Alec Goebel
Artificial Intelligence
10/30/07

Data Structures:

Nodes: The design of the node is based upon struct's. When the defnode macro is run, a creation function, a type function, an equals function, and accessors/mutators are all created. For example: (defnode 'tree ('leaf 'height)) would create:

(make-tree symbol num)
(tree? tree)
(tree-eqv? tree1 tree2)
(tree.leaf tree)
(tree.height tree)
(set!tree.leaf tree symbol)
(set!tree.height tree num)

Though very little was added and define struct could have been used, this allowed a greater amount of control over the functions generated and will allow me to modify them as needed. The actual format of the node is a simple vector with the first value being the name of the node.

Posn: Posn's are just like nodes except predefined. Originally added as a test to make sure the node code expanded properly, due to various issues it became necessary to leave the expanded form defined.

Grid: A grid is a triple nested vector. The highest form is a simple vector storing 'grid (for type) width, height, and the main data for the grid. The data is a two dimensional vector, with each value being a cell (node.) The name and properties of cells are defined as nodes and then all values are set for each point on the grid.

Path: A path is simply a list of Posn's. These can be seen more as pointers to the cells themselves as opposed to copies of the cells.

Algorithms:

All the search algorithms begin by assuring that the start and end points are valid cells on the grid (meaning they are both on the grid and not blocked.) Then, the list is initialized with the partial path of just the root node. Then the search loop begins.

The first step of the loop is simply checking if the search has been completed, successfully or not. If the current path's end node is not the goal, the node is expanded. Expansion takes place by adding the directions (posn's) in the order they were given to the existing path, shifting the current node. Before adding the path, all paths with duplicate nodes are removed, as well as paths that go off the grid or into a blocked space.

The algorithm determines where the new items are added. In the case of depth first, the new partial paths are added to the top of the path list. In breadth first they are added at the end.

In the case of best-first search, the new partial paths are added and then all the paths are sorted from least expensive up. From there, the loop repeats.

The given order of directions for all algorithms was Clockwise from North.

Depth First:

```
Loop through Path list (initially root)
        If leaf of current path is not goal
                Expand leaf
                Remove illegal leaves
                Add new paths to start of path list
                Loop
```

Breadth First:

```
Loop through Path list (initially root)
        If leaf of current path is not goal
                Expand leaf
                Remove illegal leaves
                Add new paths to end of path list
                Loop
```

Best First:

```
Loop through Path list (initially root)
        If leaf of current path is not goal
                Expand leaf
                Remove illegal leaves
                Add new paths to start of path list
                Sort path list according to cost of each partial path
                Loop
```

Test 1a

```
X - - - E
- X * * -
- * * * -
- * * X -
S - - - X
```

Depth First:

```
X + + + +
+ X * * -
+ * * * -
+ * * X -
+ - - - X
```

X = Blocked
* = Toll
+ = Path

Final Path:(0,4)->(0,3)->(0,2)->(0,1)->(1,0)->(2,0)->(3,0)->(4,0)
Path Cost: 7
Number of Paths Checked: 8
Number of Paths Currently Stored: 15
Number of Paths Stored Total: 21
cpu time: 16 real time: 15 gc time: 0

As expected, the depth first progressed as far as it could in the first direction provided and then continued with the next best, resulting in a straight shot north, a jog northeast, and a straight shot east. The algorithm also has very few paths stored since it continually expands upon the current path it was working on. It should be noted that although this is the cheapest path, this is a pure coincidence. Also, since in this example it did not need to double back, it took little to no time.

Breadth First:

```
X - - - +
- X * + -
- * + * -
- + * X -
+ - - - X
```

Final Path:(0,4)->(1,3)->(2,2)->(3,1)->(4,0)
Path Cost: 14
Number of Paths Checked: 188
Number of Paths Currently Stored: 537
Number of Paths Stored Total: 723
cpu time: 312 real time: 344 gc time: 0

Breadth first found the fewest nodes traversed, although not the most cost effective path. It also checked significantly more and had many more in memory. This was by far the least effective of the searches in Test 1a, primarily due to the fact that, without "moving backwards" the goal is the furthest it can be from the start which means the space and time it takes to run a breadth first is significant.

Best First:

```
X + + + +
+ X * * -
+ * * * -
+ * * X -
+ - - - X
```

```
Final Path:(0,4)->(0,3)->(0,2)->(0,1)->(1,0)->(2,0)->(3,0)->(4,0)
Path Cost: 7
Number of Paths Checked: 24
Number of Paths Currently Stored: 43
Number of Paths Stored Total: 65
cpu time: 109 real time: 109 gc time: 0
```

Best first took a significant amount of time and resources compared to the simple depth first search for the same results. This is of course a coincidence. The number of paths stored is almost twice what was checked since every time one of the simple paths grew longer the algorithm would switch. For example, at times it would toggle between two equal partial paths, first expanding one then the other.

Test 1b

```
X - - - -
- X * * -
- * E * -
- * * X -
S - - - X
```

Depth First:

```
X + + + +
+ X * * +
+ * + * +
+ * + X +
+ - + + X
```

Final Path:(0,4)->(0,3)->(0,2)->(0,1)->(1,0)->(2,0)->(3,0)->(4,0)->(4,1)->(4,2)->(4,3)->(3,4)->(2,4)->(2,3)->(2,2)
Path Cost: 173/5
Number of Paths Checked: 15
Number of Paths Currently Stored: 28
Number of Paths Stored Total: 41
cpu time: 31 real time: 47 gc time: 0

Yet again depth first search did not take up a lot of time or resources, yet in this case the path is far less likely to be effective, and one of the more expensive costs. The method the algorithm uses and it's bias to move counterclockwise due to the initial direction inputs are made very obvious in this example.

Breadth First:

```
X - - - -
- X * * -
- * + * -
- + * X -
+ - - - X
```

Final Path:(0,4)->(1,3)->(2,2)
Path Cost: 113/5
Number of Paths Checked: 10
Number of Paths Currently Stored: 34
Number of Paths Stored Total: 42
cpu time: 31 real time: 32 gc time: 0

In this case, the search again found the path with the fewest nodes. It ran significantly faster this time, mostly due to the fact that the goal was 2 steps deep instead of 4, shortening the number of places needed to check. In very small and simple searches, breadth is possibly the best due to its simple nature and accurate results. But, if the scale of the grid grows even slightly, it's usefulness rapidly decreases due to memory and time allocation.

Best First:

```
X - - - -
- X * * -
- * + * -
- + * X -
+ - - - X
```

Final Path:(0,4)->(1,3)->(2,2)
Path Cost: 63/5
Number of Paths Checked: 85
Number of Paths Currently Stored: 161
Number of Paths Stored Total: 244
cpu time: 1484 real time: 1625 gc time: 0

Although it did manage to find the correct and cheapest path, it took a great deal of time and memory comparatively. This is simply because the search actively avoided costly paths, meaning it took at least ten steps around the outside of the grid before it became cheaper to just go up a hill into a toll zone. That being said, it is certainly one of the more useful results.

Test 2

```
E X * -
X X * -
* * * -
- * * S
```

This test was designed so that the search algorithm would be able to bash its head against a simple wall and never reach the goal. It also tried to add a bias due to height and toll, forcing it to move in a predictable direction, at least at first.

Upon running the search, several interesting things occurred. First, there appeared to be a pattern in the cost where every so often it would cycle up through a sequence of numbers. Also, despite being informed, it still required the search to generate and test every possible path before failing. Lastly, although the paths never doubled back, there were many obvious redundancies (such as weaving back and forth starting in opposite directions.)

Test 3

```
- X - - - - -
X - - - - - -
- - - - X X X
- - - X - - -
- - - X - - -
- - - X - - -
```

```
+ X + + - - -
X + + + - - -
- - - + X X X
- - - X + - -
- - - X - - -
- - - X - - -
```

```
Final Path:(4,3)->(3,2)->(3,1)->(3,0)->(2,1)->(2,0)->(1,1)->(0,0)
Path Cost: 7
Number of Paths Checked: 58208
Number of Paths Currently Stored: 14
Number of Paths Stored Total: 58220
cpu time: -183481 real time: 270969 gc time: 134718
```

This test was designed to point out how even requiring only 5 moves, if the moves were at the end of the possibilities checked, it would take a very long time. Were the order of the directions simply reversed, this same map would not take nearly as much time.

Although the final path was not overly excessive, it took a lot of tries to find the right one. The last direction checked, northwest, took a very long time to get to twice. It should be noted however, that although it took a long time, the number of stored paths did not dramatically increase.