Complexity of Planar Graph Orientation Consistency, Promise-Inference, and Uniqueness, with Applications to Minesweeper Variants

MIT Hardness Group[1]
Massachusetts Institute of Technology, Cambridge, MA, USA

Della Hendrickson
Massachusetts Institute of Technology, Cambridge, MA, USA

Andy Tockman
Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

We study three problems related to the computational complexity of the popular game Minesweeper. The first is consistency: given a set of clues, is there any arrangement of mines that satisfies it? This problem has been known to be NP-complete since 2000 [4], but our framework proves it as a side effect. The second is inference: given a set of clues, is there any cell that the player can prove is safe? The coNP-completeness of this problem has been in the literature since 2011 [6], but we discovered a flaw that we believe is present in all published results, and we provide a fixed proof. Finally, the third is solvability: given the full state of a Minesweeper game, can the player win the game by safely clicking all non-mine cells? This problem has not yet been studied, and we prove that it is coNP-complete.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases NP, coNP, hardness, minesweeper, solvability, gadgets, simulation

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.25

Related Version *Full Version*: https://arxiv.org/abs/2404.14519

Acknowledgements This paper was initiated during open problem solving in the MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.5440) taught by Erik Demaine in Fall 2023. We thank the other participants of that class for helpful discussions and providing an inspiring atmosphere.

1 Introduction

The puzzle-based video game Minesweeper was popularized by its inclusion in the default installation of Windows 3.1 in 1992, and to this day is one of the most widely recognizable computer games. Though the premise is very simple, it has spawned an endless stream of spinoffs, and a number of communities dedicated to challenges such as completing a randomly generated game as quickly as possible.

From a computational complexity perspective, Minesweeper is interesting in that there are multiple natural decision problems to study. The simplest is *consistency* ("given some clues, is there a possible arrangement of mines?"), which was proved NP-complete in 2000 [4].

But consistency doesn't capture the essence of Minesweeper as a game, where the player has some partial information and tries to find a cell that they can click on safely, leading to more information. This suggests the *inference* problem ("given some clues, is there a

[1] Artificial first author to highlight that the other authors (in alphabetical order) worked as an equal group. Please include all authors (including this one) in your bibliography, and refer to the authors as "MIT Hardness Group" (without "et al.").

cell that is provably safe?"), which was shown coNP-complete in 2011 [6]. Unfortunately, this proof of coNP-hardness of Minesweeper inference is incorrect, and Thieme and Basten's proof [7], which is designed to use very small gadgets, suffers from the same issue.

While inference asks about a single "turn" of Minesweeper, it is also natural to ask a related question about the entire game. We introduce the *solvability* decision problem, in which we are given both the current state of a Minesweeper game and also the secret arrangement of mines, and asked whether the player can make a sequence of safe clicks to solve the game.

In this paper, we develop a framework based on graph orientation to prove coNP-completeness of Minesweeper inference and solvability. As a side effect, the same gadgets also suffice to prove NP-hardness of Minesweeper consistency. Of particular note, we prove that Minesweeper solvability is coNP-complete even "after a single click": from a nearly empty initial state with only one cell revealed.

Specifically, we define three graph orientation decision problems (consistency, promise-inference, and uniqueness) related to the Minesweeper problems, and show that each is hard with a particular set of simple abstract gadgets. It follows that finding well-behaved constructions in Minesweeper which behave like those gadgets is enough to prove hardness for all three Minesweeper problems.

In Section 2, we define each decision problem carefully, summarize the previously known results, and explain the flaw in the existing reduction for inference. In Section 3, we develop a framework of gadgets that makes it easy to prove hardness results for all three decision problems. In the full version of the paper, we apply the framework to Minesweeper and to many variants of Minesweeper from the video game *14 Minesweeper Variants*. For the most part, each application consists entirely of constructions of the relevant gadgets in the Minesweeper variant under consideration.

2 Prior work and definitions

2.1 Consistency

Research on the computational complexity of Minesweeper began when Sadie Kaye [4] posed the Minesweeper consistency problem. Informally, this problem asks whether a partially completed Minesweeper board has a legal arrangement of mines. We provide a formal definition here.

▶ **Definition 1.** *A **partial board** is a two-dimensional rectangular array, where each entry is either a **covered cell** or an **uncovered cell**. A covered cell is an unknown cell which may or may not be a mine. An uncovered cell has an integer (and is known to not contain a mine), representing a Minesweeper clue. For a partial board B, we denote the set of covered cells $\mathscr{C}(B)$, and the set of uncovered cells $\mathscr{U}(B)$.*

▶ **Definition 2** (Minesweeper consistency problem). *A partial board B is **consistent** if there exists a set $M \subseteq \mathscr{C}(B)$, representing all locations of mines, such that $M \cap \mathscr{U}(B) = \emptyset$ and the integer in each uncovered cell $c \in \mathscr{U}(B)$ counts the number of cells in M which are orthogonally or diagonally adjacent to c. Otherwise, B is **inconsistent**. See Figures 1 and 2. The input to the **Minesweeper consistency problem** is a partial board, and the problem asks whether it is consistent.*

Kaye proves that the consistency problem is NP-complete [4]. We also prove NP-completeness as a side effect of our framework.

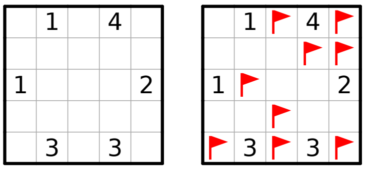


Figure 1 Left: an example of a consistent board. Right: one possible way to satisfy it.

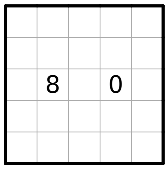


Figure 2 An example of an inconsistent board.

▶ **Proposition 3.** *The Minesweeper consistency problem is in NP.*

Proof. Given a partial board B, a certificate of consistency is the M of Definition 2. We can iterate over $\mathscr{U}(B)$ and check that each clue is satisfied in polynomial time. ◀

We leave the proof of hardness to Section 3.

2.2 Inference

The Minesweeper inference problem was first posed by Allan Scott, Ulrike Stege, and Iris van Rooij [6]. Informally, this problem asks whether a partially completed Minesweeper board has a logical deduction available to the player that lets them click on a cell which is guaranteed to not have a mine. We provide a formal definition of a slight variation on the problem as originally posed.

▶ **Definition 4** (Minesweeper inference problem)**.** *Given a partial board B, an **inference** is a cell $c \in \mathscr{U}(B)$ such that for all consistent arrangements of mines $M \subseteq \mathscr{C}(B)$, we have $c \notin M$. The input to the **Minesweeper inference problem** is a partial board, and it asks whether there is an inference. See Figures 3 and 4.*

Note that this definition has two key differences from the one originally given by Scott, Stege, and van Rooij [6]:
- Deducing the location of a mine does not count as an inference.
- No positions of known mines are given in the input.

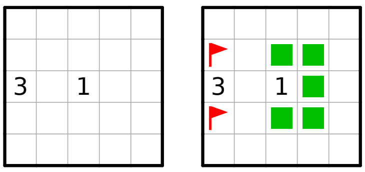


Figure 3 Left: an example of a board with an inference. Right: green squares mark cells that can be inferred.

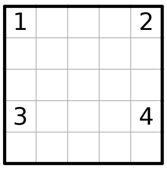


Figure 4 An example of a board that doesn't have an inference.

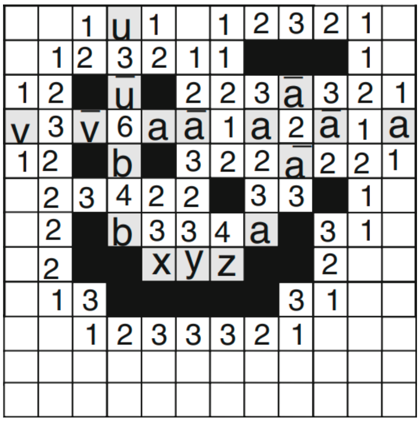


Figure 5 The OR gate from [6] Figure 16. There is a minor typo: the lower cell with b should have $\bar{b}$.

We prefer Definition 4 because it eliminates the need to place conditions on the input such as "all given mines must be deducible from the clues," which is otherwise necessary to avoid placing mines that could never be deduced in a real game. Furthermore, the flags representing known mines can be viewed as simply a player aid – one could in principle play a full game of Minesweeper without marking a single mine. Though we will proceed with Definition 4 only, we remark that all results in this paper work under either definition.

▶ **Proposition 5.** *The Minesweeper inference problem is in coNP.*

Proof. Given a partial board B, a certificate of noninference is, for each cell $c \in \mathscr{C}(B)$, a consistent arrangement of mines M with $c \in M$. We can iterate over the polynomially many arrangements given by the certificate and check consistency in polynomial time. ◀

Again, the proof of hardness will be in Section 3.

Error in existing proofs

Scott, Stege, and Rooij's proof of coNP-hardness [6] has an issue where there is sometimes an unintended inference. Their OR gate is shown in Figure 5. A cell has a mine when the literal marking it is true. The inputs are u and v the output is a. The 6 enforces $a + b = u + v$, and the section at the bottom enforces $a \geq b$. Together this forces $a = u \vee v$ and $b = u \wedge v$.

The issue occurs when we know that u and v can't both be true. This might happen if the OR gate is used inside an AND gate which merges two clauses which can't simultaneously be false, such as if one contains x and the other contains $\neg x$.

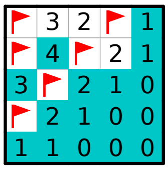


Figure 6 An example of a solvable board. Cyan cells indicate elements of K, and white cells are unknown to the player. Red flags indicate elements of M. From the initial state, the only provably safe cell is the fourth cell in the second row. After clicking it, the player can now deduce the third cell in the first row. Finally, this reveals enough information to deduce the second cell in the first row.

In this case, we can deduce that $b = u \wedge v$ is false, and thus the (higher) cell labeled b is safe to click. This is an inference. One strategy for resolving this issue would be to reveal that cell in the input, assuming one can find all inferences like this. But this doesn't work: That cell has either a 3 or a 4 depending on the value of u, so revealing it tells the player the value of u, allowing them to make more inferences and possibly learning further information.

Thieme and Basten's more compact proof [7] has a very similar issue.

Our approach to avoiding this issue is twofold. meaning clicking an inferred cell never reveals information. This allows us to eliminate inferences by revealing those cells in the input. This handles unintended "local" inferences, which we are able to detect.

Second, to prevent unintended larger-scale inferences, we design the network of gadgets carefully. Specifically, we ensure that for every gadget (OR gate, crossover, etc.), every locally valid combination of values can be achieved. The only exception is the "final" gate, which has a forced output when the input formula is unsatisfiable. We maintain this property across simulations by introducing a problem we call "promise-inference", which also partially relaxes the constraint that every locally valid solution is achievable.

2.3 Solvability

Informally, the Minesweeper solvability problem asks whether a player can make a sequence of inferences from a partially completed Minesweeper game and win by clicking on all non-mine cells. We provide a formal definition of this problem, which to our knowledge has not been studied.

▶ **Definition 6** (Minesweeper solvability problem). *Let B be a partial board with uncovered cells $K = \mathscr{U}(B)$, and let M be a set of mines consistent with B. Note that B is determined by M and K. Here M represents the secret set of mines, which is thought of as unknown to the player, and K represents the set of known cells.*

Consider an ordering O of $G \setminus (M \cup K)$, meaning a list containing each element of $G \setminus (M \cup K)$ exactly once. For an element $o \in G \setminus (M \cup K)$, let $O = O_{init} +\!\!+ [o] +\!\!+ O_{tail}$, where $+\!\!+$ denotes concatenation. We say M is ***solvable*** *from K if there exists an O such that for all $o \in O$, o is a inference for the (unique) partial board consistent with M whose uncovered cells are $K \cup O_{init}$. Otherwise, M is* ***unsolvable*** *from K. See Figure 6.*

The input to the ***Minesweeper solvability problem*** *consists of the dimensions of a rectangular grid G and two disjoint subsets of cells $M \subseteq G$ and $K \subseteq G$. The problem asks whether M is solvable from K.*

Intuitively, solvability simulates a player who is given a Minesweeper puzzle on their computer to solve. If such a player wishes to guarantee that they do not click a mine (to win without any luck, or because they have antagonistic implementation which repositions mines to make the player lose if they click a cell that isn't provably safe, such as "expert mode" in *14 Minesweeper Variants*), they must click covered cells in an order O that satisfies Definition 6.

▶ **Proposition 7.** *The Minesweeper solvability problem is in coNP.*

Proof. Given a game (M, K), a certificate of unsolvability is a set $K' \supseteq K$ together with a certificate of noninference for K'.

Suppose there is such a certificate. As long as the information available to the player is a subset of K', they cannot infer that a cell outside K' is safe. Any order has a first cell outside K', which by assumption is not an inference.

Conversely, suppose an instance is unsolvable. Starting from K, repeatedly make inferences and add them to the known information. By assumption, this must get stuck before all non-mine cells are uncovered, meaning we reach a point with no inferences. Then take K' to be the uncovered cells at that point. ◀

Once again, we prove hardness in Section 3.

3 The Minesweeper gadget framework

In this section, we describe an abstract framework which we will later apply to Minesweeper. The wires in Minesweeper hardness proofs generally have two states, which can be thought of as two orientations of an edge, so our framework is a general form of graph orientation.

▶ **Definition 8.** *A* ***gadget*** *is an abstract object which has*
- *a finite set of* ***ports****, in a specified cyclic order (we will generally list the ports in this order, and describe it more explicitly when relevant).*
- *a* ***constraint****, which is a set of subsets of the ports.*

Gadgets will interact via directed edges connecting ports. The constraint says which sets of edges pointing towards the gadget should be considered legal.

The gadgets we name are collected in Table 1, and will also be described as they come up.

▶ **Definition 9.** *A* ***network*** *of gadgets from a set S is an undirected graph where*
- *each vertex is labeled with a gadget from S*
- *if $G \in S$ has k ports, each vertex labeled G has degree k and its edge incidencies are labeled in a bijection with ports of G.*

A ***planar network*** *is such a graph equipped with a planar embedding, such that the cyclic order of edges around each vertex matches the order of the ports of the corresponding gadget.*

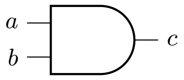
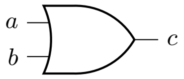
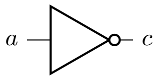
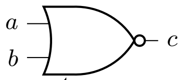
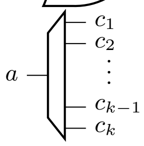
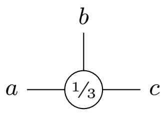
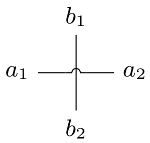
We often equivocate between vertices and gadgets, and between edge incidencies and ports – think of each vertex as a copy of its label, and think of edges as connecting ports to ports in a matching.

We draw planar networks using the icons in Table 1, which also indicate the correspondence between edges and ports (except for when it doesn't matter by symmetry).

▶ **Definition 10.** *An* ***assignment*** *to a (planar) network assigns a direction to each edge. A vertex is* ***satisfied*** *if the set of (labels of) edges pointing into it is in its (label's) constraint. An assignment is* ***satisfying*** *if every vertex is satisfied.*

Planar Graph Orientation (PGO) is the study of satisfying assignments of planar networks.

Table 1 The gadgets we define in this paper, the icons we use to draw them (for those we show in networks), and their constraints.

Name	Icon	Constraint
fixed (true) terminal	a — F	$\{\{\}\}$
fixed (false) terminal	a — T	$\{\{a\}\}$
free terminal	a —	$\{\{\},\{a\}\}$
AND gate		$\{\{c\},\{a,c\},\{b,c\},\{a,b\}\}$
OR gate		$\{\{c\},\{a\},\{b\},\{a,b\}\}$
NOT gate		$\{\{\},\{a,c\}\}$
NOR gate		$\{\{\},\{a,c\},\{b,c\},\{a,b,c\}\}$
(k-way) fanout gate		$\{\{a\},\{c_1,\dots,c_k\}\}$
1-in-3 gadget		$\{\{a\},\{b\},\{c\}\}$
crossover		$\{\{a_1,b_1\},\{a_2,b_1\},\{a_1,b_2\},\{a_2,b_2\}\}$

3.1 Gates as gadgets

One important kind of gadget is logic gates, which can be interpreted as gadgets: inputs are edges entering from the left and output are edges exiting on the right. Interpret pointing right as true. The gadget's constraint allows the inputs to be arbitrary, but forces the correct outputs for each combination of inputs. More precisely, for each subset S of input ports, the constraint contains $S \cup T$, where T is the set of outputs that are *false* when precisely the inputs in S are true.

It's important to keep in mind distinction between gate gadgets, which compute a value, and "normal" gadgets, which enforce a constraint.

For example, the ***OR gate*** has ports $\{a, b, c\}$ and constraint $\{\{c\}, \{a\}, \{b\}, \{a, b\}\}$. This constraint allows any subset of the input ports a and b to have edges pointing in, and enforces that the edge incident to the output port c to point out exactly when at least one input port points in. In other words, it computes the OR of a and b, and outputs it at c.

On the other hand, the ***OR gadget*** (which we don't need beyond this example) has only two ports $\{a, b\}$ and constraint $\{\{a\}, \{b\}, \{a, b\}\}$. It enforces that at least one edge points in, or that at least one input is true. This is equivalent to an OR gate with the output forced to be "true" (pointing away).

Any boolean circuit can be represented as a network of gate gadgets, by attaching input ports to output ports in the natural way. Allow us to leave inputs and outputs to the full circuit as "dangling" edges for now. It follows from the definition of gate gadgets – formally, one can induct on the depth of a gate – that this network has exactly one satisfying assignment for each combination of orientations of the dangling input edges, and in each the orientations of the output edges encode the output of the circuit (and internal edges encode the internal state of the circuit).

If an output connects to $k > 1$ inputs, we need to use a k-way ***fanout gate***, which has ports $\{a, c_1, \ldots, c_k\}$ and constraint $\{\{a\}, \{c_1, \ldots, c_k\}\}$. This is the gadget representing the gate that duplicates its input k times.

A drawing of a circuit in the plane becomes a planar network of gate gadgets. A crossing pair of wires is translated to the ***crossover gate***, which has two inputs and two outputs matching the inputs in the opposite order. As a gadget, the crossover gate has ports $\{a_1, b_1, a_2, b_2\}$, in that cyclic order, and constraint $\{\{a_1, b_1\}, \{a_1, b_2\}, \{a_2, b_1\}, \{a_2, b_2\}\}$. One can think of it as two wires $a_1 \to a_2$ and $b_1 \to b_2$ that cross in a circuit. However, the directions of these wires are arbitrary because the gadget is highly symmetric. As a gadget outside the perspective of networks of logic gates, the crossover gate is two edges that can independently be assigned orientations, and which cross each other. So we will also call it simply the ***crossover***.

3.2 Decision problems

We consider three decision problems about planar graph orientation, corresponding to the Minesweeper decision problems we're interested in. Each of them is parameterized by a set of gadgets S – throughout we consider only finite sets of gadgets – and takes as input a planar network N of gadgets from S.

The problem corresponding to Minesweeper consistency is straightforward.

▶ **Problem 11.** PGO ***consistency*** with S asks whether N has a satisfying assignment.

For Minesweeper inference, we need to avoid a subtle issue, which is the error in prior claims of coNP-hardness [6, 7]. If there is a gadget for which we can deduce that some legal combination of edge orientations can't be extended to a full satisfying assignment, this information may allow us to infer the value of a Minesweeper cell internal to the gadget. This can happen even if we can't deduce the orientation of any particular edge, so the most obvious PGO inference problem fails to reduce to Minesweeper, and thus isn't useful.

Our strategy for resolving this issue will ultimately be to "click" all cells in the Minesweeper instance that could be deduced in this way. To make this work, we will need the values of those cells to not reveal any additional information (a property we will call "silence"), and we need the reduction to find all such cells in polynomial time. For our gadget framework to help here, we need to define the inference problem carefully, and in particular it needs to be aware of which combinations of edges are ruled out by "semilocal" deductions.

▶ **Problem 12.** PGO ***promise-inference*** with S is given a network N as well as, for each vertex in N, a ***semilocal constraint*** which is a subset of its constraint. We say an edge e is ***locally forced*** if the semilocal constraint of one of its vertices requires it has a particular orientation (because it's either in every element or in no element), and ***locally free*** otherwise.

We are promised that

- in every satisfying assignment, the set of edges pointing into a vertex is in its semilocal constraint; and
- either
  - there is a locally free edge to which every satisfying assignment assigns the same direction; or
  - for each vertex v and set of edges in its semilocal constraint, there's a satisfying assignment that makes exactly those edges point towards v.

We are asked to determine whether there is an edge whose orientation can be inferred but isn't immediate from semilocal constraints; that is, whether we are in the first option above.

Note that the two options can't simultaneously be true, since the existence of such an edge would imply that its incident vertices' semilocal constraints have elements that can't be extended to satisfying assignments.

The intent of semilocal constraints is to be between the levels of individual gadgets' "local" constraints and "global" deductions that require understanding the entire network. A semilocal constraint typically contains the sets of in-pointing edges that are attainable when looking at some constant-size neighborhood of a gadget.

The first part of the promise ensures that semilocal constraints are actually enforced by the structure of the network. In the case where there's no inferable edge, the second part says that it isn't possible to deduce more about the immediate neighborhood of a gadget than its semilocal constraint.

Finally, the PGO decision problem analogous to Minesweeper solvability is simpler.

▶ **Problem 13.** PGO ***uniqueness*** with S is given N as well as a satisfying assignment of N, and asks whether it is the unique satisfying assignment.

PGO uniqueness is related to Minesweeper solvability because it's possible to deduce the orientations of all edges if and only if there's a unique satisfying assignment.

▶ **Lemma 14.** *For any set S of gadgets,*
1. *PGO consistency with S is in NP.*
2. *PGO promise-inference with S is in promise-coNP.*
3. *PGO uniqueness with S is in coNP.*

Proof. Each problem has a straightforward certificate:
1. A satisfying assignment serves as a certificate of consistency.
2. A certificate that there is no inference consists of, for locally free edge and each orientation, a satisfying assignment that assigns the orientation to the edge.
3. A second satisfying assignment serves as a certificate of nonuniqueness. ◀

3.3 Hardness

We now prove hardness of each PGO decision problem for the appropriate class, with a specific set of gadgets. The gadget sets are chosen to make the hardness proofs simple; there are easy ways to reduce the number of different gadgets needed, and we will further simplify our gadget sets using simulation in Section 3.5. With Lemma 14, we have completeness in each case.

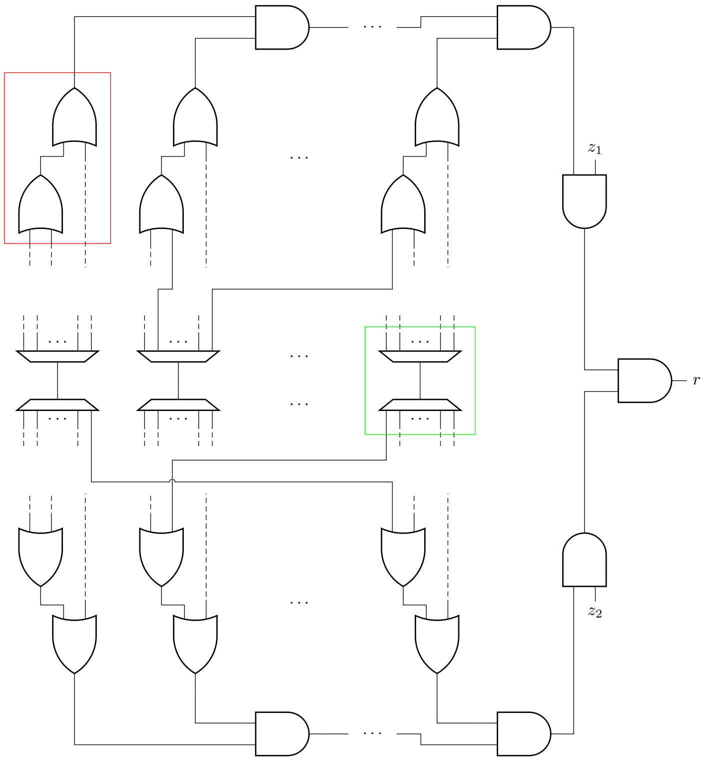


Figure 7 Our reduction for coNP-hardness of PGO promise-inference. See Table 1 for gadget notation.

▶ **Theorem 15.** *PGO satisfiability with free terminals, fanout gates, and 1-in-3 gadgets is NP-hard.*

Some of these gadgets are new: the ***1-in-3 gadget*** has three ports, and its constraint says that exactly one of them must point in. The ***free terminal*** has one port, which is allowed to point in either direction.

Proof. We reduce from planar positive 1-in-3SAT, which is NP-hard [5]. Each variable with k occurrences becomes a free terminal that leads to a k-way fanout gate. Each clause becomes a 1-in-3 gadget. We connect the outputs of the fanout gates to 1-in-3 gadgets as in the 1-in-3SAT formula, which is planar. Satisfying assignments of this network correspond to satisfying assignments of the formula. ◀

▶ **Theorem 16.** *PGO promise-inference with free terminals, fanout gates, crossovers, OR gates, and AND gates is coNP-hard.*

Proof. We reduce from the complement of monotone 3SAT, which is NP-hard [1]. The layout of gadgets is depicted in Figure 7. The green outline is an example of a variable, and the red outline is an example of a clause. We think of the top half of the circuit as the positive clauses, and the bottom half of the circuit as the negative clauses. We remove duplicate clauses before constructing the circuit. Each gadget has as its semilocal constraint its full constraint set.

The output of the formula is given by r, which is connected to a free terminal. If the formula is unsatisfiable, there is an inference; namely, r is false (points towards the AND gate). So we just need to show that if the formula is satisfiable, there is no inference; that is, for each gadget in the reduction, every set of edges in its constraint is possible to achieve (pointing in) in some consistent assignment. Because each gadget is a gate, this is equivalent to every combination of input values being attainable.

There is a unique consistent assignment for each choice of values for variables, z_1, and z_2. We will describe such a choice that achieves each combination of input values for each gadget.

Each fanout depends on a single variable, which can have either value. Each crossover is between two wires connected to different variables, so all combinations are possible. The inputs of each OR gate in a clause can be chosen by setting the input variables as desired.

Consider any of the AND gates combining the outputs of the clauses on one half of the circuit, and assume for simplicity that it's on the positive half. We can make both inputs true (pointing in) or both inputs false (pointing out) by setting all variables true or all variables false, respectively. To make exactly one input true, note that we can choose a single (positive) clause to be unsatisfied: make the three variables in that clause false and all other variables true. Then the clause is unsatisfied, but (since clauses have exactly three literals and there aren't duplicates), every other positive clause is satisfied. By choosing a clause that feeds into either side of the AND gate in question, we can make either of its inputs false and the other true. The same argument shows that the AND gates with z_1 and z_2 can also have any combination of inputs.

Finally, consider the rightmost AND gate, with output r. By hypothesis, the formula is satisfiable, so both inputs can be made true by satisfying the formula and setting z_1 and z_2 to both be true. We can then independently choose to make either input false by flipping z_1 or z_2 as appropriate. ◀

▶ **Theorem 17.** *PGO uniqueness with free terminals, fanout gates, crossovers, NOT gates, OR gates, AND gates, and fixed terminals is coNP-hard.*

The ***fixed terminal*** has one port, and its constraint forces the port's value. There are two kinds of fixed terminal; we will use the ***fixed true terminal***, which forces the edge to point in.

Proof. We reduce from the complement of 3SAT, which is NP-hard [3]. Refer to Figure 8. Each variable becomes an edge connected to a fanout gate pointing down: that edge pointing down represents the variable being true. Each clause becomes a pair of OR gates connected in the natural way. We place edges connecting variables to clauses in the structure of the formula. If edges cross, we use a crossover gadget. For negated literals, we place a NOT gate along the edge.

The outputs of the clauses are merged with AND gates, so the edge in the bottom right of Figure 8 points right (and up) exactly when the assignment (based on the orientations of edges representing variables) is satisfying.

In the top section of Figure 8, we merge the other ends of the variable edges with AND gates. The top right edge points right (and down) exactly when all variables are false.

On the right, there is an edge z which points towards one of two AND gates, with the results merged by an OR gate and then run into a fixed true terminal. For the output of that OR gate to be true (point right), either

- z points up, and all variables are false; or
- z points down, and the 3SAT formula is satisfied.

Note that the orientations of all edges are uniquely determined by those of z and variables, even ignoring the fixed terminal. In particular, the network has exactly one satisfying assignment (with z pointing down) for each satisfying assignment of the 3SAT formula, plus exactly one more, which has z pointing up and all variable edges pointing up.

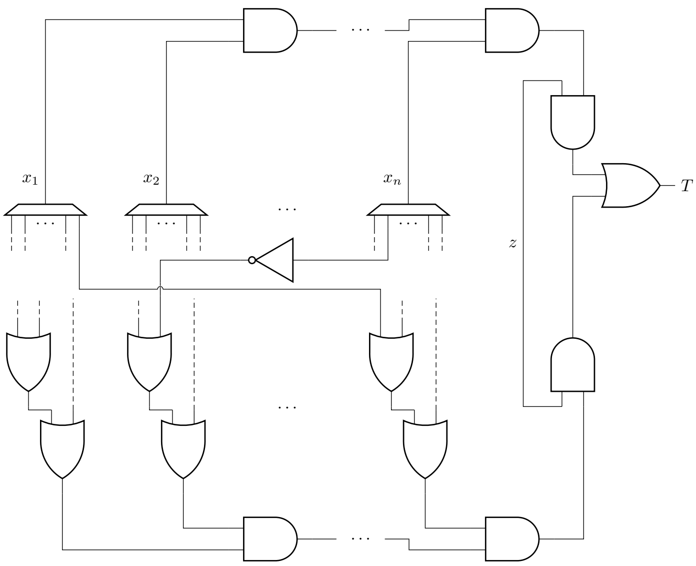


Figure 8 Our reduction for coNP-hardness of PGO uniqueness. See Table 1 for gadget notation.

The input to PGO uniqueness is the network described above and the satisfying assignment with z pointing up. This is the unique satisfying assignment exactly when the 3SAT formula is not satisfiable. ◀

3.4 Simulation

A key feature of our framework is that it allows us to abstractly construct gadgets out of other gadgets, greatly reducing the complexity of the gadgets we need to actually implement in Minesweeper. In particular, the results in Section 3.3 use many gadgets, some of which are hard to build directly, particularly with the properties our hardness proofs require.

▶ **Definition 18.** *A **simulation** using gadgets from S is a network of gadgets from S, except it may have some "dangling" edges incident to only one vertex. Equivalently, the graph contains one special vertex called the "outside world" (which has the trivial constraint).*

*For **planar** simulations, we require that the dangling edges are in the external face, or equivalently that the graph including the outside world is planar.*

See Section 3.5 for several examples of simulations.

▶ **Definition 19.** *Given a simulation, the **simulated gadget** has dangling edges as ports, and its constraint contains each set of dangling edges for which there is a satisfying assignment making exactly those edges point into the simulation (away from the outside world).*

In the planar case, the order of the ports is the order of the dangling edges around the simulation, or the reverse of their order around the outside world.

▶ **Definition 20.** *We say S **simulates** G if there is a simulation using gadgets from S where the simulated gadget is G. For a set T, we say that S **simulates** T if S simulates each gadget in T.*

For PGO uniqueness and Minesweeper solvability, we will need our simulations to be even better behaved.

▶ **Definition 21.** *A simulation of G is* ***parsimonious*** *if for each legal configuration of the edges of G, there is exactly one satisfying assignment of the simulation that orients the dangling edges that way.*

We say that S ***parsimoniously*** *simulates G if the relevant simulation is parsimonious, and S* ***parsimoniously*** *simulates T if S parsimoniously simulates each element of T.*

Much of the point of having a theory of simulations is that they can be composed. This reduces conceptual complexity by letting us break down complicated simulations into a sequence of simpler ones.

▶ **Lemma 22.** *If S simulates T and T simulates G, then S simulates G. Moreover, this composition preserves parsimony.*

Proof. In the simulation of G using gadgets from T, replace each gadget with its simulation using gadgets from S. In any satisfying assignment of the new simulation, each component simulation is consistent with the gadget it's simulating, so we can construct a satisfying assignment of the original simulation with the same orientations for dangling edges by looking at only the edges between simulated gadgets.

Conversely, any satisfying assignment of the original simulation can be extended to one of the new simulation by filling in each simulated gadget with an appropriate satisfying assignment. Thus we have a simulation of G using gadgets from S.

If each gadget is replaced with a parsimonious simulation, there is only one way to fill in simulated gadgets this way. So we have defined a bijection between satisfying assignments of the original and new simulations of G. If the original is parsimonious, so is the new simulation. ◀

The other half of the point of simulations is to simplify hardness proofs, so we need them to preserve hardness. This is straightforward for satisfiability.

▶ **Lemma 23.** *If S simulates T, then there is a polynomial-time reduction from PGO satisfiability with T to PGO satisfiability with S.*

Proof. Given a network N of gadgets from T, replace each gadget with a (constant-size) simulation using gadgets from S to construct a network N' of gadgets from S.

If there is a satisfying assignment of N', looking only at the edges connecting simulated gadgets gives a satisfying assignment of N.

If there is a satisfying assignment of N, we can set the edges connecting simulated gadgets to match it, and then each simulated gadget has a local solution compatible with those edges. ◀

This is somewhat more complicated for promise-inference, but it's not so bad with the right definition for the decision problem.

▶ **Lemma 24.** *If S simulates T, then there is a polynomial-time reduction from PGO promise-inference with T to PGO promise-inference with S*

Proof. We are given an instance of promise-inference with T, which is a network N of gadgets from T and semilocal constraints. Construct a network N' of gadgets from S in the same way as above.

To complete the instance of PGO promise-inference with S, we must define semilocal constraints for the vertices in N'. Consider a vertex v, which is inside a simulation of $G \in T$. The semilocal constraint of v shall contain the sets the of edges that point into v in satisfying assignments of the simulation that are compatible with the semilocal constraint of G (as a vertex of N). These semilocal constraints can be computed in polynomial time because each one requires considering only a simulation, and simulations have constant size.

It remains to check that this instance satisfies the promise, and falls into the same option as the input instance. For the first part of the promise, consider a vertex v in N' and a satisfying assignment. Then v is inside a simulation of some $G \in T$, the corresponding assignment of N (which has only edges between simulated gadgets) is compatible with the semilocal constraint of G. So we have a satisfying assignment of the simulation of G compatible with the semilocal constraint of G, and thus by construction it is compatible with the semilocal constraint of v.

Suppose N has a locally free edge whose orientation is forced. Since every satisfying assignment of N' contains a satisfying assignment of N in the inter-simulation edges, the corresponding edge of N' also has forced orientation. It must also be locally unforced: each orientation is compatible with the semilocal constraints of its vertices in N, and therefore each orientation is compatible with some appropriate satisfying assignment of the simulations of its vertices. Hence N' also has a locally free edge with forced orientation.

Now suppose N falls into the second option, namely there are satisfying assignments achieving every element of its semilocal constraints. Consider a vertex v in N' which is inside a simulation of G, and consider a set L in the semilocal constraint of v. By definition, there is a satisfying assignment of the simulation which makes exactly the edges in L point towards v, and which is compatible with the semilocal constraint of G as a vertex of N. By assumption, there is a satisfying assignment of N which orients the dangling edges of the simulation in the same way. Combining these, and filling in the assignment for other simulations, we obtain a satisfying assignment of N' which directs exactly the edges in L towards v, as desired. ◀

To prove an analogous result for PGO uniqueness, we need our simulations to preserve unique solutions. That is, they should be parsimonious.

▶ **Lemma 25.** *If S parsimoniously simulates T, then there is a polynomial-time reduction from PGO uniqueness with T to PGO uniqueness with S*

Proof. We are given a network N of gadgets from T and a satisfying assignment A. Construct a network N' of gadgets from S in the same way as the two previous proofs – replace each gadget with its simulation.

Our instance of PGO uniqueness with S also needs a satisfying assignment of N'. To construct one, direct inter-simulation edges to match A, and extend this to a full satisfying assignment A' by consistently orienting edges inside simulations. Since the simulations are parsimonious, there is a unique way way to do this for each simulation, and since simulations are constant-size, A' can be computed in polynomial time.

As before, there is a correspondence between satisfying assignments of N and satisfying assignments of N'. This time, however, parsimony ensures that the correspondence is a bijection, since A' is well-defined. In particular, there is a satisfying assignment of N other than A if and only if there is a satisfying assignment of N' other than A'. ◀

3.5 Simpler gadget sets

Now we put the theory of simulations into practice: we will demonstrate several simulations and use them with the results of Section 3.4 to improve the results of Section 3.3 to use more convenient sets of gadgets. The simulations we use are summarized in Figure 9.

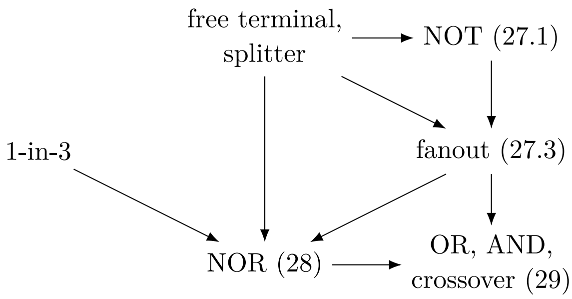


Figure 9 The (parsimonious) simulations we use to simplify our gadget set. Each gadget is simulated by the collection of gadgets pointing towards it, except that we will need to build 1-in-3 gadgets, free terminals, and splitters directly.

When we build gadgets in variants of Minesweeper, we will construct gadgets that are like the fanout gate, but may have more than three ports and may have some ports reversed.

▶ **Definition 26.** *A* ***generalized fanout*** *is a gadget with at least three ports whose constraint has exactly two sets, which are compliments. That is, it has two legal configurations, which differ by flipping all edges.*

For instance, the fanout gate is a generalized fanout, and the NOT gate is almost a generalized fanout, except it has only two ports.

▶ **Lemma 27.** *Let F be a generalized fanout. Then F and free terminals parsimoniously simulate*

1. *the NOT gate*
2. *any generalized fanout F'*
3. *fanout gates (with any number of outputs).*

Proof. We describe each simulation.

1. Let F have ports P and constraint $\{T, F\}$, where $T \sqcup F = P$. Since $|P| \geq 3$, at least one of T and F has size at least 2; assume without loss of generality that $|T| \geq 2$, and $a, b \in T$. Attach free terminals to all ports of F other than a and b. This simulation has two satisfying assignments, corresponding to T and F. The simulated gadget has ports a and b, and constraint $\{\{a, b\}, \emptyset\}$, so it is the NOT gate.
2. Connect copies of F in a tree until there are at least as many dangling edges as ports of F'. Note that there are exactly two satisfying assignments; all copies of F must flip state together. Now assign each port of F' to a dangling edge, respecting cyclic order. If there are extra dangling edges, put free terminals on them.
   The result is a generalized fanout with the same ports as F', but the partition into the two legal configurations may be wrong. For each port on the wrong side of the partition, attach a NOT gate (composing simulations using Lemma 22). This changes which of the two satisfying assignments has the edge at that port pointing in. Now the simulated gadget partition its ports into two legal configurations in the same way as F', so it is in fact F'.
3. This is a special case of the above. ◀

▶ **Lemma 28.** *1-in-3 gates, free terminals, and fanouts parsimoniously simulate the NOR gate.*

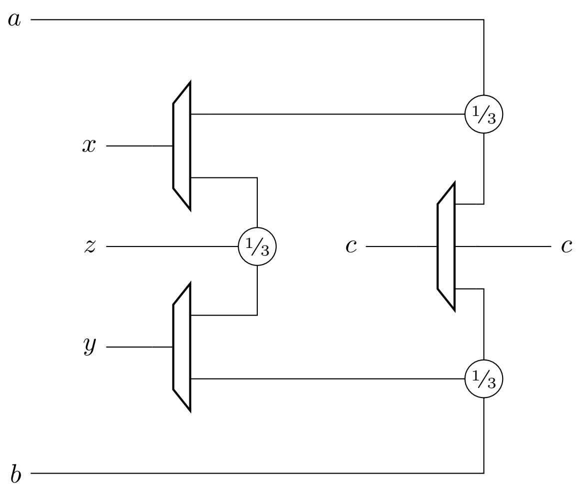


Figure 10 A simulation of a NOR gate using 1-in-3 gadgets, free terminals, and fanouts.

Proof. The simulation consists of three 1-in-3 gadgets, four free terminals, two 2-way fanouts, and a 3-way fanout, and is shown in Figure 10. The inputs are a and b and the output is c. True means "pointing right", for inputs, outputs, and labeled free terminals.

The simulation enforces a 1-in-3 constraint on each of $\{a, x, c\}$, $\{b, y, c\}$, and $\{x, y, z\}$. The easiest way to verify that this is a parsimonious simulation of a NOR gate is to list all satisfying assignments:

a	b	x	y	z	c
T	T	F	F	T	F
T	F	F	T	F	F
F	T	T	F	F	F
F	F	F	F	T	T

We see that c is always $\neg(a \vee b)$, and there is a unique assignment for each combination of values for a and b. When comparing against the definition of the NOR gate in Table 1, recall that the correspondence between true/false and in/out is reversed for c relative to a and b. ◀

▶ **Lemma 29.** *NOR gates and fanout gates parsimoniously simulate OR gates, AND gates, and crossovers.*

Proof. In Section 3.1 we observed that boolean circuits can be converted to gadget networks; these networks can be thought of as parsimonious simulations. The NOR operation is logically complete and can build crossovers in planar circuits [2]. Hence for each gate we want to simulate, we can find a planar circuit of NOR gates that computes it, then convert this into a planar network of NOR-gate gadgets and fanout-gate gadgets. ◀

We now pull everything together by applying the results on simulation in Section 3.4 to the hardness results of Section 3.3, using the simulations in this section. We can either compose reductions or compose simulations using Lemma 22; either way, we obtain our main result about planar graph orientation decision problems.

▶ **Corollary 30.** *PGO consistency with any generalized fanout, fixed terminals, free terminals, and 1-in-3 gadgets is NP-hard. PGO promise-inference and PGO uniqueness with the same set of gadgets are coNP-hard.*

3.6 Applying the framework

We now conclude the development of our gadget framework by discussing what it takes to use it to prove hardness for a game like Minesweeper. We do not attempt to precisely define "like Minesweeper", and this results of this section are not stated precisely. Ultimately, one needs a polynomial-time reduction from the appropriate PGO problem to the corresponding question for the game under consideration, but the games we will consider are similar enough that the reductions share most of their structure, and the discussion here applies to many or all of them.

It is conceptually helpful to distinguish between gadgets as abstract formal objects and the constructions we build in concrete games to embed gadgets in them. We call the latter "implementations", to parallel the distinction between the specification and the implementation of a function.

▶ **Definition 31.** *An **implementation** of a gadget is a construction in a game like Minesweeper which behaves like the gadget, in that there are covered cells (usually on the boundary of the construction) corresponding to ports, and the construction has a solution exactly when the configuration of edges represented by the values of those cells is legal for the gadget.*

Unfortunately, because it matches how the term is typically used, we will frequently call implementations "gadgets" when the intended meaning is clear from context.

For implementations to interact in a way that simulates gadget networks, we need some kind of ***wire***. Each of our wires will have two possible states, corresponding to the two orientations of the edge it represents. We must be able to route our wires in an arbitrary planar graph – the ability to extend and turn them is sufficient. We need to be able to plug wires into the ports of implementations, which sometimes requires the ability to adjust the alignment of a wire by a single cell.

Our wires and implementations also need to have the following properties.

- Every solution uses the same number of mines. This means the total number of mines in the puzzle doesn't reveal any information that might break our reduction.
- Every uncovered cell has a clue, as in most implementations of Minesweeper. Some implementations and variants allow cells that are known to be safe but don't have additional information (e.g. showing a question mark instead of a number), but we don't want to rely on this feature.
- All given mines can be deduced from uncovered non-mine cells. This provides robustness against changes in the definition, e.g. whether the locations of mines can be provided as part of a puzzle in addition to uncovered cells without mines.

Implementing some gadgets in a game allows us to a build network of those gadgets in the game. The resulting instance of the game has a solution if and only if the network is satisfiable.

▷ Claim 32. If a game like Minesweeper has implementations of the gadgets in S, then there is a polynomial-time reduction from PGO consistency with S to the game's consistency problem.

Note that our definition of Minesweeper consistency in Section 2 is specific to Minesweeper. For other games or variants, "partial board" and "consistent" need to be defined appropriately, but the other definitions (for inference and solvability) work as is.

For the other two decision problems, we need well-behaved implementations.

▶ **Definition 33.** *An implementation is **silent** if clicking a known-safe internal cell can never reveal information that wasn't already known. In other words, for each covered internal cell, the information that cell provides when clicked (e.g. the number of adjacent mines) is the same in every solution in which it doesn't have a mine.*

▷ Claim 34. If a game like Minesweeper has silent implementations of the gadgets in S, then there is a polynomial-time reduction from PGO promise-inference with S to the game's inference problem.

Proof. Embed a network of gadgets from an instance of PGO promise-inference with S in the game. For each (constant size) gadget in the network, consider all solutions to its implementation which are consistent with its semilocal constraint. If there are ***commonalities***, or cells which are safe in all such solution (or mines in all such solutions), "click on" them, revealing them in the instance of the game. Silence guarantees that the information revealed is the same in all such solutions, so the information to put on that cell can be determined from the semilocal constraint. If a commonality dictates the orientation of an edge, we click on all covered cells in the edge in the same way. It doesn't matter whether this includes the cell(s) representing the port on the other end of the edge – if that gadget's semilocal constraint doesn't force the edge in the same way, we must be in the first case of the promise so there's an inferable locally unforced edge anyway.

If the network has an inferable locally unforced edge, then cells inside the wire representing that edge (or cells representing the ports that edge connects to) can be inferred.

Otherwise, we are in the second case of the promise. We've already clicked all cells that can be deduced from each semilocal constraint – for any cell that is still covered, the gadget (or wire) it's in has solutions compatible with its semilocal constraint in which that cell has a mine and in which it doesn't. Thus the entire instance also has both of these kinds of solutions because we are promised that every element of a semilocal constraint is achieved by a satisfying assignment. That is, the cell in question can't be inferred. This is where it is crucial that we use promise-inference, and not a simpler PGO inference decision problem. ◁

For PGO uniqueness and Minesweeper solvability, we also need parsimony as we did for Lemma 25.

▶ **Definition 35.** *An implementation is **parsimonious** if it has exactly one solution corresponding to each legal configuration of the gadget.*

▷ Claim 36. If a game like Minesweeper has silent parsimonious implementations of the gadgets in S, then there is a polynomial-time reduction from PGO uniqueness with S to the game's solvability problem.

Proof. Embed a network N of gadgets from an instance of PGO uniqueness with S in the game. Thanks to parsimony, satisfying assignments of the network are in bijection with consistent solutions to this instance of the game. The secret solution is the solution corresponding to the given satisfying assignment of N, but cells are only uncovered if they are uncovered in the embedding of N, which doesn't depend on the satisfying assignment.

If N has another satisfying assignment, the game instance has multiple solutions. The player may be able to safely click some cells – perhaps the orientation of some edge is forced, or they can deduce that a cell inside a gadget is safe. However, since our implementations are silent, the player can't gain any information by doing this. In particular, all solutions consistent with the initial state of the game are also consistent after the player makes any sequence of safe clicks. In order to solve the instance, they would need to distinguish between these consistent solutions, which is impossible.

Conversely, suppose the only satisfying assignment to N is the one given to the reduction. Then the secret solution is the only solution consistent with the initial state of the game (we need parsimony to ensure there is only one). Thus the value of every cell can be deduced, and all cells without mines can be safely clicked in any order. ◁

It follows that if we can silently and parsimoniously implement that gadgets needed by Corollary 30, we have hardness of consistency, inference, and solvability for the game in question. To simplify a little further, fixed terminals are trivial to construct: just let a wire end, and reveal some cells in it to force its orientation. Alternatively, modify a free terminal (possibly including a bit of wire) by revealing cells that determine its configuration.

▶ **Corollary 37.** *Suppose that for some game like Minesweeper, we have silent parsimonious implementations of any generalized fanout, free terminals, and 1-in-3 gadgets, and we are able to route, turn, and adjust the alignment of wires enough to embed any planar network of those gadgets. Then consistency is NP-hard, and inference and solvability are coNP-hard.*

We now go through one final layer of abstraction, which will handle routing of wires and filling empty space. We lay a network of the gadgets above on a square grid, where edges run horizontally and vertically and can turn. In particular, each tile contains either one of the gadgets above, a straight edge section, and turning edge section, or nothing.

This lets us reduce from problems about planar networks on grids, so we can construct straight and turning tiles instead of describing general wire routing.

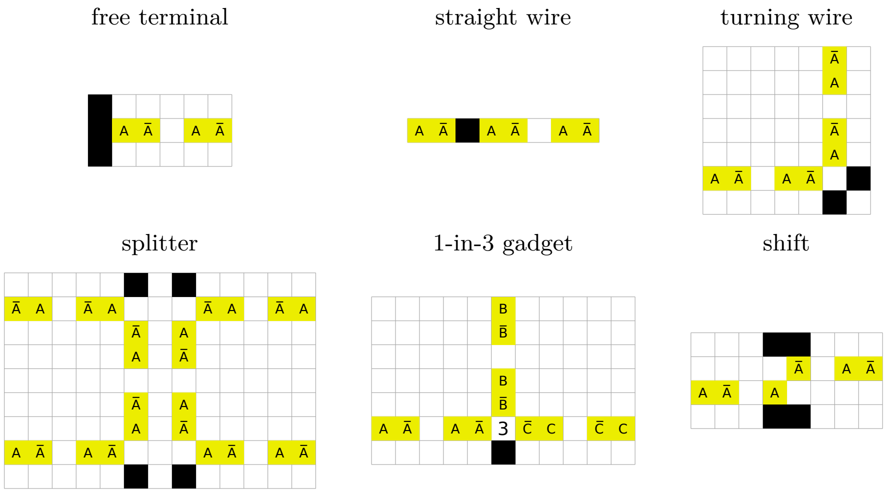
▶ **Corollary 38.** *Suppose that for some game like Minesweeper, we have silent parsimonious implementations of gadgets which are all square tiles that fit in a grid and interact appropriately with adjacent tiles. Suppose in particular that we have empty tiles, straight wires, turning wires, any generalized fanout, free terminals, and 1-in-3 gadgets. Then consistency is NP-hard, and inference and solvability are coNP-hard.*

For a few of our applications, it is more convenient to build a 2-in-3 or a 1-in-4 instead of a 1-in-3 gadget. These are also sufficient, since they can simulate a 1-in-3 gadget, using NOT gates and a free terminal, respectively.

4 Hardness proofs

To demonstrate the utility of this PGO framework and the ease with which it facilitates writing hardness proofs, we provide a number of example applications that prove hardness of Minesweeper and Minesweeper-like games, found in the full version of this paper. We reproduce some gadgets for standard Minesweeper here, as a representative example.

In the original Minesweeper game, the player does not start with any cells revealed, and the player starts by clicking any cell on the entirely unrevealed board (which the game guarantees to be safe). Hence, we construct the following "transparent" gadgets, where all clues can be deduced starting from this initial state. This shows that Minesweeper solvability is coNP-complete even in the setting where the player is initially given no clues. (To guarantee the click is safe, we can double the width of the grid, and delay the decision of which half to place the circuit on until after the click is made.)



References

1 E. Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.

2 Leslie M. Goldschlager. The monotone and planar circuit value problems are log space complete for p. *ACM sigact news*, 9(2):25–29, 1977.

3 Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

4 Sadie Kaye. Minesweeper is np-complete. *The Mathematical Intelligencer*, 22:9–15, 2000. doi:10.1007/BF03025367.

5 Philippe Laroche. Planar 1-in-3 satisfiability is np-complete. *Comptes Rendus de L Academie des Sciences Serie I-Mathematique*, 316(4):389–392, 1993.

6 Allan Scott, Ulrike Stege, and Iris van Rooij. Minesweeper may not be np-complete but is hard nonetheless. *The Mathematical Intelligencer*, 33:5–17, 2011. doi:10.1007/s00283-011-9256-x.

7 Alex Thieme and Twan Basten. Minesweeper is difficult indeed! *A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, 13560:472, 2022.