Maze Classification

Mazes in general (and hence algorithms to create Mazes) can be organized along seven different classifications. These are: Dimension, Hyperdimension, Topology, Tessellation, Routing, Texture, and Focus. A Maze can take one item from each of the classes in any combination.

Dimension: The dimension class is basically how many dimensions in space the Maze covers. Types are:

- 2D: Most Mazes, either on paper or life size, are this dimension, in which it's always possible to display the plan on the sheet of paper and navigate it without overlapping any other passages in the Maze.
- 3D: A three dimensional Maze is one with multiple levels, where (in the orthogonal case at least) passages may go up and down in addition to the four compass directions. A 3D Maze is often displayed as an array of 2D levels, with "up" and "down" staircase indicators.
- Higher dimensions: It's possible to have 4D and higher dimension Mazes. These are sometimes rendered as 3D Mazes, with special "portals" to travel through the 4th dimension, e.g. "past" and "future" portals.
- Weave: A weave Maze is basically a 2D (or more accurately a 2.5D) Maze, but where passages can overlap each other. In display it's generally obvious what's a dead end and what's a passage that goes underneath another. Life size Mazes that have bridges connecting one portion of the Maze to another are partially Weave.

Hyperdimension: The hyperdimension class refers to the dimension of the object you move through the Maze, as opposed to the dimension of the Maze environment itself. Types are:

- Non-hypermaze: Virtually all Mazes, even those in higher dimensions or with special rules, are normal non-hypermazes. In them you work with a point or small object, such as a marble or yourself, which you move from point to point, and the path behind you forms a line. There's an easily countable number of choices at each point.
- Hypermaze: A hypermaze is where the solving object is more than just a point. A standard hypermaze (or a hypermaze of the 1st order) consists of a line where as you bend and move it the path behind it forms a surface. A hypermaze can only exist in a 3D or higher dimension environment, where the entrance to a hypermaze is also a line instead of a point. A hypermaze is fundamentally different since you need to be aware of and work with multiple parts along the line at the same time, where there's nearly an infinite number of states and things you can do with the line at any time. The solving line is infinite, or the

endpoints are fixed outside of the hypermaze, to prevent one from crumpling the line into a point, which could then be treated as a non-hypermaze.
- Hyperhypermaze: Hypermazes can be of arbitrarily high dimension. A hyperhypermaze (or a hypermaze of the 2nd order) increases the dimension of the solving object again. Here the solving object is a plane, where as you move it the path behind you forms a solid. A hyperhypermaze can only exist in a 4D or higher dimension environment.

Topology: The topology class describes the geometry of the space the Maze as a whole exists in. Types are:

- Normal: This is a standard Maze in Euclidean space.
- Planair: The term "planair" refers to any Maze with an abnormal topology. This usually means connecting the edges of the Maze in interesting fashions. Examples are Mazes on the surface of a cube, Mazes on the surface of a Moebius strip, and Mazes that are equivalent to being on a torus with the left and right sides wrapping and the top and bottom wrapping.

Tessellation: The tessellation class is the geometry of the individual cells that compose the Maze. Types are:

- Orthogonal: This is a standard rectangular grid where cells have passages intersecting at right angles. In the context of tessellations, this can also be called a Gamma Maze.
- Delta: A Delta Maze is one composed of interlocking triangles, where each cell may have up to three passages connected to it.
- Sigma: A Sigma Maze is one composed of interlocking hexagons, where each cell may have up to six passages connected to it.
- Theta: Theta Mazes are composed of concentric circles of passages, where the start or finish is in the center, and the other on the outer edge. Cells usually have four possible passage connections, but may have more due to the greater number of cells in outer passage rings.
- Upsilon: Upsilon Mazes are composed of interlocking octagons and squares, where each cell may have up to eight or four possible passages connected to it.
- Zeta: A Zeta Maze is on a rectangular grid, except 45 degree angle diagonal passages between cells are allowed in addition to horizontal and vertical ones.
- Omega: The term "omega" refers to most any Maze with a consistent non-orthogonal tessellation. Delta, Sigma, Theta, and Upsilon Mazes are all of this type, as are many other arrangements one can think up, e.g. a Maze composed of pairs of right triangles.
- Crack: A crack Maze is an amorphous Maze without any consistent tessellation, but rather has walls or passages at random angles.
- Fractal: A fractal Maze is a Maze composed of smaller Mazes. A nested cell fractal Maze is a Maze with other Mazes tessellated within each cell, where the process may be repeated multiple times. An infinite recursive fractal Maze is a true fractal, where the Maze contains copies of itself, and is in effect an infinitely large Maze.

Routing: The routing class is probably the most interesting with respect to Maze generation itself. It refers to the types of passages within whatever geometry defined in the categories above.

- Perfect: A "perfect" Maze means one without any loops or closed circuits, and without any inaccessible areas. Also called a simply-connected Maze. From each point, there is exactly one path to any other point. The Maze has exactly one solution. In Computer Science terms, such a Maze can be described as a spanning tree over the set of cells or vertices.
- Braid: A "braid" Maze means one without any dead ends. Also called a purely multiply connected Maze. Such a Maze uses passages that coil around and run back into each other (hence the term "braid") and cause you to spend time

going in circles instead of bumping into dead ends. A well-designed braid Maze can be much harder than a perfect Maze of the same size.
- Unicursal: A unicursal Maze means one without any junctions. Sometimes the term Labyrinth is used to refer to constructs of this type, while "Maze" means a puzzle where choices are involved. A unicursal Maze has just one long snake-like passage that coils throughout the extent of the Maze. It's not really difficult unless you accidentally get turned around half way through and make your way back to the beginning again.
- Sparseness: A sparse Maze is one that doesn't carve passages through every cell, meaning some are left uncreated. This amounts to having inaccessible locations, making this somewhat the reverse of a braid Maze. A similar concept can be applied when adding walls, resulting in an irregular Maze with wide passages and rooms.
- Partial braid: A partial braid Maze is just a mixed Maze with both loops and dead ends in it. The word "braid" can be used quantitatively, in which a "heavily braid Maze" means one with many loops or detached walls, and a "slightly braid Maze" means one with just a few.

Texture: The texture class is subtle, and describes the style of the passages in whatever routing in whatever geometry. They're not really on/off flags as much as general themes. Here are several example variables one can look at:

- Bias: A passage biased Maze is one with straightaways that tend to go in one direction more than the others. For example, a Maze with a high horizontal bias will have long left-right passages, and only short up-down passages connecting them. A Maze is usually more difficult to navigate "against the grain".
- Run: The "run" factor of a Maze is how long straightaways tend to go before forced turnings present themselves. A Maze with a low run won't have straight passages for more than three or four cells, and will look very random. A Maze with a high run will have long passages going across a good percentage of the Maze, and will look similar to a microchip.
- Elite: The "elitism" factor of a Maze indicates the length of the solution with respect to the size of the Maze. An elitist Maze generally has a short direct solution, while a non-elitist Maze has the solution wander throughout a good portion of the Maze's area. A well designed elitist Maze can be much harder than a non-elitist one.
- Symmetric: A symmetric Maze has symmetric passages, e.g. rotationally symmetric about the middle, or reflected across the horizontal or vertical axis. A Maze may be partially or totally symmetric, and may repeat a pattern any number of times.
- Uniformity: A uniform algorithm is one that generates all possible Mazes with equal probability. A Maze can be described as having a uniform texture if it looks like a typical Maze generated by a uniform algorithm. A non-uniform algorithm may still be able to potentially generate all possible Mazes within whatever space but not with equal probability, or it may take non-uniformity further in which there exists possible Mazes that the algorithm can never generate.
- River: The "river" characteristic means that when creating the Maze, the algorithm will look for and clear out nearby cells (or walls) to the current one being created, i.e. it will flow (hence the term "river") into uncreated portions of the Maze like water. A perfect Maze with less "river" will tend to have many short dead ends, while a Maze with more river will have fewer but longer dead ends.

Focus: The focus class is obscure, but shows that Maze creation can be divided into two general types: Wall adders, and passage carvers. This is usually just an algorithmic difference when generating, as opposed to a visual difference when observing, but is still useful to consider. The same Maze can be often generated in both ways:

- Wall adders: Algorithms that focus on walls start with an empty area (or an outer boundary) and add walls. In real life, a life size Maze composed of hedges,

tarps, or wood walls, is a definite wall adder.
- Passage carvers: Algorithms that focus on passages start with a solid block and carve passages. In real life, a Maze composed of mine tunnels, or running in the inside of pipes, is a passage carver.
- Template: Mazes can of course be both passage carved and wall added, and some computer algorithms do just that. A Maze template refers to a general graphic that isn't a Maze, which is then modified to be a valid Maze in as few steps as possible, but still has the texture of the original graphic template. Complicated Maze styles like interlocking spirals are easier to do on a computer as templates, as opposed to trying to create a valid Maze while keeping it conforming to whatever style at the same time.

Other: The above is by no means a comprehensive list of all possible classes or items within each class. They're just the types of Mazes I've actually created. :-) Note most every type of Maze, including Mazes with special rules, can be expressed as a directed graph, in which you have a finite number of states and a finite number of choices at each state, which is called Maze equivalence. Here are some other classes and types of Mazes:

- Direction: This is where certain passages can only be traveled in one way. In Computer Science terms, such a Maze would be described by a directed graph, as opposed to an undirected graph like all the others.
- Segmented: This is where a Maze has different sections of its area falling in different classes.
- Infinite length Mazes: It's possible to create an infinitely long Maze (a finite number of columns by as many rows as you like) by only keeping part of the Maze in memory at a time and "scrolling" from one end to the other, discarding earlier rows while creating later rows. One way is with a modified version of the Hunt and Kill algorithm. Visualize the potentially infinitely long Maze as a long film reel, composed of individual picture frames, where just two consecutive frames are kept in memory at a time. Run the Hunt and Kill algorithm, however give bias to the top frame so it gets finished first. Once finished, it's no longer needed, so can be printed out, etc. Either way, discard it, make the partially created bottom frame be the new top frame, and clear a new bottom frame. Repeat the process until you decide to stop, at which point let Hunt And Kill finish both frames. The only limitation is the Maze can never have a path that doubles back toward the entrance for a length greater than two frames. An easier way to make an infinite Maze is with Eller's or the Sidewinder algorithms, as they already make Mazes one row at time, so simply keep letting them add rows to the Maze forever.
- Virtual fractal Mazes: A virtual Maze is one where the whole Maze isn't stored in memory at once. For example only store the 100x100 section of passages or so nearest your location, in a simulation where you walk through a large Maze. An extension of nested fractal Mazes can be used to create virtual Mazes of enormous size, such as a billion by a billion passages. Note a life size version of a billion by billion Maze (with six feet between passages) would cover the Earth's surface over 6000 times! Consider a 10^9 by 10^9 passage Maze, or a 10x10 Maze nested with 9 levels total. If we want at least a 100x100 section around us, we only need to create the 100x100 passage submaze at the lowest level, and the seven 10x10 Mazes it's nested within, to know exactly where the walls lie within a 100x100 section. (Actually it's best to have four adjacent 100x100 sections forming a square, in case you're near the edge or corner of a section, but the same concept applies.) To ensure the Maze remains consistent and never changes as you move around, have a formula to determine a random number seed for each coordinate at each nesting level. Virtual fractal Mazes are similar to a Mandelbrot set fractal, in which the pictures in a Mandelbrot exist virtually, and you just need to visit a particular coordinate at a high enough zoom level for them to reveal themselves.

Maze Creation Algorithms

Here's a list of general algorithms to create the various classes of Mazes described above:

- Perfect: Creating a standard perfect Maze usually involves "growing" the Maze while ensuring the no loops and no isolations restriction is kept. Start with the outer wall, and add a wall segment touching it at random. Keep on adding wall segments to the Maze at random, but ensure that each new segment touches an existing wall at one end, and has its other end in an unmade portion of the Maze. If you ever added a wall segment where both ends were separate from the rest of the Maze, that would create a detached wall with a loop around it, and if you ever added a segment such that both ends touch the Maze, that would create an inaccessible area. This is the wall adding method; a nearly identical way to do it is passage carved, where new passage sections are carved such that exactly one end touches an existing passage.
- Braid: To create a Maze without dead ends, basically add wall segments throughout the Maze at random, but ensure that each new segment added will not cause a dead end to be made. I make them with four steps: (1) Start with the outer wall, (2) Loop through the Maze and add single wall segments touching each wall vertex to ensure there are no open rooms or small "pole" walls in the Maze, (3) Loop over all possible wall segments in random order, adding a wall there if it wouldn't cause a dead end, (4) Either run the isolation remover utility at the end to make a legal Maze that has a solution, or be smarter in step three and make sure a wall is only added if it also wouldn't cause an isolated section.
- Unicursal: One way to create a random unicursal Maze is to take a perfect Maze, seal off the exit so there's only the one entrance, then add walls bisecting each passage. This will turn each dead end into a U-turn passageway, and there will be a unicursal passage starting and ending at the original Maze's beginning, that will follow the same path as someone wall following the original Maze. The new unicursal Maze will have twice the dimensions of the original perfect Maze it was based on. Small tricks may be done to have the start and end not always be next to each other: When creating the perfect Maze, never add segments attached to the right or bottom walls, so the resulting Maze will have an easy solution that follows that wall. Have the entrance at the upper right, and after bisecting to create the unicursal routing, remove the right and bottom wall. This will result in a unicursal Maze that starts at the upper right and ends at the lower left.
- Sparseness: Sparse Mazes are produced by choosing to not grow the Maze in areas that would violate the rule of sparseness. A consistent way to implement this is to, whenever considering a new cell to carve into, to first check all cells within a semicircle of chosen cell radius located forward in the current direction. If any of those cells is already part of the Maze, don't allow the cell being considered, since doing to would be too close to an existing cell and hence make the Maze not sparse.
- 3D: Three and higher dimensional Mazes can be created just like the standard 2D perfect Maze, except from each cell you can move randomly to six instead of four other orthogonal cells. These Mazes are generally passage carved due to the extra dimensions.
- Weave: Weave Mazes are basically done as passage carved perfect Mazes, except when carving a passage you're not always blocked by an existing passage, as you have the option to go under it and still preserve the "perfect" quality. On a monochrome bitmap, a Weave Maze can be represented with four rows per passage (two rows per passage is enough for a standard perfect Maze) where you have one row for the passage itself and the other three rows to make it unambiguous when another nearby passage goes under instead of just having a dead end near the first passage. For aesthetics you may want to look ahead before carving under an existing passage, to ensure you can continue to carve once you're completely under it, so there won't be any dead ends that terminate under a passage. Also, after carving under a passage, you may want to invert the pixels adjacent to the intersection, making it so newer passages can go over instead of always under existing ones.

- Crack: Crack Mazes are basically done as wall added perfect Mazes, except there are no distinct tessellation cells other than random pixel locations. Pick a pixel that's already set as a wall, pick another random location, and "shoot" or start drawing a wall toward the second location. However, make sure you stop just before running into any existing wall, so as not to create an isolation. Stop after you haven't been able to add any significant walls in a while. Note that random locations to draw to that may be anywhere else in the Maze, will make it so there will be several straight lines going across the Maze, and other proportionally smaller walls as you look between them, the number of walls only being limited by the pixel resolution. This makes the Maze look very much like the surface of a leaf, so this is technically a fractal Maze.
- Omega: Omega style Mazes involve defining some grid, defining how the cells link up with each other, and how to map the vertexes that surround each cell to the screen. For example, for the triangular Delta Maze with interlocking triangular cells: (1) There's a grid where each row has a number of cells that increases by two. (2) Each cell is connected to the cells adjacent to it in that row, except the third passage is linked to an appropriate cell in the row above or below based on whether it's in an odd or even column (i.e. whether the triangle is pointing up or down). (3) Each cell uses the math for a triangle to figure out where to draw it on the screen. You can draw all walls on the screen ahead of time and passage carve the Maze, or keep some modified array in memory and render the whole thing when complete.
- Hypermaze: A hypermaze in a 3D environment is similar to the reverse of a standard 3D non-hypermaze, where blocks become open spaces and vice versa. While a standard 3D Maze consists of a tree of passages through a solid area, a hypermaze consists of a tree of bars or vines through an open area. To create a hypermaze, start with solid top and bottom faces, then grow tangled vines from these faces to fill the space between, to make it harder to pass a line segment between the two faces. As long as each vine connects with either the top or bottom, the hypermaze will have at most a single solution. As long as no vine connects with both the top and bottom (which would form an impassable column), and as long as there are no vine loops in the top and bottom sections that cause them to be inextricably linked with each other like a chain, the hypermaze will be solvable.
- Planair: Planair Mazes with unusual topology are generally done as an array of one or more smaller Mazes or Maze sections, where it's defined how the edges connect with each other. A Maze on the surface of a cube is just six square Maze sections, where when the part being created runs into an edge it flows onto another section and onto the right edge appropriately.
- Template: Mazes based on templates are done by simply starting with the base template image, then running the isolation remover to ensure the Maze has a solution, followed by the loop remover to ensure the Maze is hard enough, resulting in a perfect Maze that still looks very similar to the original image. For example, to create a Maze composed of interlocking spirals, just create some random spirals without worrying whether it's a Maze or not, then run it through the isolation and loop removers.

Perfect Maze Creation Algorithms

There are a number of ways of creating perfect Mazes, each with its own characteristics. Here's a list of specific algorithms. All of these describe creating the Maze by carving passages, however unless otherwise specified each can also be done by adding walls:

- Recursive backtracker: This is somewhat related to the recursive backtracker solving method, and requires stack up to the size of the Maze. When carving, be as greedy as possible, and always carve into an unmade section if one is next to the current cell. Each time you move to a new cell, push the former cell on the stack. If there are no unmade cells next to the current position, pop the stack to the previous position. The Maze is done when you pop everything off the stack.

This algorithm results in Mazes with about as high a "river" factor as possible, with fewer but longer dead ends, and usually a very long and twisty solution. When implemented efficiently it runs fast, with only highly specialized algorithms being faster. Recursive backtracking doesn't work as a wall adder, because doing so tends to result in a solution path that follows the outside edge, where the entire interior of the Maze is attached to the boundary by a single stem.

- Kruskal's algorithm: This is Kruskal's algorithm to produce a minimum spanning tree. It's interesting because it doesn't "grow" the Maze like a tree, but rather carves passage segments all over the Maze at random, but yet still results in a perfect Maze in the end. It requires storage proportional to the size of the Maze, along with the ability to enumerate each edge or wall between cells in the Maze in random order (which usually means creating a list of all edges and shuffling it randomly). Label each cell with a unique id, then loop over all the edges in random order. For each edge, if the cells on either side of it have different id's, then erase the wall, and set all the cells on one side to have the same id as those on the other. If the cells on either side of the wall already have the same id, then there already exists some path between those two cells, so the wall is left alone so as to not create a loop. This algorithm yields Mazes with a low "river" factor, but not as low as Prim's algorithm. Merging the two sets on either side of the wall will be a slow operation if each cell just has a number and are merged by a loop. Merging as well as lookup can be done in near constant time by using the union-find algorithm: Place each cell in a tree structure, with the id at the root, in which merging is done quickly by splicing two trees together. Done right, this algorithm runs reasonably fast, but is slower than most because of the edge list and set management.
- Prim's algorithm (true): This is Prim's algorithm to produce a minimum spanning tree, operating over uniquely random edge weights. It requires storage proportional to the size of the Maze. Start with any vertex (the final Maze will be the same regardless of which vertex one starts with). Proceed by selecting the passage edge with least weight connecting the Maze to a point not already in it, and attach it to the Maze. The Maze is done when there are no more edges left to consider. To efficiently select the next edge, a priority queue (usually implemented with a heap) is needed to store the frontier edges. Still, this algorithm is somewhat slow, because selecting elements from and maintaining the heap requires log(n) time. Therefore Kruskal's algorithm which also produces a minimum spanning tree can be considered better, since it's faster and produces Mazes with identical texture. In fact, given the same random number seed, identical Mazes can be created with both Prim's and Kruskal's algorithms.
- Prim's algorithm (simplified): This is Prim's algorithm to produce a minimum spanning tree, simplified such that all edge weights are the same. It requires storage proportional to the size of the Maze. Start with a random vertex. Proceed by randomly selecting a passage edge connecting the Maze to a point not already in it, and attach it to the Maze. The Maze is done when there are no more edges left to consider. Since edges are unweighted and unordered, they can be stored in a simple list, which means selecting elements from the list is very fast and only takes constant time. Therefore this is much faster than true Prim's algorithm with random edge weights. The texture of the Maze produced will have a lower "river" factor and a simpler solution than true Prim's algorithm, because it will out spread equally from the start point like poured syrup, instead of bypassing and flowing around clumps of higher weighted edges that don't get covered until later.
- Prim's algorithm (modified): This is Prim's algorithm to produce a minimum spanning tree, modified so all edge weights are the same, but also implemented by looking at cells instead of edges. It requires storage proportional to the size of the Maze. During creation, each cell is one of three types: (1) "In": The cell is part of the Maze and has been carved into already, (2) "Frontier": The cell is not part of the Maze and has not been carved into yet, but is next to a cell that's already "in", and (3) "Out": The cell is not part of the Maze yet, and none of its neighbors are "in" either. Start by picking a cell, making it "in", and setting all its neighbors to "frontier". Proceed by picking a "frontier" cell at random, and carving into it from one of its neighbor cells that are "in". Change

that "frontier" cell to "in", and update any of its neighbors that are "out" to "frontier". The Maze is done when there are no more "frontier" cells left (which means there are no more "out" cells left either, so they're all "in"). This algorithm results in Mazes with a very low "river" factor with many short dead ends, and a rather direct solution. The resulting Maze is very similar to simplified Prim's algorithm, with only the subtle distinction that indentations in the expanding tree get filled only if that frontier cell is randomly selected, as opposed to having triple the chance of filling that cell via one of the frontier edges leading to it. It also runs very fast, faster than simplified Prim's algorithm because it doesn't need to compose and maintain a list of edges.

- Aldous-Broder algorithm: The interesting thing about this algorithm is that it's uniform, which means it generates all possible Mazes of a given size with equal probability. It also requires no extra storage or stack. Pick a point, and move to a neighboring cell at random. If an uncarved cell is entered, carve into it from the previous cell. Keep moving to neighboring cells until all cells have been carved into. This algorithm yields Mazes with a low "river" factor, only slightly higher than Kruskal's algorithm. (This means for a given size there are more Mazes with a low "river" factor than high "river", since an average equal probability Maze has low "river".) The bad thing about this algorithm is that it's very slow, since it doesn't do any intelligent hunting for the last cells, where in fact it's not even guaranteed to terminate. However since the algorithm is simple it can move over many cells quickly, so finishes faster than one might think. On average it takes about seven times longer to run than standard algorithms, although in bad cases it can take much longer if the random number generator keeps making it avoid the last few cells. This can be done as a wall adder if the boundary wall is treated as a single vertex, i.e. if a move goes to the boundary wall, teleport to a random point along the boundary before moving again. As a wall adder this runs nearly twice as fast, because the boundary wall teleportation allows quicker access to distant parts of the Maze.
- Wilson's algorithm: This is an improved version of the Aldous-Broder algorithm, in that it produces Mazes with exactly the same texture as that algorithm (the algorithms are uniform with all possible Mazes generated with equal probability), however Wilson's algorithm runs much faster. It requires storage up to the size of the Maze. Begin by making a random starting cell part of the Maze. Proceed by picking a random cell not already part of the Maze, and doing a random walk until a cell is found which is already part of the Maze. Once the already created part of the Maze is hit, go back to the random cell that was picked, and carve along the path that was taken, adding those cells to the Maze. More specifically, when retracing the path, at each cell carve along the direction that the random walk most recently took when it left that cell. That avoids adding loops along the retraced path, resulting in a single long passage being appended to the Maze. The Maze is done when all cells have been appended to the Maze. This has similar performance issues as Aldous-Broder, where it may take a long time for the first random path to find the starting cell, however once a few paths are in place, the rest of the Maze gets carved quickly. On average this runs five times faster than Aldous-Broder, and takes less than twice as long as the top algorithms. Note this runs twice as fast when implemented as a wall adder, because the whole boundary wall starts as part of the Maze, so the first walls are connected much quicker.
- Hunt and kill algorithm: This algorithm is nice because it requires no extra storage or stack, and is therefore suited to creating the largest Mazes or Mazes on the most limited systems, since there are no issues of running out of memory. Since there are no rules that must be followed all the time, it's also the easiest to modify and to get to create Mazes of different textures. It's most similar to the recursive backtracker, except when there's no unmade cell next to the current position, you enter "hunting" mode, and systematically scan over the Maze until an unmade cell is found next to an already carved into cell, at which point you start carving again at that new location. The Maze is done when all cells have been scanned over once in "hunt" mode. This algorithm tends to make Mazes with a high "river" factor, but not as high as the recursive backtracker. You can make this generate Mazes with a lower river factor by choosing to enter "hunt"

mode more often. It runs slower due to the time spent hunting for the last cells, but isn't much slower than Kruskal's algorithm. This can be done as a wall adder if you randomly teleport on occasion, to avoid the issues the recursive backtracker has.

- Growing tree algorithm: This is a general algorithm, capable of creating Mazes of different textures. It requires storage up to the size of the Maze. Each time you carve a cell, add that cell to a list. Proceed by picking a cell from the list, and carving into an unmade cell next to it. If there are no unmade cells next to the current cell, remove the current cell from the list. The Maze is done when the list becomes empty. The interesting part that allows many possible textures is how you pick a cell from the list. For example, if you always pick the most recent cell added to it, this algorithm turns into the recursive backtracker. If you always pick cells at random, this will behave similarly but not exactly to Prim's algorithm. If you always pick the oldest cells added to the list, this will create Mazes with about as low a "river" factor as possible, even lower than Prim's algorithm. If you usually pick the most recent cell, but occasionally pick a random cell, the Maze will have a high "river" factor but a short direct solution. If you randomly pick among the most recent cells, the Maze will have a low "river" factor but a long windy solution.
- Growing forest algorithm: This is a more general algorithm, that combines both tree and set based types. It's an extension of the Growing Tree algorithm, that basically involves multiple instances of it expanding at the same time. Start with all cells randomly sorted into a "new" list, and also each cell in its own set similar to the start of Kruskal's algorithm. Begin by selecting one or more cells, moving them from the "new" to an "active" list. Proceed by picking a cell from the "active" list, and carving into an unmade cell in the "new" list next to it, adding the new cell to the "active" list, and merging the two cells' sets. If an attempt is made to carve into an existing part of the Maze, allow it if the cells are in different sets, and merge the sets as done with Kruskal's algorithm. If there are no unmade "new" cells next to the current cell, move the current cell to a "done" list. The Maze is complete when the "active" list becomes empty. At the end, merge any remaining sets together as done with Kruskal's algorithm. You may periodically spawn new trees by moving one or more cells from the "new" list to the "active" list like at the beginning. The number of initial trees and the rate of newly spawned trees can generate many unique textures, combined with the already versatile settings of the Growing Tree algorithm.
- Eller's algorithm: This algorithm is special because it's not only faster than all the others that don't have obvious biases or blemishes, but its creation is also the most memory efficient. It doesn't even require the whole Maze to be in memory, only using storage proportional to the size of a row. It creates the Maze one row at a time, where once a row has been generated, the algorithm no longer looks at it. Each cell in a row is contained in a set, where two cells are in the same set if there's a path between them through the part of the Maze that's been made so far. This information allows passages to be carved in the current row without creating loops or isolations. This is actually quite similar to Kruskal's algorithm, just this completes one row at a time, while Kruskal's looks over the whole Maze. Creating a row consists of two parts: Randomly connecting adjacent cells within a row, i.e. carving horizontal passages, then randomly connecting cells between the current row and the next row, i.e. carving vertical passages. When carving horizontal passages, don't connect cells already in the same set (as that would create a loop), and when carving vertical passages, you must connect a cell if it's a set of size one (as abandoning it would create an isolation). When carving horizontal passages, when connecting cells union the sets they're in (since there's now a path between them), and when carving vertical passages, when not connecting a cell put it in a set by itself (since it's now disconnected from the rest of the Maze). Creation starts with each cell in its own set before connecting cells within the first row, and creation ends after connecting cells within the last row, with a special final rule that every cell must be in the same set by the time we're done to prevent isolations. (The last row is done by connecting each pair of adjacent cells if not already in the same set.) The best way to implement the set is with a circular doubly linked list of cells (which can

just be an array mapping cells to the pair of cells on either side in the same set) which allows insertion, deletion, and checking whether adjacent cells are in the same set in constant time. One issue with this algorithm is that it's not balanced with respect to how it treats the different edges of the Maze, where connecting vs. not connecting cells need to be done in the right proportions to prevent texture blemishes.

- Recursive division: This algorithm is somewhat similar to recursive backtracking, since they're both stack based, except this focuses on walls instead of passages. Start by making a random horizontal or vertical wall crossing the available area in a random row or column, with an opening randomly placed along it. Then recursively repeat the process on the two subareas generated by the dividing wall. For best results, give bias to choosing horizontal or vertical based on the proportions of the area, e.g. an area twice as wide as it is high should be divided by a vertical wall more often. This is the fastest algorithm without directional biases, and can even rival binary tree mazes because it can finish multiple cells at once, although it has the obvious blemish of long walls crossing the interior. This algorithm is a form of nested fractal Mazes, except instead of always making fixed cell size Mazes with Mazes of the same size within each cell, it divides the given area randomly into a random sized 1x2 or 2x1 Maze. Recursive division doesn't work as a passage carver, because doing so results in an obvious solution path that either follows the outside edge or else directly crosses the interior.
- Binary tree Mazes: This is basically the simplest and fastest algorithm possible, however Mazes produced by it have a very biased texture. For each cell carve a passage either leading up or leading left, but not both. In the wall added version, for each vertex add a wall segment leading down or right, but not both. Each cell is independent of every other cell, where you don't have to refer to the state of any other cells when creating it. Hence this is a true memoryless Maze generation algorithm, with no limit to the size of Maze you can create. This is basically a computer science binary tree, if you consider the upper left corner the root, where each node or cell has one unique parent which is the cell above or to the left of it. Binary tree Mazes are different than standard perfect Mazes, since about half the cell types can never exist in them. For example there will never be a crossroads, and all dead ends have passages pointing up or left, and never down or right. The Maze tends to have passages leading diagonally from upper left to lower right, where the Maze is much easier to navigate from lower right to upper left. You will always be able to travel up or left, but never both, so you can always deterministically travel diagonally up and to the left without hitting any barriers. Traveling down and to the right is when you'll encounter choices and dead ends. Note if you flip a binary tree Maze upside down and treat passages as walls and vice versa, the result is basically another binary tree.
- Sidewinder Mazes: This simple algorithm is very similar to the binary tree algorithm, and only slightly more complicated. The Maze is generated one row at a time: For each cell randomly decide whether to carve a passage leading right. If a passage is not carved, then consider the horizontal passage just completed, formed by the current cell and any cells to the left that carved passages leading to it. Randomly pick one cell along this passage, and carve a passage leading up from it (which must be the current cell if the adjacent cell didn't carve). While a binary tree Maze always goes up from the leftmost cell of a horizontal passage, a sidewinder Maze goes up from a random cell. While binary tree has the top and left edges of the Maze one long passage, a sidewinder Maze has just the top edge one long passage. Like binary tree, a sidewinder Maze can be solved deterministically without error from bottom to top, because at each row, there will always be exactly one passage leading up. A solution to a sidewinder Maze will never double back on itself or visit a row more than once, although it will "wind from side to side". The only cell type that can't exist in a sidewinder Maze is a dead end with the passage facing down, because that would contradict the fact that every passage going up leads back to the start. A sidewinder Maze tends to have an elitist solution, where the right path is very direct, but there are many long false paths leading down from

the top next to it.

Algorithm	Dead End %	Type	Focus	Bias Free?	Uniform?	Memory	Time	Solution %
Unicursal	0	Tree	Wall	Yes	never	N^2	379	100.0
Recursive Backtracker	10	Tree	Passage	Yes	never	N^2	27	19.0
Hunt and Kill	11 (21)	Tree	Passage	Yes	never	0	100 (143)	9.5 (3.9)
Recursive Division	23	Tree	Wall	Yes	never	N*	10	7.2
Binary Tree	25	Set	Either	no	never	0*	10	2.0
Sidewinder	27	Set	Either	no	never	0*	12	2.6
Eller's Algorithm	28	Set	Either	no	no	N*	20	4.2 (3.2)
Wilson's Algorithm	29	Tree	Either	Yes	Yes	N^2	48 (25)	4.5
Aldous-Broder Algorithm	29	Tree	Either	Yes	Yes	0	279 (208)	4.5
Kruskal's Algorithm	30	Set	Either	Yes	no	N^2	33	4.1
Prim's Algorithm (true)	30	Tree	Either	Yes	no	N^2	160	4.1
Prim's Algorithm (simplified)	32	Tree	Either	Yes	no	N^2	59	2.3
Prim's Algorithm (modified)	36 (31)	Tree	Either	Yes	no	N^2	30	2.3
Growing Tree	49 (39)	Tree	Either	Yes	no	N^2	48	11.0
Growing Forest	49 (39)	Both	Either	Yes	no	N^2	76	11.0

This table summarizes the characteristics of the perfect Maze creation algorithms above. The Unicursal Maze algorithm (unicursal Mazes are technically perfect) is included for comparison. Descriptions of the columns follow:

- Dead End: This is the approximate percentage of cells that are dead ends in a Maze created with this algorithm, when applied to an orthogonal 2D Maze. The algorithms in the table are sorted by this field. Usually creating by adding walls is the same as carving passages, however if significantly different the wall adding percentage is in parentheses. The Growing Tree value can actually range from 10% (always pick newest cell) to 49% (always swap with oldest cell). With a high enough run factor the Recursive Backtracker can get lower than 1%. The highest possible dead end percentage in an 2D orthogonal perfect Maze is 66%, which would be a unicursal passage with a bunch of one unit long dead ends off either side of it.
- Type: There are two types of perfect Maze creation algorithms: A tree based algorithm grows the Maze like a tree, always adding onto what is already present, having a valid perfect Maze at every step. A set based algorithm builds where it pleases, keeping track of which parts of the Maze are connected with each other, to ensure it's able to link everything up to form a valid Maze by the time it's done. Some algorithms like Growing Forest do both at once.
- Focus: Most algorithms can be implemented by either carving passages or adding walls. A few can only be done as one or the other. Unicursal Mazes are always wall added since they involve bisecting passages with walls, although the base Maze can be created either way. Recursive Backtracker can't be done as a wall adder because doing so tends to result in a solution path that follows the outside edge, where the entire interior of the Maze is attached to the boundary by a single stem. Similarly Recursive Division can only be done as a wall adder due to its bisection behavior. Hunt and Kill is technically only passage carved for a similar reason, although it can be wall added if special case effort is made to grow inward from all boundary walls equally.

- Bias Free: This is whether the algorithm treats all directions and sides of the Maze equally, where analysis of the Maze afterward can't reveal any passage bias. Binary Tree is extremely biased, where it's easy traveling toward one corner and hard to its opposite. Sidewinder is also biased, where it's easy traveling toward one edge and hard to its opposite. Eller's algorithm tends to have a passage roughly paralleling the starting or finishing edges. Hunt and Kill is bias free, as long as hunting is done column by column as well as row by row to avoid a slight bias along one axis.
- Uniform: This is whether the algorithm generates all possible Mazes with equal probability. "Yes" means the algorithm is fully uniform. "No" means the algorithm can potentially generate all possible Mazes within whatever space, but not with equal probability. "Never" means there exists possible Mazes that the algorithm can't ever generate. Note that only bias free algorithms can be fully uniform.
- Memory: This is how much extra memory or stack is required to implement the algorithm. Efficient algorithms only require and look at the Maze bitmap itself, while others require storage proportional to a single row (N), or proportional to the number of cells (N^2). Some algorithms don't even need to have the entire Maze in memory, and can be added onto forever (these are marked with a asterisk). Eller's algorithm requires storage for a row, but more than makes up for that since it only needs to store the current row of the Maze in memory. Sidewinder also only needs to store one row of the Maze, while Binary Tree only needs to keep track of the current cell. Recursive Division requires stack up to the size of a row, but other than that doesn't need to look at the Maze bitmap.
- Time: This gives an idea of how long it takes to create a Maze using this algorithm, lower numbers being faster. The numbers are only relative to each other (with the fastest algorithm being assigned speed 10) as opposed to in some units, because the time is dependent on the size of the Maze and speed of the computer. These numbers are from creating 100x100 passage Mazes in the latest version of Daedalus. Usually creating by adding walls is the same speed as carving passages, however if significantly different the wall adding time is in parentheses.
- Solution: This is the percentage of cells in the Maze that the solution path passes through, for a typical Maze created by the algorithm. This assumes the Maze is 100x100 passages with the start and end in opposite corners. This is a measure of the "windiness" of the solution path. Unicursal Mazes have maximum windiness, since the solution goes throughout the entire Maze. Binary Tree has the minimum possible windiness, where the solution path simply crosses the Maze and never deviates away from or ceases to make progress toward the end. Usually creating by adding walls has the same properties as carving passages, however if significantly different the wall adding percentage is in parentheses.

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.

Algorithm	Solutions	Guarantee?	Focus	Human Doable?	Passage Free?	Memory Free?	Fast?
Random Mouse	1	no	You	Inside / Above	no	Yes	no
Wall Follower	1	no	You	Inside / Above	Yes	Yes	Yes
Pledge Algorithm	1	no	You	Inside / Above	Yes	Yes	Yes
Chain Algorithm	1	Yes	You +	no	Yes	no	Yes
Recursive Backtracker	1	Yes	You	no	Yes	no	Yes
Trémaux's Algorithm	1	Yes	You	Inside / Above	no	no	Yes
Dead End Filler	All +	no	Maze	Above	no	Yes	Yes
Cul-de-sac Filler	All +	no	Maze	Above	no	Yes	Yes
Blind Alley Sealer	All +	Yes	Maze	no	no	no	Yes

Blind Alley Filler	All	Yes	Maze	Above	no	Yes	no
Collision Solver	All Shortest	Yes	You +	no	no	no	Yes
Shortest Paths Finder	All Shortest	Yes	You +	no	Yes	no	Yes
Shortest Path Finder	1 Shortest	Yes	You +	no	Yes	no	Yes

This table summarizes the characteristics of the Maze solving algorithms above. Maze solving algorithms can be classified and judged by these criteria. Descriptions of the columns follow:

- Solutions: This describes the solutions the algorithm finds, and what the algorithm does when there's more than one. An algorithm can pick one solution, or leave multiple solutions. Also the solution(s) can be any path, or they can be the shortest path. The dead end and cul-de-sac fillers (and the blind alley sealer when considering its inaccessible sections) leave all solutions, however they may also leave passages that aren't on any solution path, so are marked "All +" above.
- Guarantee: This is whether the algorithm is guaranteed to find at least one solution. Random mouse is "no" because it isn't guaranteed to terminate, and wall follower and Pledge algorithm are "no" because they will fail to find a solution if the goal is within an island. The dead end and cul-de-sac fillers are "no" because they may not do anything to the Maze at all in purely braid Mazes.
- Focus: There are two general types of algorithms to solve a Maze: Focus on "you", or focus on the Maze. In a you-focuser, you have a single point ("You" above) or a set of points ("You +" above) and try to move them through the Maze from start to finish. In a Maze-focuser, you look at the Maze as a whole and invalidate useless passages.
- Human Doable: This refers to whether a person could readily use the algorithm to solve the Maze, either while inside a life sized version, or while looking at a map from above. Some you-focuser algorithms can be implemented by a person inside (or above) the Maze, while some Maze-focusers can be implemented by a person, but only from above. Other algorithms are complicated or intricate enough they can only reliably be done by a computer.
- Passage Free: This is whether the algorithm can be done anywhere. Some algorithms require the Maze to have obvious passages, or distinct edges between distinct vertices in graph terms, or one pixel wide passages when implemented on a computer. The wall follower, Pledge algorithm, and chain algorithm only require a wall on one side of you. The recursive backtracker and the shortest path(s) finders make their own paths through open spaces.
- Memory Free: This is whether no extra memory or stack is required to implement the algorithm. Efficient algorithms only require and look at the Maze bitmap itself, and don't need to add markers to the Maze during the solving process.
- Fast: This is whether the solving process is considered fast. The most efficient algorithms only need to look at each cell in the Maze once, or can skip sections altogether. Running time should be proportional to the size of the Maze, or in Computer Science terms O(n^2) where n is the number of cells along one side. Random mouse is slow because it isn't guaranteed to terminate, while the blind alley filler potentially solves the Maze from each junction.

Other Maze Operations

There are more things that can be done with Mazes beyond just creating and solving them, as described below:

- Flood fill: A quick and dirty yet useful utility can be implemented with a single call to a graphics library's Fill or FloodFill routine. FloodFill the passage at the beginning, and if the end isn't filled, the Maze has no solution. For Mazes with an

entrance and exit on the edges, FloodFill one wall, and the remaining edge marks out the solution. For Mazes with the start or goal inside the Maze, FloodFill the surrounding wall, and if the exit wall isn't erased, wall following won't work to solve it. Many Maze creation methods, solving methods, and other utilities involve "flooding" the Maze at certain points.

- Isolation remover: This means to edit the Maze such that there are no passage sections that are inaccessible from the rest of the Maze, by removing walls to connect such sections to the rest of the Maze. Start with a copy of the Maze, then flood the passage at the beginning. Scan the Maze (preferably in a random order that still hits every possible cell) for any unfilled cells adjacent to a filled cell. Remove a wall segment in the original Maze at that point, flood the Maze at this new point, and repeat until every section is filled. This utility is used in the creation of braid and template Mazes.
- Loop remover: This means to edit the Maze such that there are no loops or detached walls within it, every section of the Maze reachable from any other by at most one path. The way to do this is almost identical to the isolation remover, just treat walls as passages and vice versa. Start with a copy of the Maze, then flood across the top of the outer walls. Scan the Maze (preferably in a random other that still hits every possible wall vertex) for any unfilled walls adjacent to a filled wall. Add a wall segment to the original Maze at that point connecting the two wall sections, flood the Maze at this new point, and repeat until every section is filled. This utility is used in the creation of template Mazes, and can be used to convert a braid Maze to a perfect Maze that still looks similar to the original.
- Bottleneck finder: This means to find those passages or intersection points in a Maze such that every solution to that Maze passes through them. To do this, run the left hand wall follower to get the leftmost solution, and run the right hand wall follower to get the rightmost solution, where places the two solutions have in common are the bottlenecks. That technique however only works for Mazes that wall following will successfully solve. For other Mazes, to find bottleneck passages, find any solution to the Maze, and also run the blind alley sealer (which may make the Maze unsolvable if it treats an entrance or exit within the Maze as a large blind alley). Parts of the solution path that go through sealed off passages, are bottlenecks.

Algorithm Implementations

- Daedalus: All the Maze creation and solving algorithms described above are implemented in Daedalus, a free Windows program available for download. Daedalus comes with its complete source code, which can be viewed to see more information about and specific implementations of these algorithms.

Back to Think Labyrinth!

This site produced by Walter D. Pullen (see Astrolog homepage), hosted on astrolog.org and Magitech, created using Microsoft FrontPage, page last updated November 20, 2015.