Project: Convoluted Kernel Maze (in Python)

Author: Pat Wrinn

Description:

This program builds a maze based on a grid structure and populates randomly selected cells with treasure or "swag" items. The x and y dimensions of the maze grid and a list of swag items are provided by the user (or a default swag list is used). Using an A* algorithm with a Euclidean heuristic, the shortest route between randomly-assigned start and end points is mapped. As the path is traced, swag items encountered are collected in a list. Once the path is finished, the swag list is sorted using radix sort. The unsorted and sorted swag lists, a tabulated swag list with item counts, and the maze with shortest path are outputted to the console.

The program can be executed by typing "python maze.py" in the command line.

Discussion - maze solving:

For this program, the maze is initially set to be a 2D grid consisting entirely of walls, except for one cell which is the starting point. The mow function creates paths by removing walls: a random cardinal direction is picked to explore, and if the grid location 2 moves in that direction is within the grid and is a wall, that cell is set to empty as is the cell 1 move in that direction. This procedure continues until there are no further directions to explore within the grid - the grid is now a recognizable maze with walls and paths. If one moved more than 2 cells at a time, path creation would be more restricted because one would encounter grid boundaries more frequently, so the maze would be mostly walls with fewer, more isolated paths. If the distances moved varied, the maze might have some "rooms" or isolated columns within it rather than simply one-cell-wide corridors. Of course, if one moved one cell at a time, all walls would eventually be removed resulting in an empty field.

Two possible methods for tracing the shortest path through the maze are Dijkstra's algorithm and the A* algorithm. Dijkstra's algorithm, a variant of Breadth-First Search, finds the shortest distance from a source vertex to all other vertices by traversing a graph, keeping track of the minimum sum of weights along each traversed path as it builds a solution. The A* algorithm finds the minimum cost from a source vertex to a specific end vertex. It represents an optimization of Dijkstra's by incorporating a heuristic, which is the estimated "crow flies" distance from the current vertex to an end vertex. With A*, only the lowest-cost path that includes the heuristic, taking into account the end goal, is pursued.

Since the maze in this program has randomly-assigned, but predetermined start and end points, the better algorithm to employ for finding the shortest path is A*. The Euclidean heuristic is used because diagonal movement is permitted in this program, thus rendering the Manhattan heuristic inadmissible. Dijkstra's algorithm would be preferable if, for example, the position of end vertex was unknown or there were different "values" to different end points, for it would check every shortest possible route.

Discussion - sorting:

In this program, as swag items are collected, they are stored in a Python list data structure. This works because the swag items are simply values, there are relatively few of them, and they are not sub-categorized (i.e., "food swag", "cheap swag", "rare swag", etc.). The time/space complexity for querying the swag list is O(n). If there were many more swag items or subcategories of swag items, querying a list would be inefficient and/or inadequate and it would be better to store swag in a Python dictionary data structure, for which the complexity of querying hash keys (swag names) would be O(1).

The list of swag is initially in chronological order, built up as swag is encountered along the path to the end. It is then sorted alphabetically using radix sort, with Unicode digits substituted for letters (i.e., 26 + 1 (empty space) buckets). The time complexity of radix sort is O(n*k), with k representing the number of buckets. Another suitable sorting algorithm would be quick sort, which has an average complexity of O(n*log(n)) if a good random pivot selection is used. Merge sort and heap sort, with worst case complexities of O(n*log(n)) that are better than the worst case of quick sort (O(n^2)), could nonetheless be considered less preferable because of the resources required to maintain the heap and execute the merge operation.