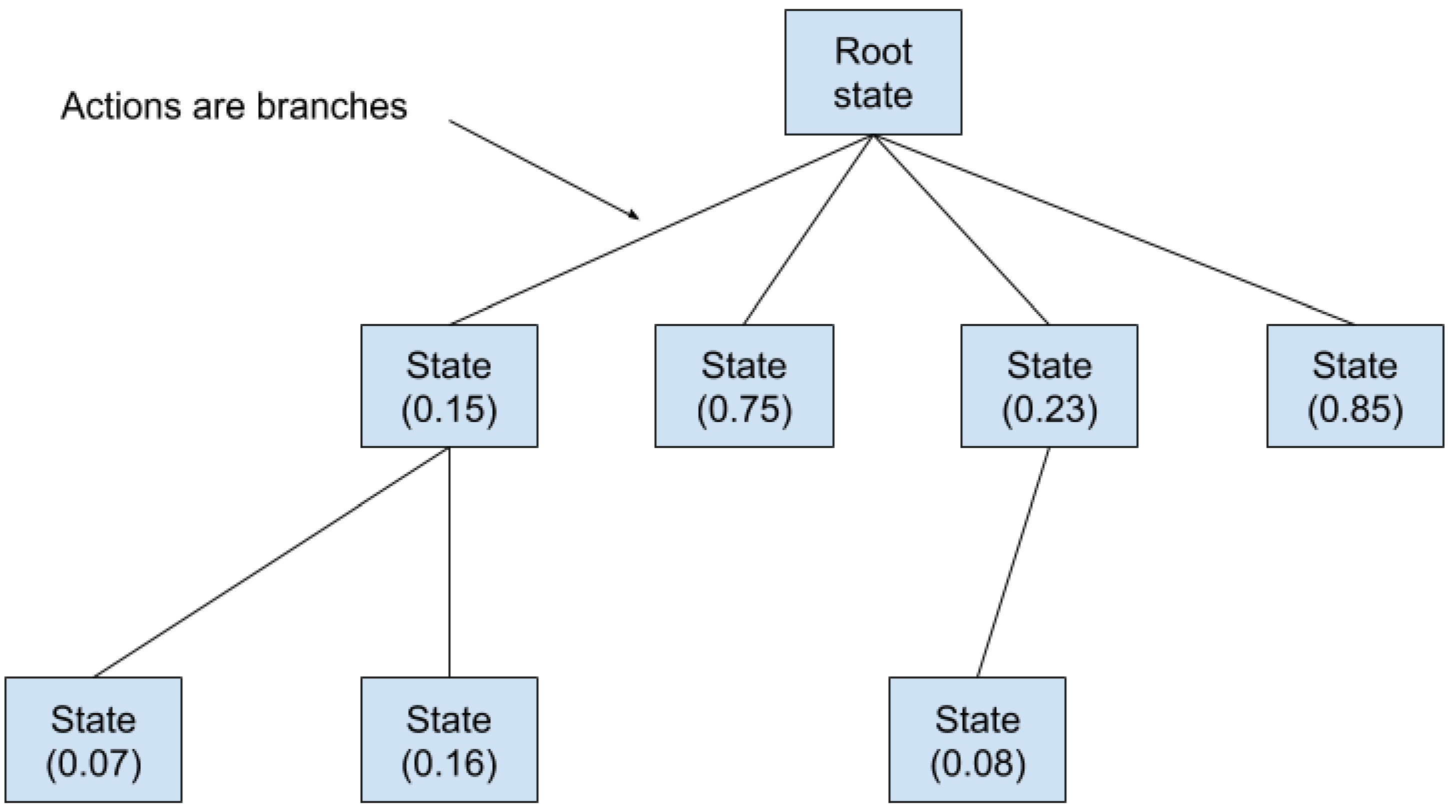
Data Structures and Algorithms (DSA) Assignment 1 Algorithm Analysis

By Marcus Belcastro (19185398)

Outline of the search algorithm and other processes

The search function used for my mini-SHRDLU game is the best-first-search (BFS) algorithm. It is a depth-first, tree-based, recursive algorithm that picks the next best node in a tree based on the smallest heuristic. A tree structure is not physically created in my program, however, it can be diagrammed as a tree where each node is a state and each branch from a node is an action that leads to a new node with a particular heuristic associated with it.



For any state in the tree, all associated possible actions are created and put into a queue based on heuristically how well they solve the board. When each action is tested, a hash of the resulting board is loaded into a hash map (std::unordered_set) in order to prevent duplicate states and from reversing the previous action (hence resolving loops in the graph).

Heuristics are calculated using a linear distance from the goal to the tile it relates to. Specific heuristical distance calculations for conjunctive/disjunctive goal lists and atom/neighbour goals are specified elsewhere in this document.

Numerical definitions

These definitions will be used to determine the functional complexity for each function of the program and a big-O representation for both the explicit case (each variable as it is) and the general case (in terms of n). Hence I have provided each variable other than n a value in terms of n for the worst case.

n - The width of the state, creates an $n \times n$ board

t - The amount of tiles that are specified to go on the board (worst case $n^2 - n$)

g - The number of goals that are specified (worst case ∞ so we will use an average case of $\frac{n}{2}$)

Algorithmic analysis

The following will walk through an analysis of the complexity of the function:

```
bool bestFirstSearch(State* node, int maxRecurse)
```

It will first walk through all the relevant functions per class and then assemble them all into the final operations.

State

Construction

The constructor used in the BFS is the copy constructor:

```
FUNCTION Constructor(State obj s)
     Let size = the size of the board in state s
     Let numTiles = the number of tiles in s

     Initialise the board's memory
     Let otherBoard = the board of s
     FOR x = 0 to size
          FOR y = 0 to size
               This.board[x][y] = otherBoard[x][y]
          ENDFOR
     ENDFOR

     this.maxHeuristic = the max heuristic of state s
ENDFUNCTION
```

- Any assignments of values is a constant operation of 1
- Initialisation of memory is similar to the loop provided in this function, it's functional complexity is $n(1+n)+1 = n^2+n+1$
- The nested for-loop creates a functional complexity of n^2

Hence the total functional and general complexity is

$$2n^2 + n + 5 \Rightarrow O(n^2)$$

void State::find(int num, int& x, int& y)

This function finds the coordinates of num as x and y.

```
FUNCTION find(int num, int x, int y)
     x = 0
     y = 0

     FOR cx = 0 to size
          FOR cy = 0 to size
               IF (this.board[cx][cy] equals num) THEN
                    x = cx
                    y = cy
                    RETURN
               ENDIF
          ENDFOR
     ENDFOR
ENDFUNCTION
```

- Any assignments of values is a constant operation of 1
- The nested for-loop creates a functional complexity of $n(n(3)) = 3n^2$

Hence the total functional and general complexity is

$$3n^2 + 2 \Rightarrow O(n^2)$$

void State::getPossibleMoves(vector<Action>& actionList)

This functions gets all possible moves and stores them into a std::vector structure

```
FUNCTION getPossibleMoves(array struct actionList)
     FOR col = 0 to size
          IF the column is empty THEN
               FOR otherCol = 0 to size
                    IF col is not otherCol AND col has room for tile THEN
                         Add the action of moving the tile from col to the
                         otherCol to the actionList
                    ENDIF
               ENDFOR
          ENDIF
     ENDFOR
ENDFUNCTION
```

- Determining emptiness and if a column has room have a constant functional complexity of 4
- Comparisons all have a functional complexity of 1
- Creating a new action has a functional complexity of 2
- Adding an action to the list has a functional complexity of the sum of i from 1 to the number of times the pushing occurs, where i is the size of the list at that particular instant.
- The total possible actions for any state using one column at worst case is $n-1$ hence the complexity of the inner loop is
$$\sum_{i=1}^{n-1}(i+7) = \sum_{i=1}^{n-1} i + \sum_{x=1}^{n-1} 7 = \frac{(n-1)^2}{2} + \frac{n-1}{2} + 7n - 7$$
- Multiplying the outer loop we get
$$\sum_{i=1}^{n(n-1)} (i+7) + 4n = \sum_{i=1}^{n(n-1)} i + \sum_{x=1}^{n(n-1)} 7 + 4n = \frac{(n^2-n)^2}{2} + \frac{n^2-n}{2} + 7n^2 - 3n$$
$$= \frac{n^4-2n^3+n^2}{2} + \frac{n^2-n}{2} + 7n^2 - 3n$$
- Generalising with big-O we get
$$\frac{n^4-2n^3+n^2}{2} + \frac{n^2-n}{2} + 7n^2 - 3n \Rightarrow O(n^4)$$
- And a worst case size for the total number of actions being n^2-n

void State::performAction(Action& a) (pushToCol() + popFromCol())

The performAction() function relies on a single push to a column and a single pop from a column So it is the addition of the complexity of the two functions.

```
FUNCTION popFromCol(int col)
     If the array is empty, raise an error.
     Let i be the size of the board - 1
     Let current be the number at row i, column col
     WHILE current equals 0 AND i > 0
          Decrement i
          Set current to the number at row i, column col
     ENDWHILE
     IF current is not 0 THEN
          Let temp be the number at row i, column col
          The number at row i, column col is set to 0
          RETURN temp
     ENDIF
     RETURN 0
ENDFUNCTION
```

```
FUNCTION popFromCol(int val, int col)
     If the top of the column is not 0, the value is less than 1 or the
     value is greater than the number of tiles, raise an error.
     i = 0
     Let current be the number at row 0, column col
     WHILE current not equal 0 AND i < size of board - 1
          Increment i
          Set current to the number at row i, column col
     ENDWHILE
     IF current is 0 THEN
          The number at row i, column col is set to val
     ENDIF
ENDFUNCTION
```

- The functions are the same apart from pop having a constant +7 and push having a constant +12
- The assertions at the head of each function cause the while loop to iterate a maximum of n-1 times
- Adding the two functional complexities together for the performAction() we get

$$2(n-1) + 12 + 2(n-1) + 7 = 4n + 15 \Rightarrow O(n)$$

string State::getHash()

This function returns a string representing the current state of the board. Used as a unique key for tree pruning via hash-maps.

```
FUNCTION getHash()
     Let hash be an empty string
     FOR x = 0 to size
          FOR y = 0 to size
               Append the character of the tile at board[x][y] to hash
          ENDFOR
     ENDFOR
     RETURN hash
ENDFUNCTION
```

- Lets and returns are constant 1
- Appending has a functional complexity of i where i is the size of the list at that particular instant.
- The for loop is run n by n times and hence the complexity is

$$\sum_{i=1}^{n^2} i + 2 = \frac{n^4}{2} + \frac{n^2}{2} + 2 \Rightarrow O(n^4)$$

Destruction

The destructor simply deletes n arrays from the board and then deletes the board array itself. Hence the complexity is:

$$n^2 + n \Rightarrow O(n^2)$$

Goal

A goal is split into two polymorphic types, an atom goal and neighbour goal. The following algorithms will show the implementation of the neighbour goal as it is of higher complexity than an atom goal.

A goal is split into a tuple of 3 numbers, for a neighbour goal, the first and last numbers are the two tiles that need to be close to each other and the middle number represents a direction. This is stored in the variable goalTupe[3].

bool NeighbourGoal::isSatisfied(State* gameState)

This function checks if the goal has been satisfied.

```
FUNCTION isSatisfied(game state s)
     Let x and y = 0
     Find the coordinates of goalTuple[2] as x and y
     IF The direction is ABOVE AND the above tile is on the board THEN
          RETURN true if the above tile equals goalTuple[0]
     ENDIF
     IF The direction is BELOW AND the below tile is on the board THEN
          RETURN true if the below tile equals goalTuple[0]
     ENDIF
     IF The direction is LEFT AND the left tile is on the board THEN
          RETURN true if the left tile equals goalTuple[0]
     ENDIF
     IF The direction is RIGHT AND the right tile is on the board THEN
          RETURN true if the right tile equals goalTuple[0]
     ENDIF
ENDFUNCTION
```

- Each let, return and comparison is of constant complexity of 1
- Only one of the if-statements will ever run
- The find function’s complexity is defined above
- Hence the functional complexity is
  $1 + 3n^2 + 2 + 4 = 3n^2 + 7 \Rightarrow O(n^2)$

double NeighbourGoal::getHeuristic(State* gameState)

This function returns a floating point number that represents how far the state is from the goal. 0 means it has completed the goal and 1 means it is the farthest away possible.

```
FUNCTION isSatisfied(game state s)
     Let x, y, goalNumX and goalNumY = 0
     Let Final be 1.0
     Find the coordinates of goalTuple[2] as x and y
     Find the coordinates of goalTuple[0] as goalNumX and goalNumY
     IF The direction is ABOVE AND the above tile is on the board THEN
          IF the tile above is free THEN
               Final = linear distance between x+1,goalNumX and y,goalNumY
          ENDIF
     ENDIF
     IF The direction is BELOW AND the below tile is on the board THEN
          Final = linear distance between x-1,goalNumX and y,goalNumY
     ENDIF
     IF The direction is LEFT AND the left tile is on the board THEN
          IF the tile left is free THEN
               Final = linear distance between x,goalNumX and y-1,goalNumY
          ENDIF
     ENDIF
     IF The direction is RIGHT AND the right tile is on the board THEN
          IF the tile right is free THEN
               Final = linear distance between x,goalNumX and y+1,goalNumY
          ENDIF
     ENDIF
     RETURN final
ENDFUNCTION
```

- Each let, return and comparison is of constant complexity of 1
- Only one of the if-statements will ever run, every one but the BELOW one is the most complex
- The find function's complexity is defined above
- The Complexity of the linear distance is 7
- Hence the functional complexity is

$$5 + 6n^2 + 4 + 11 = 6n^2 + 20 \Rightarrow O(n^2)$$

GoalList

The goal list is split into two different polymorphic types, a conjunctive goal list or a disjunctive goal list. The following will show the implementations of the disjunctive list as it is of slightly higher complexity.

bool DisjunctiveGoalList::isSatisfied(State* gameState)

This function checks if at least one of the goals of the goal list are satisfied.

```
FUNCTION isSatisfied(game state s)
     FOR every goal in the goal list
          IF the goal is satisfied given s THEN
               RETURN false
          ENDIF
     ENDFOR
     RETURN false
ENDFUNCTION
```

- Each return and comparison is of constant complexity of 1
- The goal satisfaction complexity is defined above
- The worst case assumes that the very last goal in the list is correct and all others are not satisfied
- Hence the functional complexity is

$$g(3n^2 + 7) + 1 = \frac{3n^3}{2} + \frac{7n}{2} + 1 \Rightarrow O(n^3)$$

void DisjunctiveGoalList::getActionHeuristic(State* gameState, Action* act)

This function gets the aggregate heuristic for an action given all the goals that must be achieved and applies it to the action object.

```
FUNCTION isSatisfied(game state s, current action a)
    Let min be 1.0
    FOR every goal in the goal list
        Let temp be the heuristic from the current goal given s
        IF temp is smaller than min THEN
            min = temp
        ENDIF
    ENDFOR
    Set the heuristic of a to be min
ENDFUNCTION
```

- Each let and comparison is of constant complexity of 1
- The worst case assumes that every goal including the last have a smaller heuristic than the previous
- Calculating the heuristic has the complexity described above
- Setting the heuristic has a complexity of 1
- Hence the functional complexity is

$$1 + g(6n^2 + 20 + 3) + 1 = 3n^3 + \frac{23n}{2} + 2 \Rightarrow O(n^3)$$

Solver

The solver class implements a variety of algorithms and helper functions that solve the mini-SHRDLU game. The following will focus on the best first search algorithm.

bool Solver::hashExists(string hash)

This function abstracts the hash-map that prunes the game tree by preventing duplicate states. It takes a hash of a game state and stores it in a hash-map if it doesn't already exist. Else it says it already exists.

```
FUNCTION hashExists(string hash)
     IF the hash exists in the hash-map THEN
          RETURN true
     ELSE
          Insert the hash into the hash-map
          RETURN false
     ENDIF
ENDFUNCTION
```

- Comparisons and returns have a functional complexity of 1
- Insertion and checking membership of the hash map has a functional complexity of 1 (by definition of a hash-map)
- Hence the functional complexity is

  $$2 \Rightarrow O(1)$$

void Solver::getHeuristicActions(State* currentState, priority_queue<Action>& q)

This function calculates the heuristics for all the possible actions for a particular state. It performs the action and reverses it in order to calculate how well it did.

```
FUNCTION getHeuristicActions(game state s, queue where actions go Q)
    Let Acts be a dynamic array of actions
    Get all the actions from s and put it into Acts
    FOR every action in Acts
        Perform the action on s
        IF the hash of s does not exist in the hash-map THEN
            Using the final goal, get the heuristic for the action
            Push the action to Q
        ENDIF
        Reverse the action on s
    ENDFOR
ENDFUNCTION
```

- Lets and comparisons have a functional complexity of 1
- The following process complexities have been calculated above:
    - Getting all possible actions
    - Performing the action
    - Reversing the action (same as performing it)
    - Whether the hash exists
    - Getting the heuristics for an action
- The amount of actions that will be iterated is at worst $n^2 - n$
- Hash checking and tree pruning is ignored for worst-case analysis
- Pushing the action to an ordered queue (according to [c++ documents]) is in the complexity $i + log(i)$ where i is the size of the queue at that moment in time
- Hence the complexity of the for loop is

$$\frac{n^4 - 2n^3 + n^2}{2} + \frac{n^2 - n}{2} + 7n^2 - 3n + (n^2 - n)(4n + 15) + \sum_{i=1}^{n^2-n} i + \sum_{i=1}^{n^2-n} log(i)$$
$$= n^4 - 2n^3 + n^2 + \frac{n^2}{2} - \frac{n}{2} + 7n^2 - 3n + 4n^3 + 15n^2 - 4n^2 - 15n + l(n)$$
$$= n^4 - 2n^3 + \frac{39n^2}{2} + \frac{25n}{2} + l(n) \Rightarrow O(n^4)$$

Where l(n) is the sum of the logarithms over an interval. This is beyond the scope of the course and hence I left it as a functional constant.

bool Solver::bestFirstSearch(State* node, int maxRecurse)

This is the recursive function that runs the best first search. It gets the actions into a priority queue, runs through them and unless they are the winning state, it will recurse into a new instance of the function.

```
FUNCTION bestFirstSearch(game state s, int maxRecursion)
     IF maxRecusion is less than 1 THEN
          RETURN false
     ENDIF
     Get the queue Q of the actions and heuristics
     WHILE Q is not empty
          Let A be the next action in Q
          Create a new state called Node as a copy of s
          Perform the current action
          Add the action to the plan
          IF Node is solved THEN
               Make the main state equal to Node
               RETURN true
          ELSE IF bestFirstSearch(Node, maxRecursion-1) is true THEN
               RETURN true
          ENDIF
          Free the memory of Node
          Remove the latest action from the plan
          Move to the next action in the queue
     ENDWHILE
     RETURN false
ENDFUNCTION
```

- Freeing of memory, comparisons, lets, getting the next action and plan pushes and pops all have functional complexity of 1
- The following functional complexities have been calculated previously:
  - Getting Q
  - Copy constructor of State
  - Destruction of State
  - Checking if the state is satisfied
  - Performing actions
- The assignment brief allows us to assume that queue operations are of log(n) hence the sum of queue pops will be of complexity l(n)
- Combining all of the complexities together we get the complexity of the BFS as

$$n^4 - 2n^3 + \frac{39n^2}{2} + \frac{25n}{2} + l(n) + 1 + (n^2 - n)(3n^2 + 6n + 24 + \frac{3n^3}{2} + \frac{7n}{2} + l(n))$$
$$= \frac{3n^5}{2} + \frac{5n^4}{2} + \frac{9n^3}{2} + 41n^2 - \frac{23n}{2} + 1 + n^2 l(n) - nl(n) + l(n)$$

In the assignment brief, we are told to assume that queue operations have a complexity of log(n), so the function l(n) now = log(n)

- Hence if we ignore recursion, we get a big-O complexity for only the algorithm of $O(n^5)$