COMP332 Assignment 1

Conor Apcar

4462 4336

Introduction

The assignment was built around the concept of B-trees and creating a path planner using the A* algorithm. A framework for the project was provided, with code already describing the form of a B-tree. This framework also provided most of the implementation for the path planner, as well as the code for the A* algorithm.

The project required that certain elements would be designed and tested. The parts that required our implementation in the BTree.scala module were the 'find()' method, the 'pprint()' method, and the 'inOrder()' method. Code was also required for the PathPlan.scala module, to create an image of the path plan.

This report presents my own implementation of the methods that are described above, as well the testing process of these methods.

But First, this…

I think it's important to mention the call 'this' which is used in all three methods that I have carried out in the BTree.scala module.

```
def find(v : T): Option[T] = this match {
```

In the above code we can see the part stating 'this match'. For all of the methods in the BTree.scala module that I have had to solve, I have used a match and case recursion. Anytime we wish to use one of these methods on a given tree, the call is 'tree.find(v)'. Since we are trying to match a node of the tree to a value, 'this' is used to reference the current object, or node in this case that the tree is on.

So when we recursively call a method, whatever value is next in the tree becomes ‘this’ since it is the current object.

Find Method

The find method was implemented to find an entry in the B-tree, based off a straightforward variant of binary tree searching. The Find method has a parameter value ‘v’ with type T. The method should return the value as ‘Some(v)’, as it is some of an entry that has been found equal to v.

The main part of the code involves a match and case call, with the three cases being an ‘EmptyNode()’, a ‘TwoNode(t1, u1, t2)’ B-tree, or a ‘ThreeNode(t1, u1, t2, u2, t3)’ B-tree. The ‘EmptyNode’ should return the value ‘None’. This will occur if there is no B-tree, or the B-tree has been recursively searched already and the value has not been found. For the case of TwoNode I have recursively called both t1 and t2 trees and checking if the value u1 is equal to v.

```
case TwoNode(t1, u1, t2) =>
  if (u1 equiv v)  {
    Some(v)
  } else {
    t1.find(v) orElse t2.find(v)
  }
```

From the code I’ve implemented above, the main thing to note is the code ‘(u1 equiv v)’. The equiv method is helpful for finding the equivalent value of type T. using ‘==’ would only work for number based values, however we also need to be able to find values that of type string in a B-tree.
The last part of that code shows the recursion on both t1 and t2, utilising the orElse call to combine both the trees and be able to search both of them recursively.

```
case ThreeNode(t1, u1, t2, u2, t3) =>
  if (u1 equiv v) {
    Some(v)
  } else if (u2 equiv v) {
    Some(v)
  } else {
    t1.find(v) orElse t2.find(v) orElse t3.find(v)
  }
```

the case of ThreeNode, uses the same concepts as TwoNode, except it has an extra branch to recursively search through as well. This means that we need to check both our values at u1 and u2 for the answer, if not we may traverse through the tree without ever finding the value. The t3 tree is simply added onto the orElse call so we can recursively search through all three branches of u1 and u2.

Pretty Print Method

The pretty print method was applied to take the B-tree as a string and print each node of the tree as a new line with indentations relative to its position in the tree. The method takes a parameter value of 'tab' which is of type Int, and returns a String of the B-tree that has been called. The Match and case creates three possible cases, the same as the find method. An EmptyNode, a TwoNode, and a ThreeNode case.

```
case EmptyNode() => {
  val s = " "
  (s*tab) + "EmptyNode()"
}
```

Starting with the EmptyNode case, we first declare a value s, which is simply one space. This will be done for each case. The (s*tab) call, takes however many tab spaces we need to indent before printing "EmptyNode()". The reason we do not have a recursive call here to add more 'tab _width', is because we want this string to be in the same indentation as the entry that is between it.

```
case TwoNode(t1, u1, t2) => {
  val s = " "
  (s*tab) + "TwoNode(" + "\n" +
  t1.pprint(tab + tab_width) + "," + "\n" +
  (s*(tab + tab_width)) + "Entry: " + u1 + "," + "\n" +
  t2.pprint(tab + tab_width) + ")"
}
```

Above is the case TwoNode, where we begin printing the actual values that are in the tree. We first print how many nodes this part of the tree has, in this case it is “TwoNode(“. We leave the bracket open since we will close it when we have finished this section of the tree. This is recursively done, adding tab_width to each line (tab_width = 4 spaces). Each time we have a value, the tab spaces plus the extra tab_width are added and the value are printed as an “Entry”. Finally this tree is closed once all values are printed with it and the last EmptyNode has been found.

```
case ThreeNode(t1, u1, t2, u2, t3) => {
  val s = " "
  (s*tab) + "ThreeNode(" + "\n" +
  t1.pprint(tab + tab_width) + "," + "\n" +
  (s*(tab + tab_width)) + "Entry: " + u1 + "," + "\n" +
  t2.pprint(tab + tab_width) + "," + "\n" +
  (s*(tab + tab_width)) + "Entry: " + u2 + "," + "\n" +
  t3.pprint(tab + tab_width) + ")"
}
```

The case for ThreeNode looks very similar except for the extra code which prints u2 as well as u1, and traverses through t2, whilst using t3 to go and find the final node to close the whole string.

In Order Method

The inOrder method converts the given B-tree into an ordered list by applying a depth first, in order traversal. The method does not take any parameters and returns a list of type T. Another match and case recursive method that has three cases; EmptyNode, TwoNode and ThreeNode.

The EmptyNode case simply returns Nil if there is nothing in the B-tree.

```
case TwoNode(t1, u1, t2) => {
  t1.inOrder() ::: u1 :: t2.inOrder()
}
```

The case for TwoNode is only one line, but does do quite a lot in that one line. It starts by calling the inOrder function recursively on the t1 tree. This traverses the tree as far left as it can possibly go before it becomes an EmptyNode. This is the first step of a depth first traversal, before beginning to go down the right branches. The key part of this code is the ':::'. This piece of code is instrumental in creating the list of in order values. The ':::', helps concatenate the list that is currently being built. So u1 is added to the list, and then the '::' code places the next value in the list, which will be whatever is found going down the t2 tree. Once we have this list it will be merged with a new list starting with value u1, every time it calls the method again thanks to the ':::'.

```
case ThreeNode(t1, u1, t2, u2, t3) => {
  t1.inOrder() ::: u1 :: t2.inOrder() ::: u2 :: t3.inOrder()
}
```

The ThreeNode case concatenates the second value of a ThreeNode tree, and then traversing down the t3 tree as well. This code concatenates three lists into one.

Path Plan Map Image

The final code that was required to be completed was to be able to map and image in the class pathPlan in the module pathPlan.scala. The class pathPlan takes in the information that has been created in AStar.scala. This is the basis for the path, the obstacles, the start and finish of the path, and finally a collection of all visited cells whilst calculating the optimal path.

```
lazy val cells: Seq[Seq[Image]] = {
  for {
    y <- 1 to 24
    row = for {
      x <- 1 to 24
      image = {
```

The first thing to do to start creating the image is creating each of the cells. This creates a value called cell, and is a sequence that is 2-dimensional list. It is a nested loop and it creates the row and columns for the image.

```
if(map.obstacles.member(x, y)) {
  rectangle(cellSize,cellSize).lineWidth(cellEdgeWidth).fillColor(obstacleColour)
```

The image contains multiple if statements to fill the cell with the correct colour based on its coordinates. The code above is only showcasing the creation of the obstacles in the image. thee cell size, edge width, and colour have already been determined in the framework. The code ‘member’ checks if any of the locations in the map.obstacles cells are the same coordinates that are currently x and y in this loop. I’ve left out the rest of the code for all the other cells, since it would all look relatively similar and do the same thing.

```
    } yield image
  } yield row
```

The final part of the loop is to yield the image and the row. The yield call stores the image in the row loop, whilst the yield row is what is finally stored in the sequence.

The Sequence of images need to be collapsed down into one single image, which can be done using two new methods stored in the object pathPlan, one called ‘collapseToRowImage’ and the other ‘collapseToMapImage’.

```
def collapseToMapImage(images : Seq[Seq[Image]]) : Image = images.toList match {
  case Nil => Image.empty
  case(x :: xs) => collapseToRowImage(x) below collapseToMapImage(xs)
}
```

Looking at the collapseToMapImage first, it takes a sequence of sequences and returns an Image. Images needs to be converted to a list so that we can traverse the list of images. The case where there is a head and a tail or (x::xs) calls the other method, collapseToRowImage and places it below the another row. This recursion allows for a row to be created with a sequence, and then once it is completed, create another row and so on.

```
def collapseToRowImage(images : Seq[Image]) : Image = images.toList match {
  case Nil => Image.empty
  case (x :: xs) => (x beside collapseToRowImage(xs))
}
```

When the collapseToRowImage is called, the above code is what is used to run for Seq[x]. It does the same thing as the previous method my converting the sequence of images into a list, and then on case (x::xs), it puts a single cell, which is x in this case beside another single cell, until the row is complete. This is constantly recalled until the max amount of rows are reached in the collapsetoMapImage method.

```
lazy val grid = collapseToMapImage(cells)

lazy val mapImage : Image = grid
```

The grid value is where we store this image of the path plan, and then it is called in mapImage where it is printed when we run the whole project in the console of sbt.

Problems

Unfortunately the correct path cells are not showing with the correct colour in them at the end of this. Everything else in the image has appeared except for the gold path.

```
} else if(pathSet.member(x, y)) {
  rectangle(cellSize,cellSize).lineWidth(cellEdgeWidth).fillColor(pathColour)
```

This is currently the code for the pathSet. I have tried a few other variations but nothing has been able to work. At the moment it is currently only displaying what the terminus location is, and doesn't seem to be recursively working from the tail to the start when toPath is called with it.

Testing

The aim of my testing was to check that different variations of trees could be passed through the methods that have been implemented. I wanted to check both integer values and strings in the tree, and make sure they both give the correct output.

Starting with the find method testing, I started with looking at two different trees both of which are small in size. These trees contained a value that I gave the find method to find and return as Some(v). This was done for both string and Int trees. I also tested the B-trees provided in the framework under the object EgBTrees, with both t1 and t2 tested both returning successful results. I finally tested searching a tree where the value the method was asked to find, was not in the tree at all. This correctly returned 'None'.

The inOrder method first tested whether it would return a B-tree with three nodes in it on the correct order and the same for the two node trees, both of which were executed correctly with a tree of type value int. I also checked to see if it would ignore duplicates when ordering a tree, which it does correctly. It correctly orders a tree of strings in alphabetical order. I also used the EgBTrees as test subjects as well.

The pretty print method testing was mainly focused on making sure certain values appear in the string that are found in the string. I started with a B-tree and when converted it to a tree, I made sure it started with the correct value, which was “TwoNode(“ in this case. I also chose random values in the tree and checked that they were included in the string. I also made sure the ending was correct. I did this same process for the EgBTrees example, utilising t2 so we can also test a string.