The Problem

Assumptions:

* The duct must pass through each room exactly once.

* The duct must pass through all rooms, starting at the starting room and ending at the goal room.

* From a given room you can explore at most three subsequent rooms (since you can't backtrack to the previously entered room).

* There is exactly one starting room and exactly one goal room.

Since the duct must pass through every room and can only pass through a room exactly once, the problem stood out as a hamiltonian path problem. A hamiltonian path is a path between two vertices (in our case the start room and end/goal room) of a graph that visits each vertex exactly once.

My first stab at the problem was an exhaustive depth first search which explored all distinct paths from the given start vertex. If the search encountered a goal vertex and all nodes have been explored (that is, the number of verticies in the graph is equal to the number of verticies in the path), then a hamiltonian path has been found and the the path counter is incremented.

Although this was a working solution it was definitely not optimal, taking several minutes to find paths on even a 5x5 room grid. This prompted me to research other hamiltonian path search algorithms which yield better performance times (not surprisingly there are a plethora of algorithms out there, see http://cs.swan.ac.uk/~csoliver/ok-sat-library/OKplatform/ExternalSources/sources/Graphs/Hamiltonian/Vandegriend/vandegriend.ps).

When researching ways to improve hamiltonian path search times I came across a number of approaches, some intuitive (such as ensuring that each vertex in a graph has a degree >= 2), and some requiring more thought (testing for biconnectivity). I describe the approaches that I implemented below:

Pruning Strategies: One way to dramatically reduce the search space when searching for hamiltonian cycles is to prune away paths that will never contain a hamiltonian path. Since this is an exponential problem performing this pruning has a huge impact on search times.

Testing a partial path for connectivity: The first pruning strategy I implemented involved testing a partial path for connectivity; that is, ensuring that for a current vertex v in graph G, every vertex u in G can be visited from v (excluding any verticies that have already been visited). Consider the following example of a connected graph where the starting vertex is 0:

0 -> {1, 2}
1 -> {0, 2, 3}
2 -> {0, 1, 3}
3 -> {1, 2}

The graph above is connected since every vertex can be reached from 0. If the edges from {2, 1} and {1, 3} were removed, the graph would no longer be connected:

0 -> {1, 2}
1 -> {0, 3}
2 -> {0}

3 -> {1}

The way I test for connectivity is by exploring the set of verticies accessible from each neighbor of v, then testing them for disjointedness between them. For example in the graph above 0 has neighbors 1 and 2, with the following verticies accessible from each neighbor (excluding v and any previously explored verticies):

n1verts: 1 -> {1, 3}
n2verts: 2 -> {2}

since n1verts intersect n1verts = empty set, the partial path is not connected.

Testing a partial path for biconnectivity: The pruning strategy that yielded the best results involved testing a given partial path for biconnectivity, that is, ensuring that no articulation points exist in a given partial path. An articulation point is a vertex that when removed, disconnects the previously connected graph into separate components. As an example:

0 -> {1}
1 -> {2}

In the graph above 1 is an articulation point since removing it separates the graph into two graphs (0 and 1). If an undirected edge from 0 to 2 was added then 1 would no longer be an articulation point since you would be able to access 2 from 0 and vice versa. To perform the biconnectivity test I'm using Hopcroft and Tarjan's modified depth first search algorithm (a good explanation of it can be found here: http://en.wikipedia.org/wiki/Biconnected_component)

Checking the degrees of each vertex in a partial path: There are a number theorems regarding the degree requirements a given vertex for a hamiltonian path/cycle to exist. The two that I focused on involve ensuring that each vertex in a potential path has at least a degree of 2 (in other words, it has at least two incident edges), and ensuring that no vertex which exists in a potential path has three neighbors that are of degree 2:

* In a graph with a hamiltonian cycle the degree of each vertex must be >= 2.
* If a vertex has three neighbors of degree 2, that graph cannot contain a hamiltonian cycle.

The way that I check for these conditions to be met is by performing a depth first search against the graph, excluding any previously visited verticies in the path.

Removing spurious edges from an initial graph or partial path: A number of theorems exist which address removing edges that cannot be part of a hamiltonian cycle, either before conducting the search, or as the search progresses. Performing this initial pruning can significantly reduce the search space for certain types of graph. The edge removal approach that I implemented involved examining each vertex v in the graph, and if v has two neighbors of degree 2, removing any additional edges incident to v:

* If vertex v has 2 neighbors a and b which are both of degree 2, then all edges (v, x), where x not in {a, b} are not in any possible hamiltonian cycle.

Converting a hamiltonian path to a hamiltonian cycle: Except for the connectivity test, all of the strategies above act upon hamiltonian cycles instead of hamiltonian paths so I needed an approach to convert a given path to a cycle. The way I do this is by adding an 'unofficial', virtual undirected edge between a starting vertex and a goal. As an example consider the following path where the starting vertex is 0 and the goal vertex is 3:

0 -> {1}
1 -> {2}
2 -> {3}

Although this is a hamiltonian path, this path will fail the biconnectivity test as removing 1 or 2 will increase the number of connected components in the graph. To mitigate this I create a virtual undirected edge from a given starting vertex to the goal vertex. For example, if we have explored 0 and our current vertex is 1, the transformed graph would look like:

1 -> {2, 3}
2 - {3}
3 -> {1}

Resources

When researching hamiltonian path search algorithms I came across a number of helpful resources that I wanted to mention:

Rubin, F. 1974. A Search Procedure for Hamilton Paths and Circuits. J. ACM 21, 4 (Oct. 1974), 576-580. DOI= http://doi.acm.org/10.1145/321850.321854

Although it's a fairly short paper, it describes all of the strategies noted above (converting a ham path to a ham cycle, checking for articulation points, testing connectivity of the unexplored graph, admissibility of a partial path based on the degree of verticies in it, etc.)

Basil Vandegriend, Finding Hamiltonian Cycles: Algorithms, Graphs and Performance: http://cs.swan.ac.uk/~csoliver/ok-sat-library/OKplatform/ExternalSources/sources/Graphs/Hamiltonian/Vandegriend/vandegriend.html

An excellent overview of hamiltonian cycle algorithms, pruning strategies, and theorems. One particularly interesting search algorithm that the paper describes is the MultiPath algorithm, in which the initial set of paths is formed by obtaining all paths formed from forced edges (a forced edge is an edge between two verticies that doesn't allow for exploration of an alternate path. That is, an intermediate vertex in a forced edge path has a degree of 2), and constructing admissible paths by combining segments as the search progresses. Since you start with a set of segments, rather than a fixed set of paths from s (the starting point), at first glance it seems like this approach would lend itself well to threading since each segment in the initial path set could be spun off in a separate thread in a given thread pool. I started to implement the multipath algorithm but due to time constraints I didn't finish it.

Andrew Chalaturnyk, A Fast Algorithm For Finding Hamilton Cycles: ftp://www.combinatorialmath.ca/g&g/chalaturnykthesis.pdf

Describes an interesting multipath implementation which uses a turing machine.

Wikipedia - Of course wikipedia was one of the first resources that I turned to. I noted a number of helpful wikipedia entries in my code comments.

Implementation

Although I knew that doing so would incur a performance penalty, I decided to implement the path

counter in Java since it would allow me to utilize the all of the helpful data structures that Java provides such as sets, maps, etc. In retrospect writing it in C would have most likely yielded better search times.

To perform set differences, unions, etc. that some of the pruning strategies require I'm using of google-collections (http://code.google.com/p/google-collections/) which I have been using heavily at work and absolutely love. It also made the adjacency list code much cleaner than it might have been (the adjacency list is backed by a Multimap which allows you to associate multiple values with a single key. The alternative would be to maintain a Map of Sets (for instance, Map<E, Set<E>>) which would make insertions and removals a little more verbose.)

Results

In general testing for biconnectivity gave the best results out of any combinations of the other pruning strategies, even when combining some of them in a single depth first search. Since the reduction in search space that these checks provided didn't offset the added time complexity of including them, I commented them out in my mainSearch() method (all except the biconnectivity test). If you want to test them feel free to uncomment them.

Running the 7x8 sample grid finds 301,716 paths, and on my laptop (2 cores, 4 gigs of ram) runs in a little under 2 minutes. On my 2 core xeon, 6 gigs of ram workstation it runs in about 1.5 minutes.

Future Work - Performance

While 2 minutes for a 7x8 grid is not horrible considering that this is an exponential problem there are definitely improvements that could be made which would significantly cut down run time:

Port it to C: A C version of the path counter would most likely be faster than the Java one, at the same time you would lose some of the built in data structures that java provides, and arguably the code would be more difficult to extend, maintain, and test (arguably ;)).

Investigate different hamiltonian search algorithms: It would be worth looking into some of the other hamiltonian path search algorithms out there, including some of the ones that use a heuristic. The down side with using a heuristic algorithm is that there would be no guarantee that you have found all of the paths which might be permissible in some cases but probably not this one.

Make better use of threading, thread pools: Although I do make use of a thread pool when submitting an initial path to explore, it would no doubt yield better performance to run some of the pruning strategies, checks, etc. in a separate thread as the search progresses. The problem with this is that you will quickly bump up against the max thread count allowable by the JVM if you don't keep a bounded thread pool, and if you use a fixed/bounded thread pool, tasks not being processed by an active thread will be queued which will significantly hinder performance time.

Future Work - Robustness

Command line options to select different pruning strategies, including a debug and verbose mode, etc. would be a very useful feature.

Programatically inject different pruning strategies and hamiltonian path search algorithms: If you look at HamiltonianPathCounterFactory.java you'll notice that the hamiltonian path counter takes a number of fixed, hard coupled pruning strategies. These pruning strategies are fixed, (that is, they are not configurable at run or configuration time) which makes mixing/matching different pruning strategies and testing the path counter in isolation difficult. What would be really nice would be to use a

dependency injection framework like spring's ioc container or google guice, which would allow you to programatically configure what pruning strategies to use.

Tests: I added a few tests as I went but it is definitely not comprehensive.

Requirements

Java 6: I've been using java 6 (specifically java 1.6.0_11-b03) which you can get here: http://java.sun.com/javase/6/.
Apache Ant: I'm using ant version 1.7.1, though I bet that 1.8 works as well. You can get it here: http://ant.apache.org/bindownload.cgi

Usage

To compile, build, and run the project against the 7x8 example grid first download and install Java/Ant, then run 'ant clean run-default' on the command line from the challenge directory. If you want to specify your own input, you can also run 'any clean run' which will prompt for an input string.

If you are running on a *nix box, you can also run countducts.sh located in the challenge directory (make sure you build the jar first by running 'ant clean jar')

Here's a quick summary of the interesting ant tasks that one might consider running:

ant clean - cleans build files
ant clean run - will run the duct counter, prompting for user specified input (in the form 'cols rows room1 room2 ... roomN')
ant clean run-default - will run the duct counter test against the 7x8 example grid.
ant clean jar - will just build the jar if you want to run it manually (i.e. do a java -cp "foo" com.quora.challenge.command.DuctPathCounter)
ant clean test - will run the tests for the project
ant clean javadoc - will generate the javadoc for the project

Sample Output

For sample output for the 7x8 grid, please see the sample-output file (note that I modified the output so that it will only display the path count)