (c), if A corresponds to the input and B and C correspond to the outputs, the gadget functions as a FANOUT vertex in Bounded NCL.*
4. *For the CHOICE gadget (d), if A corresponds to the input and B and C correspond to the outputs, the gadget functions as a CHOICE vertex in Bounded NCL.*

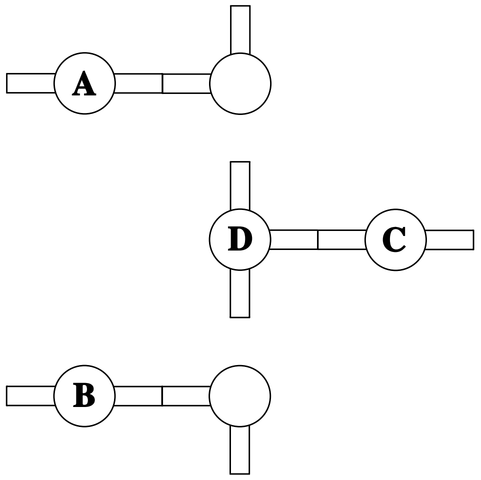
Proof. (1) If either A or B is powered, then D can be powered, which will power C.
(2) If A is powered, the D can be rotated, and then powering B will power C. Since each cell can only be rotated once, this implies that C can only be powered when both A and B are powered.
(3) If A is powered, then both B and C are powered.
(4) If A is powered, D can rotate either clockwise or counterclockwise once. After rotating, D will connect to the center cell, which will allow powering of either B or C. □

Theorem 3.3.2. *Restricted Power Grid is NP-complete.*

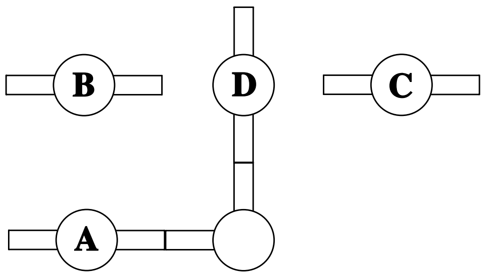
Proof. Given a bounded planar Constraint Graph, we can construct a corresponding instance of Power Grid with the same properties. By Lemma 3.3.1 the Power Grid gadgets have the properties as the vertices we needed in order to show that NCL is NP-hard. By a similar reduction from 3SAT, Puzzle-Grid is NP-Hard.

Also, Power Grid is in NP since there are only a linear number of pieces that can be rotated, so a solution can be checked in polynomial time. Thus, Power Grid is in NP, and so Power Grid is NP-complete.[1] □

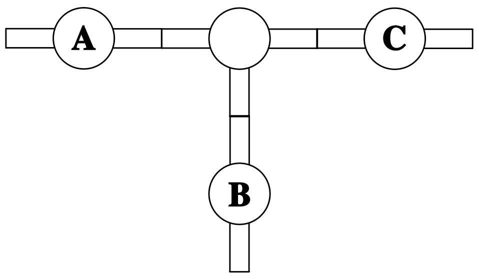
[1] $\tau\epsilon\tau\acute{\epsilon}\lambda\epsilon\sigma\tau\alpha\iota$



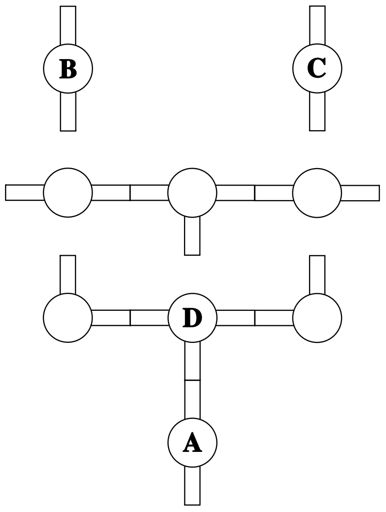
(a) The OR gadget.



(b) The AND gadget.



(c) The FANOUT gadget.



(d) The CHOICE gadget.

Figure 3.15: The Constraint Logic gadgets in Restricted Power Grid

References

[1] E. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways*. Academic Press Inc., 1982.

[2] E. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways For Your Mathematical Plays*, volume 2. Academic Press Inc., 1982.

[3] G. W. Flake and E. B. Baum. Rush hour is pspace-complete, or "why you should generously tip parking lot attendants". *Theoretical Computer Science*, (270(1-2)):895–911, January 2002.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[5] R. A. Hearn. *Games, Puzzles, and Computation*. Doctoral thesis, MIT, 2006.

[6] R. A. Hearn and E. Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, (343(1-2)):72–96, October 2005. Special issue "Game Theory Meets Theoretical Computer Science".

[7] R. A. Hearn and E. D. Demaine. *Games, Puzzles, and Computation*. A K Peters, LTD, 2009.

[8] A. Hintze. loops of zen, September 2008. `http://www.alproductions.us/home/home/home.html`.

[9] M. James. Grid game. `http://fizzlebot.com/gridgame.php`.

[10] D. Kral, V. Majerech, J. Sgall, T. Tichy, and G. Woeginger. It is tough to be a plumber. *TCS: Theoretical Computer Science*, 313(3):473–484, 2004.

[11] L. Levine. *The Rotor-Router Model*. Undergraduate thesis, University of California, Berkeley, 2002. `http://math.mit.edu/~levine/rotorrouter.pdf`.

[12] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Co., Inc., 1994.

[13] J. Propp. Rotor router model. `http://www.cs.uml.edu/~jpropp/rotor-router-model/`.

[14] E. Rich. *Automata, Computability and Complexity*. Pearson Prentice Hall, 2008.

[15] C. Sheeler and J. Lapointe. Grid, April 2009. `http://www.atomiccicada.com/flash.html`.

[16] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Co., 2nd edition, 2005.

[17] R. Uehara and S. Iwata. Generalized hi-q is np-complete. *TRANS of the IEICE*, pages 270–273, February 1990.

[18] M. van der Lee. Linkz, 2001. `http://www.vanderlee.com/software_linkz_download.html`.