Final Project Report
March 14, 2018
Julia Connelly

Introduction

There's a puzzle game called Hexcells. The board looks something like this:

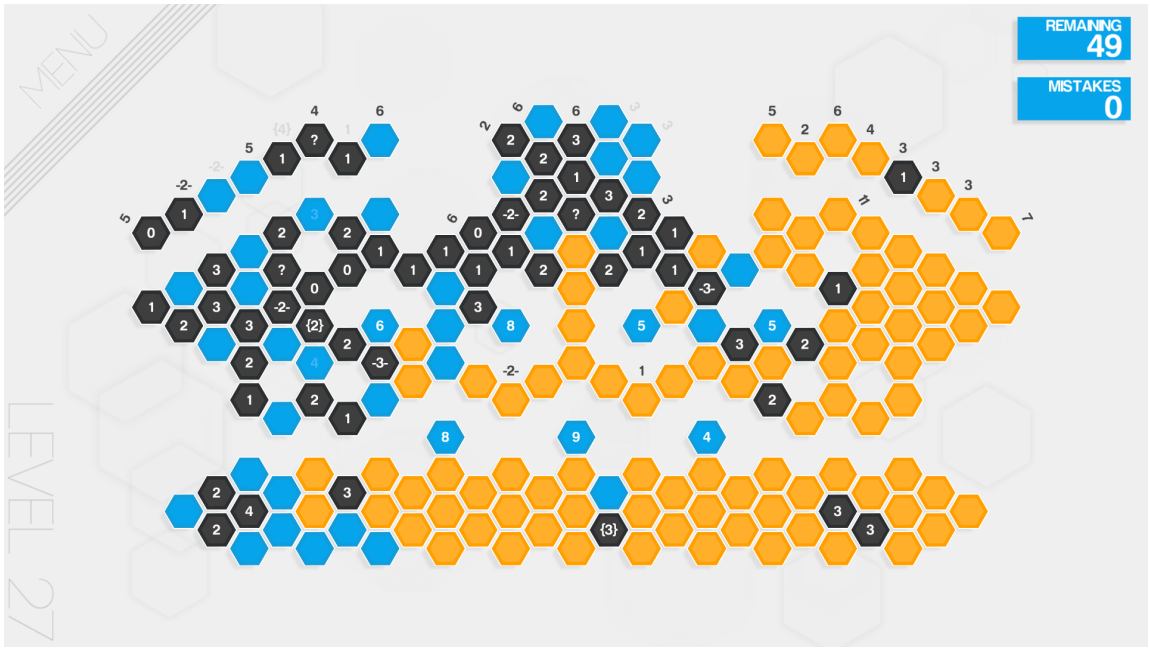


Figure 1: A Hexcells board: they aren't all shaped the same, but they all follow the same rules.

All Hexcells boards aren't shaped the same. They can be pretty much any shape you can think of. The goal is to get rid of all the gold hexes. Either you turn a gold hex into a blue hex or a black hex. The numbers on the columns and diagonals tell you how many blue hexes can be in that line. Similarly, the numbers on the black hexes tell you how many blue hexes can be touching that hex. Numbers with {} around them mean the blue hexes involved in that constraint have to be touching each other. On the other hand, the numbers with −− around them mean that not all the blue hexes involved in the constraint can be touching (although some can be touching, just not all). This constraints are enough to decide which cells are blue and which are black. When you reveal black hexes, they provide you with more constraint information.

The goal of this project is to write a program that can solve Hexcells puzzles.

Methodology

Overview

This problem is clearly a constraint satisfaction problem (CSP). The algorithm I decided to try was a CSP search with forward-checking. A CSP search is very similar to a regular search like depth-first search. The difference is that in a CSP search you can easily prune branches off of the search tree.

In a normal search, you don't know which branch the answer will be down until you get to the end of it. But, in a CSP, you can use the constraints to guide you. Even when you aren't at the end of the branch, you can check to see of the current state is valid given all constraints. If it is, then keep searching. Otherwise you can stop searching that branch entirely.

Forward-checking is an optimization on CSP search. In regular CSP search, at every node in the search tree you check to see if the state you're in is valid. Forward-checking moves this validation back a level. Instead, before you add any new nodes to the search frontier, you check to see if whatever change you're making to the current state to create that new node is valid. This further reduces the number of nodes explored during the search process.

I also added another optimization specific to this problem. Some Hexcells puzzles are really simple. In simple puzzles, you can solve a given hex by only looking at one other constraint. For example, say a column is labeled with a 5 and there are only 5 hexes in that column. Then you know that all 5 of those hexes have to be blue. With that in mind, I have a preprocessing step that aims to solve some hexes using basic hard-coded strategies which only consider one of the constraints affecting a given hex. This shrinks the state space for search.

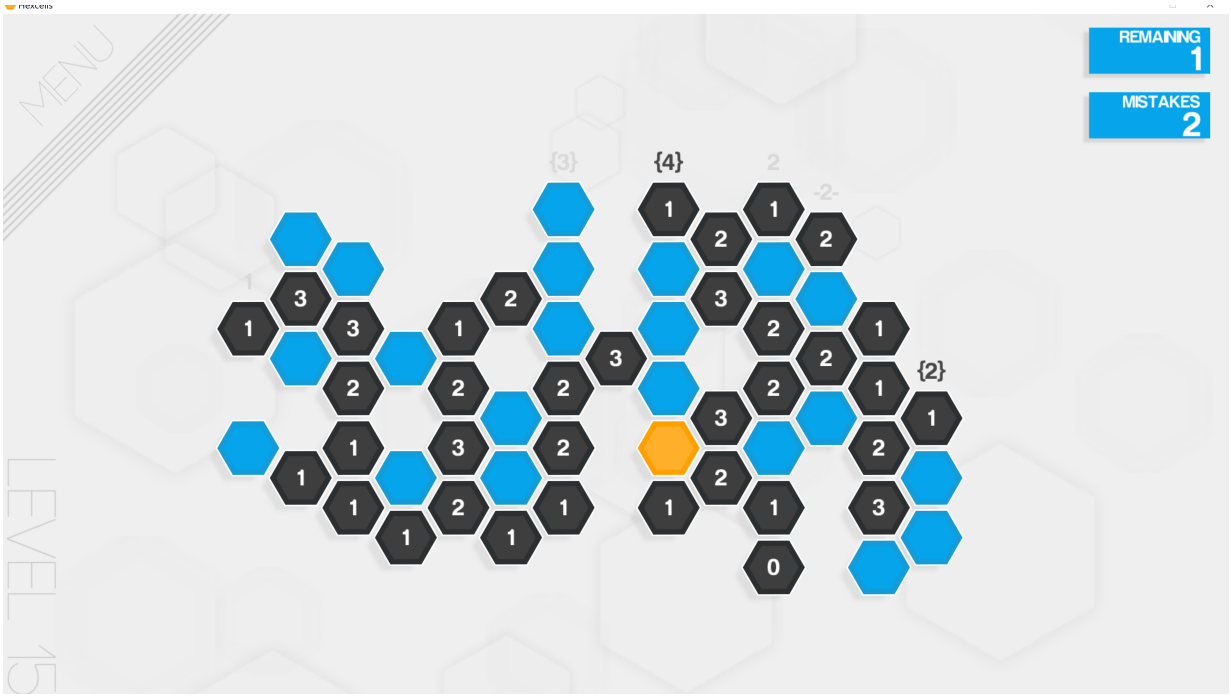
I implemented this algorithm using Python.

Implementation

Preprocessing

First, I needed some way to take in a puzzle as input. I encode a puzzle by turning it on its side so the columns become rows. Then I have different symbols that act as different hexes. Below that are more values with details about column and diagonal constraints. Here's an example:

```
_ _ y _ .1
 _ .1 _ y 3 y
_ .1 .1 .2 .3 y
 .1 y _ y
_ .2 3 2 .1
 .1 y y _ 2
_ 1 .2 .2 y y y
 _ _ _ 3
_ 1 y y y y .1
 _ .2 3 _ .3 .2
0 .1 y .2 .2 y 1
 _ _ y 2 y .2
y .3 .2 1 .1
 y y .1
+
+
0 1
6 {3
8 {4
10 2
11 -2
13 {2
```



The _ are empty parts of the board. The y's are gold hexes that are blue underneath. The numbers are self-explanatory. When there's a . in front of anything, that means that value starts outs as a gold hex.

The numbers after the plus signs describe column and diagonal constraints. The first number in each tuple is the index of the given column or diagonal once the information gets parsed. The second is the constraint on that column or diagonal.

After reading and parsing this input, I created a dictionary that mapped each hex to each constraint the hex was a part of. To do the preprocessing part of the algorithm, I essentially loop over each constraint for each hex and check if I can solve that hex. Here are the simple strategies I hard-coded:

1. If the hex is part of a constraint where all the blue cells have already been revealed, make the hex black.

2. If the hex is part of a constraint that has as many gold hexes as hexes left to fulfill the constraint, make the hex black.

3. If the hex is a part of a $--$ constraint and there are $n-1$ revealed blue hexes that are all adjacent to each other where n is the number of blue hexes there should be in the constraint and the hex is adjacent to any of those blue hexes, make the hex black.

4. If the hex is part of a {} constraint and there are some revealed blue hexes and the hex is far enough away that it couldn't be blue and still retain adjacency with the blue hexes, make the hex black.

After looping through all the constraints for each hex, I then update the constraints to include any newly revealed constraints and loop again. This repeats until either the puzzle is solved, or the algorithm goes an entire loop without being able to solve a hex.

Search

Once the preprocessing is done, I perform a CSP search to fill in the rest of the puzzle. The search is a variation on depth first search. I decided to use DFS instead of BFS because BFS is concerned about the shortest path through the search space. In this case, all branches to a completed board state would be the same length, so I don't care about the shortest route. Then I can eliminate any overhead BFS has by choosing DFS. Here is the psuedo code for the algorithm I used:

```
constraints = get_constraints()
hexToGuess = get_unsolved_hex()
frontier = [{hexToGuess: BLUE}, {hexToGuess: BLACK}]
visited = []

while frontier is not empty:
    cur = pop the first item from the frontier
    add cur to visited
    hexToGuess = get_unsolved_hex()

    if hexToGuess is null, then we're done

    for all constraints on hexToGuess:
        check if hexToGuess could be colored blue or black without violating constraints

    if hexToGuess can be blue:
        guess = copy cur
        add {hexToGuess: BLUE} to guess
        add guess to the front of the frontier

    if hexToGuess can be black:
        guess = copy cur
        add {hexToGuess: BLACK} to guess
        add guess to the frontier
```

Notice that this algorithm does not account for new constraints revealed during search. This is because accounting for the initially hidden constraints is not possible using this algorithm. More on this later.

Runtime

Let c be the number of constraints on a board, and n be the number of unsolved hexes.

The preprocessing step takes $O(cn)$ time. In the worst case, every hex is involved in every constraint, so we do $O(c)$ work n times. This n-times loop may repeat multiple times, but that has no affect on the $O(cn)$ runtime as it would be a constant number of loops.

DFS has a runtime of $O(V + E)$ where V is the number of vertexes and E is the number of edges in the graph. In this case, the number of vertexes is the number of possible ways to fill in a board. Each unsolved hex can be one of two colors, so there are 2^n possible configurations. I'm running this search on a search tree, so each node only has 1 incoming edge and 2 outgoing edges at most (one for coloring a given hex blue, the other for black). That means there are about $2V$ edges in the tree. Then, $O(V + E)$ becomes $O(2^n)$, which is not great. However, the pruning done during search greatly limits the runtime in reality.

Results

Here are some examples of puzzles this program successfully solves:

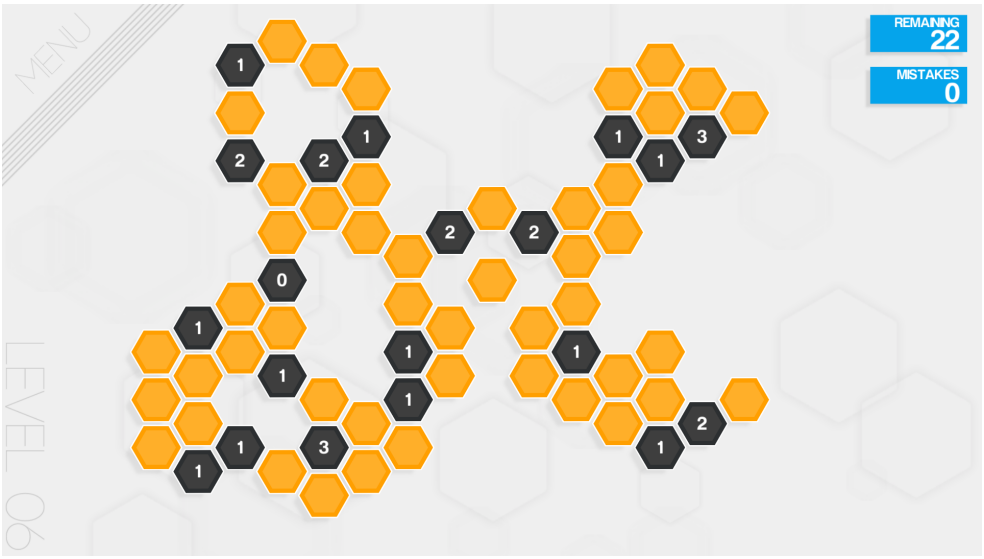


Figure 2: An easy Hexcells puzzle

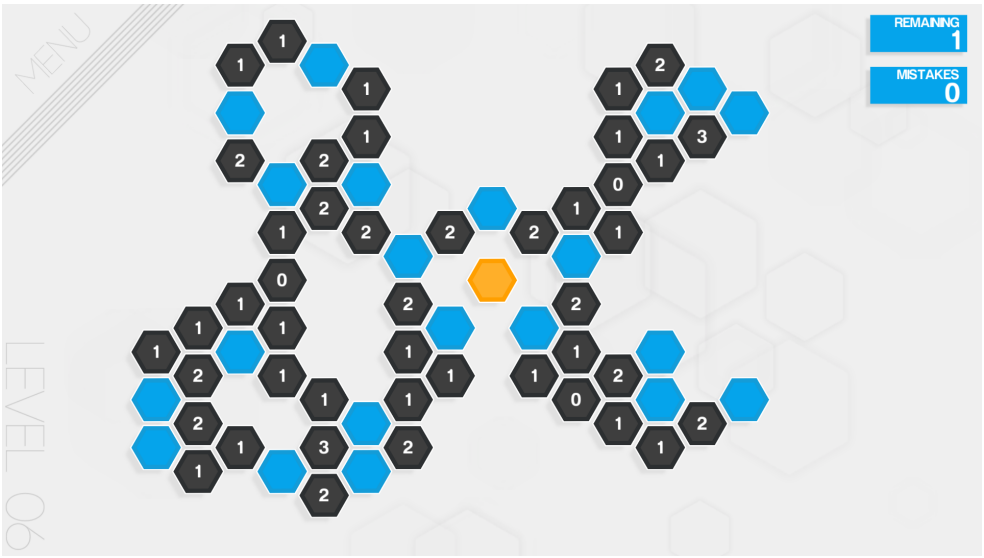


Figure 3: The above Hexcells puzzle solved

Figure 4: An easy Hexcells puzzle

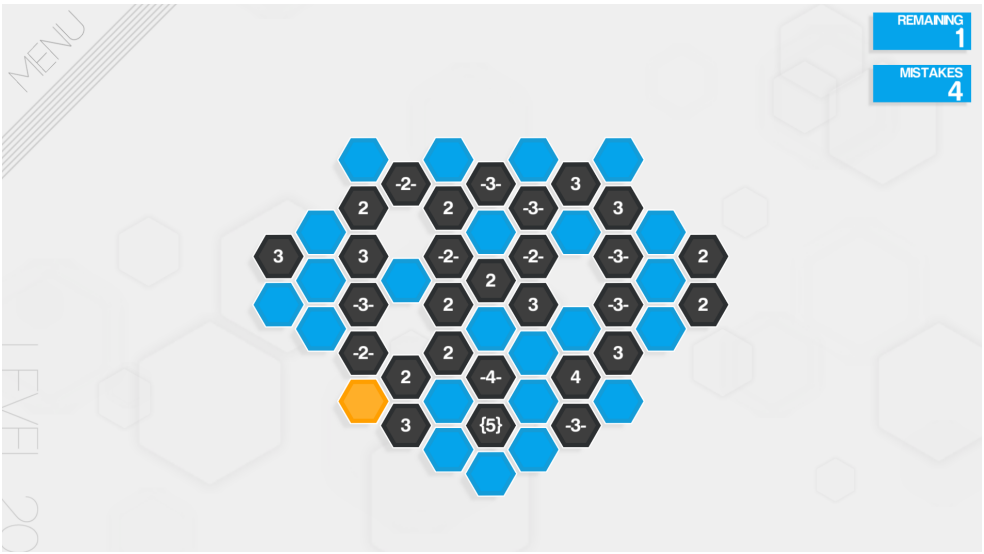
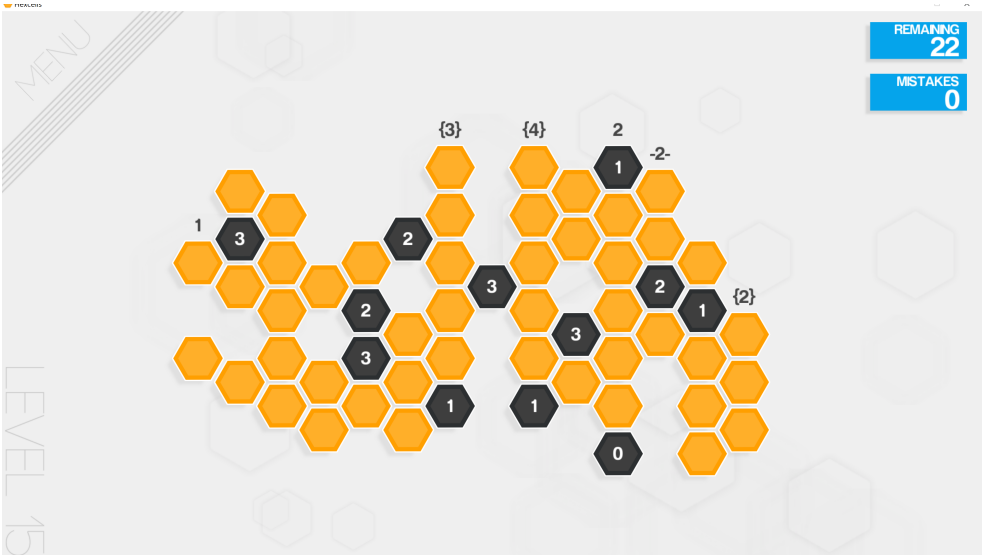
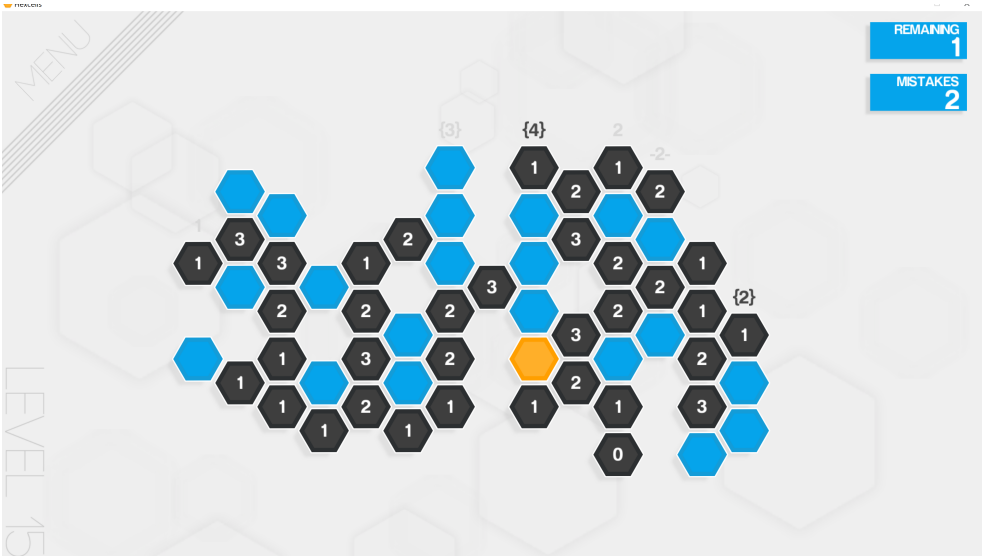
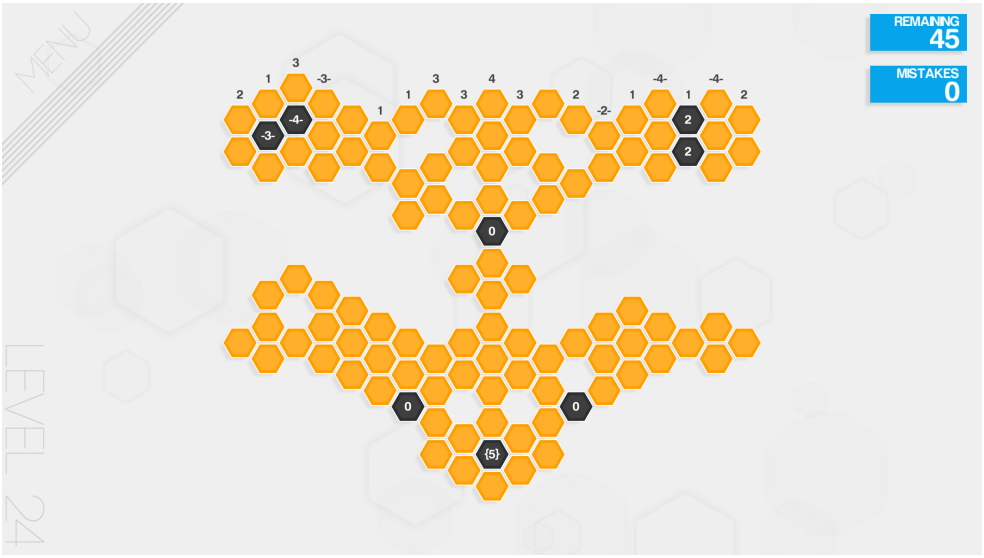


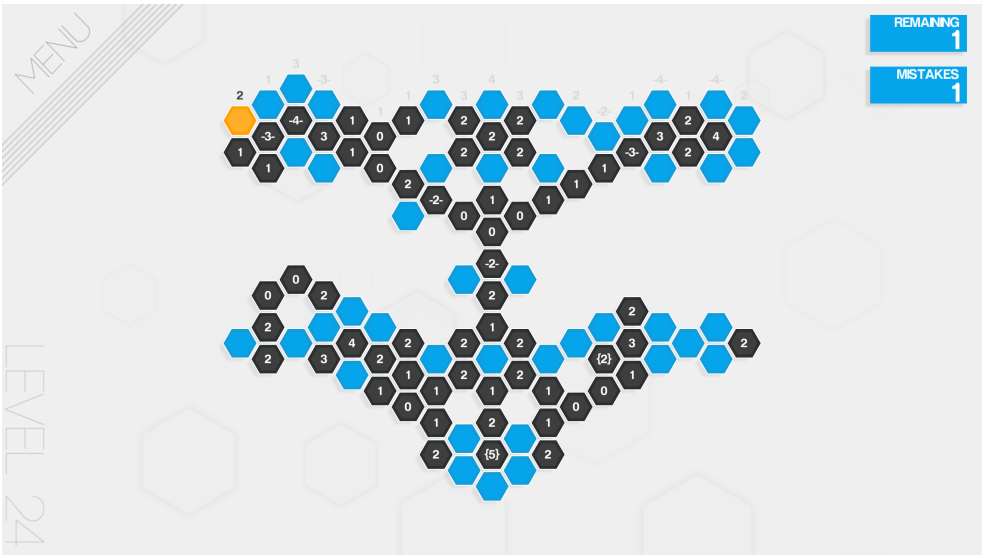
Figure 5: The above Hexcells puzzle solved

And here are some examples of puzzles it cannot solve:






MENU
LEVEL 24
REMAINING
45
MISTAKES
0


MENU
LEVEL 24
REMAINING
1
MISTAKES
1

Analysis

Preprocessing

The preprocessing step often either solves the entire puzzle on its own, or doesn't do very much at all. A lot of the early levels in Hexcells provide you with enough constraints and a small enough board that the entire board can be solved with the very simple preprocessing strategies. One thing preventing this algorithm from solving more puzzles is that it can't handle filling in the first blue hex on a given constraint (except in the very basic case when all gold hexes in a constraint must be blue). If I added the following strategies, this algorithm would actually be pretty robust:

1. For the diagonal and column {} constraints, imagine filling in the adjacent hexes from one edge of the line and then the other. If those two lines overlap at all, those hexes that are in the overlap can be blue.

2. If a $--$ constraint has $n + 1$ gold hexes touching it (where n is the constraint number), and n of those hexes are all adjacent, then the one gold hex that is not adjacent can be colored blue.

Note that all of these simple strategies are informed from my experience playing the game. I know how the game developer likes to build his puzzles, so I can tailor strategies towards that end.

Providing lots of data for this problem is difficult, because I have to hand-translate the hexcells puzzles into a parse-able format. However I can say that the puzzle in Figure 2 looped through all the constraints 6 times to solve the puzzle. The one in Figure 4 looped 5 times to solve it. In contrast, the puzzle in Figures 6 and 8 could not have any hexes solved by preprocessing. The strategies are not complex enough, although adding the two I mentioned earlier may help with the puzzle in Figure 6.

Search

Running this CSP search on more complex puzzles is not enough on its own to solve them. With just the strategy I outlined previously, this search has no way to reveal any new constraint hexes. Searching is basically just guessing and checking. They way you check if a given hex is solved correctly is by filling in other hexes assuming that one hex is a given value. If the filling in leads to a completed board, you know that assignment was valid. But, if you have to fill in the whole board to know if your assignment was correct, there's no way you can reveal any new constraints. If you have to wait until you assign all the hexes to a value before you can reveal them, you'll never be able to see the constraints when you're assigning values to hexes. This means that whatever assignment you come up with will likely not be valid because it's ignoring lots of constraints.

So, this basic CSP search fails on this problem. But there are some potential modifications that could be made to solve it. First, we should constrain the search space to only consider hexes that are affected by constraints. The game was designed to be solvable, so by looking at all the constrained hexes it should be possible to solve some hexes for sure. Then, once those hexes are solved for sure we can reveal new constraints and expand the search area.

Second, the only way to know if a hex can be made blue is if it's not black, and vice versa. So we don't want to fill in any hexes that could be both, because that means we don't have enough information to solve them. The current CSP search just takes a guess, and if it works it rolls with it. Since the search is not using all the constraints of the board, it's possible that assignment could lead to a "valid" board state.

So, here's what we want to be able to do: find a hex that, when assigned to a given value, causes some constraint break later. Basically, try to code the strategy of "If I color this hex black, then this one for sure has to be blue, and this one has to be black and ... but then this one can't be blue or black, so the original assignment was wrong".

Adding a Most-Constrained First (MCF) strategy to this search would bring the algorithm closer to this idea, but it still wouldn't guarantee a successful solve. However, this problem is difficult to solve from a search perspective because of the guess and check nature of search.

I did some research to see if others have solved this problem in a different way, and I came across Sixcells [2]. Someone coded a hexcells puzzle creator that has a hexcells solver included. The way they solved this problem was by using PuLP, which is a linear programming toolkit for Python. At a high level, it seems like the Sixcells author was able to formulate the hexcells problem in a more concrete way such that he was able to pass in constraints and variables into a library which would spit out solutions. I did not have time to look into this approach, but it seems simpler and closer to a true CSP formulated problem then the search was.

Ethical Impact

If artificial intelligence can reliably solve puzzles better than humans, that could result in a sense of lost uniqueness. If a core part of your identity involves how good you are at solving puzzles, then this type of AI may leave you wondering what makes you special.

Similarly, puzzles are supposed to be fun. They're designed to challenge peoples' minds. But if there's an AI out there that can solve all this complex, well-defined puzzles extremely well, that may take the excitement out of solving a puzzle.

On a more general level, artificial intelligence is good at solving highly-constrained problems like this one. There are lots of jobs in the world that involve doing highly constrained tasks e.g. bookkeeping. So these types of job markets may be taken over by AI in the future, which would be bad for the people losing jobs and could be bad for the economy.

If you gave an AI like this enough constraints and enough time, it could probably solve some of mathematics most difficult problems. If AI can solve difficult math problems, then needing to learn the behind those calculations would potentially become obsolete. There are jobs that don't require any sort of math degree to get, but math is an important part of the job, e.g. economists. Currently, economists may have to learn how to solve some stats problems. But if there exists an AI that can take in a bunch of information and spit out the answer, the economists may think there's no reason to learn the math backing it up. This would be bad. Knowing the math and what it means allows you to do the proper analysis on the data. The AI can spit out a lot of numbers, but you would have a much better understanding of what those numbers mean if you knew how the AI had solved the problem. Not knowing the numbers origins could lead to inaccurate analysis which could negatively affect whoever is using the analysis to inform their actions.

Similarly, if AI can solve really *really* difficult problems in mathematics, some math theorists may be out of a job. If you can toss an AI a bunch of constraints and know it will come up with an answer after a set amount of time, then math theorists would be obsolete. Although, theoretically all the theorists could be helping the team that builds the AI to solve difficult math problems. Also, these complex theory questions in math are not easy to constrain concretely, so if this happens at all it's a long way away.

Overall, an AI that can solve a really simple puzzle in one specific video game is not going to take over the world. But looking at extensions of this type of problem solving leads to some potential ethical issues.

Conclusion

In conclusion, hard-coding simple strategies is enough to solve simple hexcells puzzles. However, CSP-search is not enough on its own to solve more complex puzzles. More optimizations and strategies need to be added.

References

[1] RUSSELL, STUART NORVIG PETER. ARTIFICIAL INTELLIGENCE: a modern approach. PEARSON, 2018.

[2] oprypin, Sixcells, https://github.com/oprypin/sixcells