Computer Science Basics: Algorithms

Behnam Nikkhah

Contents

- Maze movement
  - Moving more than two cells
  - Random movement number
- Pathfinding algorithms
- Which pathfinding algorithm is best?
- Swag
  - Storing
  - Sorting
  - Quantity & Variety

Maze Movement

MOVING MORE THAN TWO CELLS

When moving a larger number of cells, we can clearly see that the maze has a lot more *walls* than *empty* areas (implicitly making the maze easier to solve).

This is due to the fact that we are moving a larger amount of cells without removing a *wall*.

If we had only explored only one move at a time, then every *wall* cell type will be removed; hence rendering the maze completely wall free.

MOVING MORE THAN TWO CELLS (CONT.)

If the cell movement was not constant, there are four possible scenarios:

- The maze will consist of **all walls** (only an endpoint)
  - Only an endpoint is generated and the maze is already solved
- The maze will consist of **no walls**
  - Any direction is plausible to solve the maze
- The maze will have **more walls** than empty spaces
  - A simpler maze is generated with less complexity to solve
- The maze will have **more empty spaces** than walls
  - A more difficult maze is generated with greater complexity to solve

Pathfinding Algorithms

Pathfinding Algorithms

Two possible pathfinding algorithms that can be used to find a path through the maze:

- Dijkstra’s algorithm
- A* algorithm

We’ll discuss these two algorithms in the proceeding slides.

Dijkstra's Algorithm

Dijkstra’s Algorithm is used to find the shortest path(s) in a graph. The algorithm is a special case of the Breadth-first Search (BFS) algorithm on unweighted graphs, whereupon the priority queue debases into a First-in First-out queue (FIFO).

Dijkstra’s algorithms exhausts all possible vertices before moving on to the next node. The algorithm is optimal for finding all possible shortest routes without knowing an endpoint or end vertex.

A* Algorithm

A* algorithm is a slight variant to Dijkstra's algorithm, which is most popular in artificial intelligence (AI). Rather than checking all neighboring vertices, the algorithm optimizes Dijkstra's algorithm by looking for the shortest distance from a *single start vertex* to a *single end vertex*.

A* algorithm uses special approximate distances called *heuristics* to estimate the distance from one vertex to the next. The algorithm makes a local optimal choice at every vertex (greedy approach).

Which one do we choose?

Since a maze always has a fixed start vertex and a fixed end vertex, the best pathfinding algorithm to choose is the *A* algorithm*.

We are choosing the A* algorithm because we know our start vertex and we know our end vertex. We would only use Dijkstra’s algorithm if we did not know where the exit is; therefore, Dijkstra’s algorithm would check for every possible short route. But since we know the start and end vertex, we would use the A* algorithm to achieve the shortest route.

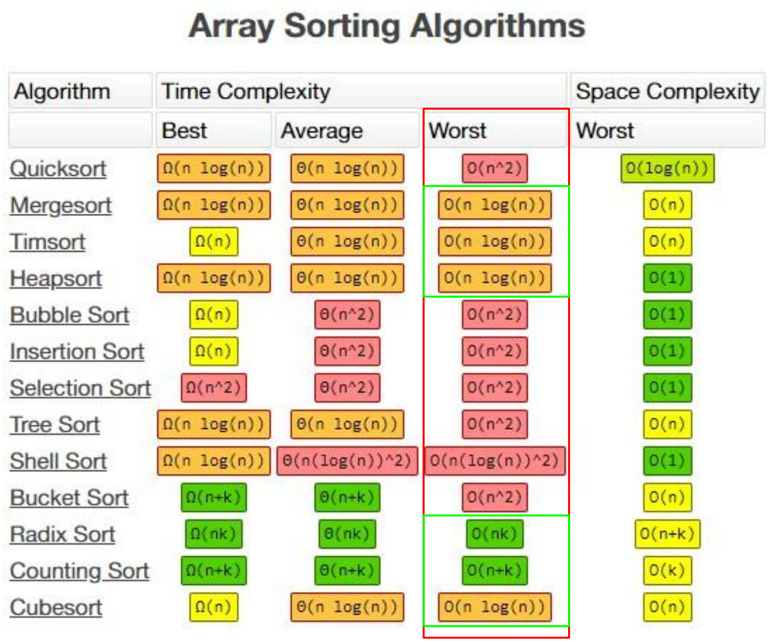
Swag

Storing Swags

We can use a Python dictionary (a.k.a Hash Map) to store the swag items. The dictionary will map the swag item as the *key* and the amount picked up to be the *value*.

For example, assume we have a swag item called *pumpkin* and we picked up *n* amount of pumpkins. Once the mapping is done, we can simply retrieve the *pumpkin* from the dictionary which only costs ***O(1)*** of time complexity.

Sorting Swags



Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

More information at http://bigocheatsheet.com/

As you can see on the diagram to your left, there are lots of sorting algorithms to choose from.

However, we would like a sorting algorithm (in its worst case) with a time complexity of **O(n log(n))** or faster.

Therefore, common sorts such as **Merge Sort**, **Heap Sort**, or **Radix Sort** are good candidates to use if we had to sort a large amount of swag items.

Quantity & Variety

Lots of Quantity + limited Variety:

Assume we have collected a lot of the same item without much variety.

Since we know that some swag items dominate in quantity than other ones and they must be sorted alphabetically (ascending order), we would expect faster sorting run-times because most of the items are already alphabetically arranged (repeated items with the same starting letter).

Quantity & Variety (Cont.)

Limited Quantity + lots of Variety:

Assume we have collected a limited amount of the same item with a lot of variety.

Since we do not know how these items are arranged and they must be sorted alphabetically (ascending order), there may be varying run-times depending on the swag items list.

Conclusion

What we've learned in this course

- Recursion makes for elegant solutions (even if it can be done in a standard loop)
- There are many sorting algorithms, but with different time complexities
- Pathfinding algorithms such as Dijkstra's algorithm and A* algorithm have one common goal: to find the shortest path in a graph
- There are many ways to solve a problem, but the time complexity and efficiency must always be accounted for (especially for larger, more complex projects)