Constraint Satisfaction Analysis

GitHub: https://github.com/EWilliams2002/CSP-Solvers-and-Analysis

Evan L. Williams

School of Information, University of Arizona

ISTA 450: Artificial Intelligence

Prof. Tyler Millhouse

May 7, 2024

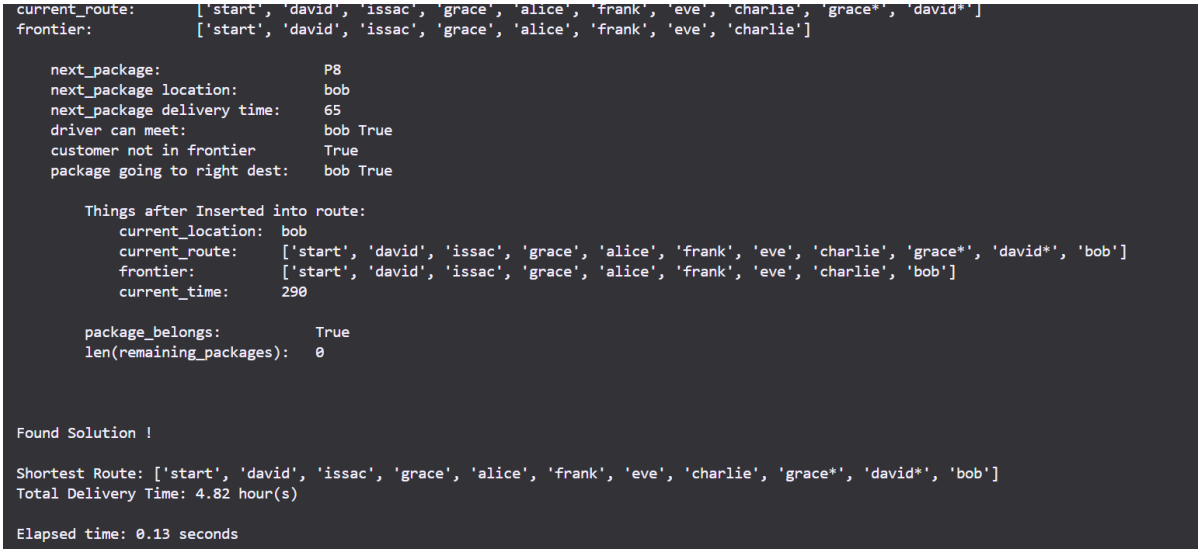
Evan Williams
7 May 2024
ISTA 450: Artificial Intelligence
Final Project Report

Overview:

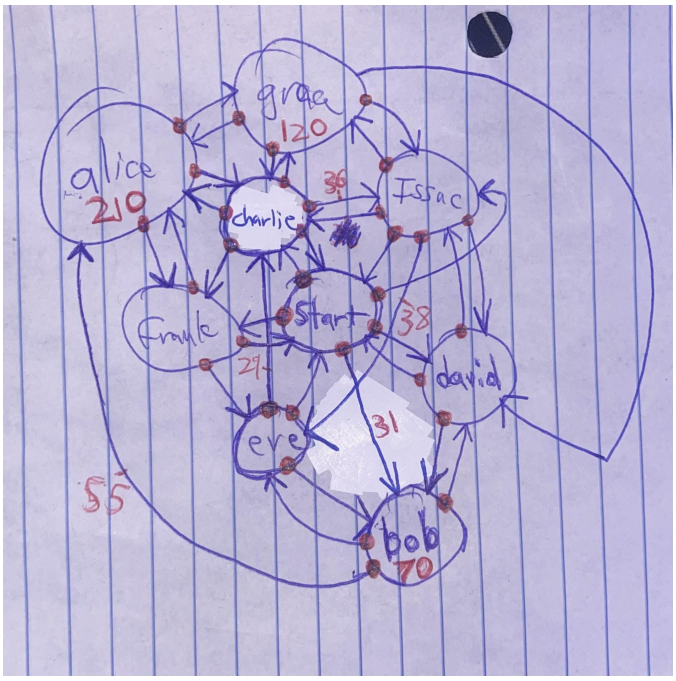
Overview: For my final project, the two algorithms that I chose to solve my Constraint Satisfaction Problem (CSP) were local search (Hill Climbing) and Backtracking. I chose these algorithms because I had some experience with them before and I felt like it was best to start with the basics. The CSP that I chose to solve with these algorithms is a Scheduled Delivery Routes system. As a bonus I also included an extra backtracking search on Sudoku.

Report:

In the delivery routes system, there is a hypothetical delivery driver who must deliver packages to eight different customers. Their constraints are that they must visit each unique node once (which represents delivering each package) and make it to each customer on time (represents their generated delivery times). On this node graph that is their map, each edge represents a distance in (minutes) away from each customer and each other, their "cost." The distances have multiple special cases that are preset which are longer routes that could take the driver across the map. For instance, Bob and Alice are at opposite ends of the map and they have



an edge that has a 55-minute cost whereas the rest of the inner edges are set randomly with a range from 5 to 50 minutes. The delivery time works the same, and there are special cases as well. These special cases work differently, I need to add special cases since some customers' times in real life are gauged by their distance away from a delivery depot because if a customer is further away from a delivery depot from which their package would get delivered, the company would set their delivery time greater than most. So to implement this, I set the furthest away locations to have slightly, medium, to higher times, and then the ranges for the random delivery times are 25 to 90 minutes. For instance, Bob's delivery time is 70 minutes (1 hour 10 minutes). To sum up, before I go into more detail about the code, the driver must try and find the shortest route that does so. To implement this, I first created a dictionary that has the node locations as keys and dictionaries as the values that have each edge with its respective distance. I also have another dictionary that has a package code as keys and dictionaries as the values with location (customer name) and the delivery time needed. First, I fill up the edges and delivery



times with a function called fill_random_vairables(line 171). It takes the node graph and packages dictionaries and sets random values using the random.randint(). Then after this, it enters the recursive process where it loops while there are remaining packages; for each neighbor from a starting point, it checks whether there is a package to be delivered to that location and if the driver can make

it on time, given the distance and delivery time. It does this until it finds the shortest solution. On line 116, I implemented a special part, which was one of the challenges I faced, where if it cannot reach the next destination where a package could be delivered, they travel to a next neighbor so that the code can evaluate whether a package can be delivered from that next destination to another destination. These steps taken in the drivers' routes are marked with an '*' like so "grace*". When using backtracking for the package delivery routes, I noticed depending on what edges are available the algorithm takes less time to visit nodes. I also noticed how increasing the max random number range of how long it could take to travel between each node filtered the solutions down to the same 1 or 2 solutions.

The local search algorithm took me the longest to implement as it has the most working parts. For the local search algorithm, the first thing that happens is setting the max_iterations for a stopping condition, then it generates the graph being used, along with the packages and an initial solution route to generate neighbors off of it. After, I generate 2000 neighbors to compare to by shuffling every name in a route array. After generating the neighbors I go into a for loop that iterates over each neighbor and validates the neighbors for connecting routes and if the driver can meet the time of each route given the random distances. Funny enough, I struggled greatly on validating the neighbors. Every neighbor I passed through to check their fitness ended

```
Original Time:   3.7 hour(s)
Original Route:  ['start', 'bob', 'david', 'issac', 'grace', 'alice', 'frank', 'eve', 'charlie']


Achieved Iterations:       100
Neighbors Per Iteration:   2000


Shortest Solution:

Route: ['start', 'issac', 'david', 'bob', 'eve', 'charlie', 'frank', 'alice', 'grace']
Total Delivery Time: 2.67 hour(s)

Solution Imporved !

Elapsed time: 0.01 seconds
```

up being false and not possible. During that, I figured out that about 90% of my current calculate_total_distance() function could possibly be a solution to my problems and it ended up working out.

The fix was instead of using a crazy drawn-out function that I was writing. In the newer version of it, from the distance function, I incorporated the way I got the distance between edges. I calculated the distance by setting distance in a try/except using a current location and next location to get the distance from the graph dictionary. If it failed, it was going to be a KeyError which meant that the next location wasn't a neighbor of the current location a.k.a. you couldn't reach whatever next location from where you were. With this, when I iterated over each neighbor and tried to validate it, if it failed then that meant that it was invalid anyway. If it didn't, that meant that in that route, you could get from one to the next. To complete the constraints, I also calculated the total distance and total delivery time during this function since it was going to be available from having no KeyError. Now with both those variables, if the total distance, which was the total time taken, was lower than the total delivery time then it was a valid neighbor to compare its fitness. Once I validated all the neighbors and compared their fitness with my actual calculated distance function I printed my found solution. To go more in-depth about my calculated distance function, what I do is get the first location in their route and next, I pass it to the node graph and recurse with a kept track of total distance and the route array with the first element removed so that the next recursive stack would have what would be the next pair of locations.

For the bonus problem Sudoku, I chose to implement the board as a 2D array where each blank element on the Sudoku board is marked with zeros, and the rest are marked with the pre-existing numbers like how a regular board will have. I got my sample board from the

extreme difficulty of the sudoku challenge on this Sudoku website for an initial board. First, in the backtrack version, I run a function called find_empty_cell(line 34) which takes the board as a parameter input. It parses through the board with embedded for loops and returns the first row and column index where the blank spot (a zero) was found, and if none were found it returns a none value. The return value of the previous function will serve as a boolean variable to check whether the board is solved or not as well as its actual purpose of finding the next empty cell to input into when placing down answers. With the return variables row and col, I check if adding a number to the board is valid by checking each row, column, and 3x3 square for all unique values 1-9. After getting whether the board was valid or not, I set the number that was checked and then recurse back using an if statement to keep filling out the board. If that statement returns true, then I know the board is finished and solved, or if false then it removes the last word that was set and either keeps going and eventually finishes, then I print my results. To go more in-depth on the use of valid_board(line 43), I call it using the board, the current number to insert, and the current position. In valid_board() I check each row by checking to see if the number I am about to set is already in the board. In that check, I return false if it was already added, I do this for both rows and columns in the puzzle. How I check the 3x3 squares is I divide the x and y coordinates by 3 so I can get the current box that the coordinates are in and then I use that information to read only the values in the box to see if the number I'm about to place is already on the board. After all is done I print my results and the elapsed time the algorithm took to run. Using backtracking on the sudoku board showed me that backtracking can work very efficiently.

```
3 1 0   0 0 0   2 0 0
7 0 0   8 0 0   4 0 0

0 0 0   0 0 0   0 0 6
5 0 0   0 0 0   0 0 0
0 0 0   2 8 4   0 0 3

0 6 5   0 0 0   0 3 0
9 2 0   7 0 0   8 0 0
0 0 0   4 1 0   0 0 0

-----------------------

4 8 9   6 7 2   3 5 1
3 1 6   5 4 9   2 8 7
7 5 2   8 3 1   4 6 9

2 3 8   1 5 7   9 4 6
5 4 7   3 9 6   1 2 8
6 9 1   2 8 4   5 7 3

1 6 5   9 2 8   7 3 4
9 2 4   7 6 3   8 1 5
8 7 3   4 1 5   6 9 2

Board Solved !

Elapsed time: 1.42 seconds
```

To compare the results and efficiency, implementing the local search took a long time, and implementing the backtracking functions was a lot easier for me. That being said I think the backtracking functions took relatively longer because the one local search algorithm that had fully gotten results, ran faster than its backtracking counterpart for delivery routes, but also ran faster than the sudoku backtracking. Comparing the searches for each CSP, as already stated, the local search algorithm for the delivery routes ran a lot faster than the backtracking did, it was also better at finding routes, especially since the random distance range is increased from 5-50 minutes (backtracking) to 7-50 minutes (local search). In general, local search algorithms can be more efficient when dealing with large search spaces and optimization problems, while backtracking algorithms are suitable for constraint satisfaction problems with a smaller search space and clear constraints.