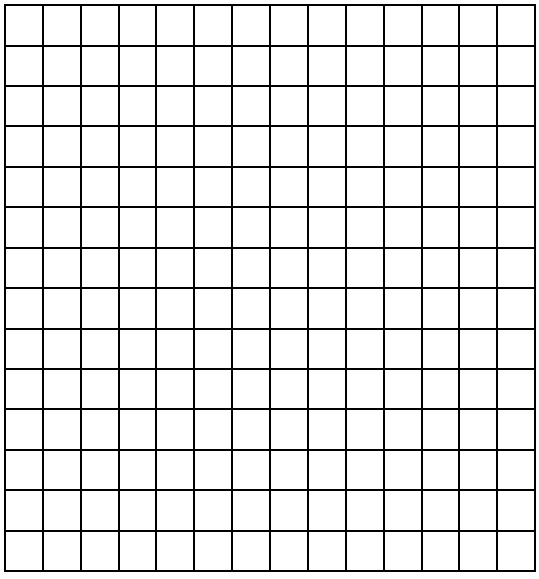
I. Search

AI Search for Autonomous Vehicle Navigation

You work as an engineer for the entrepreneur Melon Husk in his business that makes electric semi-autonomous cars. The eventual goal of the business is to create an entirely autonomous vehicle that is able to drive itself from one place to another safely and avoid traffic jams. Your task is one part of the overall goal: to find efficient routes through cities that avoid traffic jams. Which algorithm should you use?

The prototype system will be tested in Manhattan. The streets of Manhattan are famously laid out on a grid. Suppose we have a grid with 15 streets running North-South, and 15 streets running East-West, as in the picture below. The distance between two adjacent intersections is the same throughout the grid.



The autonomous vehicle starts at an intersection A, and takes its passengers to a different intersection B. However, some other intersections are blocked by traffic jams. The task is to find a route from A to B that avoids all blocked intersections and minimises the length of the journey.

The file `grids.txt` contains five grids with different start and goal locations and traffic jams. The grids are represented as Java arrays filled with integers, where each integer represents one intersection. 0 means the intersection is clear, 1 represents a traffic jam, 2 represents the starting point, and 3 represents the goal.

This activity is to implement and evaluate an AI search algorithm. There are three main criteria for evaluating search algorithms: the quality of the solution (in this case, the length of the route), the number of search states visited by the algorithm (which is related to its efficiency), and the number of search states stored in memory as the algorithm runs (which is another aspect of its efficiency).

Implement the uninformed search algorithm **iterative deepening** for this problem. This problem is a **graph search** problem, and the graph search version of iterative deepening should be used.

To implement iterative deepening, you will first need to represent a search node. A search node should contain three integers: the *x* and *y* location, and the depth of the node in the search tree. The *x* and *y* location are used to index into the grid arrays. For example, `grid1[2][3]==2`, so the first node when searching in `grid1` should have `x=2`, `y=3` and `depth=0`.

Your implementation may be in Java, and should (assuming you're using an object-oriented language) include the following classes and methods:

A class called `SearchNode` with the following data and methods:

- The integers x, y, and depth, as described above.
- A suitable constructor.
- An equals method that compares the `SearchNode` with another object and returns true when they are of the same type and have the same state (the depth is not part of the state).
- An expand method that creates the neighbours of the `SearchNode`.
- You may wish to implement the `toString` method (or the equivalent in your chosen language) to allow SearchNodes to be printed out conveniently.

A class called `IDSearch` with the following methods:

- A method called `iterativeDeepening` that contains the main loop of the iterative deepening algorithm (i.e. the loop that calls `depthLimitedSearch` repeatedly with increasing depth limits).
- A method called `depthLimitedSearch` that implements a depth-first graph search with a depth limit.
- A `main` method to start the program (or the equivalent in your chosen language).

The `depthLimitedSearch` method will need data structures for the following:

- The frontier;
- All nodes generated within the current depth-first search (the *explored set*).

For this activity, correctness is considered to be more important than efficiency. Java ArrayLists may be used for both the frontier and the explored set without losing any marks. Other suitable data structures may also be used.

It may be helpful for your program to output the expanded node at each step, and the frontier directly following the expansion.

Suppose your program has completed the searches with depth limits of 0, 1, and 2, and has just started the search with a depth limit of 3. The following is an example output for the first few steps of search with depth limit 3.

In the following, each node is printed out as (x,y,depth). Nodes are added and removed at the end of the data structure representing the frontier:

```
Starting depth limited search with depth limit 3

Initial frontier: [(2,2,0)]

Expanded node: (2,2,0). New frontier: [(1,2,1),(2,1,1),(3,2,1),(2,3,1)]

Expanded node: (2,3,1). New frontier:
[(1,2,1),(2,1,1),(3,2,1),(1,3,2),(3,3,2),(2,4,2)]

……
```

The output should also clearly show when the search has reached its goal.

A depth-limited depth-first graph search is somewhat tricky to implement. When a node is expanded and a new node (x,y,d) is created, the algorithm should then check the explored set to see if location x,y has been visited already. However, it may have been visited at a higher depth (i.e. a longer path to get to x,y) – in which case the algorithm will need to visit it again (can you see why?). Therefore, when checking the explored set, check for the existence of nodes (x,y,d_2) where d_2 is less than or equal to d. If there are any such nodes (x,y,d_2), then (x,y,d) should not be added to the frontier.

Try out your iterative deepening search on the grids provided in `grids.txt`. How does the algorithm perform? It should be able to find an optimal-length path, avoiding all the roadblocks, in a short time. How does it compare to a tree-search version (without the explored set)?