CISC 204 Modelling Project Report

Group 12: Michael Cassidy, Kieran Green, Cooper Moses, Mike Stefan

**Project Summary**

Our project aims to solve the possible different routes a vehicle could take to get to a destination given a grid of intersections which is randomly generated with red-lights, one-way roads, two-way roads, and busy pedestrian traffic. The grid of intersections will be given the number of rows and columns . This number is easily changeable and will make it easy to test wildly different test cases.

The roads will be assumed to be two-way roads. A 2D-array will then be made to fit the size of the grid. This will serve as the map the car will drive through. Each index in the array will serve as an intersection that has generated rules based on the roads that cross there. Some examples of these rules are a light blocking traffic going either North/South or East/West, as well if the intersection is at the edge of the grid the car will also not be able to go off the map.

**Propositions:**

* Mx, y: where x and y correspond to the grid location that car is at.
* Gx, y: position of the goal the car must get to.
* L: the car turns left at the intersection
* R: the car turns right at the intersection
* S: the car goes straight at the intersection
* E: the car has reached a dead end
* W: the car has reached the goal
* C: the light is red in front of the car
* Di: Where i is the direction the car is facing (N, E, S, W)

**Constraints:**

* (G2, 2 M2, 2) W
* If the car is at the goal, then the car has reached the goal.
* (~C  **~** ((DN Mk,2) (DS Mk,0)( DE M2, k)( DW M0, k))) S
* The car goes straight if the light is not red, and the direction of the car is not pointed off the map. K represents any value that correlates to the map. The example takes place in 3x3 grid with the bottom left corner being (0, 0) and the top left corner being (2,2), so k is a integer between 0 and 2. If it is at the bottom of the map then it’s position is Mk, 0 and if it is direction is DS (South). That means if it goes straight, it will go off the map so it is not allowed to go straight.
* (~S ~L ~R) E
* If the car cannot move then it has reached a dead end.

**Jape Proof Ideas:**

NOTE: jape proofs are just if the turns are possible, they do not take into account the path that needs to be taken or that one turn must be selected, they just use the Boolean of if the option is available (x and y for the position and D for direction are dropped).

To add complexity to our proofs we added ideas not in our program but are still related to our problem (busyness, one-ways, stop signs, and other layouts for each street). This is to explore our model further through more complex ideas and look at different ideas we would implement if we had more time (if it's busy with people then you can't turn right and busy with cars you cannot go left even if it's green).

Because we can’t use full words in jape, we define our variables with (some are not used in the examples):

* T = left
* R= right
* S = Straight
* G= green
* B = busy (cars)
* P = busy (people)
* A = left turn advance
* D = lights
* E = stop sign
* H = one-way (no crossroads)
* Q = four-way
* F = four-way (with one way across)
* C = T-shaped road
* The idea is that: on a four-way road given that a left turn is possible, a right turn is possible and a straight-through is never possible, the light must be an advance on a non-busy street.

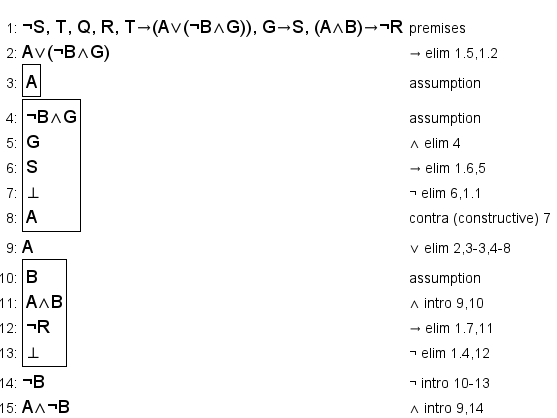
This means we get the following using only the necessary constraints:

¬S, T, Q, R, T→(A∨(¬B∧G)), G→S,(A∧B)→¬R ⊢ A∧¬B

Left implies an advance or it's not busy and green.

Green implies you can go straight.

advance and busy implies you cannot turn right as there will be cars turning into that lane.



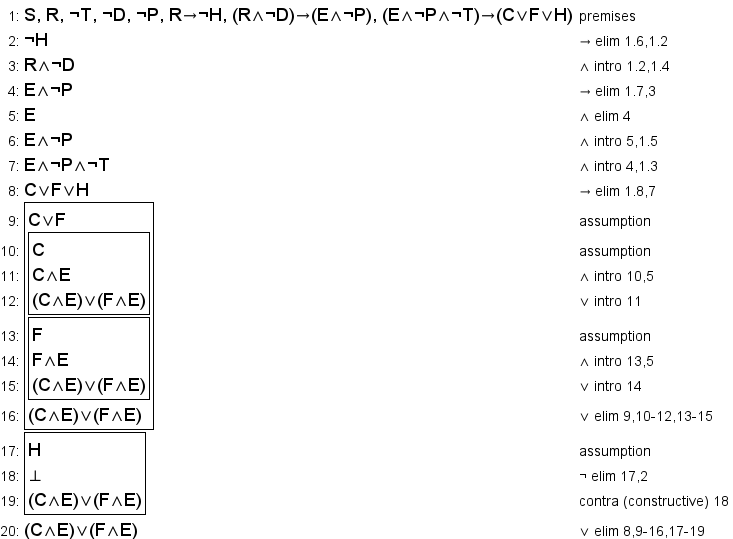
* The idea is that: if you can turn both straight and right but not left, there is no light, and it is not people busy, it must be a stop sign on a T-shaped road or a four-way with a one-way street.

The equation for this problem is: S, R, ¬T ,¬D , ¬P, R→¬H, (R∧¬D)→(E∧¬P), (E∧¬P∧¬T)→(C∨F∨H) ⊢ (C∧E)∨(F∧E) with the constraints as ordered:

right turn implies it is not a one-way street with no crossroads.

if you can go right and there are no streetlights, it must be a stop sign and not people busy.

if there’s a stop sign and it is not people busy, but you cannot turn left, it must be the street layout having no lane for left making it either a one-way (with crossroads) or a T-shaped road configuration.



* The idea is that: at a four-way If you can go straight and right, the road must be car-busy, but not people busy. The light must also be green to go straight.

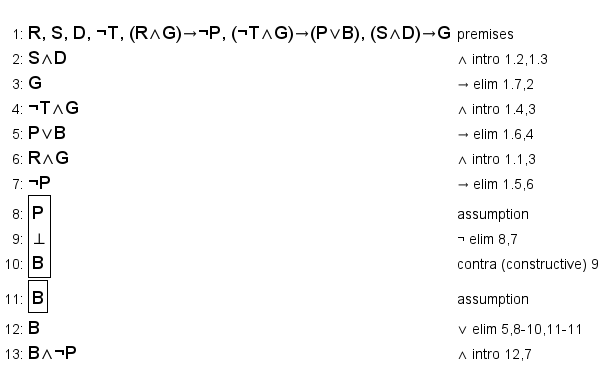
The Equation for this is: R, S, D, ¬T, (R∧G)→¬P, (¬T∧G)→(P∨B), (S∧D)→G ⊢ B∧¬P

The constraints are as follows:

If you can go right and it is green, it cannot be people busy

If you cannot go left and its green, it must be people or car busy

If you can go straight and there's lights, the light must be green



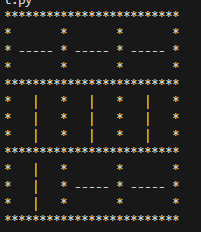
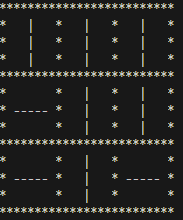
**Model Exploration**

We take the cars current position and evaluate all three possible answers, checking can the car turn left, turn right or go straight. We then make that step, simulating the car has made that move, then complete the same checks for the new intersection. Due to the nature of this problem, we decided that solving this recursively was the best option. Our program will keep checking every possible route recursively until a route finds the target or an invalid route. A route is not valid if it ends in some position that all possible movements (left, right or straight) are all invalid, or it reaches an intersection it has already been in from the same direction you entered it the first time. For example, you entered intersection [1,1] from the south as the second move, make 4 right turns and enter [1,1] from the south as the 6th move. This would create a loop, which would cause us to have infinite possible paths. The moment an end is found, either the target point, or an invalid route the recursion branch is ended. Any path the end in an invalid answer is ignored, and all valid paths are combined into a giant constraint by performing an exclusive or on all the paths (path1 xor path2 xor path3). For all relevant places in this document xor will be used rather than the equation to maintain readability. Really, each path is a combination of our left, right and straight propositions. For example, a winning path that took 2 lefts and no other turn would look like this: (left0 & ~right0 & ~straight0 & left1 & ~right1 & ~straight1). To translate that, your first move was to turn left, not turn right and not go straight and your second move was the same, but with a second set of propositions. With this system programmed in a basic example we discovered a logical issue; this system will over count the number of possible routes if there are routes that are different lengths. For this testing we defined a simplified example, our car will reach the goal the moment it makes a left turn and lose if it makes 1 straight movement or 2 right turns. This would create the constraint of ((left0 & ~right0 & ~straight0) xor (~left0 & ~right0 & straight0 & left1 & ~right1 & ~straight1)). To translate this into English, you turned left on your first movement, winning that path or you went straight then turned left. The issue with this system stems from the different number of propositions. The left side of the xor does not constraint ANY of the possible movement propositions from a second movement. This creates several truth tables which satisfy this constraint but would not be valid car actions. For example, both left0 and right1 could be true and that equation would be satisfied, but turning right on the second motion would not actually satisfy the system.

This issue was not evident to use until we after we ran the program. To help figure out why this was an issue we used the Bauhaus pprint() function which prints a nicely formatted string representing our constraint in NNF form. Unfortunately, this function has a slight bug in the docker terminal, instead of printing the logical operator characters “∧” and “∨” it errored and prints the ascii missing character “�”. We fixed this issue by locating the bauhaus/core.py as installed by pip file in the docker, force giving us read/write/execute permissions on this file with chmod 777 and editing the file to replace the broken characters with “&&” and “||” respectively. This fixed the bug in the function and allowed us to print a full NNF version of our constraint which we could use to generate a truth table and see what caused our logical issue mentioned above.

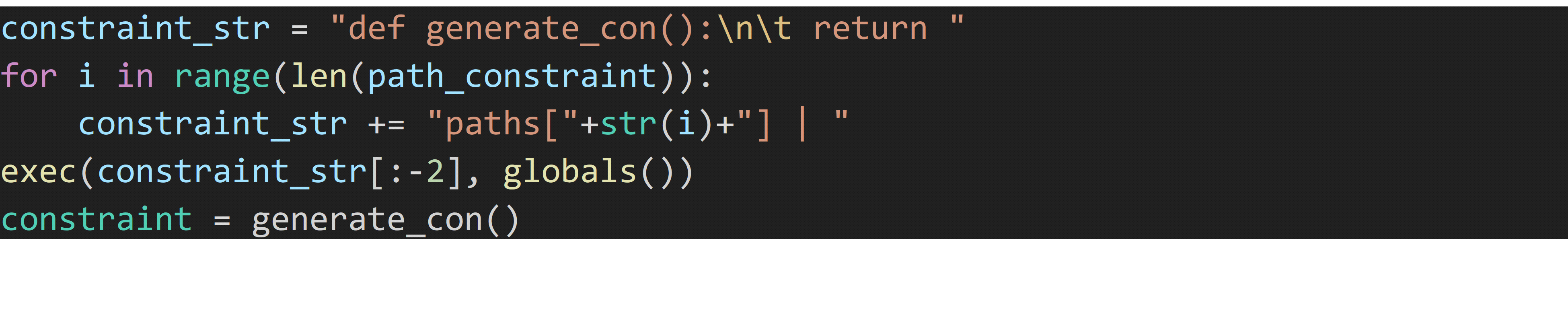
We spent a significant portion of time trying to debug this issue, but we eventually concluded that there was no good way to solve it for this test case. This is because depending on what we set the win and loss conditions to, there are infinite numbers of possible propositions. Instead, we pivoted to using our actual use case as the testing grounds for our code. We set up the map but ignored factors like traffic lights and one-way streets to keep things simple in the beginning. Since we have a map of a pre-specified size, we know exactly how many propositions there can possibly be for a given run of our code. There will be three propositions per direction we can enter each intersection, since we can move left right or straight. There are four directions per intersection and the intersection are on an M by N grid. This means there are (M\*N\*4\*3) propositions for a given map. In our new approach, we predefined all the possible propositions, then used our same recursive algorithm for the car to move around. As the car moves around, we store a list of all the positions it’s been to and the proposition which would have to be satisfied for that movement to work. We then conjunct all the propositions from a given path. We then conjunct that with the negation of all the other propositions in the grid. Finally, we disjunct our current constraint with every other possible path.

To help us understand and decode problems we created a quick visualizer to show how the map is made with the red lights. the boxes with "---" meant that there was no red light going west or east. "|" meant that there was red light going North or South. This allowed use to quickly visualize the map that was made to spot errors.



This leaves us with a constraint which is only satisfied when exactly one correct path is taken. Just like before a path constraint will look like this: (left0 & ~right0 & ~straight0 & left1 & ~right1 & ~straight1) just with a few extra cases. Our propositions are now in the form DFXY, where D is the direction the car will be turning (L, R, S), F is the direction the car is facing (N, S, E, W) and XY is the x and y coordinates of the current intersection. This means that the first intersection [0,0] will have all of the following proposition: LN00, LE00, LS00, LW00, SN00, SE00, SS00, SW00, RN00, RE00, RS00, RW00. In the case where we are facing North and want to go left our constraint will look like this: LN00 & ~LE00 & ~LS00 & ~LW00 & ~SN00 & ~SE00 & ~SS00 & ~SW00 & ~RN00 & ~RE00 & ~RS00 & ~RW00. This same pattern is repeated for all possible turns, where we create a giant conjunction of all possible turns, where all the ones we don’t make are negated. This conjunction, here on notated by paths[i] is created for all possible valid paths and combined. It is combined by generating a disjunction of all paths like this paths[0] | paths[1] | paths[2] |…| paths[x] for all possible paths.

Unfortunately this disjunction was non-trivial to create. Since we do not know the number of paths we must generate this iteratively. The initial plan to generate this looked something like this: This plan was not possible at it would generate the constraint as (((paths[0] | paths[1]) | paths[2])| …) which is not what we wanted to happen. With it being generated procedurally like this, it would insert () around every new term, which when simplified out would change the final constraint. To fix this we must iteratively generate the paths[0] | paths[1] | paths[2] |…| pattern but not configure it as a proper constraint until after it has been generated. Figuring out how to generate this was non-trivial. Eventually this was solved with the python exec function. This function allows you to pass it a string of valid python code and it would execute it as if it was code. This allows us to define some function generate\_con() which simply returns that desired constraint which we can then use to set the value of constraint. In this code block path\_constraint is equal to our paths array from earlier.



This will generate the string:



That string is then turned into code with exec() and we call generate\_con() to access our constraint. The next thing we need to account for is redlights/impassable roads. The first thing we need to is associate each intersection proposition with a light. Such that moving straight north through intersection [0,0] will also have to consider the state of the light at [0,0] in the N/S direction. Then, returning to where we generate each path in paths. Then each constraint is also paired with a light state. Returning to our going straight at [0,0] example, the constraint for that will now look like (SN00 & LightN00). This means that both we must go straight and that light allows it. That same conjunction is added to all propositions which would have us moving straight or turning left. We do not take it into account when turning right, as right on a red light is still a valid move. We then create a master conjunction of all lights. This conjunction will take the form (Green[X1,Y1] & Green[X2,Y2] &…& Green[X,Y] & ~Red[X1, Y1] & ~Red[X2, Y2]). This conjunction will include all green lights and negate all redlights. We finally take our path constraint from the generate\_con function and our light constraints and find their conjunct.

That is the final step in our algorithm, it is passed into the library to compile and calculate solutions. In the end, we decided that we would not be implementing one-way roads into our project. This is mainly because our constraints would become too big and could cause the library to crash. In the end, it will output the correct number of possible paths based on the randomly generated map it is given. The only thing to note is that at bigger maps over 3x3, there is a high chance of there being zero solutions based on map generated.

**First-Order Extension**

If we were to extend our ideas to first-order logic we would have a new set of constraints and functions. The best way for us to use these functions would be by using the positions directly to affect them. The functions we would have over the set of positions we have (each variable is a position) would be:

* M(y): The position of the car, returns T if the car is in the position x
* G(x): if the goal is at position x then T
* L(i,j): returns T if the car can turn left entering j from i
* R(i,j): returns T if the car can turn right entering j from i
* S(i,j): returns T if the car can go straight entering j from i
* E(i,j) returns T if the car cannot move anywhere in j from i (dead end)
* A(i,j) returns T if i and j are adjacent
* W(y,x): returns T if y and x are the same, y being the car’s position and x being the goal position.
* C(i,j): returns T if j has a green light entering from i

We would also receive a new set of constraints given this set of functions:

If there exists a car and goal are in the same position it implies a win in that position:

* ∃x(M(x) ∧ G(x)) W(x,x))

Also, if the car is next to the goal and you can move to the goal cell from the car cell it also implies a win at position y:

* ∃x∃y( ( M(x) ∧ G(y) ∧ A(x,y) ∧ (L(x,y) ∨ R(x,y) ∨ S(x,y))) W(y,y))

If there is no left, right, or straight option when entering j from i, and they are adjacent, there is a dead end in j from i:

* ∀i∀j( (A(i,j) ∧ ~L(i,j) ∧ ~R(i,j) ∧ ~S(i,j)) E(i,j) )

If the light is not red and there is no dead end the car can go straight given it is adjacent to that cell:

* ∀i∀j( ( A(i,j) ∧ C(i,j) ∧ ~E(i,j) ) S(i,j) )

If a car is not adjacent to a cell it cannot move into that cell at the time:

* ∀i∀j( ~A(i,j) (~L(i,j) ∧ ~R(i,j) ∧ ~S(i,j)) )

The car and goal must exist in a cell:

* ∃y∃x(M(y) ∧ G(x))

Cells must be adjacent to at least one other cell to exist:

* ∀i∃j( A(i,j) ) ∧ ∀j∃i( A(i,j) )

Expanding to first order could make our problem a lot easier to solve as we could eliminate things like the direction of where the car is facing and use the adjacency function with turn functions to easily check if the pathing is available. Issues may arise in setting the state of each cell with its layout for a pathfinding algorithm, but it wouldn’t be too hard of a transfer given our model. The domain would be a set of all of our positions for whatever sized area we would want to get the solutions for, making it very easy to access its cells and determine adjacency as they would be the positions directly next to them or the ones a row size away. In conclusion, a First-Order extension wouldn’t be that hard to implement and could easily make our problem simpler in the end.