AI Coursework 1 Edinburgh Group 47

F29AI Coursework Report Edinburgh Group 47

Participants: Keir Ballance (H00431095) & Lewis Turner (H00430662)

The source code & solution files generated by pyperplan can be found in the zip file.
Demo video link: link to demo video on SharePoint

Part 1A:

Sudoku can be defined as a constraint satisfaction problem relatively simply. The puzzle consists of a 9x9 grid with 9 3x3 sub grids, and each cell in the grid holding a number value. A cell must be a number 1-9, and the value of a cell cannot be equal to the value of any other cell that is: on the same row of the cell, on the same column of the cell or in the same 3x3 sub grid of cell. The problem is solved when every cell is assigned a number, and the rules are followed.

Variables:

The variables can be defined as the location of each cell; in our representation we will use the x(row) and y (column) in the form of a tuple to represent this. (Note: (0,0) would be the top left cell)

Set of variables would be:

V = {(0,0), (0,1), (0,2), … ,(8,8)}

Domains:

The domain of the puzzle would be a set of values {1-9} for each cell location in the puzzle. If a cell was to have an initial value, for example 3, the domain of that cell would be {3}. So, for our representation, the domains would be the set of possible values for each variable. In python this would be represented as a dictionary with our variables as the key and the domain set as the value.

An example domains dictionary could be:

D={ (0,0): {1,2,3,4,5,6,7,8,9}, (0,1):{1,2,3,4,5,6,7,8,9}, (0,2): {2}, … , (8,8): {1,2,3,4,5,6,7,8,9} }

Where the value of cell (0,2) is 2 in the starting state of the puzzle

Constraints:

The constraints of the puzzle are simple. The value of each cell cannot be equal to any other cell on the same row, same column or same sub grid, of course this doesn't include itself. Just like the domain the constraints will be stored in a dictionary where the variable is the key, and the value is the set of constraints for that variable.

An example of the dictionary would be :

Constraints = { (0,0): {constraints for (0,0)}, … ,(8,8): {constraints for (0,0)} }

And if we take the variable (2,3) for example the set of constraints would be:

{(2,0),(2,1),(2,2),(2,4),(2,5),(2,6),(2,7),(2,8),(0,3),(1,3),(3,3),(4,3),(5,3),(6,3),(7,3),(8,3),(0,4),(1,4),(0,5),(1,5)}

Brute-force:

The Time complexity of a brute-force approach would be O(9^N) where N Is the number of empty cells. We get this number as for every cell we check 9 different combinations of numbers, and we need to do this for every empty cell. The downside of a standard brute-force approach is that we check every path even if it has already become invalid.

Constraint satisfaction (backtracking):

A backtracking approach has a worst-case time complexity of O(9^N) where N is the number of empty cells. In practice the program is much faster than a brute-force approach. This I because when a path becomes invalid it is pruned. Only valid paths are fully explored so computation isn't wasted on paths that have already become invalid. This is because whenever a new value is selected for a cell, it is checked with the value of other constraining cells to check if it is allowed in the puzzle.

Part 1B:

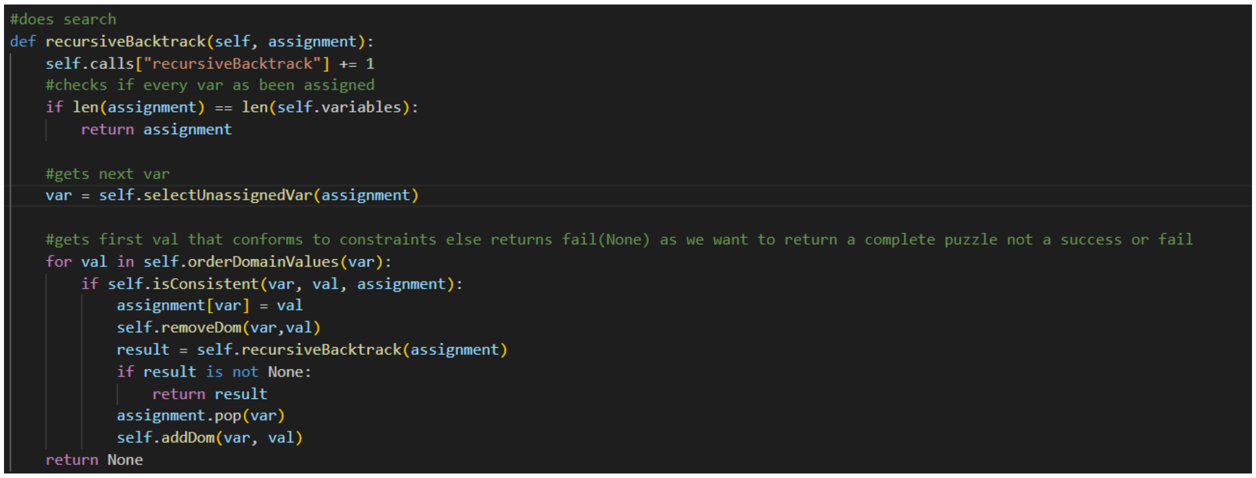
Our implementation consists of a Backtracking approach with forward checking. This is done by posing the puzzle as a constraint satisfaction puzzle. Using the starting state of the puzzle we create a set of Variables and a dictionary of domains and constraints. The problem itself is encapsulated as a class and so we do not need to pass what we just made into every function that needs it.

Using these we can recursively solve the problem. This is done step by step I the following way:

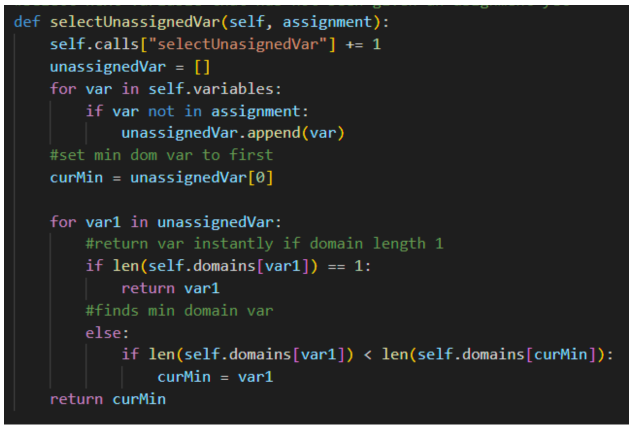
- First we create an empty dictionary which we will call the assignment. This represents our current solution as a key-value pair, where the key is a tuple (row, column) representing the cell.
- Next we call the main recursive function that will carry out the computation
- First we check if the size of the assignment dictionary is the same size as the variables, this will show if we have found a solution. As if their sizes are equal we must have a solution
- If this is not the case we then call selectUnassignedVar(assignment). This will return to us variable that has the smallest domain and is also not present in the assignment.
-
- Next for every domain value of that variable we will:
    - Check If it satisfies its constraints by calling isConsistent(var, val, assignment). This function returns true if no already known constraining variable in the assignment has the same value as the variable we are checking. Else it will return false

- If this is true we will add the variable-value pair to the assignment dictionary and then remove the value from the domain of every constraining variable by calling removeDom(var, val). This is how we implement forward checking, which significantly reduces the number of required backtracks.
- We then recursively call our backtrack function using our updated assignment.
- We then check if our recursive call returns None (None meaning an invalid path), if it does not then we can return the result of the recursive call.
- If it does return None then we have found an invalid path and need to backtrack. We remove the variable-value pair from the assignments and add back the value that we removed from the domains of constraining variables.
- Then the next value in the domain of our variable is checked

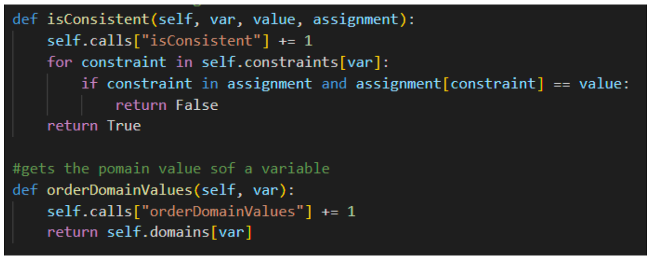
- If we have checked the path of every value in our domain then the current path is invalid, and we need to backtrack. In this case we return None.



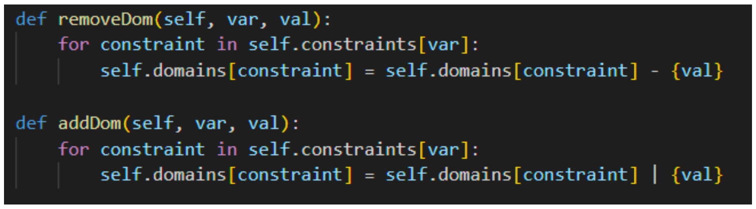
```python
#does search
def recursiveBacktrack(self, assignment):
    self.calls["recursiveBacktrack"] += 1
    #checks if every var as been assigned
    if len(assignment) == len(self.variables):
        return assignment

    #gets next var
    var = self.selectUnassignedVar(assignment)

    #gets first val that conforms to constraints else returns fail(None) as we want to return a complete puzzle not a success or fail
    for val in self.orderDomainValues(var):
        if self.isConsistent(var, val, assignment):
            assignment[var] = val
            self.removeDom(var,val)
            result = self.recursiveBacktrack(assignment)
            if result is not None:
                return result
            assignment.pop(var)
            self.addDom(var, val)
    return None
```



```python
def selectUnassignedVar(self, assignment):
    self.calls["selectUnasignedVar"] += 1
    unassignedVar = []
    for var in self.variables:
        if var not in assignment:
            unassignedVar.append(var)
    #set min dom var to first
    curMin = unassignedVar[0]

    for var1 in unassignedVar:
        #return var instantly if domain length 1
        if len(self.domains[var1]) == 1:
            return var1
        #finds min domain var
        else:
            if len(self.domains[var1]) < len(self.domains[curMin]):
                curMin = var1
    return curMin
```



```python
def isConsistent(self, var, value, assignment):
    self.calls["isConsistent"] += 1
    for constraint in self.constraints[var]:
        if constraint in assignment and assignment[constraint] == value:
            return False
    return True

#gets the pomain value sof a variable
def orderDomainValues(self, var):
    self.calls["orderDomainValues"] += 1
    return self.domains[var]
```



```python
def removeDom(self, var, val):
    for constraint in self.constraints[var]:
        self.domains[constraint] = self.domains[constraint] - {val}

def addDom(self, var, val):
    for constraint in self.constraints[var]:
        self.domains[constraint] = self.domains[constraint] | {val}
```

To test our solution, we used a selection of sudoku puzzles found online as well as a blank sudoku puzzle. The sourced puzzles were of varying difficulty and helped for an understanding of backtracking process. We also implemented a debug mode that shows the number of function calls, runtime and number of backtracks was added. A GUI was implemented that helped visualize solutions, the starting puzzle can also be changed through the GUI to allow for quickly testing different starting states to check for possible edge cases or failure cases.

Comparison:

An A* search approach would resemble a traversal through a graph. We can imagine this graph being every valid state between the start and the goal. Of course, this graph would be massive if physically stored so we can just generate the next valid states from the current state. As a starter we would require a priority queue that will contain nodes to search for and a graph that contains the starting state as a node to begin with. For each node we visit, we generate the next states. Then we can assign an F= H+G value to the state and insert it into the priority queue. For the heuristics, let's just keep it simple and say the number of empty cells is the H value and for the G value we can assign the number of cells filled to get to the state.

Comparatively, the two approaches are fundamentally different. A* is a search-based approach and so will traverse through possible states. It can suffer from exploring invalid paths with could lead to performance issues. It would also require a significantly larger amount of space. This is because it must maintain a queue of all partial states. On the other hand, backtracking with forward tracking benefits from bad paths getting pruned early, and far less space required. This is because backtracking reuses the same assignment, adding to and removing from it through as it explores. Whereas A* generates a new state for each partial solution.

From a performance perspective A* and backtracking would both have the same worst case time complexity of O(9^N) in this specific case where N is the number of empty cells and the heuristics used by the A* search is the number of empty cells. On average backtracking will tend to do less computation due to the pruning of bad paths early.

Part 2: PDDL

2A – Modelling the Domain

Types:

We define 8 new types: rover and lander with parent type of vehicle, image and scan with parent type of data, location, sample. These are used to ensure the right types of objects are used for predicates and actions. A tighter definition of actions should give the solver less combinations to attempt when finding a solution.

Predicates:

As negative preconditions aren't supported by the solver we used, we opted to define separate positive and negative predicates.

- Our first set of predicates are the (flying lander) and (landed lander) predicates, these define whether a lander is currently flying or landed. One of these must be defined at the start of the problem.
- Next we have relations between the lander and the rover. The predicates that do this are: (carrying lander rover) & (commands lander rover). Carrying defines the rover that the lander is carrying before deployment, and the commands predicate persists through the whole problem and defines a relation between the lander and rover.
- (undeployed rover) and (deployed rover) act as a positive and negative of whether a rover is deployed.
- Next we define the map as (path location1 location2) and (located object location) predicates. Path defines a connection between two locations on the map. Located defines the location of any object on the map. It allows this by using the general object type for its object parameter and by using the location type for its location parameter. We place the images, scans & samples at locations using (located object location) so that when collected the predicate can be made negative which only allows them to be collected with once.
- Next the predicates that handle the storage of data and samples. Firstly (memEmpty rover) this defines if the rover has space to store a piece of data. Next (heldData vehicle data) defines the state of a rover/lander currently holding a particular piece of data. (storeEmpty vehicle) is like memEmpty but can be used for both the rover and lander to define if they have space to store a sample. (heldSample vehicle data) defines the state of a rover/lander currently holding a particular sample.
- Lastly we have (transmitted data) and (sampleStored sample) these define final states (goals) for data & samples. They don't specify which lander to use so the solver has more freedom to determine the lander.

Actions:

- Our first action is land_lander. This changes the landers state to Landed and puts the lander at a location. As the flying and landed states are opposite, we remove the flying state as it cannot be both landed and flying, also so once landed the lander stays stationary.
- We then have deploy_rover. This requires a rover to be undeployed, check rover belongs to the lander, lander to be landed and for rover to be carried by the lander. The (located lander location) predicate ensures the right location is used for the rover's deploy position. This action results in the rover being deployed at the current location of the lander.
- move_rover. Requires the rover to be deployed at location location1 and for a path to exist from location1 to location2. The result will be the rover being moved to the new position.
- scan_radar, capture_image and collect_sample are pretty similar in function. They all require the rover to have memory or physical space, for it to be at the matching location of the data or

sample, and for it to be deployed (ensures solver deploys the rover before trying to use it). The result is that the data or sample is now being held by the rover, the rover has no memory or physical space, and the data or sample is no longer in the world.
- transmit_data. This requires the rover to be deployed, have a commanding lander and be holding data. The result is that it no longer holds the data, memory state is set to empty, data is held by lander instead, data is stated as transmitted.
- deposit_sample. Requires the rover to be at the location of its commanding lander, rover to be holding a sample, lander to have space for a sample, lander to be landed and rover to be deployed. The result is that the lander is holding the sample, lander has no physical space, rover is no longer holding the sample, rover has physical space again, and sample is stated as deposited.

The actions have strict preconditions. This is to ensure none of these actions can be done illegally, like for example a piece of data being collected by an undeployed rover or a rover being deployed by a flying lander.

2B – Modelling the Problems

Mission 1

In the initial state of mission 1 we define:
The lander as: flying, carrying the rover, commanding the rover and having space for a sample
The rover as: not deployed, having memory space and space for a sample.
We define the map as described by the specification where the waypoints are location objects & directional arrows are path predicates (implied to be one directional).
The data & samples for the goals are placed at their respective locations using located.

Goal:

For the image to be transmitted, the scan to be transmitted and the sample to be deposited in a lander. Because of how we setup the predicates & actions we don’t need to specify the lander.

Mission 2

In the initial state of mission 2 we define:
lander1 as: landed, commanding rover1, having physical space and located at wp2
rover1 as: deployed, located as wp2, having physical space and memory space
lander2 as: flying, carrying rover2, commanding rover2 and having physical space
rover2 as: not deployed, having physical space and memory space
The map is defined using same method as mission 1 following specification for mission 2.
2 images, scans & samples are also assigned & placed on their respective locations.

Goal:

Same setup as mission 1 so predicates for all data to be transmitted & all samples to be deposited.

Domain-extended/Mission3:

The domain for mission 3 is an extended version of the original domain.

AI Coursework 1 Edinburgh Group 47

Extra types:

Added an astronaut type. This astronaut can be moved between rooms in the lander and used as an additional requirement for some actions

Extra predicates:

Two extra predicates are added:
- (inControlRoom lander astronaut)
- (inDockingBay lander astronaut)

These simultaneously link an astronaut to a lander and represent where in the lander the astronaut is currently in.

Action changes:

Added two actions:

Move_astronaut_to_controls and Move_astronaut_to_Docking. These actions will move the astronaut into the room they are not in. i.e if (inControls lander astronaut) -> (not(inControls lander astronaut) and (inDocking lander astronaut)

Changed three actions:

- deploy_rover – added precondition (inDockingBay lander astronaut)
- transmit_data – added precondition (inControlRoom lander astronaut)
- deposit_sample – added precondition (inDockingBay lander astronaut)

Mission 3:

Same as mission 2 but adds astronauts Alice & Bob. Alice is initially put in the docking bay of lander 1, Bob is initially put in control room of lander 2.

Goal:

Same as mission 2.