Project 2 Report

Mikey Ferguson

October 2024

Generic Constraint Satisfaction Problem Solver:

Ultimately, all discrete constraint satisfaction problems (CSPs) can be represented with a graph, so I started with creating an arbitrary solver which can take in any such graph. CSP graphs are defined by their nodes, how those nodes are connected to each other, the set of assignable values available to each node, and the rules applied to any two connected nodes (for instance, in the case of both graph coloring and Sudoku, no two adjacent nodes can be assigned the same value). The root of a solving algorithm is recursion, in which we repeatedly assign nodes values without violating any constraints until all nodes are assigned:

```
def solve() -> bool:
    if all nodes are assigned:
        return True
    next <- next unassigned node
    for each value available to next:
        assign value to next
        update available options of each neighbor of next
        if we break desired k-consistency:
            undo updating of any nodes in the graph
            continue
        else:
            if solve():
                return True
            else:
                undo updating of any nodes in the graph
    // If we complete this loop, no variable assignments worked
    return False
```

Due to the stack-like nature of recursion, this is essentially depth-first-search to find a solution - that is, a valid assignment for all variables while meeting all constraints. Because solutions to a CSP all have the same "depth" - the number of variables - depth-first-search is always preferred over breadth-first-search because the runtime of the two algorithms is asymptotically identical, but the memory of depth-first is linear versus the exponential memory required of breadth-first.

When none of a variable's available values offer a valid assignment for said variable - whether that means each variable failed the desired k-consistency, or after assigned each variable, a recursive call to assign the rest of the graph failed - then that variable returns False. This in turn causes *previously* assigned variables to try different assignments. In this way, the algorithm tries a branch of assignments, and if that branch leads to a dead end it backtracks until it runs into a variable that has another valid option to assign to itself, which leads to a new assignment branch. The algorithm terminates when it travels along some assignment branch successfully all the way until all its variables were assigned values.

Consistency:

One of the topics mentioned above is *k-consistency*, which ensures that, without violating the problem's constraints:

$$\forall \text{ sets of } k \text{ nodes}, \forall \text{ node } a \text{ in a given set}, \forall \text{ different node } b \text{ in the set},$$

$$\exists v_2 \in \{\text{available variables for } b\} \text{ such that } b \text{ could be assigned } v_2 \textit{ and}$$

$$\text{each } v_1 \in \{\text{available variables for } a\} \text{ could still be assigned to } a.$$

For $k = 0$ consistency (backtracking, i.e. BT), we simply perform depth-first search while at each step making sure our problem's constraints are met. Once a variable is assigned a value, look at each neighbor and remove newly unavailable values from said neighbor's set of options. The simply move onto the next assignment:

```
def check_0_consistency(newly_assigned) -> bool:
    for each neighbor of newly_assigned:
        remove options from neighbor to preserve constraints
    return True
```

We won't know if we have created a problematic set of assignments until some node is told to assign itself a value, and it has no values to pick from, at which point the *solve()* algorithm backtracks.
For $k = 1$ consistency (forward checking, i.e. FC), once a variable is assigned a value, and its neighbors' available values are accordingly, updated, we return *False* if any neighbor ran out of available values:

```
def check_1_consistency(newly_assigned) -> bool:
    for each neighbor of newly_assigned:
        remove options from neighbor to preserve constraints
        if the neighbor set of available options is empty:
            return False
    return True
```

Finally, for $k = 2$ (arc-consistency, i.e. AC3) consistency, once a variable is assigned a value, update its neighbors' available values accordingly, and if any were removed, we now need to update the *neighbor's* neighbors' values, and so on until all edges of the graph involving updated nodes have been addressed:

```
def check_2_consistency(newly_assigned) -> bool:
    edge_queue <- new queue
    for each neighbor of newly_assigned:
        edge_queue.enqueue((newly_assigned, neighbor))
    while edge_queue is not empty:
        next_edge = edge_queue.dequeue()
        node, neighbor = next_edge[0], next_edge[1]
        remove options from neighbor to constraints
        if the neighbor has no options left:
            return False
        if any options were removed from neighbor:
            for each next_neighbor of neighbor:
                edge_queue.enqueue(neighbor, next_neighbor)
    return True
```

Each successive higher level of k-consistency ensures earlier stopping when a problematic branch of assignments is entered, but it is also more computationally costly to preserve, which can defeat the purpose of saving time by backtracking earlier. Hence, an algorithm to solve a CSP performs best when it has a balance of having a high enough consistency to stop relatively early on problematic branches, but not so high that the cost of maintaining said consistency takes longer than backtracking later otherwise would.

Heuristics:

A generic solver for a CSP also needs a heuristic for picking which node to assign next. This project entailed three heuristics:

- Random (R) - choose the next variable to assign randomly
- Minimum Remaining (MR) - choose the variable with the fewest remaining options to assign next
- Minimum Remaining Combined with Degree (MR_D) - prioritize both few remaining options and a high number of connections (take number of values available − number of connections, where a lower value means a higher priority)

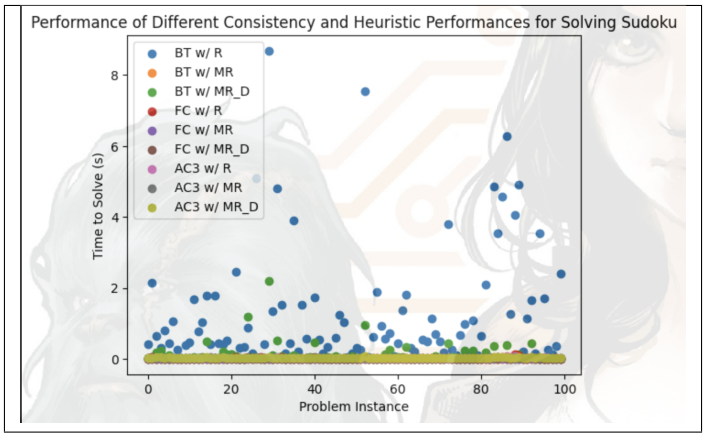
A wise heuristic can keep the algorithm from wandering into a dangerous assignment branch, as we'll see in later results.

Sudoku Implementation:

The newspaper game known as *Sudoku* is itself a CSP. Consider each box a node. A node cannot be assigned the same number as any nodes in the same row, column, or cell - those are the constraints. When a Sudoku solver sets up its underlying graph, it needs to specify the default available values for each node - the digits $1 - 9$ - and it needs to specify how to remove options from a neighbor to preserve the problem constraints. For Sudoku, that just means:

```
def preserve(node, neighbor):
    if node only has one number left in its assignments:
        remove that number from the options of the neighbor
```

Specifying the default available values and the constraint preservation function is all the Sudoku solver needs to do. In my case, I also handed the Sudoku Solver the heuristic and consistency it is to tell its underlying graph to use. Over 100 Sudoku problem instances, the results were very consistent for nearly all solving methods, but applying $k = 0$ consistency certainly took its toll in terms of time spent on useless assignment branches:



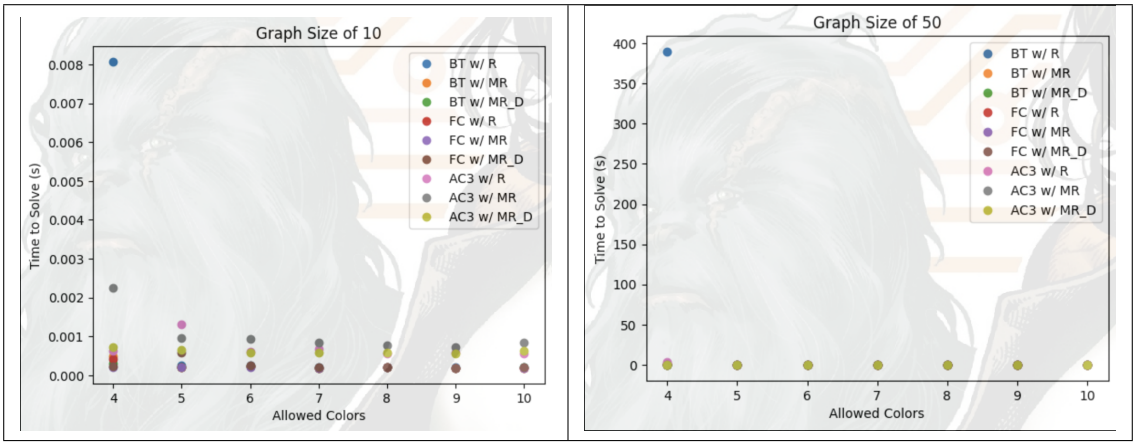
When using a random heuristic with backtracking, the random heuristic does little to help avoid dead-end assignment branches. Further, backtracking spends more time in said useless branches than forward checking or arc-consistency. As a result, when the algorithm does not get lucky in terms of the initial assignments it picks, its runtime increases dramatically, which my results clearly show.
Interestingly, backtracking while using a heuristic that combines the minimum remaining values available to a node *along* with the degree of that node also yields spikes in runtime. Compared to all other algorithms besides random selection with backtracking, such a consistency and heuristic performs poorly. This could suggest that - at least for the purposes of Sudoku - the number of options remaining for a space should be the main factor in deciding if that space should be assigned a number next.

Graph Coloring Implementation:

The famous graph coloring problem (GCP) - in which nodes can be assigned colors, but no two connected nodes may share the same color - is also by nature a CSP. A GCP solver also sets up its underlying graph by specifying the default available colors for each node (denoted as integers). Its rule for removing options from a neighbor to preserve the problem constraints is similar to Sudoku:

```
def preserve(node, neighbor):
    if node only has one color left in its assignments:
        remove that color from the options of the neighbor
```

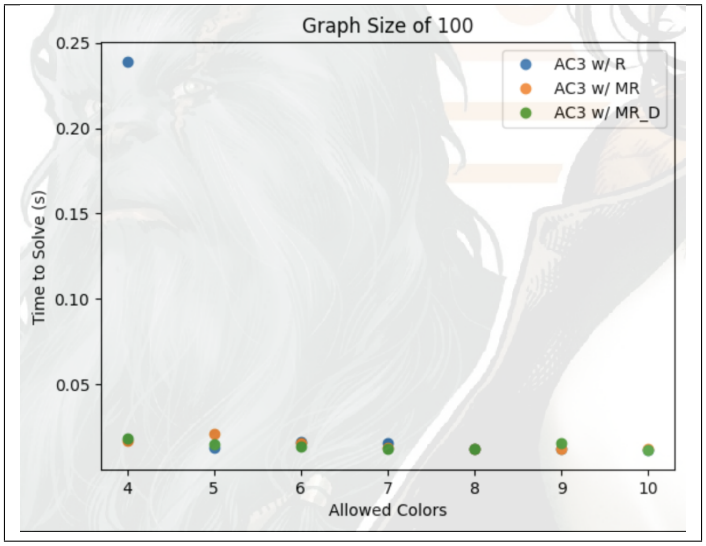
Again, a color can simply be denoted as a number. A famous theorem states that four colors guarantees that total variable assignment is possible *regardless* of the graph, but the solver I implemented allows the user to specify the number of allowed colors, which - along with graph size - yielded varying performance of the solving algorithm:



I was able to test all combinations of heuristics and consistencies with graphs up to size 50. For a graph of size 10, color assignment is almost trivial for a computer. Further, as we allow more colors for such a small problem size, the likelihood of entering a dead-end assignment branch decreases, and all algorithms perform essentially the same in terms of performance. When only allowing 4 colors, note that backtracking with a random heuristic takes significantly more time because fewer allowed colors increases the likelihood of running into a problematic assignment branch, which backtracking spends the most time in. However, for a graph size of 10, "significantly more time" still means below a hundredth of a second.

For a graph of size 50, the extremities with the results from the graph of size 10 are exacerbated. Once again, all algorithms besides backtracking with a random heuristic performed essentially identically. However, when only allowing 4 colors, the time spent backtracking and choosing variables randomly jumped all the way to 400 seconds because of the increased proportion of dead-end assignment branches. Further, the increased number of nodes in the graph means backtracking spends much more time before reaching a dead end in each branch as compared to a graph of size 10. When using a random heuristic - which fails to decrease the likelihood of entering bad assignment branches - runtime increases dramatically.

Interestingly, heuristics have thus far been the only major determining factor for the performance of a GCP solving algorithm. However, once graph size breaches 50 nodes, the consistency of an algorithm makes or breaks it:



For such large graphs, the 0− and 1−consistency solvers - backtrack and forward checking - were simply incapable of solving the graphs in a reasonable amount of time. Presumably, this is because those algorithms traversed too deeply along too many dead-end assignment branches. Therefore, I could only vary the heuristic in my analysis.

When analyzing performance of the 2−consistency solver on the 100−node graph, notice that the random heuristic - not surprisingly - had the least amount of luck in its search for productive assignment branches. However, the 2−consistency of the algorithm played its roll in ensuring that we did not spend too much time exploring any dead-end assignment branch. In this way, the algorithm still terminated in less than one second. Finally, there was no significant difference between the minimum-variables-remaining and minimum-variables-with-degree heuristics when both were used with 2−consistency.