Maze Solving Algorithms

There are a number of ways of solving Mazes, each with its own characteristics. Here's a list of specific algorithms:

- **Wall follower:** This is a simple Maze solving algorithm. It focuses on you, is always very fast, and uses no extra memory. Start following passages, and whenever you reach a junction always turn right (or left). Equivalent to a human solving a Maze by putting their hand on the right (or left) wall and leaving it there as they walk through. If you like you can mark what cells you've visited, and what cells you've visited twice, where at the end you can retrace the solution by following those cells visited once. This method won't necessarily find the shortest solution, and it doesn't work at all when the goal is in the center of the Maze and there's a closed circuit surrounding it, as you'll go around the center and eventually find yourself back at the beginning. Wall following can be done in a deterministic way in a 3D Maze by projecting the 3D passages onto the 2D plane, e.g. by pretending up passages actually lead northwest and down lead southeast, and then applying normal wall following rules.
- **Pledge algorithm:** This is a modified version of wall following that's able to jump between islands, to solve Mazes wall following can't. It's a guaranteed way to reach an exit on the outer edge of any 2D Maze from any point in the middle, however it's not able to do the reverse, i.e. find a solution within the Maze. It's great for implementation by a Maze escaping robot, since it can get out of any Maze without having to mark or remember the path in any way. Start by picking a direction, and always move in that direction when possible. When a wall is hit, start wall following until your chosen direction is available again. Note you should start wall following upon the far wall that's hit, where if the passage turns a corner there, it can cause you to turn around in the middle of a passage and go back the way you came. When wall following, count the number of turns you make, e.g. a left turn is -1 and a right turn is 1. Only stop wall following and take your chosen direction when the total number of turns you've made is 0, i.e. if you've turned around 360 degrees or more, keep wall following until you untwist yourself. The counting ensures you're eventually able to reach the far side of the island you're currently on, and jump to the next island in your chosen direction, where you'll keep on island hopping in that direction until you hit the boundary wall, at which point wall following takes you to the exit. Note Pledge algorithm may make you visit a passage or the start more than once, although subsequent times will always be with different turn totals. Without marking your path, the only way to know whether the Maze is unsolvable is if your turn total keeps increasing, although the turn total can get to large numbers in solvable Mazes in a spiral passage.
- **Chain algorithm:** The Chain algorithm solves the Maze by effectively treating it as a number of smaller Mazes, like links in a chain, and solving them in sequence. You have to specify the start and desired end locations, and the algorithm will always find a path from start to end if one exists, where the solution tends to be a reasonably short if not the shortest solution. That means this can't solve Mazes where you don't know exactly where the end is. This is most similar to Pledge algorithm since it's also essentially a wall follower with a way to jump between islands. Start by drawing a

straight line (or at least a line that doesn't double back on itself) from start to end, letting it cross walls if needed. Then just follow the line from start to end. If you bump into a wall, you can't go through it, so you have to go around. Send two wall following "robots" in both directions along the wall you hit. If a robot runs into the guiding line again, and at a point which is closer to the exit, then stop, and follow that wall yourself until you get there too. Keep following the line and repeating the process until the end is reached. If both robots return to their original locations and directions, then farther points along the line are inaccessible, and the Maze is unsolvable.

- **Recursive backtracker:** This will find a solution, but it won't necessarily find the shortest solution. It focuses on you, is fast for all types of Mazes, and uses stack space up to the size of the Maze. Very simple: If you're at a wall (or an area you've already plotted), return failure, else if you're at the finish, return success, else recursively try moving in the four directions. Plot a line when you try a new direction, and erase a line when you return failure, and a single solution will be marked out when you hit success. When backtracking, it's best to mark the space with a special visited value, so you don't visit it again from a different direction. In Computer Science terms this is basically a depth first search. This method will always find a solution if one exists, but it won't necessarily be the shortest solution.
- **Trémaux's algorithm:** This Maze solving method is designed to be able to be used by a human inside of the Maze. It's similar to the recursive backtracker and will find a solution for all Mazes: As you walk down a passage, draw a line behind you to mark your path. When you hit a dead end, turn around and go back the way you came. When you encounter a junction you haven't visited before, pick a new passage at random. If you're walking down a new passage and encounter a junction you have visited before, treat it like a dead end and go back the way you came. (That last step is the key which prevents you from going around in circles or missing passages in braid Mazes.) If walking down a passage you have visited before (i.e. marked once) and you encounter a junction, take any new passage if one is available, otherwise take an old passage (i.e. one you've marked once). All passages will either be empty, meaning you haven't visited it yet, marked once, meaning you've gone down it exactly once, or marked twice, meaning you've gone down it and were forced to backtrack in the opposite direction. When you finally reach the solution, paths marked exactly once will indicate a direct way back to the start. If the Maze has no solution, you'll find yourself back at the start with all passages marked twice.
- **Dead end filler:** This is a simple Maze solving algorithm. It focuses on the Maze, is always very fast, and uses no extra memory. Just scan the Maze, and fill in each dead end, filling in the passage backwards from the block until you reach a junction. That includes filling in passages that become parts of dead ends once other dead ends are removed. At the end only the solution will remain, or solutions if there are more than one. This will always find the one unique solution for perfect Mazes, but won't do much in heavily braid Mazes, and in fact won't do anything useful at all for those Mazes without dead ends.
- **Cul-de-sac filler:** This method finds and fills in cul-de-sacs or nooses, i.e. constructs in a Maze consisting of a blind alley stem that has a single loop at the end. Like the dead end filler, it focuses on the Maze, is always fast, and uses no extra memory. Scan the Maze, and for each noose junction (a noose junction being one where two

of the passages leading from it connect with each other with no other junctions along the way) add a wall to convert the entire noose to a long dead end. Afterwards run the dead end filler. Mazes can have nooses hanging off other constructs that will become nooses once the first one is removed, so the whole process can be repeated until nothing happens during a scan. This doesn't do much in complicated heavily braid Mazes, but will be able to invalidate more than just the dead end filler.

- **Blind alley filler:** This method finds all possible solutions, regardless of how long or short they may be. It does so by filling in all blind alleys, where a blind alley is a passage where if you walk down it in one direction, you will have to backtrack through that passage in the other direction in order to reach the goal. All dead ends are blind alleys, and all nooses as described in the cul-de-sac filler are as well, along with any sized section of passages connected to the rest of the Maze by only a single stem. This algorithm focuses on the Maze, uses no extra memory, but unfortunately is rather slow. For each junction, send a wall following robot down each passage from it, and see if the robot sent down a path comes back from the same path (as opposed to returning from a different direction, or it exiting the Maze). If it does, then that passage and everything down it can't be on any solution path, so seal that passage off and fill in everything behind it. This algorithm will fill in everything the cul-de-sac filler will and then some, however the collision solver will fill in everything this algorithm will and then some.
- **Blind alley sealer:** This is like the blind alley filler, in that it also finds all possible solutions by removing blind alleys from the Maze. However this just fills in the stem passage of each blind alley, and doesn't touch any collection of passages at the end of it. As a result this will create inaccessible passage sections for cul-de-sacs or any blind alley more complicated than a dead end. This algorithm focuses on the Maze, runs much faster than the blind alley filler, although it requires extra memory. Assign each connected section of walls to a unique set. To do this, for each wall section not already in a set, flood across the top of the walls at that point, and assign all reachable walls to a new set. After all walls are in sets, then for each passage section, if the walls on either side of it are in the same set, then seal off that passage. Such a passage must be a blind alley, since the walls on either side of it link up with each other, forming a pen. Note a similar technique can be used to help solve hypermazes, by sealing off space between branches that connect with each other.
- **Shortest path finder:** As the name indicates, this algorithm finds the shortest solution, picking one if there are multiple shortest solutions. It focuses on you multiple times, is fast for all types of Mazes, and requires quite a bit of extra memory proportional to the size of the Maze. Like the collision solver, this basically floods the Maze with "water", such that all distances from the start are filled in at the same time (a breadth first search in Computer Science terms) however each "drop" or pixel remembers which pixel it was filled in by. Once the solution is hit by a "drop", trace backwards from it to the beginning and that's a shortest path. This algorithm works well given any input, because unlike most of the others, this doesn't require the Maze to have any one pixel wide passages that can be followed. Note this is basically the A* path finding algorithm without a heuristic so all movement is given equal weight.
- **Shortest paths finder:** This is very similar to the shortest path finder, except this finds all shortest solutions. Like the shortest path finder, this focuses on you multiple

times, is fast for all types of Mazes, requires extra memory proportional to the size of the Maze, and works well given any input since it doesn't require the Maze to have any one pixel wide passages that can be followed. Also like the shortest path finder, this does a breadth first search flooding the Maze with "water" such that all distances from the start are filled in at the same time, except here each pixel remembers how far it is from the beginning. Once the end is reached, do another breadth first search starting from the end, however only allow pixels to be included which are one distance unit less than the current pixel. The included pixels precisely mark all the shortest solutions, as blind alleys and non-shortest paths will jump in pixel distances or have them increase.

- **Collision solver:** Also called the "amoeba" solver, this method will find all shortest solutions. It focuses on you multiple times, is fast for all types of Mazes, and requires at least one copy of the Maze in memory in addition to using memory up to the size of the Maze. It basically floods the Maze with "water", such that all distances from the start are filled in at the same time (a breadth first search in Computer Science terms) and whenever two "columns of water" approach a passage from both ends (indicating a loop) add a wall to the original Maze where they collide. Once all parts of the Maze have been "flooded", fill in all the new dead ends, which can't be on the shortest path, and repeat the process until no more collisions happen. (Picture amoebas surfing at the crest of each "wave" as it flows down the passages, where when waves collide, the amoebas head-butt and get knocked out, and form there a new wall of unconscious amoebas, hence the name.) Ultimately this is the same as the shortest paths finder, except this is more memory efficient (since it only needs to keep track of the coordinates of the front of each column of water) and a bit slower (since it potentially needs to be run multiple times to remove everything).
- **Random mouse:** For contrast, here's an inefficient Maze solving method, which is basically to move randomly, i.e. move in one direction and follow that passage through any turnings until you reach the next junction. Don't do any 180 degree turns unless you have to. This simulates a human randomly roaming the Maze without any memory of where they've been. It's slow and isn't guaranteed to ever terminate or solve the Maze, and once the end is reached it will be just as hard to retrace your steps, but it's definitely simple and doesn't require any extra memory to implement.

Sacado de :http://www.astrolog.org/labyrnth/algrithm.htm

PSEUDOCÓDIGO

```
if (noParedIzqierda)
        cruce 90 grados izquierda
elseif (noParedEnfrente)
        avanzar un espacio
else
        cruce 90 grados derecha
```

¿El porqué del algoritmo?
Elegimos el algoritmo de la mano derecha porque es uno de los algoritmos que salen del laberinto sin importar cuan difícil sea. El algoritmo es fácil de implementar, ya que no tiene tanta complicación a la hora de la programación. La desventaja de este algoritmo es de que tarda mucho tiempo en salir en algunas ocasiones.