# EXPLORING THE COMPLEXITY OF THE UFO PUZZLE

MARTIN HOCK

ABSTRACT. We begin by defining several fundamental concepts of computational complexity. We then define the problem (solving a puzzle game termed the UFO Puzzle) naturally and then state it in terms of a STRIPS-like domain which is known to be PSPACE-complete. We also demonstrate some methods of shortening solutions. Then, we use a reduction from CNF-SAT to show that the game is NP-hard and use this to show that a restricted version of the puzzle is NP-complete. We leave open the question of whether the original puzzle is PSPACE-complete, NP-complete, or otherwise.

## 1. BACKGROUND

What is computation? The obvious answer is that it's what a computer does. But then what is a computer? An example of a computer is something that you buy from Best Buy for $999.95, but it makes sense to characterize computation in a more abstract context, as computers have changed drastically in the past 50 years and will no doubt change drastically in the next 50. In fact, the most common model for computation was the one proposed by Alan Turing in 1936, before a general purpose computer had ever been built. His notation of computation is the following: you have a small unit which you could call the CPU which has some state and points to a specific cell in a storage unit called the tape. The tape is composed of arbitrarily many cells, each holding one character worth of data, and begins with some input string written to the first $k$ cells. The CPU starts in some initial state and points to the first cell of the tape. It reads the first cell and then acts according to the state and the cell's contents, which tell it to change to another specific state, write something new to the cell, and which cell to look at next, either the one to the left or the one to the right. This operation takes one time step. One of the states of the CPU is known as the halting state; if the CPU ever changes to the halting state it stops execution (halts). The question to ask is: will the CPU halt on a given string of input? The answer is that we cannot create any CPU which will tell in a finite amount of time whether an arbitrary CPU halts on an arbitrary string of input. Furthermore, the notion of computation of this CPU is quite similar to that which is used by computers, as well as that used by programming languages and logic systems.

An important question is to ask what kinds of programs our CPU can compute efficiently. One such program is called SAT. The question is, given a boolean expression (a series of variables which hold true or false, put together with ands, ors, and nots), can we tell quickly whether there exists an assignment of truth values to the variables such that the expression resolves to true? The fastest approach we know of is in the worst case no faster than trying all possible assignments to the variables, which an exponential number of time steps in the number of variables. On the other hand, what if we had a different kind of CPU? This CPU can split itself into multiple copies whenever it reads a cell, where each of these copies acts independently. We then ask the question: for a string of input, does some copy of the CPU halt? For SAT, we can come up with a process which will halt in a polynomial number of time steps, perhaps fewer than the square of the number of symbols in the expression plus a constant startup time. The question is, can we write a program for our normal CPU which can emulate this parallelizing CPU taking only polynomially more steps than the parallelizing CPU? It sounds like the answer is no, but in the past 30 years since the problem was first proposed, no computer scientist has been able to answer the question, and there is now a million dollar prize at stake.

This paper will not answer this question. What it will do is show that another kind of problem is at least as powerful as SAT, in the sense that if there exists a program for the CPU which halts in a polynomial number of steps in the size of any puzzle and can determine that it is solvable, then we can write a similar program for SAT. The class of problems with this property is known as NP-hard. If we also showed that the parallelizing CPU would halt in a polynomial number of steps to solve the puzzle, the problem would be in NP, and this would allow us to call the problem NP-complete.

Besides how much time a program takes to execute, we can ask questions about the space required during its execution. What about requiring that a program uses only a polynomial amount of space? We note that we can easily solve SAT in a polynomial amount of space by enumerating each possible assignment of values and trying it out.

## 2. The game

The *UFO Puzzle* (or just *UFO*) is a solitaire game created by Hiroshi Yamamoto and presented by Nob Yoshigahara in February 1998 at the Third Gathering for Gardner in Atlanta, Georgia. Yoshigahara is the creator of NOBrain and created numerous other puzzles including *Rush Hour*, a restricted movement sliding block puzzle. UFO is now being marketed in a $5 \times 5$ board version by Binary Arts called *Lunar Lockout*, but it is the general arbitrary board that we are concerned with.

**Definition 2.1.** Pieces, Positions, Configurations, and Boards.

The *piece* is the fundamental unit of UFO. Each piece is unique. A *position* is a member of $\mathbb{N} \times \mathbb{N}$, the elements referred to as the coordinates of the position. A *configuration* is an injective function from pieces to positions. A *board* is a set of pieces $p$ and a configuration $c$ called the *initial configuration*. If we examine all of the positions in the initial configuration as being points on the plane, we can bound the board by the smallest rectangle that contains all of these positions; then we say that the *size* of the board is equal to the area of this rectangle.

**Definition 2.2.** Movement.

We define *move* as a partial function from a configuration $x$ and a pair of pieces $a, b$ to a configuration. We say two pieces are collinear in some configuration if the configuration maps both of them to have the same first or second coordinate. We refer to the two coordinates as the shared and the differing coordinates and for piece $k$ its shared and differing coordinates are defined as $k.s$ and $k.d$. We define a function $dec(n) = \frac{n}{|n|}(|n| - 1)$. If $a$ and $b$ are collinear and there is no piece $c$ such that $a.s = c.s \wedge ((a.d < c.d < b.d) \vee (a.d > c.d > b.d))$ then the new configuration is defined to be the same as the old, except $a.d = a.d + dec(b.d - a.d)$, e.g. $a$ slides so that it is adjacent to $b$. We say that the piece $a$ has moved and that piece $b$ was collided with. Otherwise the function is not defined. Again appealing to the pieces on the plane, we may label the four potential moves a piece may have as right, left, up, and down corresponding to $+x, -x, +y, -y$, respectively.

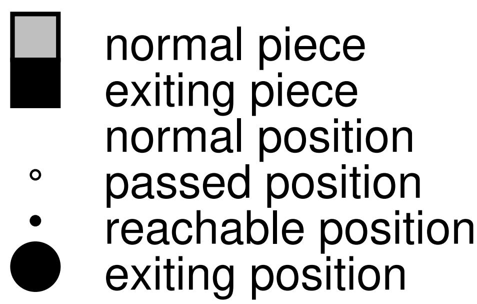
*Remark* 2.3. No sequence of movements will ever give rise to a configuration which places a piece outside of the bounding rectangle of the initial configuration. This is because all pieces begin within this rectangle, but in order to be moved outside the rectangle, a piece would have to collide with a piece outside the rectangle. Therefore, we define the board to be only the finite set of positions which comprise the bounding rectangle of the initial configuration.

**Definition 2.4.** Configuration and Solvability.

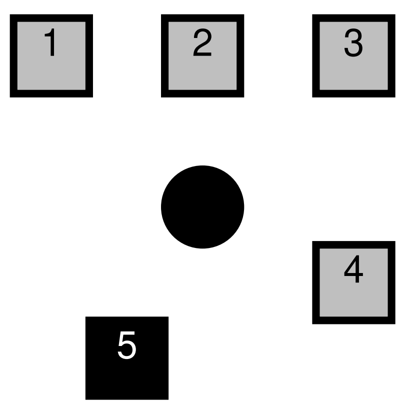
For a given board $b$ (of $n$ positions and $m$ pieces) and a configuration $c$ defined on the same set of pieces, we may wonder if there exists a sequence of moves $s$ that, if executed, $b$'s configuration becomes $c$. We call the decision problem of whether arbitrary $b$ may be reconfigured to arbitrary $c$ the *UFO configuration problem*.

For a given board $b$ (of $n$ positions and $m$ pieces) with a given piece $d$ and a given position $p$, we may wonder if there exists a sequence of moves $s$ such that, if the sequence is executed, $d$ is mapped to the position $p$. If so, we say that the triple $(b, d, p)$ is solvable. We call the decision problem of whether arbitrary $(b, d, p)$ is solvable the *UFO solvability problem*.

We may also represent boards or portions of boards visually as a two dimensional matrix of positions containing pieces. We can refer to positions as being north, south, east, and west of each other to correspond to the four spatial directions; the orientation does not matter, though north and south are along one axis and east and west are along the other. We use the following notation for such representations:




Here is an example of a board:




**Exercise 3.** Solve this board. It can be done in 12 moves. The solution is at the end of this paper.

*Remark* 3.1. We consider the complexity of UFO in terms of the size of the board. We note that letting $n$ be the number of pieces is not worth considering, as with only a constant number of pieces we may have a board which requires an arbitrary number of moves to solve:

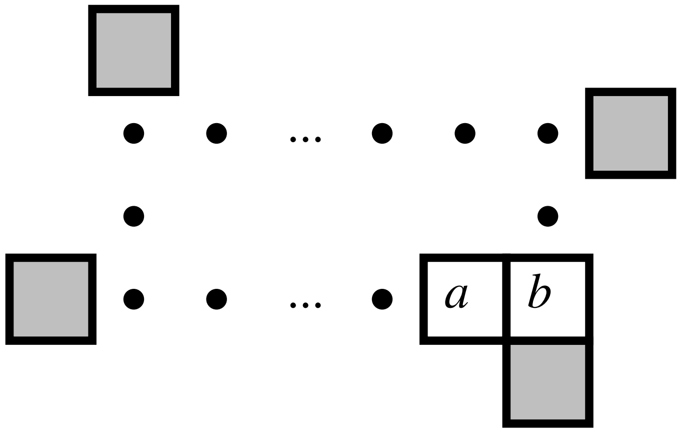



Figure 1.

In this board of size pieces, piece $a$ may move left, right, up, and down to collide with $b$; $b$ may then move left, right, up, and down to collide with $a$; the process may be repeated to allow $a$ and $b$ to traverse an arbitrary number of positions.

## 4. Classifying Configuration and Solution

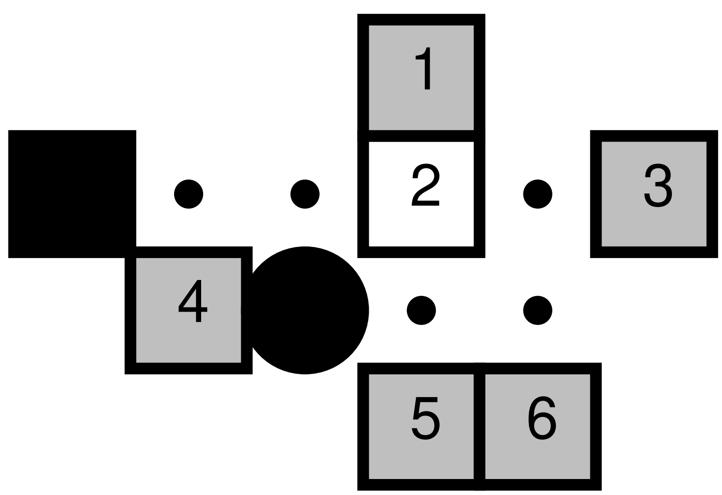
4.1. **Basic complexity arguments.** To begin with, it is easy to see that the solution and configuration problems are decidable, e.g. we can write a computer program which will stop running after a period of time and tell us whether a certain puzzle can be solved or a certain configuration reached. We can recursively enumerate all sequences of moves. No solution need repeat a board state as any move made from the second occurrence occurrence of the board state could have been made from the original occurrence. On a board with $n$ positions and $m$ pieces there is a maximum of $\frac{n!}{(n-m)!}$ different ways of arranging the pieces on the board, so this is the maximum length of the solution.

This is quite a long length, however; how can we shorten it? In particular, if we know that any solvable puzzle can be solved in some polynomially bounded solution length, we know that the puzzle is in the class of NP-solvable problems, which would be quite helpful. However, finding a polynomial bound on the length of operation-based problems like this is quite difficult. For example, despite multiple NP-hardness results being proven over the span of a few years by several computer scientists, the classification of the Push problems as either NP or PSPACE-complete is unknown for all but the most trivially bounded variety[3, 2]. In the Push problems, one has a grid full of pieces much as in this game, and one piece is designated as the mover, which takes steps of one square at a time. The question one asks is whether the mover can move to a certain position. This would be quite simple except that the mover can move into another piece, pushing it aside, but only if the adjacent square is free. If we restrict the mover so that it cannot move to a square more than once, in a game called *Push-1X*, it can only make at most $n$ moves, but otherwise, no sub-exponential upper move bound is known.

We can restrict both Configuration and Solvability to the class of problems in PSPACE. In particular, we can encode the board naturally, where a given block of cells tells us whether a particular position contains no piece or a piece having a certain identity. The number of cells necessary for each position is logarithmic in the size of the board so the entire coding is $n \log n$. Now, when we perform a move, we do not save our solution, we simply modify the board. However, instead of modifying only one copy of the board, we use a nondeterministic machine and modify the board according to all possible moves which could be made. But from[6] we know that NPSPACE = PSPACE. This is the same technique employed by[4] for showing that Sokoban is in PSPACE.

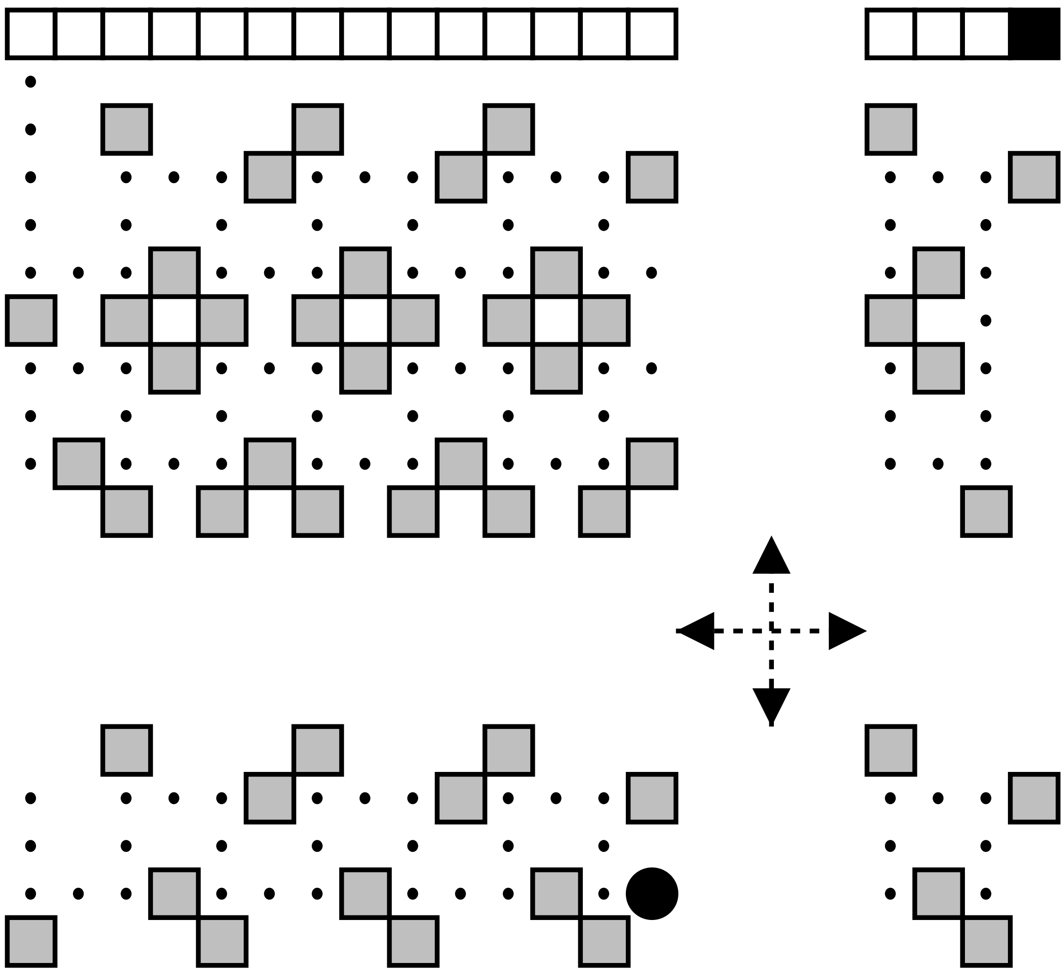
Here is a short summary of the proof that NPSPACE = PSPACE. Suppose we have a program $p$ of length $n$ which executes on the parallelizing CPU in no more than $f(n)$ space, where $f(n) \geq \log_2 n$. If our computer is binary, we note that the parallelizing CPU can take no more than $2^{f(n)}$ steps to solve the problem; if more steps than this are needed, there must be a loop, which is a contradiction since we know the CPU will always terminate. Now we will write a program $q$ which the normal CPU accepts in no more than $f(n)^2$ space, if and only if the parallelizing CPU accepts $p$. We use the notion of a snapshot of $p$ to indicate what one of the parallel CPUs is doing at a given moment. A snapshot takes up only space proportionate to $f(n)$. The first thing we note is that we can try all possible terminating states for the CPU in space no greater than checking one. We simply increment through all configurations, one at a time. For any given configuration, we break our $f(n)^2$ storage up into $f(n)$ snapshots each of size $f(n)$. Now, we consider the initial snapshot $I$ and some possible terminating snapshot $I''$ each of which is stored using one of our storage blocks. We want to know if there exists a sequence of calculations which will allow the parallelizing CPU to go from $I$ to $I''$ in $2^{f(n)}$ steps. To accomplish this, a divide and conquer strategy is employed. For each possible intermediate snapshot $I'$, the program checks if there exists a sequence of steps to get from $I$ to $I'$ in $2^{f(n)-1}$ steps, and then from $I'$ to $I''$ in $2^{f(n)-1}$ steps, recursing for each half. (We note that a step consists of selecting one of the possible spawned CPUs to look at, as we know that some particular CPU accepted the input, so we need only examine the steps needed to spawn this CPU.) Each of the halves of this simulation requires only $f(n)$ snapshots, for a total of $f(n)^2$ space necessary. We will not be able to keep a record of what the CPU did to reach the final state because it must erase itself in order to conserve space, but we know that the actions we performed consist of some execution of steps on the parallel CPU.

How could one limit the solution length? One could make the restriction that a piece never repeat a position, a variant which we call *UFO-X*. In that case, each of our $m$ pieces may move at most $n$ times for an upper bound of $mn < n^2$ moves. However, such a restriction makes some solvable puzzles into unsolvable ones. Here is such a puzzle:



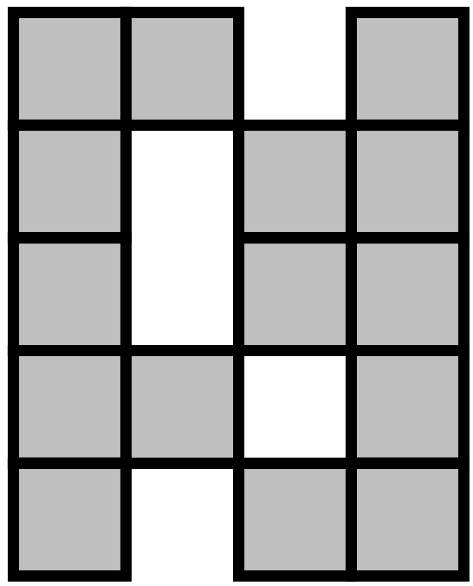

We note that piece 2 must revisit its initial position. In particular, the only piece that the exiting piece could collide with and stop on the exit is piece 4, as there must exist pieces on both sides along either the vertical or horizontal axis of the exit for such a stop to occur, but vertically no piece may appear below the exit (via a recursive reachability argument), and no piece could appear adjacent to the left of the exit except for 4, as if 4 moves the west of the exit is no longer reachable. The exiting piece must move east past piece 2 because it must approach 4 from the row that 4 is on, but then 2 blocks the exiting piece, and the only place to move it out of the way is north, repeating a state.

I conjecture that a linear bound on the number of moves does not suffice. Examine the following board (inspired by a suggestion by Venkatesan Chakaravarthy):



Consider the board to be square. Then the sides are length $\sqrt{n} \times \sqrt{n}$. The first row contains $\sqrt{n}$ pieces. The exit is $\sqrt{n}$ dots away from the end of the path indicated. I conjecture that all $\sqrt{n}$ of the pieces in the first row must be moved to the end to allow the exiting piece to exit, by following the path indicated. By this method, there are $\Omega(n)$ collisions required to move a piece to the end, for a total of $n\sqrt{n}$ moves needed. However it is likely that this is not the most efficient path for the solution, as the board is full of enough pieces to allow great flexibility. But, at least for configuration, translating the top row pieces to the end of the path should take all $\Omega(n\sqrt{n})$ moves.

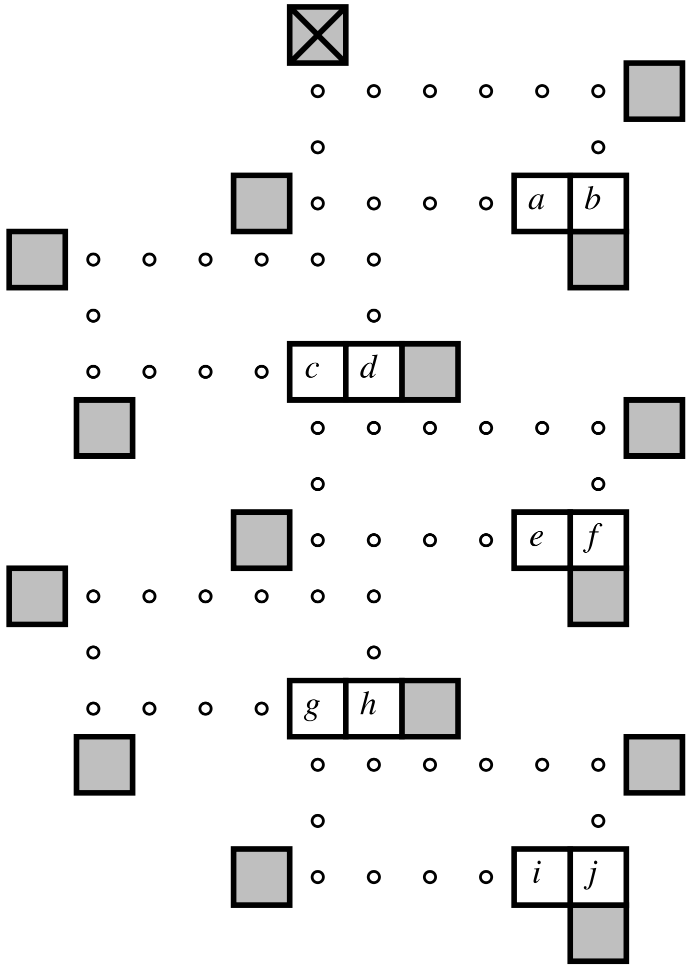
4.2. **Classes of long solutions.** Still, we can perform some reductions on the length of the solution. There are two classes of reductions I have made. The first kind of reduction keeps pieces from unnecessarily repeating their positions on an individual basis. Here is an example of a situation where this would help. Suppose we have a board which is $4$ positions wide and $n/4$ positions tall. The left and right columns are completely full of pieces; each row also contains one additional piece in one of the two central positions. The board can then represent a binary number. Here is such a board that represents $01101$:



We may then toggle through different board states at will by moving the central pieces left or right. In particular, we may cycle through all $2^n$ binary states by performing such toggles without revisiting a state. Such a traversal is equivalent to a binary Gray code of $n$ digits, but I will give a short proof here for convenience. The procedure is defined inductively. For the base case, if we have only one bit, it begins in the initial state and then we toggle it once, thus visiting both states. For the inductive step, assume we can toggle $k$ bits through all possible states without revisitation – now we show a procedure to do this for $k + 1$ bits. We toggle the low $k$ bits through all their states, then toggle the $k + 1$st bit, then again toggle the low $k$ bits. (Note that the bits will not be in their initial state, but by our inductive definition we have a procedure anyway. In fact, the procedure we used the first time will suffice.) Because this will reach $2^{k+1}$ unique states and because there are only $2^{k+1}$ such states, all states will be visited exactly once.

However, it is intuitively obvious that we need only toggle at most $n$ bits to translate the initial state to some final state – we simply toggle the $i$th bit if and only if it is different in the two numbers. When each piece need only be toggled once, it is instead toggled $2^{n-1}$ times.

The other kind of reduction occurs in the following scenario: instead of simple rows representing our bits, we represent them with larger gadgets. Now instead of our pieces merely moving back and forth, we have them moving within systems of other pieces with many intermediate states. Examine the following board:




We refer to each of the dotted rectangular regions as a cell and the lettered pieces as counters. The counters can move inside their respective cells by colliding with each other. We note that the parity of the cell determines the direction that the counters move, clockwise versus counterclockwise. We also note that although the first cell (containing $a$ and $b$) can cycle at will, the second cell can only cycle if $a$ and $b$ are in a certain position, and the third cell if $b$ and $c$ are in a certain position, and so on. We can use the same argument to have exponential movement patterns for these cells, but unlike before, although pieces will repeat their positions, they will do so only after colliding with and being collided by other pieces. The construction could be extended by removing the block marked with an X and continuing the pattern vertically.

Both kinds of reduction can be made by considering move justification, which is outlined below. We begin by assuming that we have some sequence of moves $s$ which solves the puzzle but we wish to shorten $s$ to be more efficient.

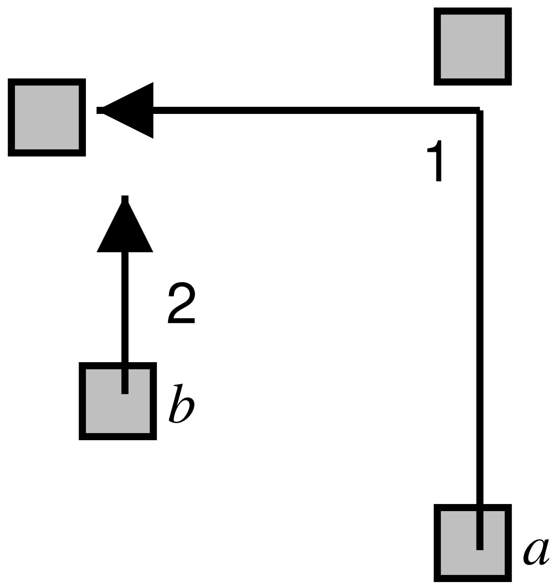
4.3. **Reduction via justification.**

**Definition 4.1.** Jumps.

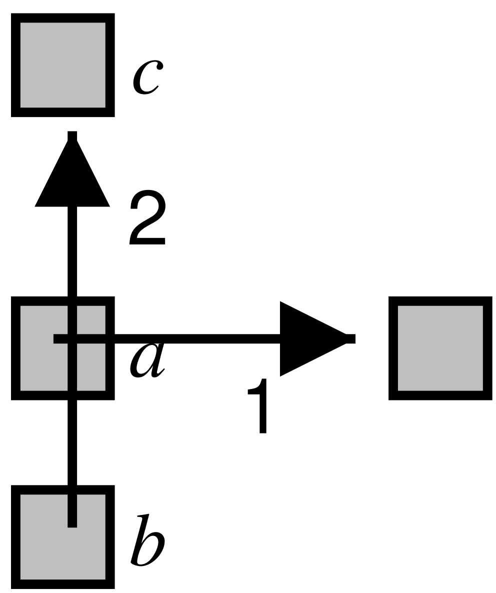
Take the sequence of moves $s$ and instead view it as a set of interleaved subsequences of movements of various pieces. What we mean by interleaved is that the moves of sequences can appear inside the bounds of other sequences. Specifically, take all moves of the piece $p$ in order as a sequence and we call this the full jump of $p$. Any subsequence of the full jump of $p$ is also a jump of $p$. For a jump of piece $a$ that is eventually moved to position $X$, we write $a \hookrightarrow X$. Similarly, an interleaved sequence of jumps may be also be viewed as a sequence of moves.

**Definition 4.2.** Justification.
A jump $a \hookrightarrow X$ is justified in one of three ways:

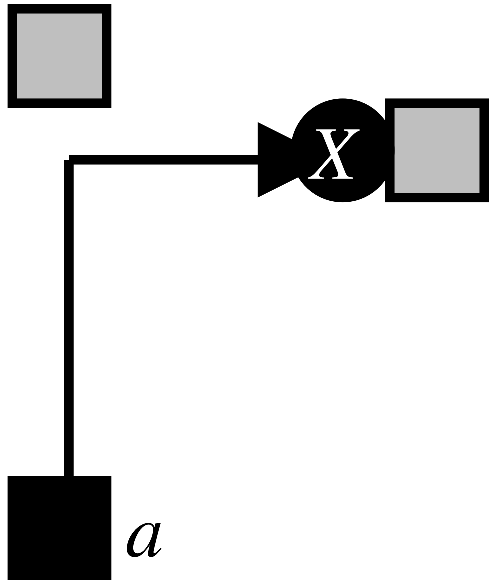
(1) Some other piece $b$ collides with $a$ in position $X$. We note that this implies that $b$ is in another jump.




(2) Some other piece $b$ collides with some other piece $c$ such that $c$ is collinear with $a$ and $b$ and $a$ is between $b$ and $c$. We note that this implies $b$ is in another jump.




(3) $X$ is $a$'s final destination, such that $a$ is never moved once it reaches $X$. We call the requirement that $a$ is never again moved the *finality condition*.




We refer to these as type-1, type-2, and type-3 justifications. A solution is justified if all of its jumps are justified. We note that any solution may be justified by taking each of its full jumps and labeling them as type-3 justified jumps.

**Definition 4.3.** Loops.

If $a$ is moved to a certain position $Y$ more than once in a jump we refer to the sequence of moves ranging from the first time it leaves $Y$ to the last time it enters $Y$ as a loop. If a loop contains no loops it is referred to as a primitive loop.

**Theorem 4.4.** *An interleaved sequence of justified jumps $s$ which maps configuration $a$ to configuration $b$ may be rewritten as $s'$ which still maps $a$ to $b$ but contains no loops.*

*Proof.* Examine a primitive loop $a_A \hookrightarrow X$. Suppose no piece collides with $a$ in any of the moves interleaved inside $a_A \hookrightarrow X$. Suppose also that no piece passes through $X$ in any of the moves interleaved inside $a_A \hookrightarrow X$. Then $a_A \hookrightarrow X$ without affecting the solution. This is because every (previously) interleaved move will still be valid; no piece collides with $a$ in that sequence, nor will $a$ interleave with any supposed empty region.

Otherwise, either some piece $b$ collides with $a$ or some piece $b$ passes through $X$. In the first instance, suppose it collides at $Y$. Then we may split $a_A \hookrightarrow X$ into two jumps: $a_A \hookrightarrow Y$, a type-1 justified jump which is justified by $b$, and $a_Y \hookrightarrow X$, which retains the type and justification of $a_A \hookrightarrow X$. In the second instance we select the position $Y$ where $a$ is located when $b$ passes through $X$, and split $a_A \hookrightarrow X$ into type-2 $a_A \hookrightarrow Y$ justified by $b$ and $a_Y \hookrightarrow X$, again which retains $a_A \hookrightarrow X$'s type and justification.

There are a few things we must note. First, in either of these cases if $a_A \hookrightarrow X$ was type-3 the finality condition still holds for the two resulting jumps. Next, any type-2 jump $j$ created by this process requires some other jump $k$ to contain some moves after $j$. Finally, note that this process reduces the number of loops in $s$ by at least one, though the splitting process may eliminate many loops at once. □

We see that the above loop elimination strategy deletes moves which are completely unused. It also shows that every solution can be turned into one where every move made has some "reason" behind it. However, it does not get rid of recursive reasoning like that utilized in the puzzle shown in Figure 1.

**Theorem 4.5.** *An interleaved sequence of jumps may be rewritten such that the jumps are no longer interleaved but still result in a valid solution.*

*Proof.* We begin by defining a primitive jump.

**Definition 4.6.** Primitive jump.
A primitive jump of $a$ from $s_1$ to $s_2$ is a jump of $a$ such that no piece collides with $a$ until it has completed the jump nor does any piece pass through a position occupied by $a$ at its initial position in the jump until it has completed the jump. We denote this by $a_{s_1} \rightharpoonup s_2$.

We may convert any jump into an interleaved sequence of primitive jumps. We use a similar tactic to the above. If a piece collides with $a$ mid-jump, we split the jump at the time of the collision, with the first part being type-1 and the second part having $a$'s old type. If a piece passes through a position occupied at the initial position of a jump, we split the jump at the time of this move, with the first part being type-2 and the second part having $a$'s old type. Note that the splitting process may require multiple passes because the creation of a new jump will create a new initial position that a piece may pass through. For convenience we also add all other utilizations of a jump as justifications, e.g. if two jumps $a$ and $b$ collide with a piece moved in a jump $c$, both $a$ and $b$ justify $c$, Note also that since the interleaved sequence of moves is identical to the new set of jumps the new set of jumps is valid. Finally note that since the new jumps are strict subsets of the old jumps, no loops will be created

All moves contained within a single primitive jump may be shifted such that they are immediately consecutive to the first move in the primitive jump. This is because no piece either collides with the piece being jumped in the middle of the jump, or moves through any intermediate space of the jump in the middle of a jump. □

Now that we have written out the moves as interleaved sequences of primitive jumps, we may use a directed graph to represent the relationships between the jumps. For each piece $p$, for each start location $s$ and end location $e$, we have a vertex representing $p_s \rightharpoonup e$. Now, for each vertex we may ascribe a set of numbers indicating the index of the primitive jump within the sequence of jumps that reaches the configuration. Throw out the vertices which are never used. By simply observing this set, we can check its validity by trying every possible sequence of moves for each jump listed, since we know the initial board configuration and the jump designates what the following board configuration must be. We note that checking a jump's validity is linear in the size of the board because for each jump, we are only moving one piece, so we may create a graph of the positions that the piece may reach, cutting out any loops. We also note that if no vertex has more than one number in its set, we have polynomially bounded the sequence of moves for the witness.

Suppose we revisit a vertex. In particular, if there are no out edges in any of the vertices contained within this loop, none of the inner vertices are used (since we have included all utilizations as justifications) so we can simply throw out these moves.

4.4. **UFO as a planning problem.** The techniques we have used to eliminate redundant moves have generalized UFO in terms of jumps and justification. In this vein, we can define UFO as a simple STRIPS[5] domain. We define a STRIPS problem as the following: we have a set of conditions $c$ and a set of operations $o$, each of which has the following form: a set of preconditions $j \subseteq c$ and a set of postconditions $k \subseteq c$. Then we define a set of preconditions $r \subseteq c$ and a set of postconditions $p \subseteq c$. The current set of conditions $a$ is set to $r$. We may apply an operation $m$ if $m.j \subseteq a$, at which point $a$ is set to $(a - m.j) \cup m.k$. We ask if there exists some sequence of operations which we may apply to transform $a$ to $c$.

The UFO puzzle may be encoded in such a domain fairly easily. In terms of conditions, we have for each position $p$, for each piece $x$, we have $on(x, p)$. We also have $empty(p)$ if the position is empty. This is only $n^2$ conditions. In terms of operations, for any two collinear positions $a$ and $b$, for every piece $x$ we have the operation $move(x, a, b)$ to move $x$ from $a$ to $b$. This is only $n^3$ operations. Despite these polynomial bounds on the operations and conditions, the domain is still PSPACE-complete[1].

## 5. Solvability is NP-Hard

### 5.1. **Preliminary Observations.**

**Definition 5.1.** Lines and sides.

The set of positions which constitute a given row or column are called a *line*. A *gap* is one or more consecutive empty positions in a line. In a board $b$, any line $l$ has two *sides*, $l_1$ and $l_2$ which are sets of positions disjoint to each other, such that $l \cup l_1 \cup l_2 = b$. If $l_1$ or $l_2$ contains no pieces, $l$ is called a *leading edge.* An board has four leading edges, northern, southern, eastern, and western.

**Definition 5.2.** Protection.

A set of positions is *protected* if no sequence of moves can cause a piece which is not contained by a member of the set initially to be contained by a member of the set. By definition, a rectangle is protected unless a piece outside the rectangle may pass through some position of the rectangle.

**Theorem 5.3.** *No piece in a board may be moved to a position which is outside of the effective board.*

*Proof.* Initially, by definition, the set of positions outside the effective board is empty. Suppose there is some first move that brings a piece to a position outside the effective board. However, in order for a piece to rest in a position outside the effective board, it would have to collide with a piece outside the effective board. (Case analysis: for a northern leading edge, a northern movement would require a piece north of the effective board, and so on.) □

**Corollary 5.4.** *Leading edges are protected.*

*Proof.* A leading edge would have to be one of the outside lines of an effective board. For a piece not within the leading edge to come to rest within the leading edge would require it to collide with a piece outside the effective board. (Case analysis: for a northern leading edge, a northern movement would require a piece northern adjacent to the leading edge, and so on.) □

**Definition 5.5.** Submission.

A protected set of positions $s$ is said to *submit* a piece when a piece contained by a member of $s$ is moved to a position outside of $s$.

5.2. **Linear UFO.** It makes sense to study protected parts of the board as separate games. In particular, a protected line has only one row or column. We refer to any game with only one row or column as *linear*. Without loss of generality, instead of thinking about the pieces and positions spatially, think of them as an order, i.e. first (piece 1), second, third, ..., last counting consecutively along some direction, with the comparatives "before" or "after" instead of "less than" or "greater than." and the movement directions "negative" and "positive." Within the context of the board, the "first position" is the one which contains the first piece, but we could also think about it in terms of some range of the board such as a gap. Since the pieces may not move past each other, the order of the pieces in the board remains constant, e.g. the first piece is always first, a fact which we call *persistence*.

**Lemma 5.6.** *(Unidirectionality) A configuration is reachable if and only if it is possible to reach the configuration by moving each of the pieces in a maximum of one direction.*

*Proof.* Suppose it is necessary to move a piece in both a positive and negative direction. Then there must be some piece $p$ and some first instance of this motion. Without loss of generality, assume that $p$ is moved positively, followed by any number of unidirectional motions of other pieces followed by a negative motion of $p$. In order for the board to be in a different state than it would have been if only the negative motion had been performed, a piece would have to collide with $p$ after it moved to its positive state. The only two pieces that could collide with $p$ are $p-1$ and $p+1$. Since the piece $p$ collided with, $p-1$, would have to perform a non-unidirectional motion to collide with $p$, it would have to be $p+1$. But then in order for $p$ to move positively, $p+1$ would have to move positively, resulting in a non-unidirectional motion before $p$. Therefore there can be no first non-unidirectional motion that results in a unique board configuration. □

**Corollary 5.7.** *(Fixed Point) Any reachable configuration may be reached via a sequence of movements that leaves at least one piece unmoved.*

*Proof.* Suppose all pieces must move. Then all pieces are translated from their original positions. Then either all pieces move in one direction, which is impossible because the last piece would have nothing to collide with, or all the pieces move outward, which is impossible because both end pieces would have nothing to collide with, or two pieces move toward each other, which is impossible because as soon as $a$ collides with $b$, $b$ may not move in the opposite direction as $a$ moved because $a$ is adjacent. □

*Note* 5.8. The fixed point theorem is not true for the two dimensional case; solving the exercise board requires moving all pieces.

**Theorem 5.9.** *The linear UFO configuration problem is polynomial in the number of positions.*

*Proof.* By unidirectionality and persistence, by simple examination of the desired configuration we can assign the necessary direction to each piece in linear time. Since each piece movement must displace the piece at least one position and pieces may not move past each other, with $m$ pieces and a total of $n$ positions, at most $n-m<n$ moves can be made. For each of these $n$ moves we note that two of them are independent if and only if they do not occur in mutually adjacent pieces which are moving in the same direction. Each independent group can be checked independently to see if its configuration can match its corresponding pieces in the desired configuration Suppose there are $k$ such pieces; then we note that the $k$th piece can move to one unique position, the $k-1$st can move twice (either colliding with $k$ in its original position or in its new position), and in general the $k-i$th can move to $i+1$ unique positions. There are a total of $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$ different configurations these pieces can enter. Assuming we have $n$ dependent groups each of $n$ pieces (over counting severely), there are a maximum of $O(n^3)$ different configurations we need to check. □

**Corollary 5.10.** *The linear UFO solvability problem is polynomial.*

*Proof.* We examine the independent group containing the exiting piece and in $O(n^2)$ time we can check to see if one of its configurations places the exiting piece on the exit. □

**Theorem 5.11.** *Suppose we have $n$ pieces on a linear board and a gap $g$ of size $n+k$ located between the $i$th and $i+1$st pieces. If $k>0$ then there does not exist a reachable configuration such that within $g$, any position ranging from $n-i+1$ to $n-i+k$ contains a piece.*

*Proof.* The positions $[n-i+1, n-i+k]$ are initially empty. Before the positions there is a gap of size $n-i$, before which there are $i$ pieces, and after a gap of size $i$, after which there are $n-i$ pieces. By unidirectionality, either the $i$th or $i+1$st piece may remain stationary; otherwise, we will only increase the size of the gap. However, by a simple counting argument, the remaining pieces on either side of the gap will not be sufficient to occupy any of the positions mentioned. □

We call these positions that no piece may occupy the *barren* positions. If a position is not barren, it is *occupiable*.

**Corollary 5.12.** *Within the gap above, the $n$th piece may occupy and is the only piece which may occupy the $n-i$th position.*

*Proof.* By the same argument above, either $i$ or $i+1$ must remain in the same position. If $i+1$ does we will not have enough pieces to fill up the adjacent positions such that we can occupy the position; if $i$ does only the last piece will fill the offset, such that if there were only $n-1$ pieces, the $n-i$th position could never be occupied. □

We call these positions that only the $n$th piece may occupy the $n$-unique positions. The case is symmetric for the first piece, and we refer to the positions that only it may occupy as the 1-unique positions.

**Corollary 5.13.** *(Linearity) If a move causes a $n$-unique position which has been passed through to become empty, there does not exist a sequence of moves such that it will contain a piece.*

*Proof.* Because all pieces are to one side of the $n$th piece, as soon as it moves past a $n$-unique position, it will not be able to collide with a piece on the other side, and thus will be unable to move back. □

5.3. **The Construction.**

**Definition 5.14.** The satisfiability problem.

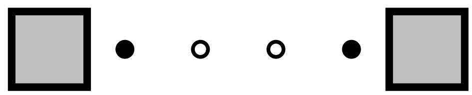
Given a boolean expression (involving variables, parentheses, negation, or, and), find values for the variables such that the expression resolves to true. Expressions can be converted to conjunctive normal form (CNF). The definition of CNF is the following: we have *literals* of the form $x$ or $\neg x$ where $x$ is a variable and $\neg$ is logical negation, *clauses* of the form $(l_1 \vee l_2 \vee ... \vee l_k)$ where $l_1, ..., l_k$ are literals and $\vee$ is logical or, and the *expression* of the form $(c_1 \wedge c_2 \wedge ... \wedge c_k)$ where $c_1, ..., c_k$ are clauses and $\wedge$ is logical and. If an expression is satisfiable, this means that there exists an assignment of truth values to the variables such that the value of the expression is true. We have some canonical ordering for the $m$ variables and refer to them as $v_1, ..., v_m$. The same goes for the $n$ clauses referred to as $c_1, ..., c_n$. We also refer to $t$ and $f$ where $t$ is the total number of appearances of true literals within the clauses and $f$ the number of appearances of false literals.

When we construct a puzzle based on a CNF expression $e$, we have a goal: some piece $d$ must be movable to some position $p$ if and only if $e$ is satisfiable. In this construction, $d$ and $p$ will begin in the same row which we call the exiting row. We will have representations of variables and clauses; each clause will be able to submit at most one piece, corresponding to a literal in the clause; each piece submitted will be able to enter the exiting row if and only if the literal is true with the assignment given; the exiting piece may exit if and only if $n+t+3f$ pieces enter the exiting row, where $n$ is the number of clauses. All pieces will fall into five categories, defined below.

For convenience, we will use a coordinate notation for the positions, with north indicating the positive $x$ direction, east positive $y$, and so on. With coordinate $(0,0)$ is not too special, we will see that the $(+,+)$ region contains variable representations, the $(-,-)$ region contains clause representations, the $(+,-)$ region contains pieces to interface clauses with variables, and the $(-,+)$ region is empty. We set row $0$ to be the exiting row.

**Definition 5.15.** The representation of variables.

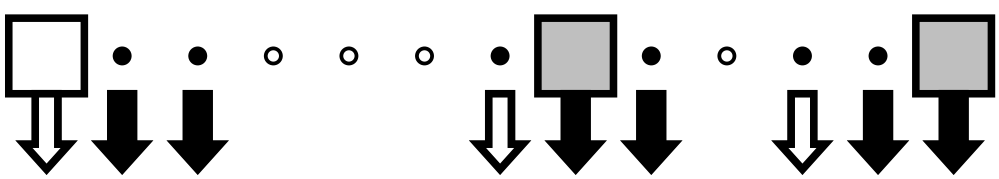
A variable is represented by two pieces in the same row with four separating them. We will assume the row is protected; this will be proven later. We call the column adjacent to the left piece the *positive column* and the column adjacent to the right piece the *negative column*. Initially the two columns are empty (not just within the variable representation but on the board itself) but by colliding one piece of the variable with the other we may fill one of them.



The location of a variable representation may be designated by its western piece. The bounding rectangle which contains all variables initially contains no other pieces. The southwestern corner of this rectangle encompasses $(1,1)$. Place the $i$th variable at $(4i, i(n+t+3f+1))$, thus bounding the rectangle at $(4m, m(n+t+3f+1))$. This places a gap between each variable of $n$ positions.

**Definition 5.16.** The representation of clauses.

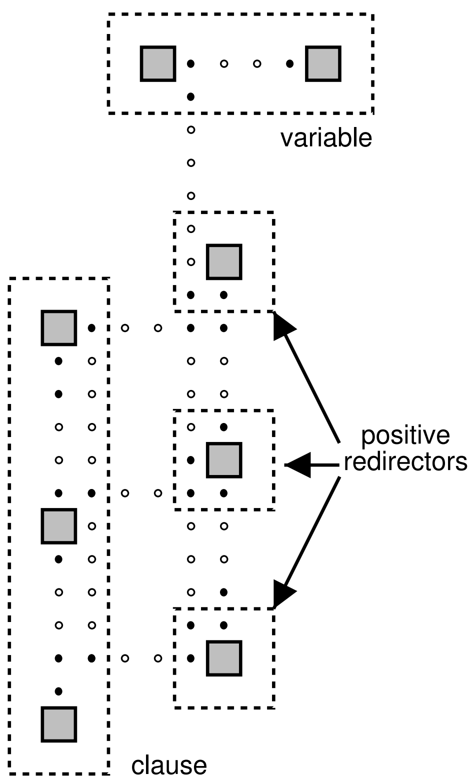
A clause of $k$ literals is represented by $k$ pieces in the same column. We will assume the column is protected; this will be proven later. Consider the first piece to be the northernmost one. The spacing between the pieces must preserve certain properties. Firstly, each position that the first piece may reach must be 1-unique. The following (horizontal instead of vertical) diagram illustrates where each of three pieces may be submitted; the white piece may only be submitted along the white arrows. Where there are no arrows, no pieces may be submitted; this will come into play shortly. We note that for a clause which contains pieces at positions $p_1, p_2, ..., p_k$, the 1-unique positions are $p_1, p_2 - 1, ..., p_k - (k-1)$. We note that positions $p_1, p_1 + 1 ... p_1 + (k-1), p_2 - 1, p_2, ..., p_2 + (k-2), ..., p_k$ may hold pieces (e.g. the interval $[p_i - (i-1), p_i + (k-i)]$) but the remaining positions may not.



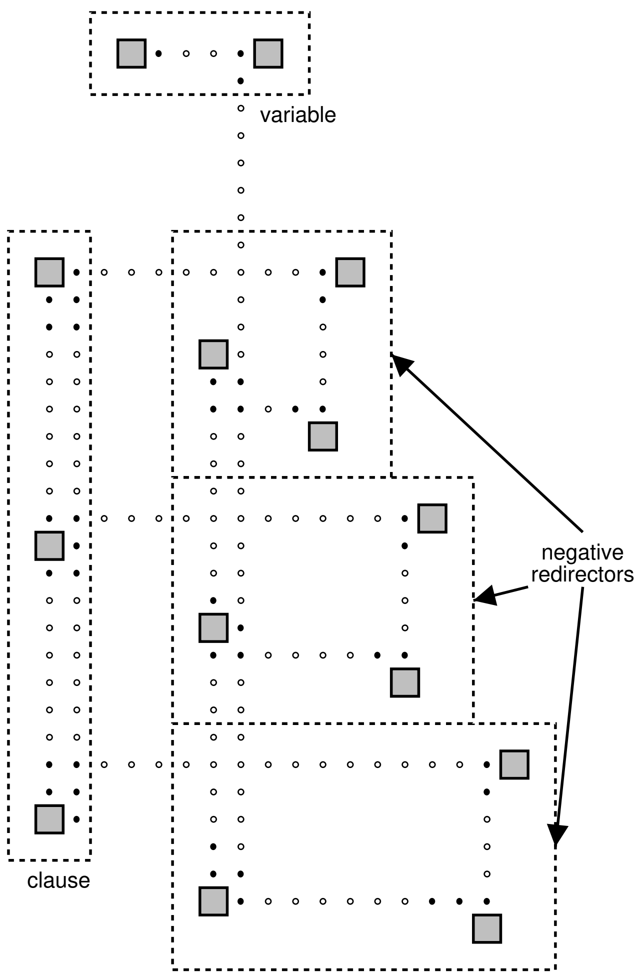
The location of a clause representation may be designated by its northernmost piece. As mentioned earlier, the clauses are confined to the southwest quadrant of the board. The $i$th clause is horizontally located at $-2i-1$. We refer to the empty filler column to the right of each clause as a *ghost column*. Vertically, the clause is defined on a piece-by-piece basis laid out downward (decreasingly) from $-1$. We keep a notion of "next position" stored in a mutable variable called $y$ which starts set as $-1$. Our subsequent definition of the redirectors will make clearer where the pieces should be laid out. Generally speaking, if a literal in a clause is the $i$th of $j$ instances of that literal, we set $y \leftarrow y - (j-i)$. We place the variable at the position designated by $y$, then we decrease $y$ to reserve the space we just made as barren and preserve 1-uniqueness. If our clause has $k$ literals, we set $y \leftarrow y - k$. If the literal was negative, we also set $y \leftarrow y - 2$ to reserve space for the extra gadget we need for the negative literal.

**Definition 5.17.** Redirectors.

We have clause and variable representations, but so far have not specified a method of tying them together. The purpose of a redirector is to allow the first piece in the $i$th 1-unique position of a clause representation to be submitted and collide with the $j$th variable representation if and only if the $i$th literal in the clause is satisfied by the assignment made to the $j$th variable representation. There are two kinds of redirectors, positive and negative, corresponding to the two assignments one can make to the variable. A positive redirector is a single piece. Effectively, we want it to redirect one of the $k$-unique positions of the $i$th clause representation of $k$ literals to the positive column of the $j$th represented variable. At this point, the submitted piece may collide with the variable representation. Since the clause piece will be traveling east, we want the redirector to be immediately east of the positive column. It would appear at first that vertically the redirector should be in the same row as a $k$-unique position. However, if we have more than one instance of the same positive literal, we will have multiple positive redirector pieces in the same column which could potentially collide. The solution, then, is to force the user to collide these pieces and strategically include barren positions. Specifically, if there are $i$ instances of a positive literal, we will need an extra gap below of $i - 1$ positions (we get the first one for free because of $k$-uniqueness) below the $k$-unique position, even if it is at the end of a clause. An example of the clause representation of $x_1 \vee x_1 \vee x_1$ appears below, along with its redirectors and its variable representation:




There are also negative redirectors. We cannot put a piece immediately east of the negative column as this would place it below the right piece of a variable. Instead, we place it directly west of the negative column. Multiple redirectors representing the same literal will have a piece in this column, so if it is the $h$th instance of the literal counting from the top, we place it $h+2$ spaces below the corresponding $k$-unique row. This piece is called a primary piece of the negative redirector. We also have two additional secondary pieces associated with a single literal. The secondary pieces of the negative redirectors are to the east of all other pieces aside from the ladder. Each pair of secondary pieces takes up two columns and has a one column gap between it and the next pair – this is a dead column which comes into play with the fifth case in the enumeration of states which follows. The first piece in a pair is in the same row as a $k$-unique position so we call it the collider piece. This allows the submitting piece to collide with the redirector. The second, which we call the turner, is one space to the west and a few south of the the first. This sets up the submitting piece to collide with the corresponding piece in the column of primary pieces. If there are $i$ instances of a given negative literal, this second additional piece will appear $i+1$ spaces below its corresponding primary piece. As before, multiple instances of the same negative literal could cause a problem with the primary redirector pieces, so we must set the gaps between literals to be large enough to accommodate the structure of several pieces within their barren rows. Specifically, the gap below the $k$-unique position corresponding to the negative redirector must be $i+2$, three more than with the positive literals: two for the extra pieces below, and one for the ghost row in between. We note that because we are coming in from below, the necessary order of piece submission is reverse from before. An example of the clause representation of $\neg x_1 \vee \neg x_1 \vee \neg x_1$ appears below, along with its redirectors and its variable representation:



**Definition 5.18.** Redirector caps.

Later we will see a need to submit all of the redirector pieces the same way we submit the 1-unique pieces. Towards this end, we place pieces at the top of each column which contains redirector pieces. The easternmost such column contains the northernmost cap; the next cap appears $n+t+3f+1$ spaces below it. The southernmost cap is still high enough such that it is $n+t+3f+1$ spaces above the northernmost variable representation.

**Definition 5.19.** Ladder and catcher.

After we have collided all of the clause representations' submitted pieces against the variable representations, we must move them to the exiting row. If $t$ is the total number of true literals that appear in the clauses and $f$ the number of false literals, the position of the ladder is $n + t + 3f$ spaces to the east of all other pieces (past the primary and secondary redirector pieces) with an additional piece being in the row below the exiting piece and immediately west of the ladder called the catcher. The pieces of the ladder are vertically oriented such that the final $m$th piece is directly below the final variable representation, the $m - 1$st 2 spaces below, and in general the $i$th is $m - i + 1$ spaces below. There are a total of $m + t + f$ pieces in the ladder so there should be enough spacing to preserve 1-uniqueness of the first piece.

This completes the list of all pieces used in the construction.

5.4. **State Enumeration.** In order for us to examine the properties of the construction, we must examine which states it can enter. We begin with a starting state and then see what classes of states various moves can produce. In particular, we want to show that the pieces interact in well-defined ways, which only allow redirector column pieces and first clause pieces from clause representations which are satisfied to enter the exiting row. The following case analysis will attempt to perform an enumeration of all reachable states relevant to the movement of the clause pieces.

*Case* 1. Structure state enumeration.

Consider all states individually enumerable by the variable representations, the clause representations, the primary redirector column, and the ladder. As each of these is a linear construction, it is not difficult to enumerate all of the states that they may enter, and within our specification we have given hints as to how to approach their usage.

*Case* 2. Initial submission.

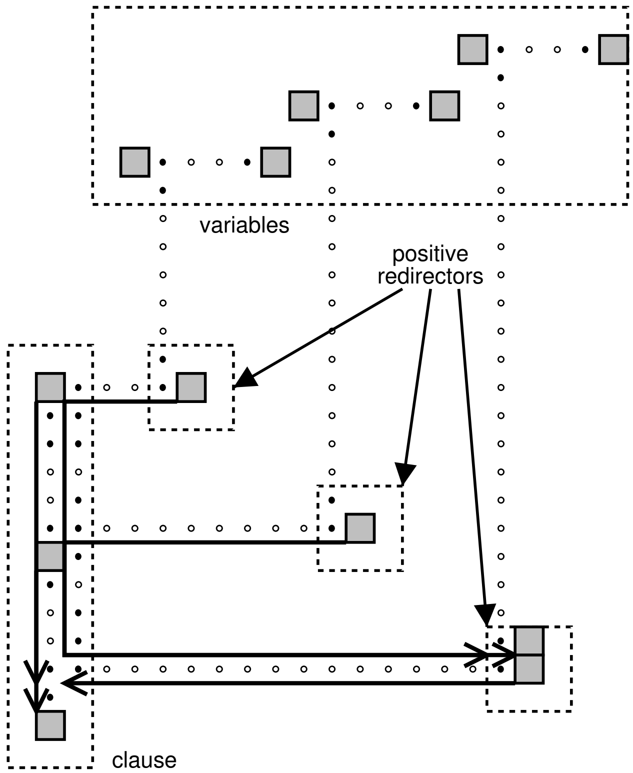
How can one of these structures interact with another structure given that we have only performed first order moves? Given the arrangement of our structures, one can see that we can only line up the first piece in a clause representation with various redirectors corresponding to the literals of that clause. Therefore, there are two possible interactions: either the clause representation collides with the redirector, or the redirector collides with the clause representation.

*Case* 3. Ghost column, secondary submission.

Suppose the clause representation collides with the redirector. Then it must have been the first piece. We are therefore in a desirable state where we can continue the construction to completion.

Suppose the redirector collides with the clause representation. Then we may place a piece in the clause representation's ghost column in a 1-unique row. In general, each literal which appears in some clause may be moved once into the ghost column of a clause it appears in. For some clause, then, the ghost columns can fill up the rows $p_1, p_2 - 1, ..., p_k - (k - 1)$. This makes the positions $p_1, p_1 + 1 ... p_1 + (k - 1), p_2 - 2, p_2 - 1, ..., p_2 + (k - 3), ..., p_k - (k - 1)$ reachable, in general $[p_i - 2(i-1), p_i - 2(i-1) + k]$. The full combined intervals are of the form $[p_i - 2(i-1), p_i + (k-i)]$.

At this point we notice a problem. Namely, a piece sent into the ghost column could cause more than one piece to be submitted to a clause. However, we notice that the extra pieces we submit must be redirector pieces since these extra submissions must occur in rows above that which the first piece is in. The following diagram illustrates the problem:




Luckily, the redirector cap allows all redirector pieces to be submitted anyway, so this gives us nothing more than a pointless alternative solution procedure. In fact, the puzzle cannot be solved once we do this, since at least one of the redirector pieces must remain in the ghost column where it cannot be submitted, and since the exiting row may contain at most $n + t + 3f$ submitted pieces, but it needs this many to be filled, the lack of a single piece will cause the puzzle to be unsolvable.

*Case* 4. Crashing variables, tertiary submission.

Suppose a piece collides with the variable. It must be an errantly submitted redirector piece, or a first piece from a clause representation. In the former case the piece could be submitted anyway so nothing is wrong; in the latter case we are solving the puzzle correctly.

On the other hand, suppose the variable collides with an errant redirector piece or a first clause representation piece. Then we can reserve a barren row such that the variable cannot perform any additional moves.

In the case of a negative literal, we could also have our first clause piece collide with the turner, or collide the turner with it. Again, a barren row solves the problem if the second literal piece collides with the first clause piece, but if the first clause piece collides with the second literal piece we are solving the puzzle correctly

*Case* 5. Quaternary and quinary submission.

In the case of a negative literal, there are two more collisions which must occur before the first clause piece is adjacent to the variable. Firstly, it must collide with the piece in the primary redirector column. Alternatively, the piece in the primary redirector column could collide with it, but reserving a dead column for this case is easy since we can horizontally isolate our secondary negative redirector pieces as much as necessary. Secondly, it must collide with the variable itself; as with the tertiary submission of a positive variable, we can reserve barren rows to prevent the collision of the variable with the first clause piece from doing anything inappropriate.

At this point, we have exhausted all movement classes, since we have shown the only things which can happen in the process of moving the first pieces from clause representations such that they have collided with their respective literals do not increase the domain of solvable puzzles from those which represent solvable puzzles.

**Theorem 5.20.** *A represented expression is solvable if and only if the original expression is solvable.*

*Proof.* We have shown above that only the redirector column pieces and first pieces from clause representations which are satisfied may collide against variables. Because we need these pieces in the exiting row to allow the exiting piece to move to the exit, the construction is solvable only if the expression is satisfiable. Once the pieces are in this position the only pieces they can interact with are the ladder pieces. Here is a solution procedure

(1) Set the variable representation to a satisfying assignment.
(2) For each clause which is satisfied by a positive variable, beginning with the first one which is, send the $k$th piece so that it collides with its positive redirector and then with the positive variable (or other $k$th pieces which have collided with the positive variable). Then, if there is a subsequent clause which has a positive instance of this variable, collide all positive redirector pieces at this clause with the subsequent one.
(3) For each clause which is satisfied by a negative variable but not a positive one, beginning with the last one which is, send the $k$th piece so that it collides with its negative redirector and then with the negative variable (or other $k$th pieces which have collided with the negative variable). Then, if there is a preceding clause which has a negative instance of this variable, collide all negative redirector pieces at this clause with the preceding one.
(4) For each redirector piece beneath the caps, send upward, starting with the northernmost ones in each column first.
(5) Beginning with the northernmost cap and working backwards, collide the piece touching the cap against the ladder, then down, and then left against the exiting piece. Then if there is another redirector piece in that column, collide it against the cap. Finally, once the redirector column's pieces are exhausted, collide all ladder pieces at that redirector cap with the next ladder piece.
(6) Beginning with the last variable and working backwards, collide the piece touching the variable against the ladder, then down, and then left against the exiting piece. Then if there is another piece which used the variable, collide it against the variable. Finally, once the variable representation's pieces are exhausted, collide all ladder pieces at that variable representation with the next ladder piece. With the very last piece we move in this way, collide down but not left against the other pieces which have collided against the exiting piece.
(7) Move all the pieces which we moved into the exiting row to the right against the one piece which we did not move left. This will fill the end of the exiting row, allow the exiting piece to exit.

. □

## 6. Conclusions

From a purely pragmatic view, we have shown that solving the UFO Puzzle is intractable, assuming that NP-complete problems are intractable. Perhaps this means that we must be careful to plan ahead with space missions involving sliding robots. Our construction also merits the creation of the aforementioned UFO-X where a piece may not revisit a position on the board. Because solving our construction does not require that a piece revisit a position, it automatically shows that UFO-X is NP-complete. As we stated earlier, there are a number of important questions which this paper leaves open. The most important is determining whether Solvability is in NP or is PSPACE-hard, a question which I intend to continue researching. Perhaps an easier question is whether Configuration is NP-hard – the fact that linear configuration has a more complex algorithm than linear solution may indicate that 2D configuration is also difficult. The question of whether the Push games are PSPACE-hard or NP-complete is also interesting and perhaps related in that they too are stated easily as simple planning problems. A less complicated NP-hardness construction would also be desirable.

The solution to the exercise given near the beginning of this paper is as follows: 2 west, 3 west, 2 south, 1 east, south, 2 east, 5 north, 2 west, 4 west, 2 north, 5 east, north. (The problem is taken from the set of puzzles included with Lunar Lockout.)

## 7. Acknowledgements

Thanks to Danny Sleator, Venkatesan Chakaravarthy, Kaustuv Chaudhuri, Scott Crosby, Tom Murphy 7, and Jason Reed for some helpful discussions.

## References

[1] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69, 1994.
[2] Erik Demaine and Michael Hoffmann. Pushing blocks is np-complete for noncrossing solution paths. *Proceedings of the 13th Canadian Conference on Computational Geometry*, pages 65–68, 2001.
[3] Martin Demaine, Erik Demaine and Joseph O'Rourke. Pushpush and push-1 are np-hard in 2d. *Proceedings of the 12th Annual Canadian Conference on Computational Geometry*, pages 211–219, 2000.
[4] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry: Theory and Applications*, 13:215–228, 1999.
[5] P.E. Fikes, R.E. Hart and N.J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3, 1972.
[6] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.