Assignment 1 Report

Kyle Kent
S5182050

Contents

Problem Formulation

Initial state: Specifies the locations of all car objects on 6x6 board

Actions: move car object x left, right, up or down

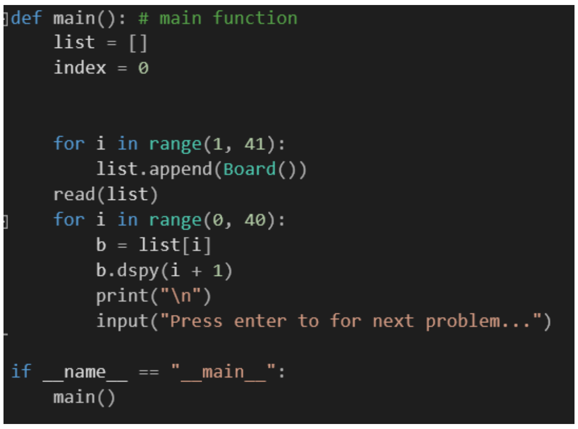
Transition model: eg. Apply left to initial state moves object x left

Goal test: Check whether car object x has reached goal state; board point (3,6)

Path cost: Step cost is 1, path cost = no of steps

Software Design

Function; Main



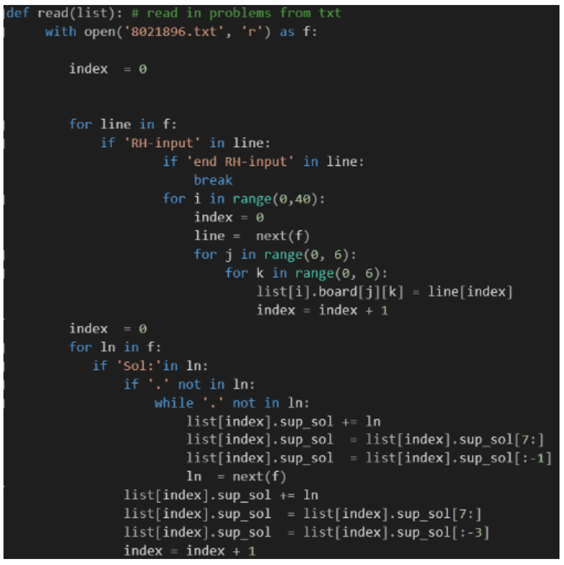
```
def main(): # main function
    list = []
    index = 0


    for i in range(1, 41):
        list.append(Board())
    read(list)
    for i in range(0, 40):
        b = list[i]
        b.dspy(i + 1)
        print("\n")
        input("Press enter to for next problem...")

if __name__ == "__main__":
    main()
```

Main is responsible for every function call and data structure initializations; all calls and operations are done within this one function.

Data; List[]

List is a list data structure within the main that contains a single board with the initial state for every problem. The length of list will always be the number of problems; 40

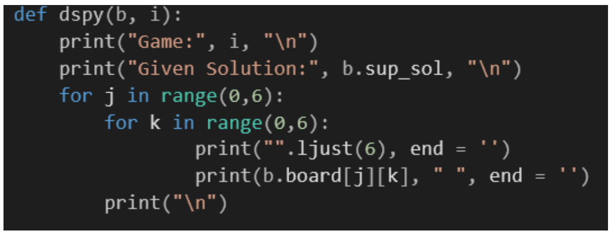
Function; Read



```python
def read(list): # read in problems from txt
     with open('8021896.txt', 'r') as f:

        index  = 0


        for line in f:
            if 'RH-input' in line:
                    if 'end RH-input' in line:
                        break
                    for i in range(0,40):
                        index = 0
                        line =  next(f)
                        for j in range(0, 6):
                            for k in range(0, 6):
                                list[i].board[j][k] = line[index]
                                index = index + 1
        index  = 0
        for ln in f:
           if 'Sol:'in ln:
               if '.' not in ln:
                   while '.' not in ln:
                       list[index].sup_sol += ln
                       list[index].sup_sol  = list[index].sup_sol[7:]
                       list[index].sup_sol  = list[index].sup_sol[:-1]
                       ln  = next(f)
               list[index].sup_sol += ln
               list[index].sup_sol  = list[index].sup_sol[7:]
               list[index].sup_sol  = list[index].sup_sol[:-3]
               index = index + 1
```

Read is responsible for reading all 40 problems from a text file into the program. Taking the problems and putting them into a data structure.

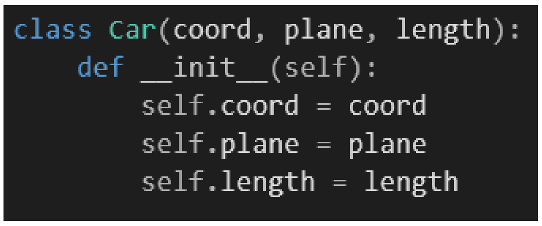
Dspy



```python
def dspy(b, i):
    print("Game:", i, "\n")
    print("Given Solution:", b.sup_sol, "\n")
    for j in range(0,6):
        for k in range(0,6):
                print("".ljust(6), end = '')
                print(b.board[j][k], " ", end = '')
        print("\n")
```

dspy allows the program to display the current problem and all relevant data to the console.

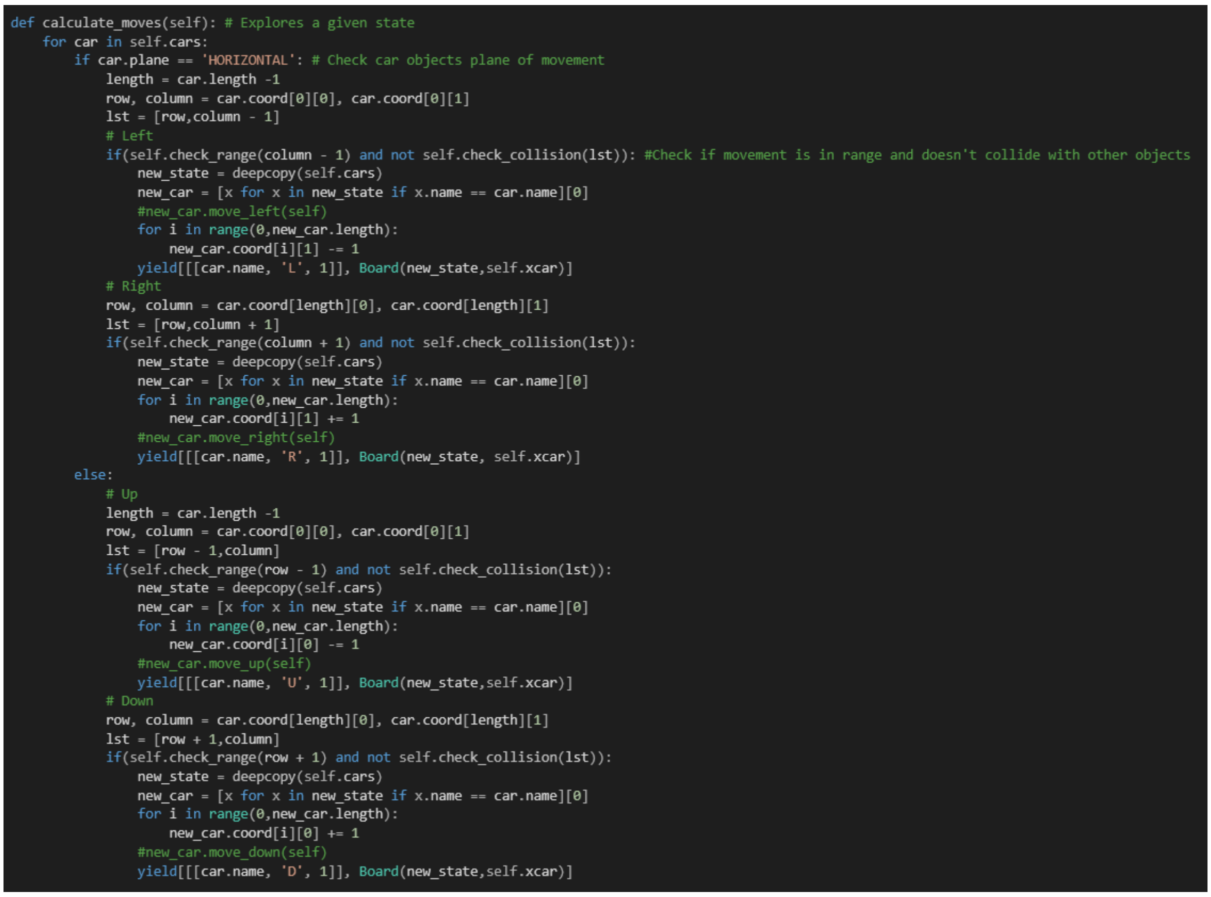
Class; Car



```python
class Car(coord, plane, length):
    def __init__(self):
        self.coord = coord
        self.plane = plane
        self.length = length
```

Car is a nested class within the class board that is responsible for mapping all car locations on the current board, allowing the program to manipulate the cars on the board.

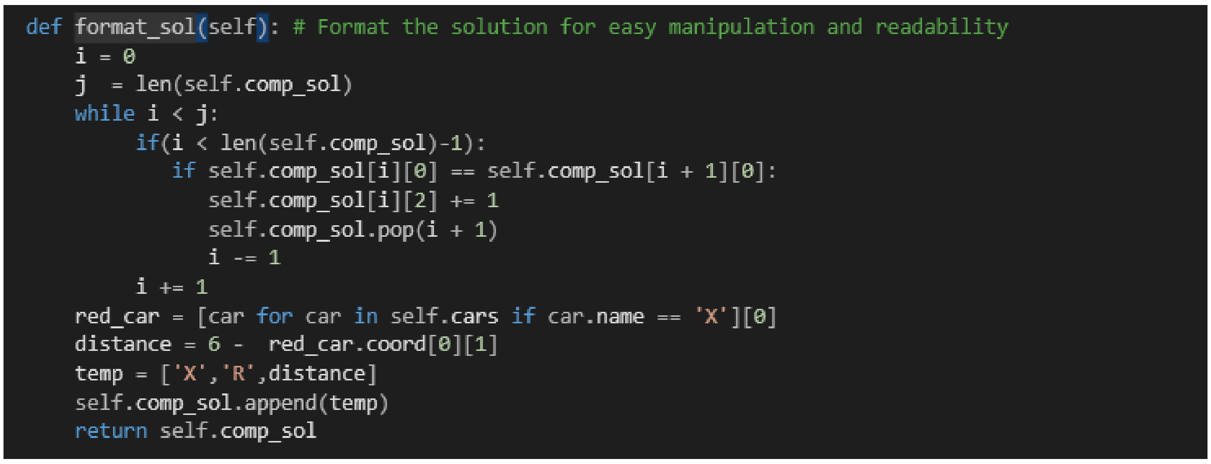
Method; calculate_moves(self)



```
def calculate_moves(self): # Explores a given state
    for car in self.cars:
        if car.plane == 'HORIZONTAL': # Check car objects plane of movement
            length = car.length -1
            row, column = car.coord[0][0], car.coord[0][1]
            lst = [row,column - 1]
            # Left
            if(self.check_range(column - 1) and not self.check_collision(lst)): #Check if movement is in range and doesn't collide with other objects
                new_state = deepcopy(self.cars)
                new_car = [x for x in new_state if x.name == car.name][0]
                #new_car.move_left(self)
                for i in range(0,new_car.length):
                    new_car.coord[i][1] -= 1
                yield[[[car.name, 'L', 1]], Board(new_state,self.xcar)]
            # Right
            row, column = car.coord[length][0], car.coord[length][1]
            lst = [row,column + 1]
            if(self.check_range(column + 1) and not self.check_collision(lst)):
                new_state = deepcopy(self.cars)
                new_car = [x for x in new_state if x.name == car.name][0]
                for i in range(0,new_car.length):
                    new_car.coord[i][1] += 1
                #new_car.move_right(self)
                yield[[[car.name, 'R', 1]], Board(new_state, self.xcar)]
        else:
            # Up
            length = car.length -1
            row, column = car.coord[0][0], car.coord[0][1]
            lst = [row - 1,column]
            if(self.check_range(row - 1) and not self.check_collision(lst)):
                new_state = deepcopy(self.cars)
                new_car = [x for x in new_state if x.name == car.name][0]
                for i in range(0,new_car.length):
                    new_car.coord[i][0] -= 1
                #new_car.move_up(self)
                yield[[[car.name, 'U', 1]], Board(new_state,self.xcar)]
            # Down
            row, column = car.coord[length][0], car.coord[length][1]
            lst = [row + 1,column]
            if(self.check_range(row + 1) and not self.check_collision(lst)):
                new_state = deepcopy(self.cars)
                new_car = [x for x in new_state if x.name == car.name][0]
                for i in range(0,new_car.length):
                    new_car.coord[i][0] += 1
                #new_car.move_down(self)
                yield[[[car.name, 'D', 1]], Board(new_state,self.xcar)]
```

Calculate moves explores every possible child state of a given state. It does this by looping over the car list of objects associated with the given board and yielding a generator when it finds a move to be valid

Function; format_sol(self)

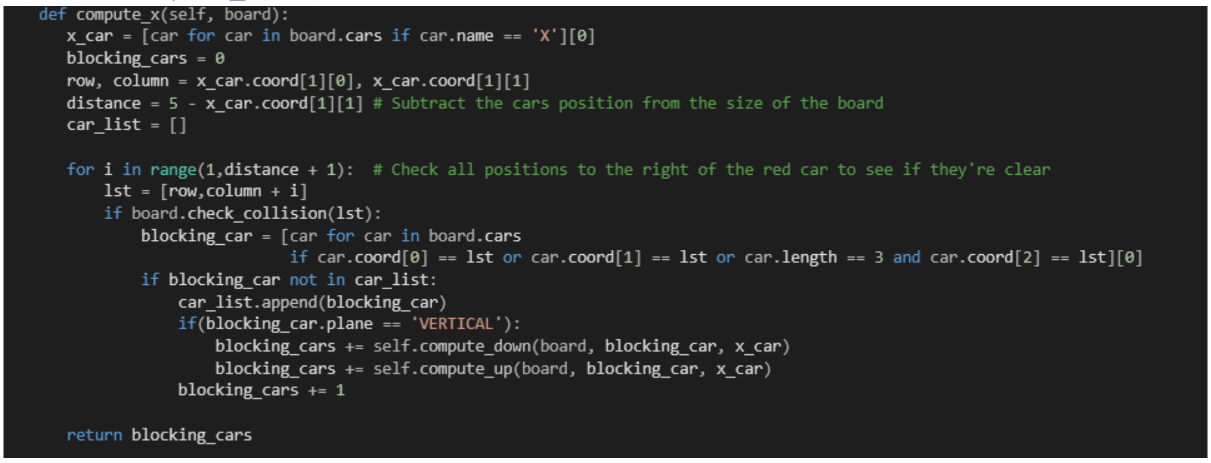


```
def format_sol(self): # Format the solution for easy manipulation and readability
    i = 0
    j  = len(self.comp_sol)
    while i < j:
        if(i < len(self.comp_sol)-1):
            if self.comp_sol[i][0] == self.comp_sol[i + 1][0]:
                self.comp_sol[i][2] += 1
                self.comp_sol.pop(i + 1)
                i -= 1
        i += 1
    red_car = [car for car in self.cars if car.name == 'X'][0]
    distance = 6 -  red_car.coord[0][1]
    temp = ['X','R',distance]
    self.comp_sol.append(temp)
    return self.comp_sol
```

Format_sol takes a provided solution to the problem and reshapes it for better readability. As a move is defined as moving 1 spot in a valid direction the solution tends to have multiple entries of Rd1, Rd1. This function formats that to simply be RD2.

Class; Adv_Blocking_Heuristic

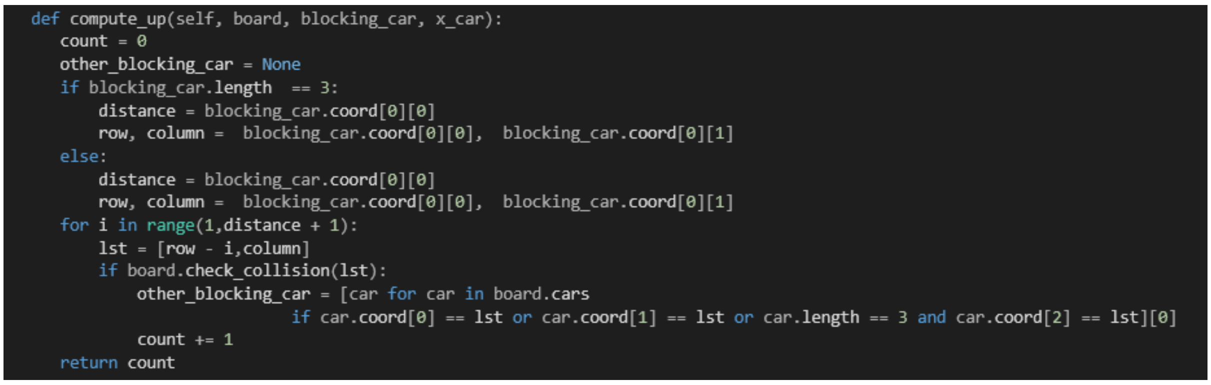
Method compute_x(self)



```python
def compute_x(self, board):
    x_car = [car for car in board.cars if car.name == 'X'][0]
    blocking_cars = 0
    row, column = x_car.coord[1][0], x_car.coord[1][1]
    distance = 5 - x_car.coord[1][1] # Subtract the cars position from the size of the board
    car_list = []

    for i in range(1,distance + 1):  # Check all positions to the right of the red car to see if they're clear
        lst = [row,column + i]
        if board.check_collision(lst):
            blocking_car = [car for car in board.cars
                            if car.coord[0] == lst or car.coord[1] == lst or car.length == 3 and car.coord[2] == lst][0]
            if blocking_car not in car_list:
                car_list.append(blocking_car)
                if(blocking_car.plane == 'VERTICAL'):
                    blocking_cars += self.compute_down(board, blocking_car, x_car)
                    blocking_cars += self.compute_up(board, blocking_car, x_car)
                blocking_cars += 1

    return blocking_cars
```

This is a more advanced form of the other class blocking_heurisitc for informed search algorithms. Not only will this class return the number of car objects blocking x from reaching the goal state but with the help of two other methods it will determine the number of cars blocking the car objects that are blocking x resulting in a more efficient and optimal heuristic function, the no. of cars blocking x represent the priority within the heuristic with the smaller no. of cars blocking x being higher than the larger no. This is passed as a variable for every informed search to base its movement off

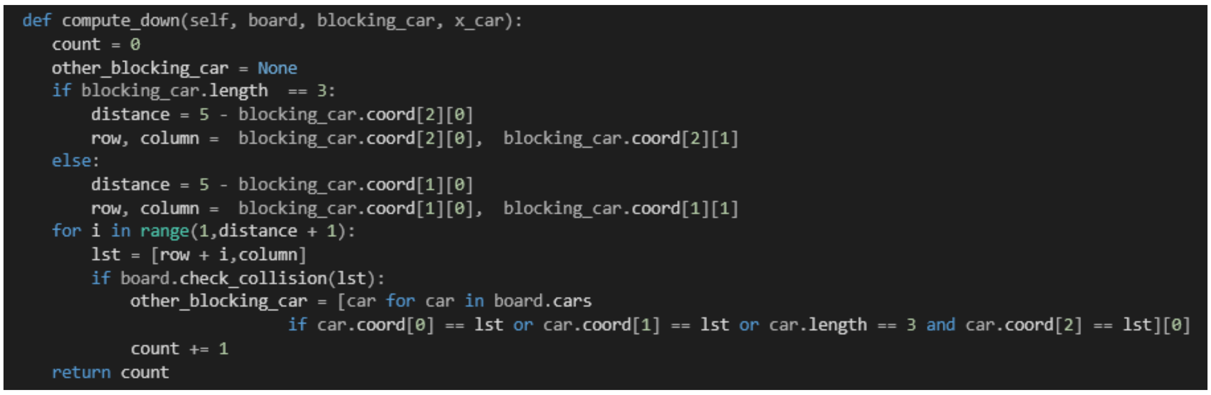
Method compute_up(self, board, blocking_car, x_car)



```python
def compute_up(self, board, blocking_car, x_car):
    count = 0
    other_blocking_car = None
    if blocking_car.length  == 3:
        distance = blocking_car.coord[0][0]
        row, column =  blocking_car.coord[0][0],  blocking_car.coord[0][1]
    else:
        distance = blocking_car.coord[0][0]
        row, column =  blocking_car.coord[0][0],  blocking_car.coord[0][1]
    for i in range(1,distance + 1):
        lst = [row - i,column]
        if board.check_collision(lst):
            other_blocking_car = [car for car in board.cars
                                  if car.coord[0] == lst or car.coord[1] == lst or car.length == 3 and car.coord[2] == lst][0]
            count += 1
    return count
```

Compute_up is called on any car objects that have a movement of vertical that are blocking the x_car from reaching the goal state it does what compute_x does for x only with a different object and upwards

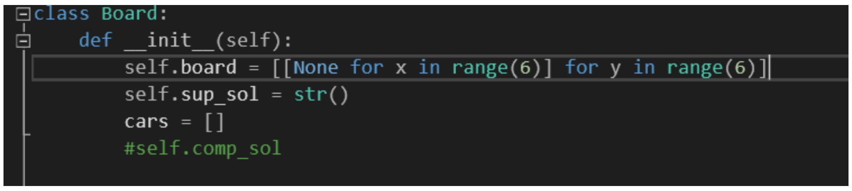
Method compute_down(self, board, blocking_car, x_car)



```python
def compute_down(self, board, blocking_car, x_car):
    count = 0
    other_blocking_car = None
    if blocking_car.length  == 3:
        distance = 5 - blocking_car.coord[2][0]
        row, column =  blocking_car.coord[2][0],  blocking_car.coord[2][1]
    else:
        distance = 5 - blocking_car.coord[1][0]
        row, column =  blocking_car.coord[1][0],  blocking_car.coord[1][1]
    for i in range(1,distance + 1):
        lst = [row + i,column]
        if board.check_collision(lst):
            other_blocking_car = [car for car in board.cars
                                if car.coord[0] == lst or car.coord[1] == lst or car.length == 3 and car.coord[2] == lst][0]
            count += 1
    return count
```

Compute_down is called on any car objects that have a movement of horizontal that are blocking the x_car from reaching the goal state it does what compute_x does for x only with a different object and downwards

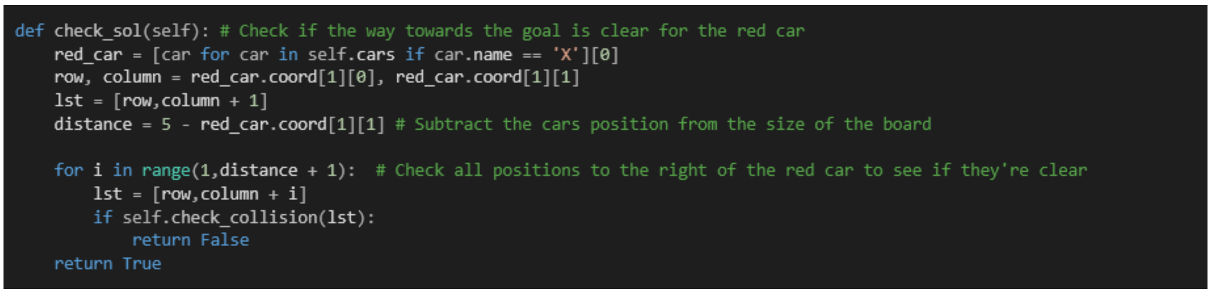
Class; Board



```python
class Board:
    def __init__(self):
        self.board = [[None for x in range(6)] for y in range(6)]
        self.sup_sol = str()
        cars = []
        #self.comp_sol
```

Board is a class that is responsible for initializing the current problems board. This allows us to manipulate the board as an object and find solutions.

Method; check(self)



```python
def check_sol(self): # Check if the way towards the goal is clear for the red car
    red_car = [car for car in self.cars if car.name == 'X'][0]
    row, column = red_car.coord[1][0], red_car.coord[1][1]
    lst = [row,column + 1]
    distance = 5 - red_car.coord[1][1] # Subtract the cars position from the size of the board

    for i in range(1,distance + 1):  # Check all positions to the right of the red car to see if they're clear
        lst = [row,column + i]
        if self.check_collision(lst):
            return False
    return True
```

Check_sol check to see if the current board object has found the goal state, it does this by checking if anything car objects exist to the right of the x_car car object.

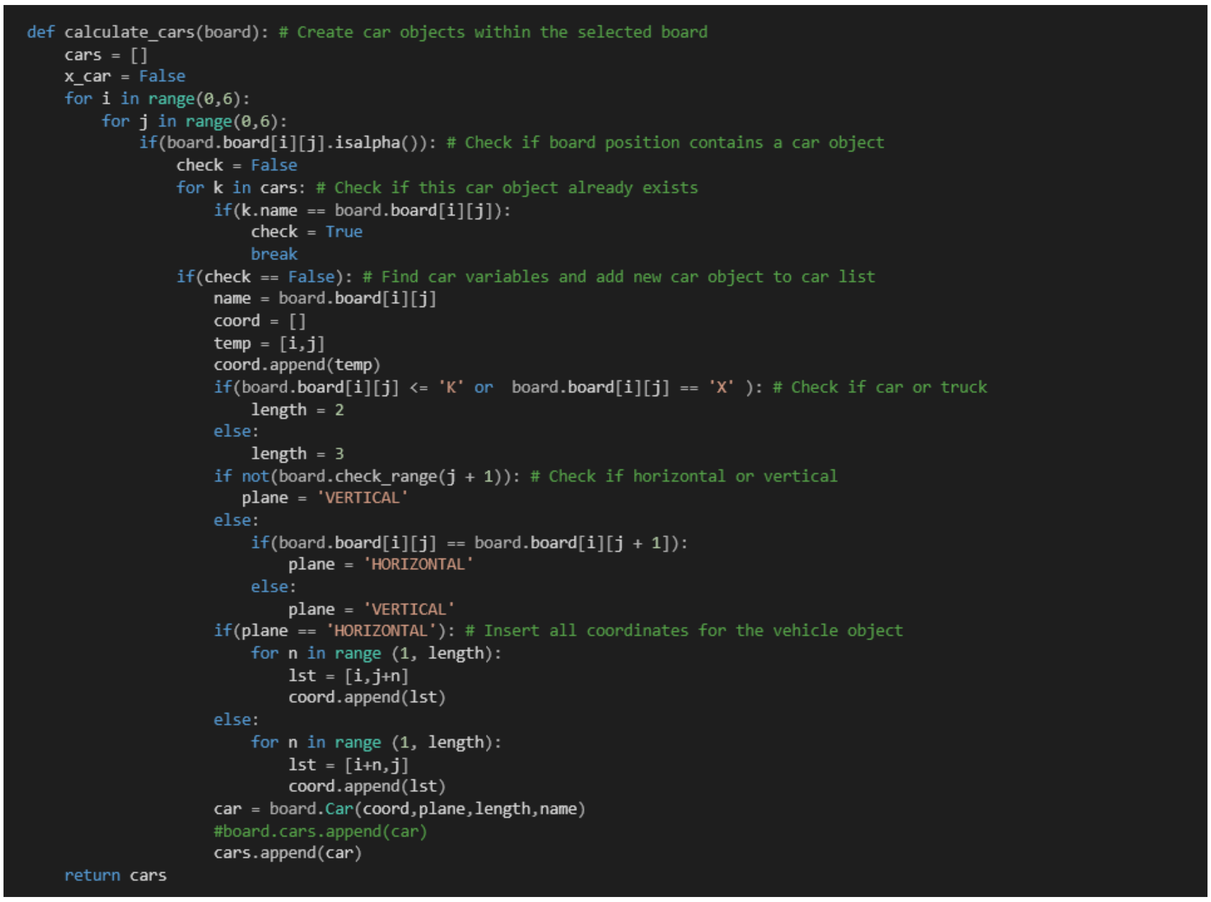
Method; check_range(i)

Check range takes an integer and checks if that value is < or = to the size of the board returning a Boolean value.

Method; check collision(self,tile)

Check_collision takes the board and an integer the loops through all car objects that exist within the given board object and compares their locations against the integer returning a boolean

Methhod; calculate_cars(board)



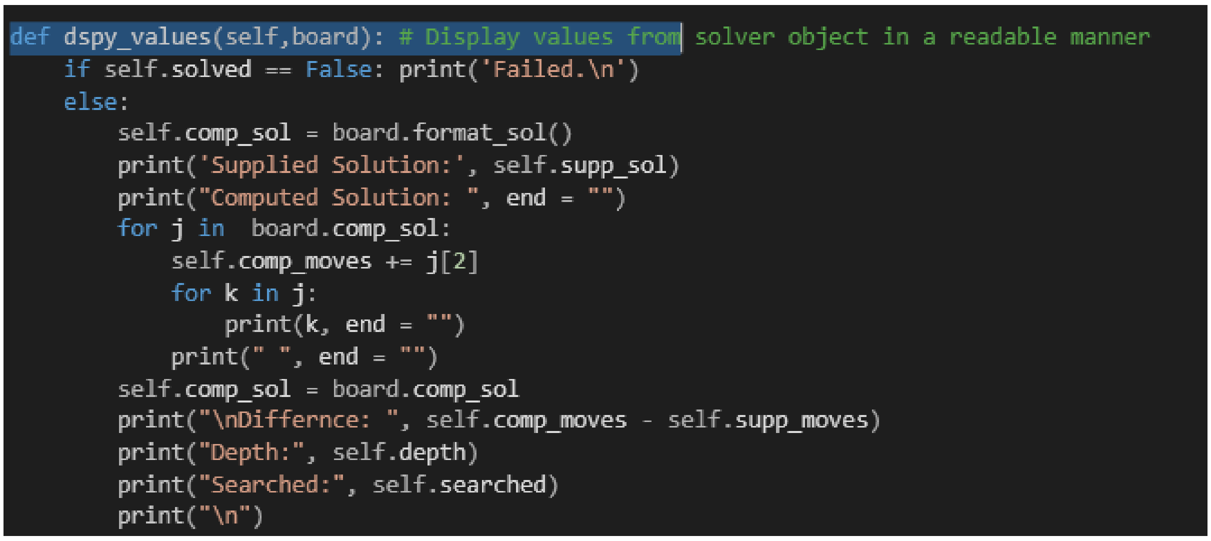
```
def calculate_cars(board): # Create car objects within the selected board
    cars = []
    x_car = False
    for i in range(0,6):
        for j in range(0,6):
            if(board.board[i][j].isalpha()): # Check if board position contains a car object
                check = False
                for k in cars: # Check if this car object already exists
                    if(k.name == board.board[i][j]):
                        check = True
                        break
                if(check == False): # Find car variables and add new car object to car list
                    name = board.board[i][j]
                    coord = []
                    temp = [i,j]
                    coord.append(temp)
                    if(board.board[i][j] <= 'K' or  board.board[i][j] == 'X' ): # Check if car or truck
                        length = 2
                    else:
                        length = 3
                    if not(board.check_range(j + 1)): # Check if horizontal or vertical
                       plane = 'VERTICAL'
                    else:
                        if(board.board[i][j] == board.board[i][j + 1]):
                            plane = 'HORIZONTAL'
                        else:
                            plane = 'VERTICAL'
                    if(plane == 'HORIZONTAL'): # Insert all coordinates for the vehicle object
                        for n in range (1, length):
                            lst = [i,j+n]
                            coord.append(lst)
                    else:
                        for n in range (1, length):
                            lst = [i+n,j]
                            coord.append(lst)
                    car = board.Car(coord,plane,length,name)
                    #board.cars.append(car)
                    cars.append(car)
    return cars
```

Calculate cars takes a board object and creates all car objects associated with that board.

Class; Solver

Solver contains everything we need to know about solving a board and the data gained from solving the board.

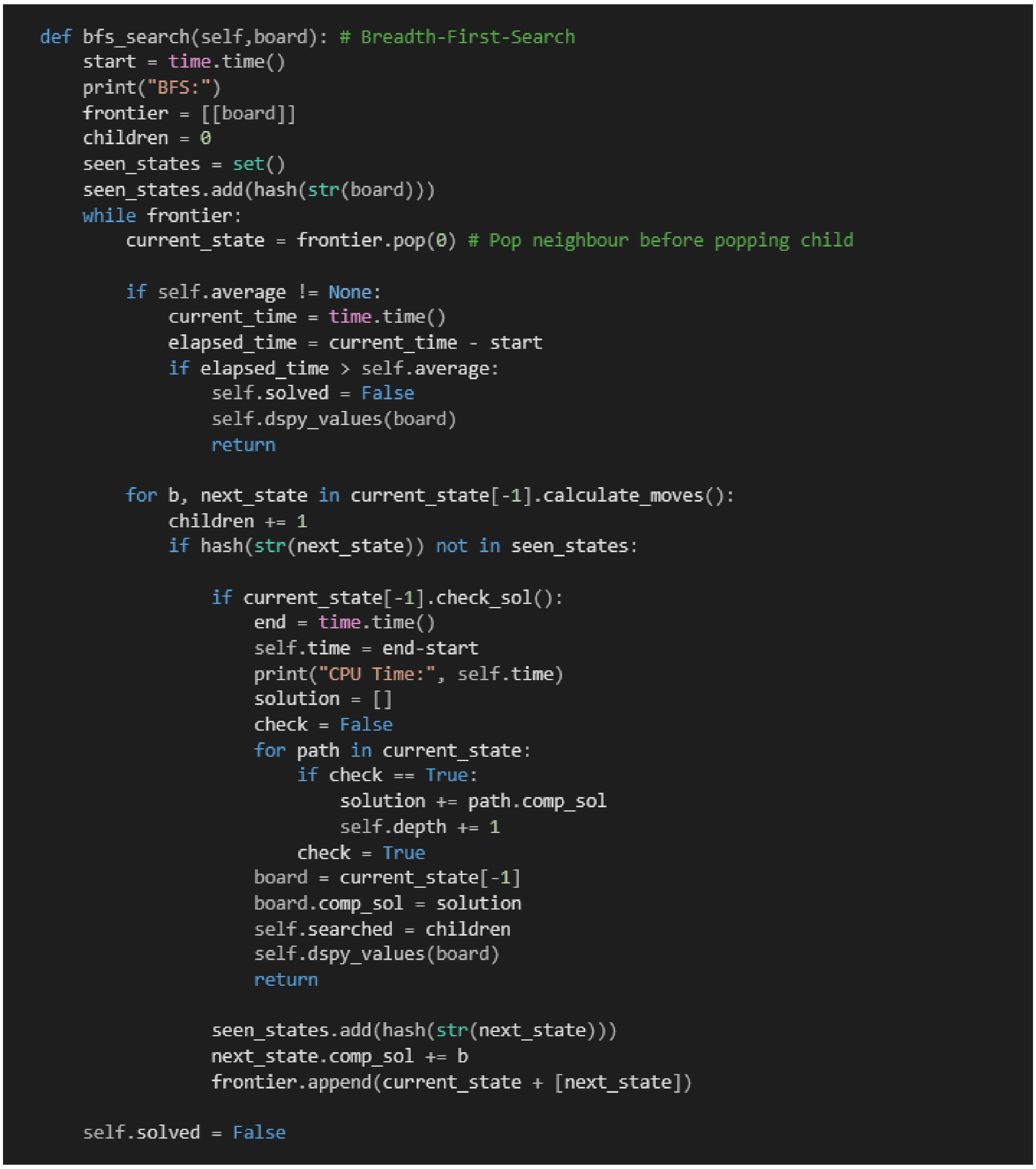
Method; dspy_values(self,board)



```
def dspy_values(self,board): # Display values from solver object in a readable manner
    if self.solved == False: print('Failed.\n')
    else:
        self.comp_sol = board.format_sol()
        print('Supplied Solution:', self.supp_sol)
        print("Computed Solution: ", end = "")
        for j in  board.comp_sol:
            self.comp_moves += j[2]
            for k in j:
                print(k, end = "")
            print(" ", end = "")
        self.comp_sol = board.comp_sol
        print("\nDiffernce: ", self.comp_moves - self.supp_moves)
        print("Depth:", self.depth)
        print("Searched:", self.searched)
        print("\n")
```

Dspy_values takes all the variables within the class solver and reformats them into a more readable format than prints them to the console for reference

Method bfs_search(self,board)



```python
def bfs_search(self,board): # Breadth-First-Search
    start = time.time()
    print("BFS:")
    frontier = [[board]]
    children = 0
    seen_states = set()
    seen_states.add(hash(str(board)))
    while frontier:
        current_state = frontier.pop(0) # Pop neighbour before popping child

        if self.average != None:
            current_time = time.time()
            elapsed_time = current_time - start
            if elapsed_time > self.average:
                self.solved = False
                self.dspy_values(board)
                return

        for b, next_state in current_state[-1].calculate_moves():
            children += 1
            if hash(str(next_state)) not in seen_states:

                if current_state[-1].check_sol():
                    end = time.time()
                    self.time = end-start
                    print("CPU Time:", self.time)
                    solution = []
                    check = False
                    for path in current_state:
                        if check == True:
                            solution += path.comp_sol
                            self.depth += 1
                        check = True
                    board = current_state[-1]
                    board.comp_sol = solution
                    self.searched = children
                    self.dspy_values(board)
                    return

                seen_states.add(hash(str(next_state)))
                next_state.comp_sol += b
                frontier.append(current_state + [next_state])

    self.solved = False
```

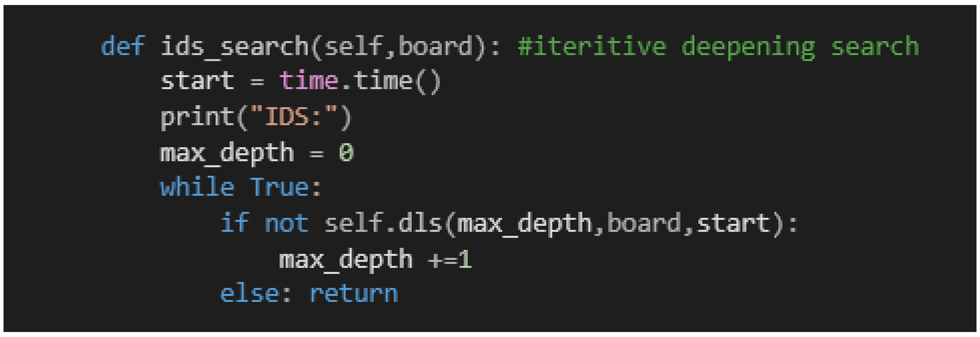
Bfs_search is where the bfs_search algorithm is employed. When this is called on a board object the method will begin trying to find the goal state using a Breadth First Search.

Method dls(self, max_depth, board)

```python
def dls(self,max_depth, board, start): # Depth-Limited-Search
    frontier = [[board]]
    cur_depth = 0
    children = 0
    seen_states = {}
    while frontier:
        current_state = frontier.pop() # Pop most recent child or neithbour if at max depth

        if self.average != None:
            current_time = time.time()
            elapsed_time = current_time - start
            if elapsed_time > self.average:
                self.solved = False
                self.dspy_values(board)
                return True

        if current_state[-1].check_sol():
            end = time.time()
            self.time = end-start
            print("CPU Time:", self.time)
            solution = []
            check = False
            for path in current_state:
                if check == True:
                  solution += path.comp_sol
                  self.depth += 1
                check = True
            board = current_state[-1]
            board.comp_sol = solution
            self.depth = len(current_state)
            self.searched = children
            self.dspy_values(board)
            return True

        cur_depth = len(current_state) - 1
        if(cur_depth < max_depth):
            for moves, next_state in current_state[-1].calculate_moves():
                    children += 1
                    if hash(str(next_state)) not in seen_states:
                        seen_states[hash(str(next_state))] = cur_depth
                        next_state.comp_sol += moves
                        frontier.append(current_state + [next_state])
                    else:
                        if (seen_states.get(hash(str(next_state))) > cur_depth):
                            seen_states.update({hash(str(next_state)): cur_depth})
                            next_state.comp_sol += moves
                            frontier.append(current_state + [next_state])
    return False
```

Dls is one part of the iterative deepening search algorithm and is responsible for performing iterative deepening to a certain depth

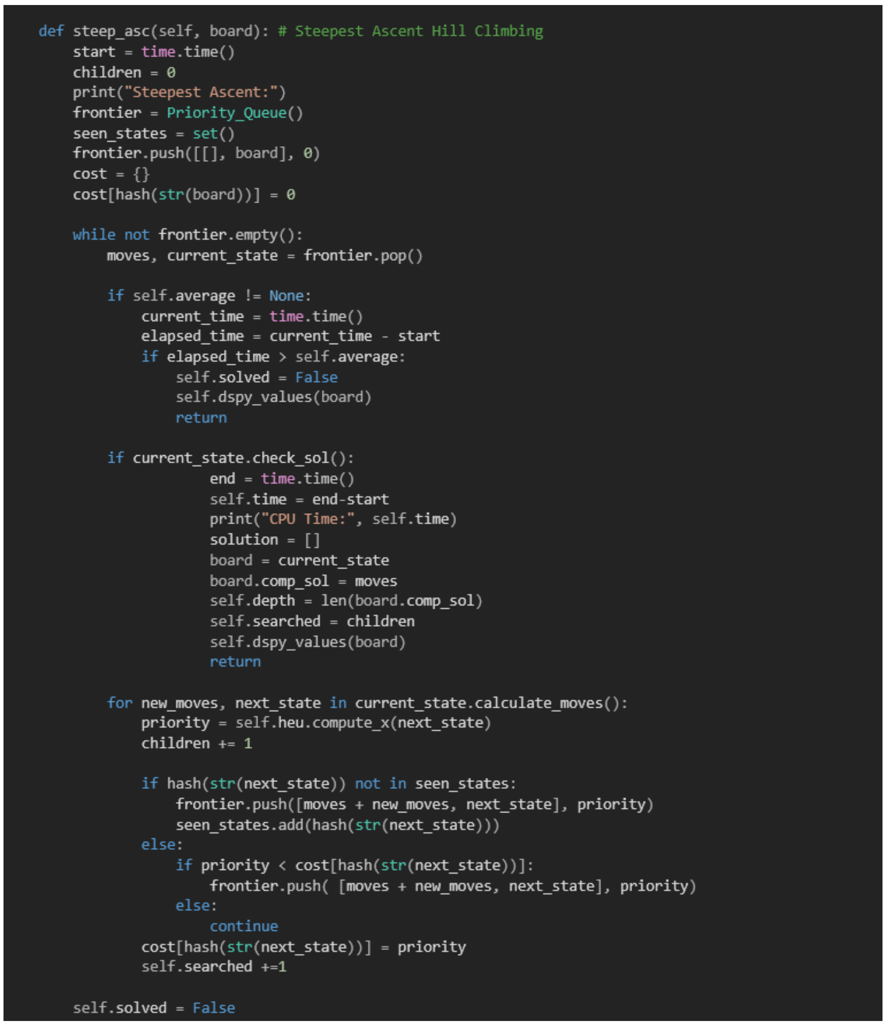
Method Ids_search



```
def ids_search(self,board): #iteritive deepening search
    start = time.time()
    print("IDS:")
    max_depth = 0
    while True:
        if not self.dls(max_depth,board,start):
            max_depth +=1
        else: return
```

Ids uses a while loop to call a depth limited search with a larger depth each time if a solution is not found at the current depth

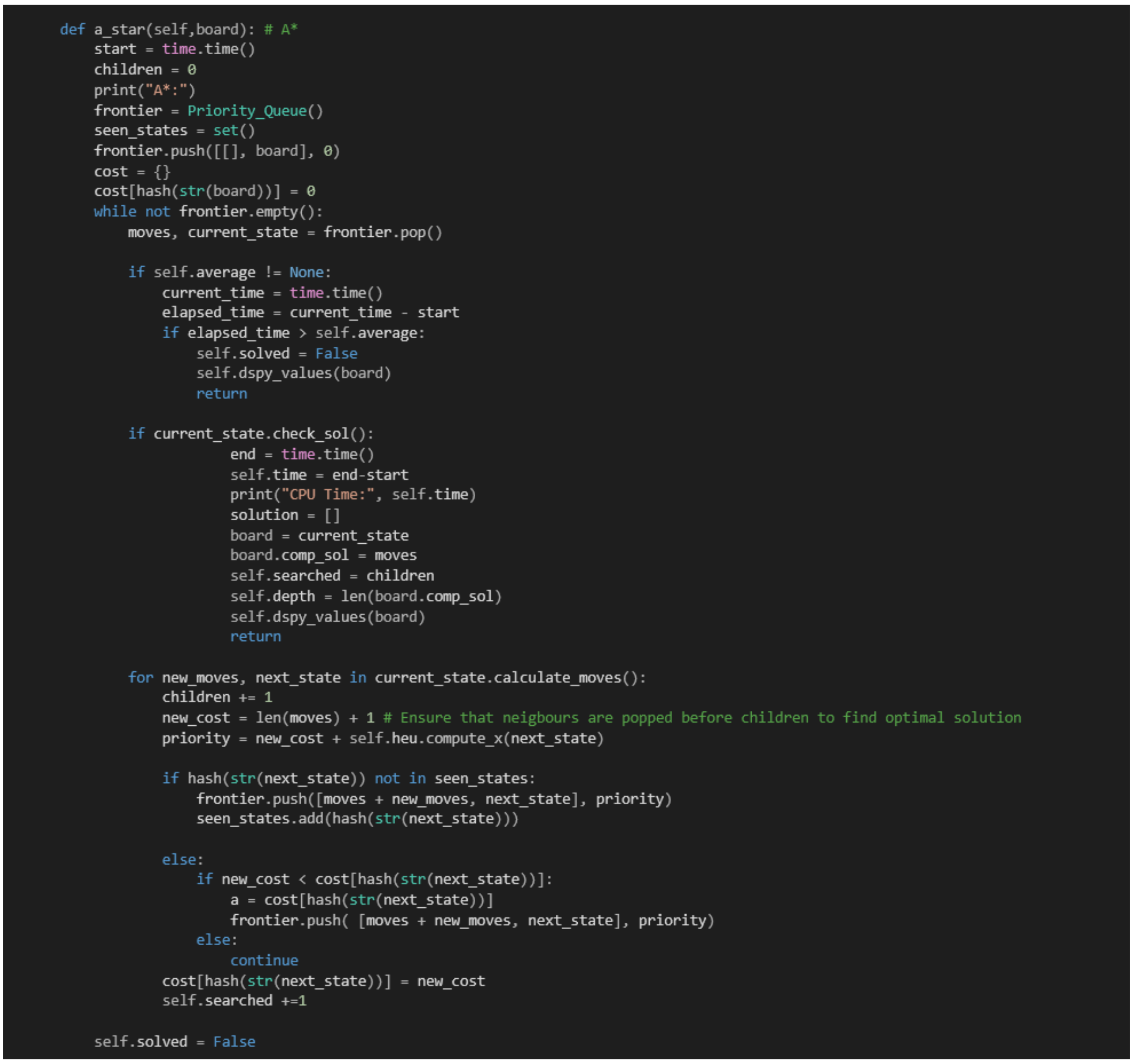
Method steep_asc(self,board)



```
def steep_asc(self, board): # Steepest Ascent Hill Climbing
    start = time.time()
    children = 0
    print("Steepest Ascent:")
    frontier = Priority_Queue()
    seen_states = set()
    frontier.push([[], board], 0)
    cost = {}
    cost[hash(str(board))] = 0

    while not frontier.empty():
        moves, current_state = frontier.pop()

        if self.average != None:
            current_time = time.time()
            elapsed_time = current_time - start
            if elapsed_time > self.average:
                self.solved = False
                self.dspy_values(board)
                return

        if current_state.check_sol():
                    end = time.time()
                    self.time = end-start
                    print("CPU Time:", self.time)
                    solution = []
                    board = current_state
                    board.comp_sol = moves
                    self.depth = len(board.comp_sol)
                    self.searched = children
                    self.dspy_values(board)
                    return

        for new_moves, next_state in current_state.calculate_moves():
            priority = self.heu.compute_x(next_state)
            children += 1

            if hash(str(next_state)) not in seen_states:
                frontier.push([moves + new_moves, next_state], priority)
                seen_states.add(hash(str(next_state)))
            else:
                if priority < cost[hash(str(next_state))]:
                    frontier.push( [moves + new_moves, next_state], priority)
                else:
                    continue
            cost[hash(str(next_state))] = priority
            self.searched +=1

    self.solved = False
```

Steep_asc Is the method responsible for implementing Steepest Ascent hill climbing.

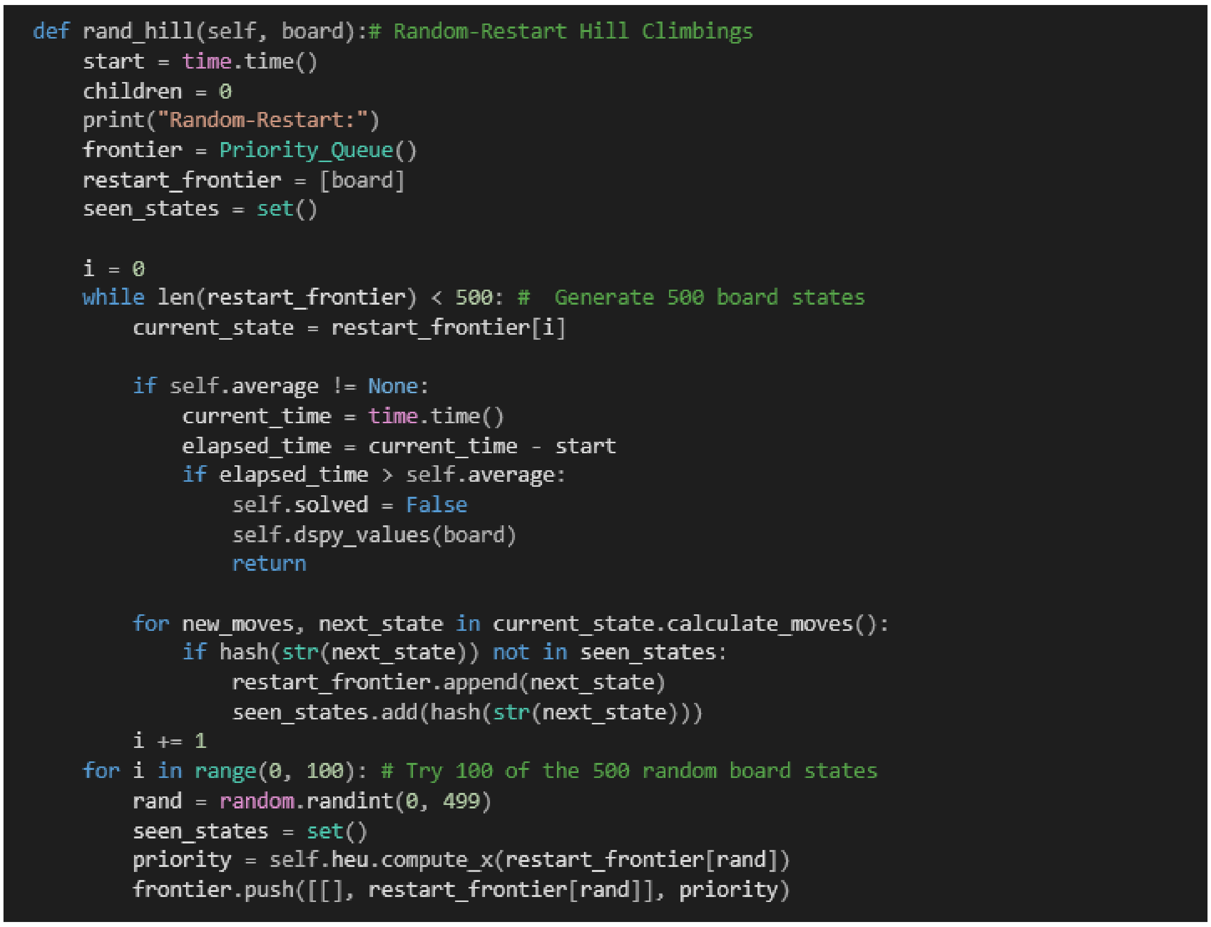
Method a_star



```
def a_star(self,board): # A*
    start = time.time()
    children = 0
    print("A*:")
    frontier = Priority_Queue()
    seen_states = set()
    frontier.push([[], board], 0)
    cost = {}
    cost[hash(str(board))] = 0
    while not frontier.empty():
        moves, current_state = frontier.pop()

        if self.average != None:
            current_time = time.time()
            elapsed_time = current_time - start
            if elapsed_time > self.average:
                self.solved = False
                self.dspy_values(board)
                return

        if current_state.check_sol():
                    end = time.time()
                    self.time = end-start
                    print("CPU Time:", self.time)
                    solution = []
                    board = current_state
                    board.comp_sol = moves
                    self.searched = children
                    self.depth = len(board.comp_sol)
                    self.dspy_values(board)
                    return

        for new_moves, next_state in current_state.calculate_moves():
            children += 1
            new_cost = len(moves) + 1 # Ensure that neigbours are popped before children to find optimal solution
            priority = new_cost + self.heu.compute_x(next_state)

            if hash(str(next_state)) not in seen_states:
                frontier.push([moves + new_moves, next_state], priority)
                seen_states.add(hash(str(next_state)))

            else:
                if new_cost < cost[hash(str(next_state))]:
                    a = cost[hash(str(next_state))]
                    frontier.push( [moves + new_moves, next_state], priority)
                else:
                    continue
            cost[hash(str(next_state))] = new_cost
            self.searched +=1

    self.solved = False
```

A_star uses both the adv_heuristic and the priority_queue classes to find a solution using an informed search.

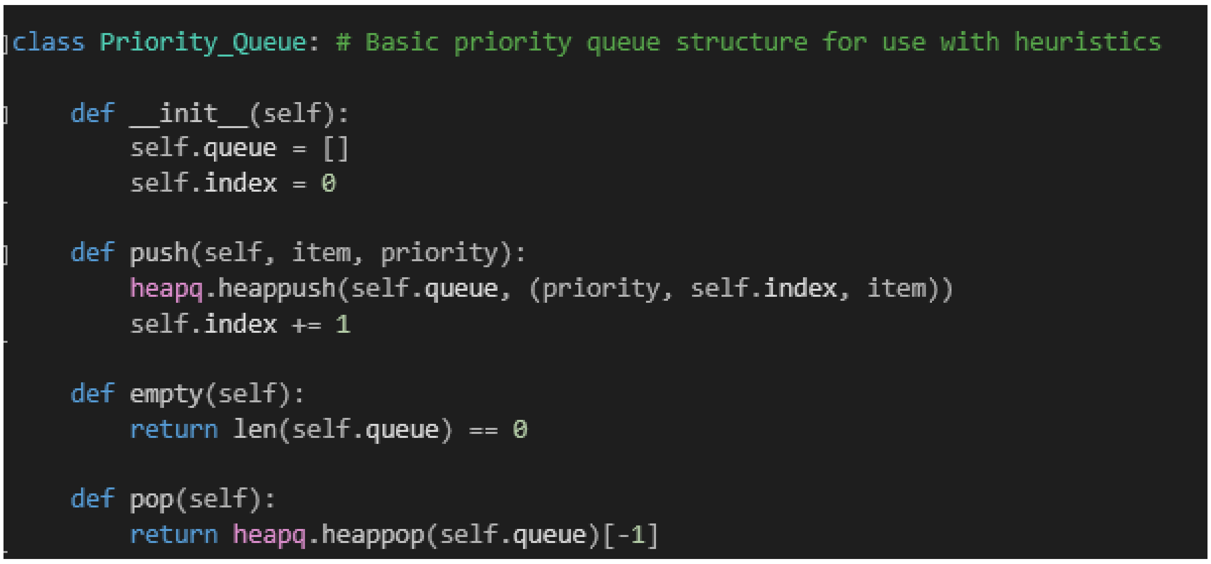
Method rand_hill()



```python
def rand_hill(self, board):# Random-Restart Hill Climbings
    start = time.time()
    children = 0
    print("Random-Restart:")
    frontier = Priority_Queue()
    restart_frontier = [board]
    seen_states = set()

    i = 0
    while len(restart_frontier) < 500: #  Generate 500 board states
        current_state = restart_frontier[i]

        if self.average != None:
            current_time = time.time()
            elapsed_time = current_time - start
            if elapsed_time > self.average:
                self.solved = False
                self.dspy_values(board)
                return

        for new_moves, next_state in current_state.calculate_moves():
            if hash(str(next_state)) not in seen_states:
                restart_frontier.append(next_state)
                seen_states.add(hash(str(next_state)))
        i += 1
    for i in range(0, 100): # Try 100 of the 500 random board states
        rand = random.randint(0, 499)
        seen_states = set()
        priority = self.heu.compute_x(restart_frontier[rand])
        frontier.push([[], restart_frontier[rand]], priority)
```

rand_hill implements a random-restart hill climbing algorithm, this is done by first generating 500 different board states and then randomly trying 100 of those 500 to see if a global maximum can be attained

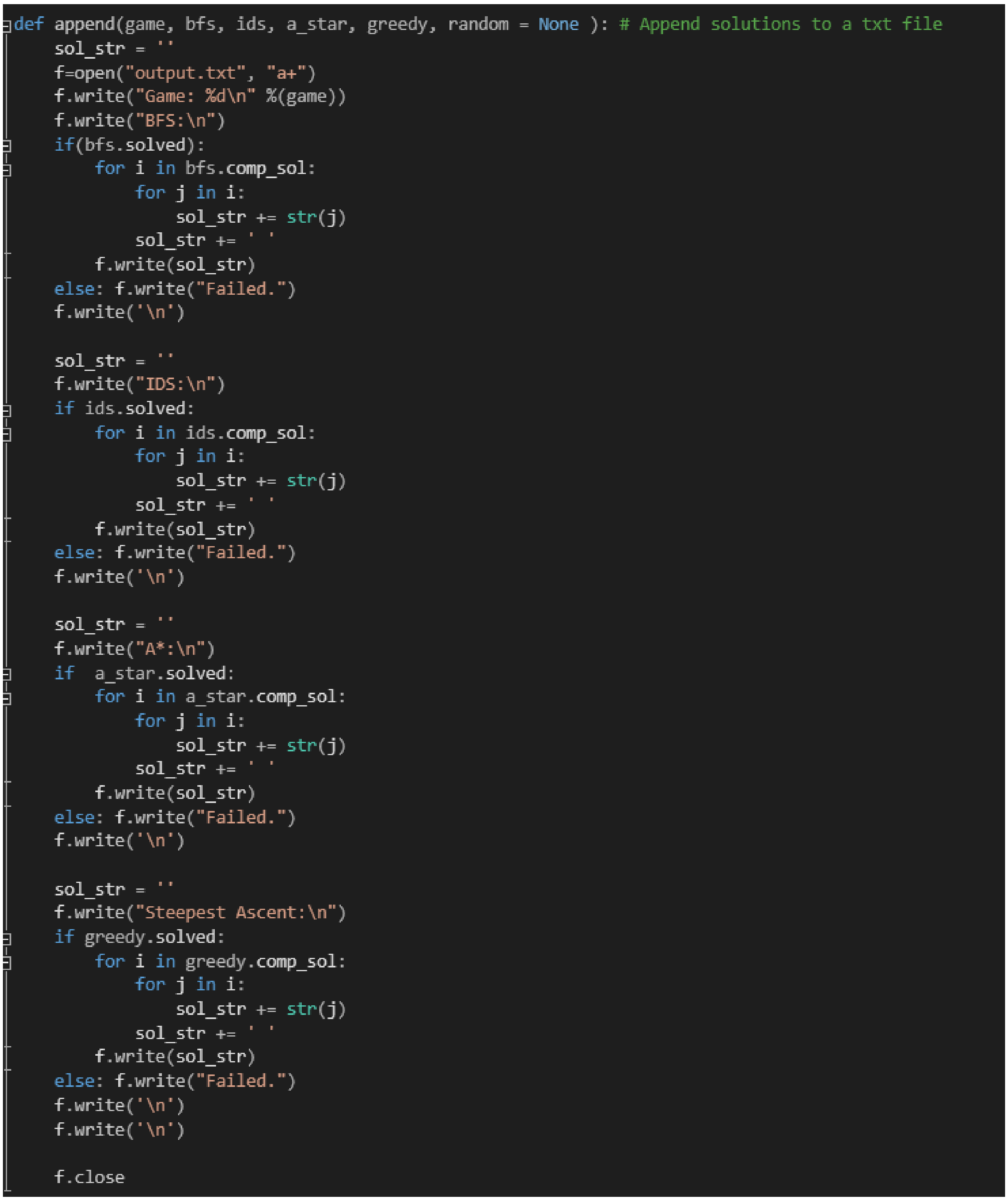
Class Priority_Queue



```python
class Priority_Queue: # Basic priority queue structure for use with heuristics

    def __init__(self):
        self.queue = []
        self.index = 0

    def push(self, item, priority):
        heapq.heappush(self.queue, (priority, self.index, item))
        self.index += 1

    def empty(self):
        return len(self.queue) == 0

    def pop(self):
        return heapq.heappop(self.queue)[-1]
```

Pritiory_queue implements a simple priority queue structure for use in informed searches

Function append()



```python
def append(game, bfs, ids, a_star, greedy, random = None ): # Append solutions to a txt file
    sol_str = ''
    f=open("output.txt", "a+")
    f.write("Game: %d\n" %(game))
    f.write("BFS:\n")
    if(bfs.solved):
        for i in bfs.comp_sol:
            for j in i:
                sol_str += str(j)
            sol_str += ' '
        f.write(sol_str)
    else: f.write("Failed.")
    f.write('\n')

    sol_str = ''
    f.write("IDS:\n")
    if ids.solved:
        for i in ids.comp_sol:
            for j in i:
                sol_str += str(j)
            sol_str += ' '
        f.write(sol_str)
    else: f.write("Failed.")
    f.write('\n')

    sol_str = ''
    f.write("A*:\n")
    if  a_star.solved:
        for i in a_star.comp_sol:
            for j in i:
                sol_str += str(j)
            sol_str += ' '
        f.write(sol_str)
    else: f.write("Failed.")
    f.write('\n')

    sol_str = ''
    f.write("Steepest Ascent:\n")
    if greedy.solved:
        for i in greedy.comp_sol:
            for j in i:
                sol_str += str(j)
            sol_str += ' '
        f.write(sol_str)
    else: f.write("Failed.")
    f.write('\n')
    f.write('\n')

    f.close
```

Append is responsible for appending all solutions to the current board into a txt file called output

Algorithm Comparison

An external excel sheet is used to methodically compare and contrast the different algorithms. The sheets averages and overall comparisons are embedded here:

Average Overall					
BFS	12.7191524	-0.45	47.825	17981.88	0
IDS	72.52795331	-0.666666667	21.66667	16565.17	34
A*	13.70432588	-0.282051282	47.74359	11967.31	1
Steepest Ascent	7.028709483	7.7	56.225	6814.825	0
Random-Restart					
	Time	Difference	Depth	Searched	Failed?

Average Total	
Time	26.49503527
Difference	1.575320513
Depth	43.3650641
Searched	13332.29359
Failed?	8.75

These averages allow us to determine a reasonable maximal time for finding solutions and from here all manner of comparisons can be done with ease.

Conclusion

We can conclude a great many things here. We can see that Steepest Ascent has on average the best performance when it comes to cpu time. However, this clearly comes at a cost in regard to optimal solutions with Steepest Ascent being the most likely to return a not optimal path towards the goal state. IDS was by far the most taxing on the cpu in regard to time however, for every problem it managed to find an optimal solution. BFS manages to find an optimal solution for every problem besides 13 which I believe to be a bug in how my program determines lengths of solutions and not an issue with the actual algorithm but has the biggest tax on memory space and needs to search through the most nodes to find a solution. A* manages the same with a slightly longer cpu time and fewer searched nodes, with one exception at problem 23 where due to the design of the heuristic used within A* it always prioritizes a not optimal move as that moves an object considered a priority. If memory is no issue, then BFS is clearly the winner for finding the optimal solution in a good cpu time. With a better Heuristic A* would most likely beat BFS in CPU time to solution, However if the optimal path is not necessary Hill climbing will find the result in much quicker time if the algorithm is allowed to break ties, but the more ties that the program breaks results in larger local maximums if it fails to reach the global maximum, which can be seen in all the advanced games (30 – 40)