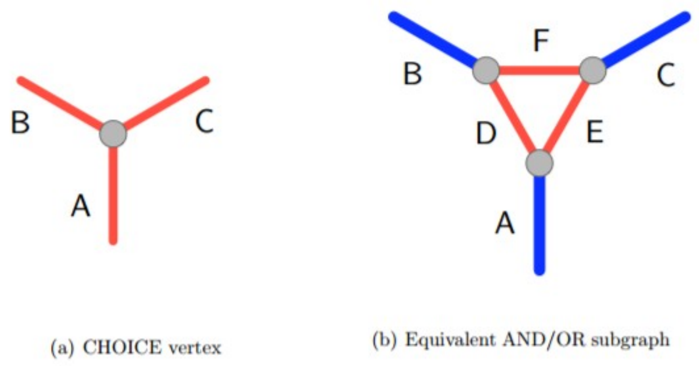
Constraint Logic Classes

Team I

Constraint Graph

A constraint graph is a directed graph, with edge weights ∈ {1, 2}. An edge is then called red or blue, respectively. The inflow at each vertex is the sum of the weights on inward-directed edges. Each vertex has a nonnegative minimum inflow. A legal configuration of a constraint graph has an inflow of at least the minimum inflow at each vertex; these are the constraints. A legal move on a constraint graph is the reversal of a single edge, resulting in a legal configuration.

- Generally, in any game, the goal will be to reverse a given edge, by executing a sequence of moves. In multiplayer games, each edge is controlled by an individual player, and each player has his own goal edge. In deterministic games, a unique sequence of reversals is forced. For the bounded games, each edge may only reverse once.

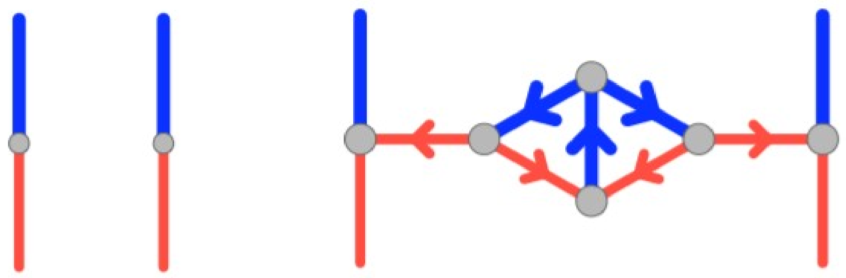


(a) CHOICE vertex (b) Equivalent AND/OR subgraph

Choice Vertex

- A choice vertex is a vertex with three incident red edges and an inflow constraint of 2. The constraint is thus that at least two edges must be directed inward. If we view A as in input edge, then if A points down, then the outputs B and C must also point down. If A is directed up, either B or C, but not both, may also be directed up. In the context of a game, a player would have a choice of which path to go through.

- The AND/OR subgraph shown in (b) has the same constraints on its A, B, and C edges as the CHOICE vertex does. Suppose A points down. Then D and E must also point down, which forces B and C to point down. If A points up, D and E may as well. F may then be directed either left or right, to enable either B or C, but not both, to point up.

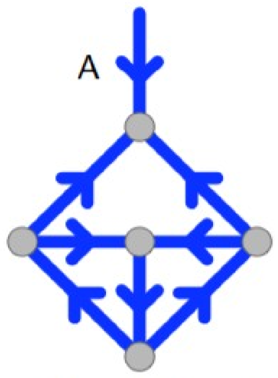


(a) Pair of red-blue vertices (b) Equivalent AND/OR subgraph

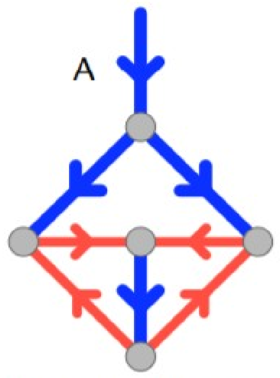
- Viewing AND/OR graphs as circuits, we might want to connect the output of an OR, say, to an input of an AND. We can’t do this directly by joining the loose ends of the two edges, because one edge is blue and the other is red. However, we can get the desired effect by joining the edges at a red-blue vertex with an inflow constraint of 1. This allows each incident edge to point outward just when the other points inward either edge is sufficient to satisfy the inflow constraint.

Edge Conversion

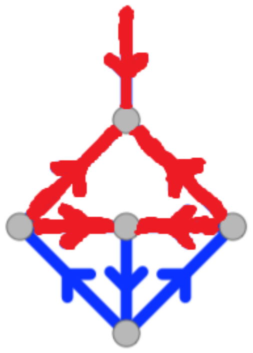
- A conversion from two red-blue vertices to an equivalent AND/OR subgraph. A red edge incident at a red-blue vertex must be one end of a chain of red edges ending at another red-blue vertex. This is shown in the figure above. The orientations shown for the edges in the middle satisfy all the constraints except for the left and right vertices. For these, an inflow of 1 is supplied, and either the red or the blue edge is necessary and sufficient to satisfy the constraints. It will occasionally be useful to use blue-blue and red-red vertices, as well as red blue. These vertices have an inflow constraint of 1, which forces one edge to be directed in.



(a) Free edge terminator



(b) Constrained edge terminator


(c) Free Red edge terminator

Terminators

- Often only one end of an edge matters; the other need not be constrained. To embed such an edge in an AND/OR graph, the subgraph shown in (a) suffices. If we assume that edge A is connected to some other vertex at the top, then the remainder of the figure serves to embed A in an AND/OR graph while not constraining it. Similarly, sometimes an edge needs to have a permanently constrained orientation. The subgraph shown (b) forces A to point down; there is no legal orientation of the other edges that would allow it to point up.

BOUNDED DETERMINISTIC CONSTRAINT LOGIC (BOUNDED DCL)

INSTANCE: AND/OR constraint graph G0; edge set R0; edge e in G0.

QUESTION: Is there an i such that e is reversed in Gi?

- P-completeness Theorem 1 Bounded DCL is P-complete.\
- Proof: Given a Boolean Circuit C, we construct a corresponding Bounded DCL problem, such that the edge in the DCL problem is reversed just when the circuit value is true. This

process is straightforward: for every gate in C we create a corresponding vertex, either an AND or an OR. When a gate has more than one output, we use AND vertices in the FANOUT configuration. The difference here between AND and FANOUT is merely in the initial edge orientation.

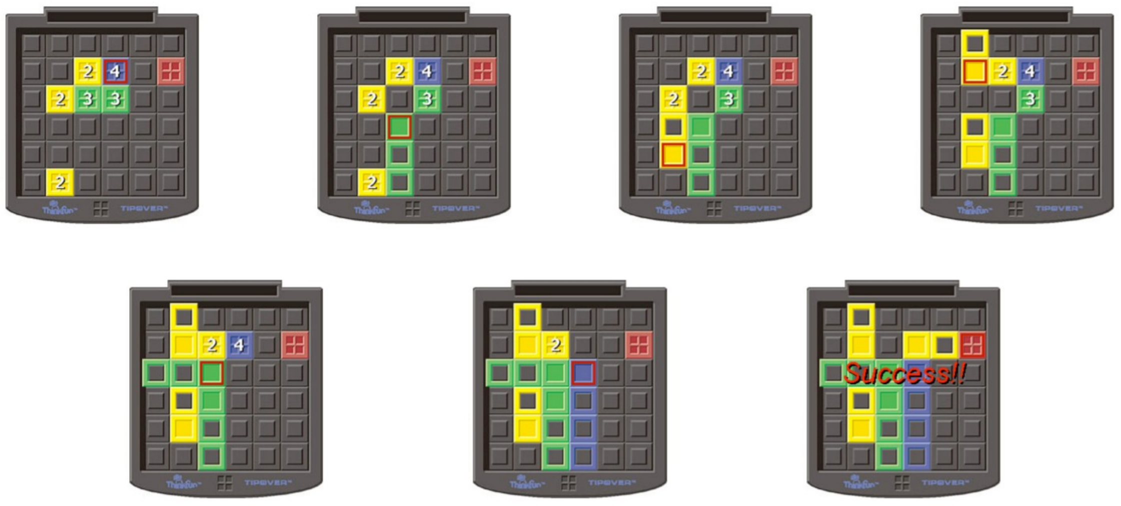
- Where necessary, we use the red-blue conversion technique. For the input nodes, we use terminators as in (a) and (b). The target edge e will be the output edge of the vertex corresponding to the circuit's output gate. We must still address the issue of potential illegal graph successors. However, in the initial configuration the only edges that are free to reverse are those in the edge terminators and in the red-blue conversion subgraphs; all other vertices are effectively waiting for input. We add the edges in the red-blue conversion graphs to the initial edge set R0, and we similarly add all edges in the edge terminators, except for the initial free edges that correspond to the Boolean circuit inputs. Then, no edges can ever reverse until the inputs have propagated through to them, and in each case the signals flow through appropriately. The only way to have an illegal graph successor would be to start in a configuration with all edges directed into an AND vertex, or with two edges directed into an OR, but these situations do not arise in the reduction. Then, the Bounded DCL dynamics exactly mirror the operation of the Boolean circuit, and e will eventually reverse if and only if the circuit value is true. This shows that Bounded DCL is P-hard. Clearly it is also in P: we may compute Gi+1 from Gi in linear time (keeping track of which edges have already reversed), and after a linear time no more edges can ever reverse).

Day 2

Tip Over

The purpose of the game is to tip over rectangles of different heights to reach a target location on the edge of the board. The game is simple, you can only tip the rectangles into empty spaces on the board and you can only move to a spot if there is a part of a rectangle on it. The trick is to tip the rectangles over in a certain order to reach the target spot. The standard TipOver board is 6x6 but can be generalized into an nxn board.[1]

1 Hearn, Robert A., and Erik D. Demaine. Games, puzzles, and computation. Wellesley (MA): Peters, 2009. Page 83



This is an example of a game wining game and the moves made to win.

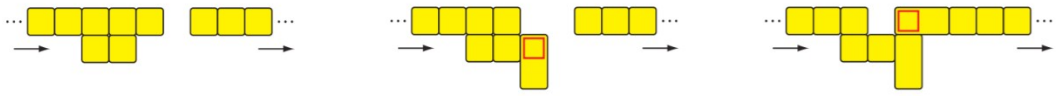
Rules

So, you start with a certain layout of the board where all the rectangles are placed vertically on a single spot of the board. These rectangles very in height but have the same length and width. Then a “tipper” is placed on a start rectangle and a red 1x1x1 rectangle is placed on a certain spot on the board making it the target location. The tipper move to any adjacent rectangles and can tip them over, given that there is enough empty space in the tipping direction that the rectangle will still be within the board.

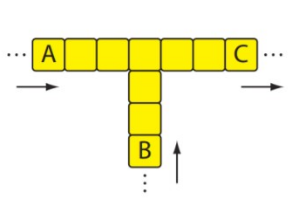
NP-completeness

To build the gadgets we will use rectangles of vertical height two. The mapping from constraint graph properties to TipOver properties is that an edge can be reversed just when a corresponding TipOver region is reachable by the tipper.[2]

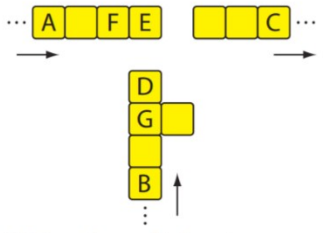
2 Hearn, Robert A., and Erik D. Demaine. Games, puzzles, and computation. Wellesley (MA): Peters, 2009. Page 84



A wire that must be initially traversed from left to right. All crates are height two.



(a) OR gadget. If the tipper can reach either A or B, then it can reach C.



(b) AND gadget. If the tipper can reach both A and B, then it can reach C.

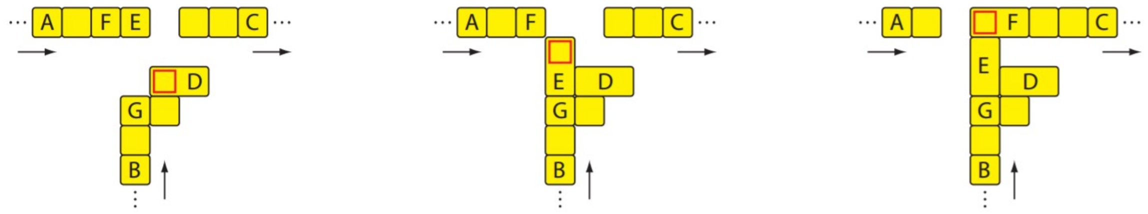
TipOver AND and OR gadgets.

OR Gadget

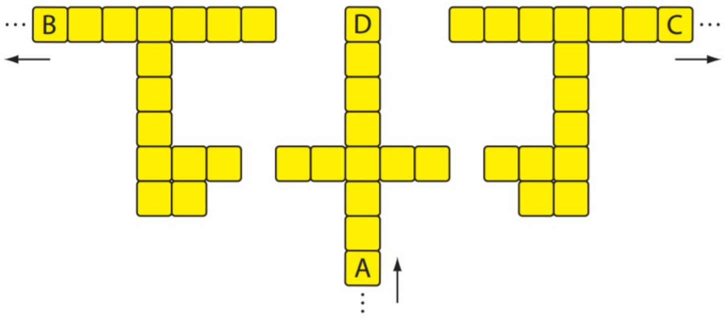
A simple intersection, protected with one-way gadgets, serves as an OR.

AND Gadget

AND is a bit more complicated. This time the tipper must be able to exit to the right only if it can independently enter from the left and from the bottom. This means that, at a minimum, it will have to enter from one side, tip some crates, retrace its path, and enter from the other side. Actually, the needed sequence will be a bit longer than that.

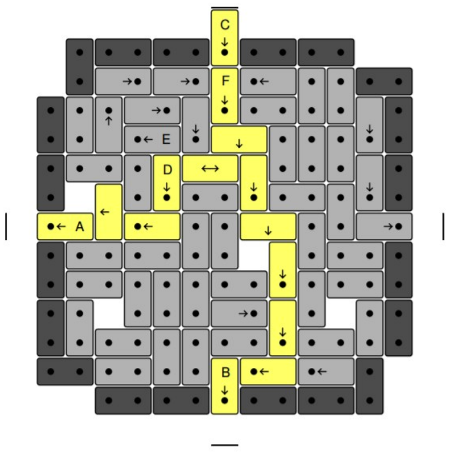


How to use the AND gadget.

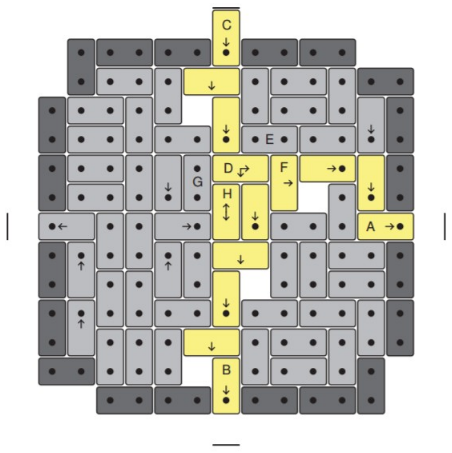


Sliding-Block Puzzles

We fill the box with a regular grid of gate gadgets, within a “cell wall” construction as shown in Figure 9-9. The internal construction of the gates is such that none of the cell-wall blocks may move, thus providing overall integrity to the configuration.[3]



AND



Protected OR

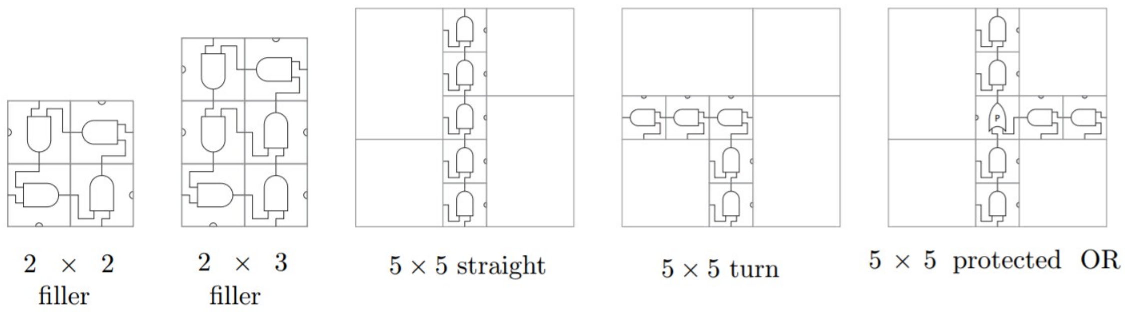
In each diagram, we assume that the cell-wall blocks (dark colored) may not move outward; we then need to show they may not move inward. The light-colored (“trigger”) blocks are the ones whose motion serves to satisfy the vertex constraints; the medium-colored blocks are fillers. Some of them may move, but

3 Hearn, Robert A., and Erik D. Demaine. Games, puzzles, and computation. Wellesley (MA): Peters, 2009. Page 89

none may move in such a way as to disrupt the vertices’ correct operation. The short lines outside the vertex ports indicate constraints due to adjoining vertices; none of the “port” blocks may move entirely out of its vertex. For it to do so, the adjoining vertex would have to permit a port block to move entirely inside the vertex, but in each diagram the annotations show this is not possible. Note that the port blocks are shared between adjoining vertices, as are the cell-wall blocks. For example, if we were to place a protected OR above an AND, its bottom port block would be the same as the AND’s top port block. A protruding port block corresponds to an inward-directed edge; a retracted block corresponds to an outward-directed edge. Signals propagate by moving “holes” forward. Sliding a block out of a vertex gadget thus corresponds to directing an edge in to a graph vertex.

Graphs

Now that we have AND and protected OR gates made out of sliding-blocks configurations, we must next connect them together into arbitrary planar graphs. First, note that the box wall constrains the facing port blocks of the vertices adjacent to it to be retracted. This does not present a problem, however, as I will show. The unused ports of both the AND and protected OR vertices are unconstrained; they may be slid in or out with no effect on the vertices. Because none of the ANDs need ever activate, all the exterior ports of these blocks are unconstrained. We may use these filler blocks to build (5 × 5)-vertex blocks corresponding to “straight” and “turn” wiring elements (Figures 9-11(c) and 9-11(d)). Because the filler blocks may supply the missing inputs to the ANDs, the “output” of one of these blocks may activate (slide in) if and only if the “input” is active (slid out). Also, we may “wrap” the AND and protected OR vertices in 5 × 5 “shells”. We use these 5×5 blocks to fill the layout; we may line the edges of the layout with unconstrained ports. The straight and turn blocks provide the necessary flexibility to construct any planar graph, by letting us extend the vertex edges around the layout as needed.[4]



Sliding Blocks wiring.

4 Hearn, Robert A., and Erik D. Demaine. Games, puzzles, and computation. Wellesley (MA): Peters, 2009. Page 92