Problem Set 4

Due: Wednesday, November 12th, 2014

Problem 1. Given a graph $G = (V, E)$, a connected dominating set $D \subseteq V$ is a set of vertices such that the subgraph of G induced by D is connected, and each vertex in V is adjacent to at least one vertex in D. The Connected Dominating Set problem takes as input a graph $G = (V, E)$ and a positive integer k, and asks whether there exists a connected dominating set of size k.

(a) Show that Connected Dominating Set is W[2]-hard with respect to the parameter k.

Solution: To show that this is W[2]-hard, we reduce from the Dominating Set problem. Given a graph $G = (V, E)$ and a desired dominating set size k, we construct the graph $G' = (V', E')$ for the corresponding Connected Dominating Set instance as follows. For each $v \in V$, construct two vertices x_v and y_v, and add both to V'. The vertices x_v will be connected in a clique; the vertices y_v will form an independent set. For each $v \in V$, add an edge (x_v, y_v) to E'. For each edge $(u, v) \in E$, add the two edges (x_u, y_v) and (x_v, y_u) to the edge set E'.

Suppose that we have a dominating set $S \subseteq V$ of size k. Then we can construct the corresponding connected dominating set by choosing $S' = \{x_v \mid v \in S\}$. By construction, the size of this set is k. Because the nodes x_v are connected in a clique, the set is also guaranteed to be connected, and all other nodes x_u are guaranteed to be dominated. For the nodes y_v, we consider two cases. If $v \in S$, then $x_v \in S'$, and therefore the edge (x_v, y_v) ensures that y_v is dominated. If $v \notin S$, then because S is dominating, there must be some edge (v, w) such that $w \in S$. As a result, x_w will be in S', and so the edge (x_w, y_v) ensures that y_v is dominated. Hence, all of the y_vs are dominated, and the set S' is therefore a dominating set.

Suppose instead that we have a connected dominating set S' of size k for the graph $G' = (V', E')$. Let $S = \{v \mid x_v \in S' \text{ or } y_v \in S'\}$. Then by construction, $|S| \leq |S'| = k$. Given a vertex $y_v \in V'$, either $y_v \in S'$, or a neighbor of y_v is in S'. In the former case, the vertex v will be contained in S, and will therefore be dominated. Suppose instead that there is some neighbor of y_v in S'. Because the y_vs form an independent set, the neighbor of y_v in S' must be x_u for some $u \in V$. The only time an edge (x_u, y_v) is added to E' is when the corresponding edge (u, v) exists in G. Because $x_u \in S'$, $u \in S$, and so v is again covered by S, and thus S forms a dominating set. □

(b) Show that Connected Dominating Set is in W[2]. **Hint:** Construct a weft-2 circuit of size $f(k)n^{O(1)}$ with nk inputs.

Solution: For each vertex $v \in V$ and each index $i \in \{1, \ldots, k\}$, we construct an input $x_{v,i}$ which should be set to true if v is the ith vertex in our connected dominating set. To enforce the desired constraints, we need three formulas: one to check for inconsistent variable settings, one to check for domination, and one to check for connectivity. As long as each of these individual constraints has weft 2, they can be combined in parallel using a single bounded-degree gate to ensure that all of the constraints are enforced.

Because we are constructing a weighted weft-2 circuit with parameter k, we know that exactly k of our inputs will be set to true. So to ensure that there is exactly one vertex assigned to each of the k indices, it is sufficient to ensure that for each index $i \in \{1, \ldots, k\}$, there do not exist two vertices $u, v \in V$ such that $x_{u,i} \wedge x_{v,i}$. This may be accomplished with the following formula:

$$\bigwedge_{i \in [1,k]} \neg \left(\bigvee_{(u,v) \in V \times V} x_{u,i} \wedge x_{v,i} \right)$$

Clearly, this formula has weft 2. (And in fact, by distributing the negation, we end up with an AND of ANDs, which can be flattened into a weft-1 circuit.)

To check whether the set is dominating, we need one formula for each vertex, asking whether it or one of its neighbors is contained in the dominating set. For each node $v \in V$, let $D(v) = \{v\} \cup N(v)$, the set of vertices in V that dominate v. This can be accomplished with the following formula:

$$\bigwedge_{u \in V} \left(\bigvee_{(v,i) \in D(u) \times [1,k]} x_{v,i} \right)$$

Again, this formula has weft 2.

Finally, to check whether the set is connected, we wish to ensure that there is a spanning tree on the k vertices in the dominating set. To accomplish this, we ensure that for each index $i \in [2, k]$, there exists some index $j < i$ and an edge $(u, v) \in E$ such that $x_{u,j} \wedge x_{v,i}$ — that is, that for each index i, there exists some edge joining the vertex chosen at index i, to some earlier vertex in the ordering. This ensures that the vertices in the dominating set are joined in a rooted tree. Because the ordering of the nodes within the set is arbitrary, we know that as long as such a spanning tree exists, there is a way to order the nodes in the dominating set to satisfy this constraint. Hence, this constraint exactly encapsulates the connectedness requirements:

$$\bigwedge_{i \in [2,k]} \left(\bigvee_{((u,v),j) \in E \times [1,i-1]} x_{u,j} \wedge x_{v,i} \right)$$

For a third time, we have a formula of weft 2, so the three formulas can be combined in parallel using a single and gate with three inputs to produce a circuit that evaluates the desired constraints. Hence, the problem is in W[2]. □

Problem 2. Recall the problem of Minesweeper. A *Minesweeper board* consists of a rectangular grid with dimensions $m \times n$, a subset $R \subseteq [1, m] \times [1, n]$ of revealed squares in the grid, and a function $f : R \to [0, 8]$ mapping from cells in R to the number of adjacent bombs. A *solution* to a Minesweeper board $\langle m, n, R, f \rangle$ is a set $X \subseteq ([1, m] \times [1, n]) \setminus R$ of bomb positions such that, for every square $(i, j) \in R$, the number of squares adjacent to (i, j) that are contained in X is $f(i, j)$. The Minesweeper problem is the obvious NP search problem: find a solution to a given Minesweeper board. The #Minesweeper problem is the corresponding problem of counting solutions.

In each part of this problem, we give a reduction from Planar Circuit SAT to Minesweeper (slight modifications from those in lecture), intended to show that #Minesweeper is #P-hard. For

each reduction, do one of the following: prove that it is parsimonious, prove that it is c-monious, or show that the number of solutions does not change by a fixed multiplicative factor (and thus that the reduction does not yield the desired #P-hardness results).

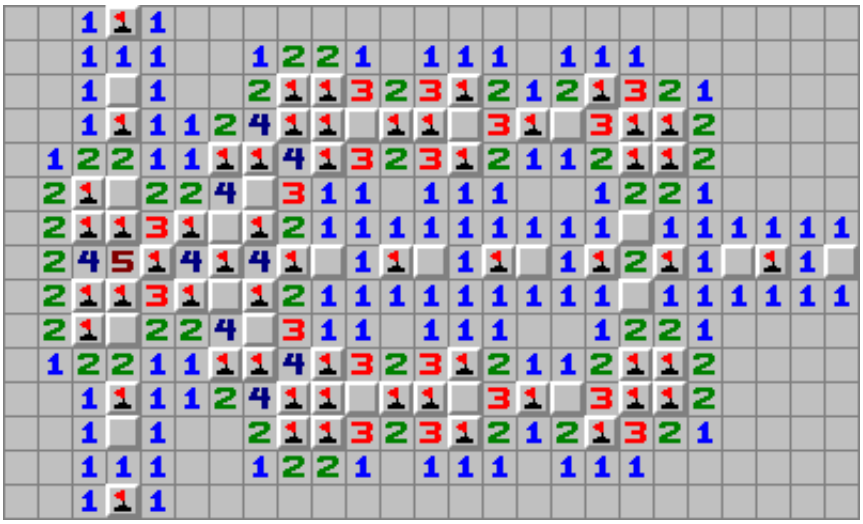
(i) In this reduction, data is conveyed by a wire like the one depicted in Figure 1(a). By construction, if there is a bomb in the leftmost empty square, then there cannot be a bomb in the rightmost empty square. If the wire is oriented from left to right, this case represents having the value TRUE conveyed along the wire. Similarly, if there is no bomb in the leftmost empty square, then there must be a bomb in the rightmost empty square. If the wire is oriented from left to right, this case represents having the value FALSE conveyed along the wire.

Wires can be terminated using one of two gadgets. The gadget in Figure 1(b) is a terminal that allows the incoming wire to carry either value. The gadget in Figure 1(c) is a terminal that forces the incoming wire to carry the value TRUE (and is used only once, to ensure that the output of the simulated circuit is TRUE). The shifter gadget depicted in Figure 1(d) can be used to adjust the length of the wire modulo 3.

The gadget in Figure 1(e) can be used for several purposes. For this gadget, if the leftmost empty square contains a bomb, then the topmost and bottommost squares cannot contain a bomb (thus creating the value of TRUE on the wires if the top wire is oriented upwards and the bottom wire is oriented downwards), and the rightmost square must contain a bomb (thus creating the value of FALSE on the right wire, if it's oriented to the right). Hence, by blocking off the top and bottom wires with a terminator, this gadget can be used to negate a value incoming from the left. By blocking off the top and right wires with a terminator, this gadget can be used to turn. By blocking off the right wire with a terminator, this gadget can be used to split a value coming in from the left. Thus, this one gadget gives us splitters, turns, and negations.

The final gadget is an AND gate, shown in Figure 1(f). The two inputs are on the left, coming in from the top and bottom; the output is on the right.

Solution: This reduction is neither parsimonious nor c-monious. The cause for this is the AND gadget. When both inputs are true, there is exactly one configuration for the gadget, depicted in the following image:



However, when both inputs are false, there are two configurations for the gadget, depicted in the following images:

As a result of this, the number of Minesweeper solutions corresponding to a single solution in the original SAT instance depends on the number of AND gates that have both inputs false, instead of being a fixed multiple (as you would get with a parsimonious or c-monius reduction). □

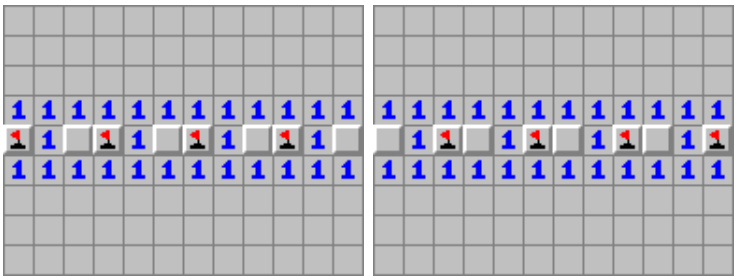
(ii) The wires for this reduction, depicted in Figure 2(a), are the same as the wires used in part (i). The terminator gadgets are set up similarly to those in part (i), with the terminator gadget in Figure 2(b) allowing the wire to take on any value, and the terminator gadget in Figure 2(c) forcing the incoming wire to be TRUE. However, instead of using a shifter to adjust the length of wires modulo 3, the shifter in this reduction is used to move the wire slightly in the direction orthogonal to the wire, as shown in Figure 2(d).

As in part (i), we have a combination splitter/negation/turn gadget, depicted in Figure 2(e). For this gadget, if there is a bomb on the leftmost empty square, then there is also a bomb on the rightmost empty square and the bottommost empty square, so if a wire is coming in from the left, then the negated value will emerge from the wire on the right and the wire on

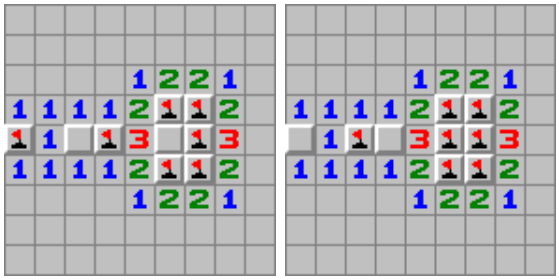
the bottom. By terminating the wire on the bottom, this can be converted into a negation gadget. By terminating the wire on the right and adding a copy of the negation gadget, this can be used to turn. By adding a copy of the negation gadget to negate the input value, this becomes a splitter.

The final gadget is an OR gate, shown in Figure 2(f). The two inputs come in from the left and from the top; the right is the output.

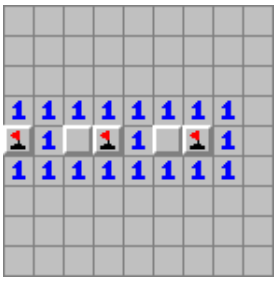
Solution: This proof is parsimonious. To see that this is true, we consider each of the gadgets individually. The wire has two possible configurations, one corresponding to a value of true, and one corresponding to a value of false:



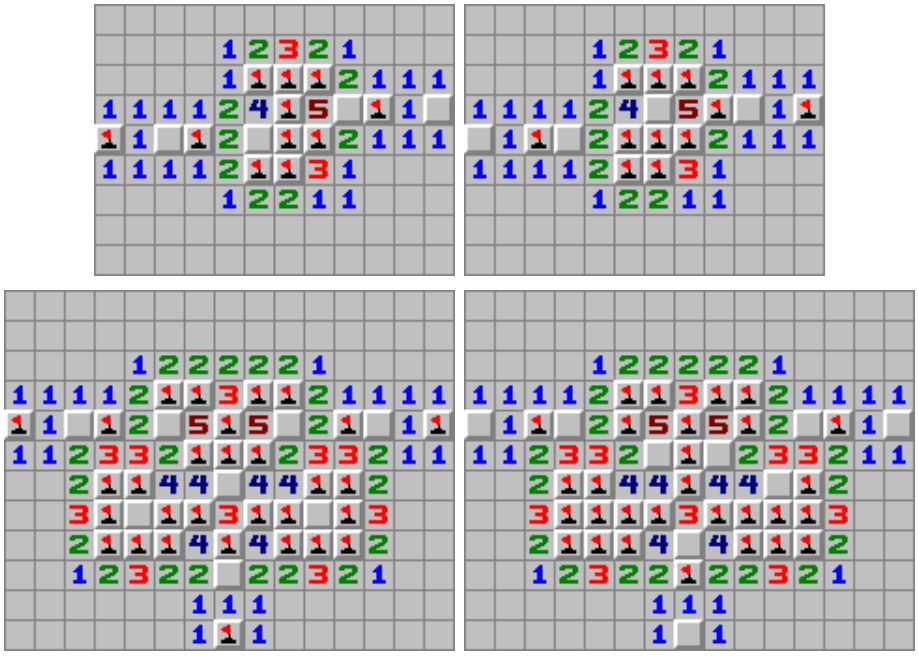
Similarly, the wire terminator has one configuration for each of the possible values carried:



The terminator forcing the wire to be true has exactly one configuration:



The shift and split gadgets, similarly, have one configuration corresponding to each of the potential values, depicted below:



So far, we have seen that for each of the wire gadgets, there is exactly one configuration for the given set of inputs. To complete the proof, we wish to show that for any combination of inputs to the OR gadget (true and true, true and false, false and true, or false and false) there is exactly one configuration of the OR gadget. The rules of Minesweeper ensure that these four configurations, corresponding to those four different cases, are the only possible ways to place mines within the OR gadget:

Hence when the gadgets are combined, there will be exactly one Minesweeper solution corresponding to each of the original SAT solutions. □

Problem 3. Given a simple directed graph $G = (V, E)$, a set of edges $E' \subseteq E$ is *cycle-removing* if $G' = (V, E \setminus E')$ contains no directed cycles. Two cycle-removing sets are *neighbors* if one set can be obtained from the other by removing a single edge (or, symmetrically, by adding a single edge).

Suppose that you are given a graph $G = (V, E)$, an integer k, and two cycle-removing sets of size k: E_{start} and E_{end}. Call a set of edges $E' \subseteq E$ *valid* if it is a cycle-removing set of size k or $k + 1$. Show that it is PSPACE-hard to determine whether there exists a path of valid sets $E_{start} = E_0, E_1, E_2, \ldots, E_m = E_{end}$ such that E_i and E_{i+1} are neighbors for all $0 \leq i < m$.

Solution: For this problem, we perform two reductions in sequence to show the PSPACE-hardness. Given a graph $G = (V, E)$, an integer k, and two vertex covers S_{start} and S_{end} of size k, the Reconfigurable Vertex Cover problem asks whether there exists a "path" of vertex covers of size k or $k+1$ from S_{start} to S_{end}, adding or removing a vertex at each step, while maintaining the invariant that each step is a vertex cover.

We may show that Reconfigurable Vertex Cover is PSPACE-hard by a reduction from Reconfiguration 3-SAT. For each variable x_i we construct two vertices x_i and $\overline{x_i}$ connected by an edge. The first vertex will be contained in the vertex cover if the variable is true; the second vertex will be contained in the cover if the variable is false. For each clause $c_t = \ell_i \vee \ell_j \vee \ell_k$, we construct three vertices $u_{t,1}$, $u_{t,2}$, and $u_{t,3}$ connected in a triangle. We then add an edge between $u_{t,1}$ and ℓ_i, an edge between $u_{t,2}$ and ℓ_j, and an edge between $u_{t,3}$ and ℓ_k. The desired vertex cover size will be $n + 2m$, where n is the number of variables and m is the desired number of clauses. Because each clause is connected in a triangle, at least two of the three vertices in the clause must be contained in the cover. Furthermore, the connection between x_i and $\overline{x_i}$ ensures that at least one vertex for every variable gadget must be contained in the cover. Hence, any cover of size $n+2m$ must contain exactly two vertices from each clause gadget and exactly one vertex from each variable gadget.

Suppose that we have a valid vertex cover of size $n + 2m$. Given a clause $c_t = \ell_i \vee \ell_j \vee \ell_k$, we know that the cover contains exactly two of $u_{t,1}$, $u_{t,2}$, and $u_{t,3}$. Without loss of generality, suppose that it does not contain $u_{t,1}$. Then because the edge between $u_{t,1}$ and ℓ_i must be covered, ℓ_i must be contained in the vertex cover. Hence, any valid vertex cover of size $n+2m$ produces a satisfying assignment. Furthermore, if we examine a path of vertex covers and use only the vertex covers of

size $n+2m$ to produce the corresponding sequence of satisfying assignments, each pair of satisfying assignments will differ by at most one variable, just as we wanted.

Suppose that we have a satisfying assignment. Then we can construct the corresponding vertex cover as follows. We start by including the set of variable nodes corresponding to true literals. Next, for each clause $c_t = \ell_i \vee \ell_j \vee \ell_k$, we find the first true literal in the clause, and then add to our vertex cover the two nodes in the clause that are not connected to that literal. This produces a vertex cover of size $n + 2m$, just as we wanted. However, to complete the proof, we must use a sequence of "neighboring" satisfying assignments to produce a sequence of "neighboring" vertex covers. If two subsequent vertex covers in our sequence differ by two elements, then we insert an intermediate vertex cover consisting of the union of the two (which must have size $n + 2m + 1$). If they differ by more, the change must be caused by vertices from the clause gadgets being added and removed from the vertex cover. In our construction, this occurs when the first true literal in the clause changes — that is, when the former first true literal becomes false (making another item in the clause become true) or when an earlier literal becomes true. In the first case, we know that there were two true literals in the clause before the first literal became false, so we can reconfigure that clause gadget before setting the literal to false by adding the third vertex in the clause to the vertex cover, then removing the vertex adjacent to the literal that will become the first true literal of the clause after the variable is changed. In the second case, we know that there will be two true literals in the clause after the chance, so we can reconfigure the clause gadget after the new literal has been set to true, using a similar method. If we do this for every clause, adding the necessary set of intermediate steps, we get a sequence of vertex covers with the desired properties.

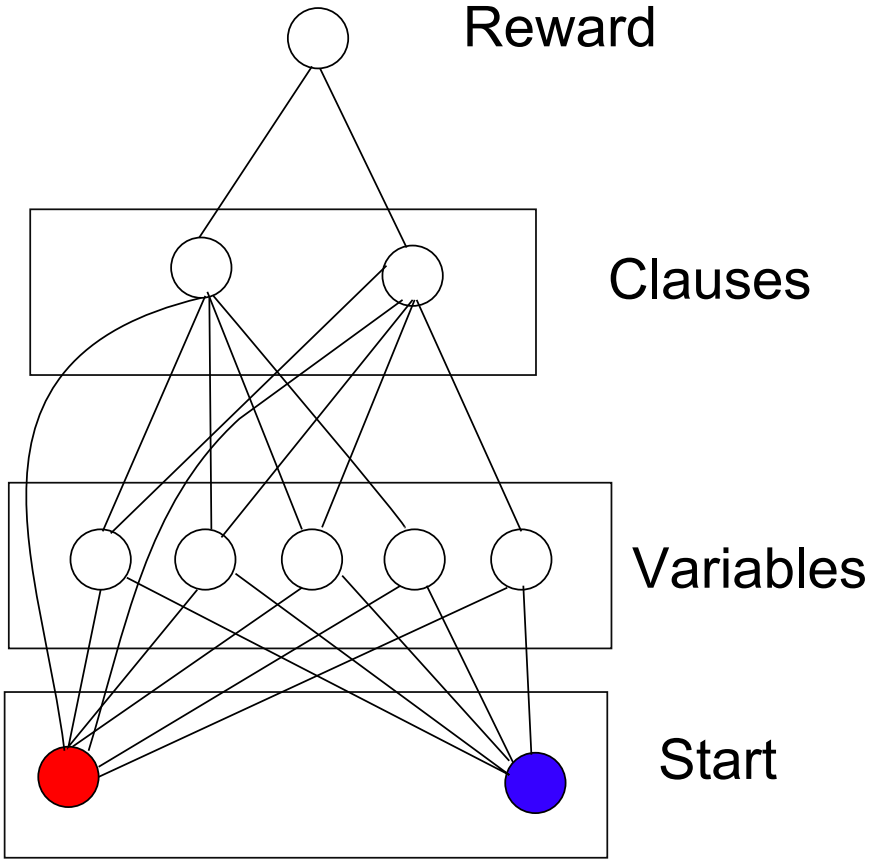
Now we wish to reduce from Reconfigurable Vertex Cover to the original problem. For each vertex v_i, construct a pair of vertices $v_{i,in}$ and $v_{i,out}$ connected by a directed edge from in to out. For each edge (v_i, v_j), add two directed edges: $(v_{i,out}, v_{j,in})$ and $(v_{j,out}, v_{i,in})$. Then for each edge (v_i, v_j) in the original graph, there exists a cycle of length 4: $v_{i,in}$, $v_{i,out}$, $v_{j,in}$, $v_{j,out}$. To remove this cycle, at least one of the four edges in the cycle must be removed. It's most efficient to remove either the edge $(v_{i,in}, v_{i,out})$ or the edge $(v_{j,in}, v_{j,out})$ (since that might eliminate other cycles). Thus a valid cycle-removing set corresponds to a vertex cover of size k or $k + 1$.

Because $v_{i,in}$ has only one outgoing edge, the only cycles involving $v_{i,in}$ must involve the edge from $v_{i,in}$ to $v_{i,out}$. Furthermore, they must involve some edge from $v_{i,out}$ to another node $v_{j,in}$, which in turn forces the cycle to use the edge from $v_{j,in}$ to $v_{j,out}$. So any cycle in the graph must contain the edges $(v_{i,in}, v_{i,out})$ and $(v_{j,in}, v_{j,out})$ where $v_{i,out}$ is connected to $v_{j,in}$ — meaning that v_i and v_j are adjacent in the original graph. Hence, any vertex cover, when convertex to a set of edges to remove, results in the removal of *all* cycles in the graph we've constructed.

Hence, there is a one-to-one correspondence between cycle-removing sets and vertex covers. So because reconfiguring vertex covers is PSPACE-hard, reconfiguring cycle-removing sets is also PSPACE-hard. □

Problem 4. *Expansion* is a two-player game played on a simple, connected graph. Each node is a different 'territory', which can be in one of three states: unowned, owned by Player 1, or owned by Player 2. Initially, each player owns one territory; the remaining territories are unowned. Players take turns claiming ownership of an unowned territory. Each unowned territory v has a number k_v which dictates how many adjacent territories a player must own in order to claim v. Play proceeds alternately until neither player can claim ownership of a territory. The player with the most owned territories at the end of the game wins. Prove that deciding whether Player 1 can win in this game is PSPACE-complete.

Solution: We will prove that deciding if Player 1 can win under optimal play in *Expansion* is PSpace-hard by a reduction from Impartial Game Positive 11-DNF SAT. We construct a single vertex for each variable. The vertex will be considered true if it is controlled by Player 1 and false if controlled by Player 2. The variable nodes have a take-over number of 1, where a take-over number is the number of adjacent, claimed territories needed to claim that territory. They are all connected to each player's starting vertex as well as each vertex representing each clause they are used in. Each clause has a take-over number of 12, one more than the total number of variables attached to it. Every clause is also connected to the reward gadget. The reward gadget is a long path of length r which we will calculate later. An example of the variable and clause gadgets is given in the figure below, although they only contain four variables for clarity. That formula would represent $(x_1 \wedge x_2 \wedge x_3 \wedge x_4) \vee (x_1 \wedge x_2 \wedge x_3 \wedge x_5)$ Player 1's starting node is connected to each variable and each clause. This allows Player 1 to capture any variable from the start of the game, and they can capture a clause if and only if they have capture the three variables connected to that clause. Player 2's start node is also connected to every variable, but it is additionally connected to an advantage gadget, which consists of a path of length $v + 1$ where v is the number of variables in the formula. The advantage gadget ensures that Player 2 can win if Player 1 is not able to capture a clause gadget and thus the reward gadget. Notice, that although Player 2 is able to capture any variable at the start of the game, they will never be able to capture any clause, since each clause is only connected to 11 variable nodes and Player 1's base, but has a take-over number of 12.The variable nodes are the only nodes which can be captured by both players, and thus it is optimal for these to all be taken before any other nodes are captured. Thus optimal play involves the player taking turns capturing variables, which we interpret setting them to true or false. Since it is Positive Game Sat formula, the satisfying player will never want to set a variable false, and the falsifying player will never want to set one true. After all variables are captured, Player 2 is free to take the nodes in the advantage gadget. If Player 1 has captured every variable gadget associated with a clause, they can then take the clause and all the nodes in the variable gadget. If we set that number of nodes r to be $2v + 2$, then Player 1 will be able to win if and only if they manage to satisfy one of the 11-DNF clauses. Thus *Expansion* is PSpace-hard.



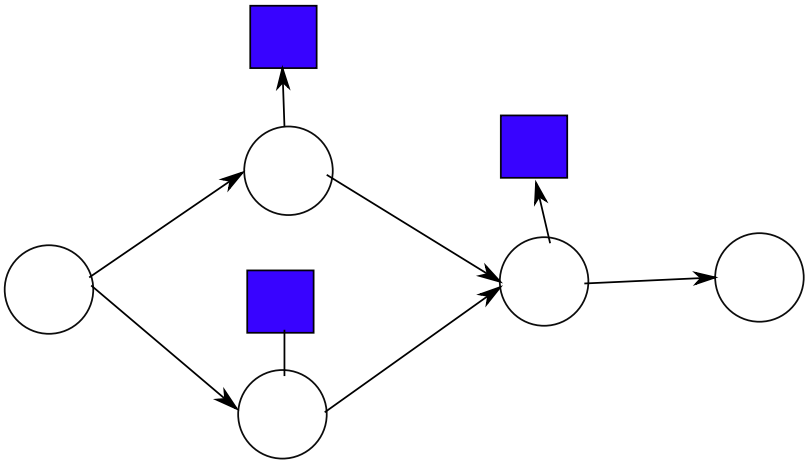
We will show *Expansion* is in PSpace by giving an NPSpace algorithm and noting Savich's Theorem shows NPSpace=PSpace. For a given board position, we non-deterministically consider the moves that player can make. We also note that the board position can always be described in a polynomial amount of space. If any of them end the game we evaluate who won, otherwise we recurse and check the resulting position. Checking for valid moves and whether the game is over can be done easily in polynomial time. Assuming every player captures a territory every turn, then there are a polynomially bounded number of moves in any game and thus a polynomially bounded depth of recursion. We now note that passing when one could have captured a territory is never optimal player. A captured territory adds to a players score, and never prevents them from capturing a different territory in the future that would have been available if they had not captured it. If the opponent would have decided to capture that territory over another, under optimal play it means that territory would be better for the opponent then the other choice. Thus, a player will always capture a territory if possible. Next, we note that if a player is unable to capture a territory at one point in the game, they will never be able to do so later. This follows from the fact that one can only make new territories available by having control of different territories.Thus this problem is in PSpace.

Since *Expansion* is in PSpace and is PSpace-hard, we have proven the game is PSpace-complete. □

Problem 5. *Graph Runner* is a game played on a simple, connected, directed graph $G = (V, E)$ with two players. Player 1 has a token on vertex v_1, the starting vertex. During their turn, Player 1 can move the token along an outgoing directed edge to an adjacent vertex, and gain 1 point; if they

cannot move, the game ends. During Player 2's turn, they can delete an outgoing edge from the node the token is currently on. Prove that it is PSPACE-complete to determine whether Player 1 can achieve a score of at least k, where k is polynomial in $|E|$.

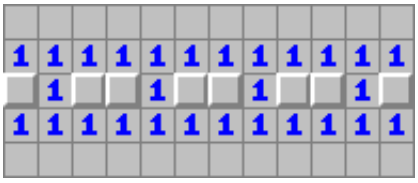
Solution: To show this game is PSpace-hard we will reduce from Positive TQBF. We represent variables with dual-rail logic. Universal quantifiers are represented by a node with two out edges to the true and false nodes. Player 2 can force Player 1 to either of them. Existentials have two out degree to the True and False side, but also have a third edge to a reward gadget. The deterrence gadget is a clique of size $k+1$. Player 2 must block that edge. The variable choice leads down a path which represents each literal. Going down this path will be interpreted as setting the variable false. We must ensure that each choice of the variable or it's negation is the same length, so we potentially pad one side with literal copies that do not connect to any clauses. Each literal has a main node, which we will describe in detail, as well as a single pass edge gadget. This gadget is comprised of two nodes, each connected to the original by an edge. Each of those nodes connects to a deterrence gadget, as well as a third node. The third node connects to the other end of the simulated edge and a reward gadget as shown in the figure below. This means the first time Player 1 visits the starting node, they can always pass over that edge (there is no edge Player 2 can remove to prevent it) and the second time they attempt to cross the edge, they cannot. We connect the last variable to a series of clause gadgets. Each clause gadget consists of a node connected to each of the three literals in the clause and the deterrence gadget. Those literals are then connected to the next clause in the line. The last clause is connected to a reward gadget. Thus if Player 1 has never visited a literal before, Player 2 must remove the edge to the deterrence gadget and Player 1 can proceed to the next clause. If they have visited one, then Player 2 could remove the edge going to the next clause. Thus if at least one literal in a clause is satisfied, Player 1 can traverse it to the next clause. We now define $k' = 3v + 2c + m$ where v is the number of variables, c is the number of clauses, and m is the number of literals. We construct the reward gadget as a clique of size k' and we set the target score equal to $2k' + 1$. Thus, to win, Player 1 must set every variable, go through the clauses, and satisfy all of them. They cannot backtrack through the variable setting, because Player 2 will be able to cut them off at the next section where a variable is set. If they attempt to leave and satisfy clauses early, then they will have skipped setting some variables and cannot achieve a sufficiently high score. This completes the reduction.



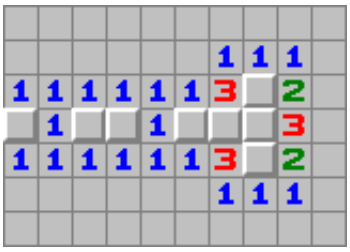
We will show *Graph Runner* is in PSpace by giving an NPSpace algorithm and noting Savich's Theorem shows NPSpace=PSpace. For a given board position, we non-deterministically consider the moves that player can make. This simply requires looking over the edges connected to the node at Player 1's location and thus only requires a polynomial amount of time. We also note that the board position can always be described in a polynomial amount of space. After each move, we evaluate if a player has won, and if not recurse on the new board position. Next we note that if the players each only pass a polynomial number of times, the depth of the recursion is polynomial. Further, we note that under optimal play, neither player will pass. If Player 1 passes, then they will have a subset of the moves that were avalbile to them. Thus if none of those moves were better than passing the first time, they will be no better in the subsequent turn. Since Player 1 want to maximize their score (unless they are already over the winning ammount) then they would prefer to have one more point then zero more points. Thus making any move is better than making no more moves for the rest of the game. Together this shows Player 1 will always move. For Player 2, their move cannot allow Player 1 to create a longer path than if they did not move. Thus, they will always delete an edge under optimal play. Since the game will terminate in a polynomial number of moves, we can now conclude it is in PSpace.

Since *Graph Runner* is in PSpace and is PSpace-hard, we have proven the game is PSpace-complete.

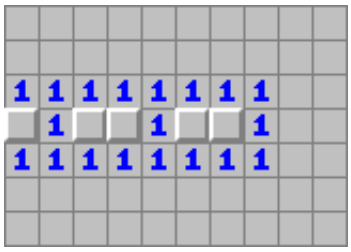
□



(a) A wire.

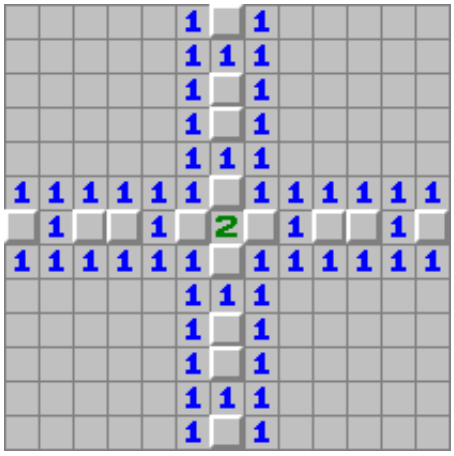


(b) Normal terminator.

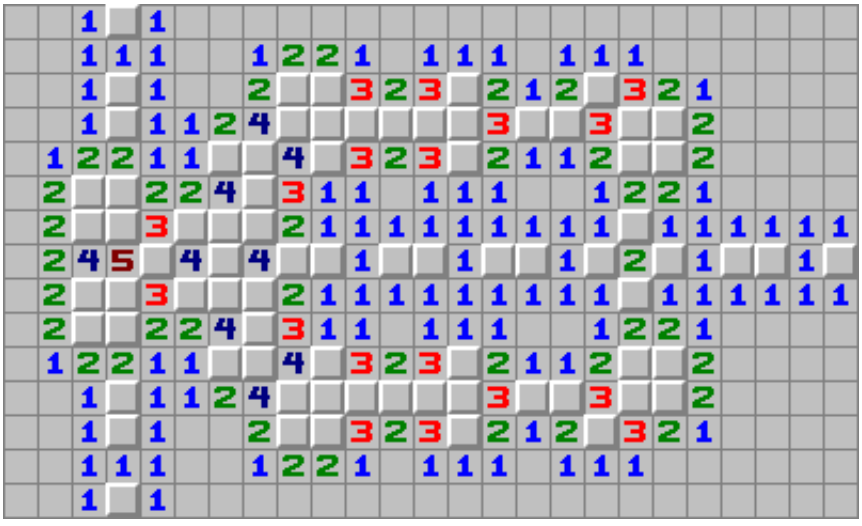


(c) Terminator forcing TRUE.

(d) Wire shifter.

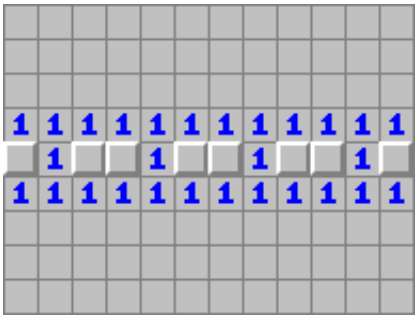


(e) Splitter gadget.

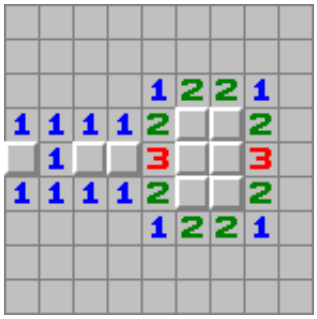


(f) AND gadget.

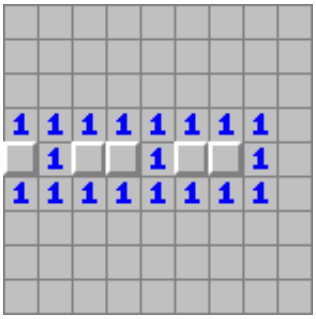
Figure 1: The gadgets used in the Planar Circuit SAT to Minesweeper reduction in Problem 2(i).



(a) A wire.

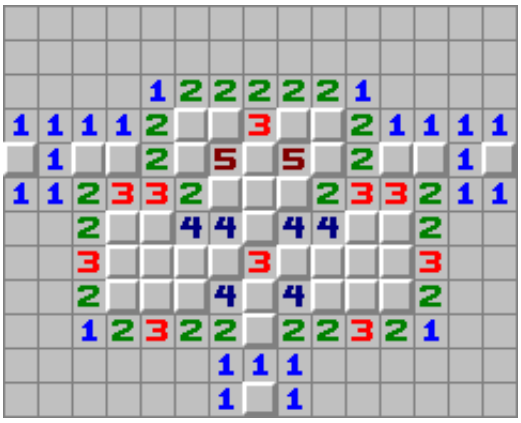


(b) Normal terminator.



(c) Terminator forcing TRUE.

(d) Wire shifter.



(e) Splitter gadget.

(f) OR gadget.

Figure 2: The gadgets used in the Planar Circuit SAT to Minesweeper reduction in Problem 2(ii).

6.890 Algorithmic Lower Bounds: Fun with Hardness Proofs

Fall 2014