CSC384 : Final Project

Christian Lessard

Robertson McClure

Jeffery McBride

Table of Contents

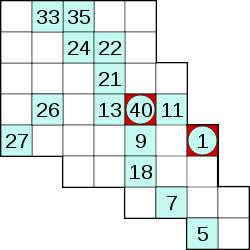
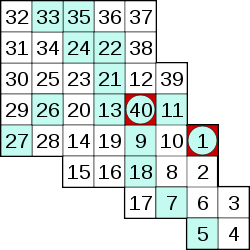
1. Project Motivation
2. Methods
   1. Formulating the CSP – A Change in Implementation
   2. Representing Assignments and Defining Constraints
   3. Comparing Assignment Heuristics
   4. Comparing Propagators
3. Evaluation and Results
4. Limitations and Obstacles
5. Conclusions
6. Works Cited

01. Project Motivation

*Hidato* is a logic puzzle game invented by Israeli mathematician Dr. Gyora M. Benedek. The game takes the form of a grid hosting several numbers: a *starting number* (typically one), an *ending number* (matching the number of cells on the grid) and some *intermediate numbers* (to limit the number of possible solutions). A player must solve the puzzle by forming a path of adjacencies such that any given number on the board has both the precedent and subsequent number as a neighbour.

Hitado is known to be NP-Complete[[1]](#footnote-1), meaning that no known polynomial time algorithm exists to solve instances of the game. Pursuing such would be extremely difficult.

Instead, we decided to interpret Hidato as a Constraint Satisfaction Problem (CSP), using techniques from the field of Artificial Intelligence to derive solutions to game instances in a reasonably short amount of time. Such an interpretation also allowed us to test various heuristics and propagators so that we may examine their comparative performance.



*Figure 1: A Hidato board with starting number 1, ending number 40, and intermediate numbers 5, 7, 9, 13, 18, 21, 22, 24, 26, 27, 33, and 35*

*Figure 2: the solution for Figure 1.*

02. Methods

**Formulating the CSP – A change in implementation**

After studying CSPs in CSC384, we implemented one for a game called Tenner using provided backend code that was extremely expensive in terms of space complexity. This code relied on each CSP constraint having access to a pre-calculated list of every possible satisfying assignment of the variables in its scope. Although this made constraint evaluation quick (with the use of a dictionary), it came at the expense of spending time generating each satisfying assignment. We decided that we could rewrite the backend code to dramatically improve this space-complexity issue, without doing so at the cost of efficient time complexity. At first, we believed it would suffice to pass an evaluation function to each constraint. We noticed, however, that constraints should only involves pairs of neighbouring numbers. We decided to give every constraint the same structure (checking that three adjacent numbers were placed next to each other on the board), allowing us to define a single, constant time evaluation function for all of them.

**Representing Assignments** **and** **Defining Constraints**

As mentioned, all constraints share the same structure: they ascertain that three adjacent numbers (such as 1, 2 and 3) are placed next to each other on the board. This would not have been possible if we had not made an important decision about encoding data into the CSPs variables. Initially, we believed it would be sensible to encode each cell on the board as a variable, using possible numeric assignments as the constituents of their current domains. Instead, we decided it would be better to encode numbers as variables, with possible coordinates on the board constituting their current domains. Had we gone with the former idea, it would have been difficult to come generalize the structure of our constraints and, more importantly, to create a single, uniform evaluation function for them.

**Comparing Assignment Heuristics**

Another result of our variable encoding what that of the assignment heuristics made available to us. The use of propagators like Forward Checking and Generalized Arc Consistency provide Backtracking Search with a heuristic – known as the Minimum Remaining Value heuristic – for making decisions about which variables to assign next. This simply has Backtracking Search pick to assign whichever variable has the smallest current domain. Because we assign coordinates to variables, we were also able to use the “Next In Line” heuristic, which simply has Backtracking Search pick the next smallest unassigned number. The comparison of these heuristics will be presented in the next section.

**Comparing Propagators**

We tested two propagators against regular Backtracking Search. These include Forward Checking and Generalized Arc Consistency, as seen in class.

03. Evaluation and Results

Here, describe your evaluation objectives and strategy, and your results. In particular, describe the way you’ve chosen to evaluate your approach (i.e. how you will determine if your approach works). Evaluation metrics could include the number of nodes expanded in a search algorithm or the amount of time or memory that you used. We encourage the use of diagrams, graphs, and/or tables to summarize experimental results and to convey important points. Note that it’s ok if your system proves to be inefficient in some way; that’s still a result and we want to know. In addition to graphs and tables, provide a written summary of your findings and their implications, if any.

To evaluate a solution, we wrote a polynomial time verifier function. This function, encoded as a method called “verify” in class CSP, begins with the variable encoding the starting number 1, and attempts to traverse the subsequent neighbours checking for their adjacencies on the board.

* Polynomial time verifier

04. Limitations and Obstacles

Here, document any obstacles you encountered during your implementation or shortcomings you discovered in your solution approach.

* NP completeness

05. Conclusions

Finally, explain what you learned and how you might improve or modify your program were you to try again in the future. Other reflections are welcome.

06. Works Cited

1. See <http://www.nearly42.org/cstheory/hidato-is-np-complete/> [↑](#footnote-ref-1)