FREE iPADS!!! - Project Report

Joseph Roth, Lucas Ausbury, Sean Fisk, and Jake Scott

CIS 467 - Fall 2010

Abstract

For our project, we wanted to create an interface to allow students to visualize various maze generation and solving algorithms. It displays a step by step demonstration of the progress of generating three different types of maze generations along with four different maze solving algorithms.

The program was written in C++ using the Q toolkit, allowing it to be cross-platform and allowing us a chance to explore a new way to make a user interface.

The GUI allows the user to select between the three different maze generators, solving a maze that has been generated, and zooming in and out to various parts of the maze.

Introduction

The two types of mazes that are generated fall within two categories. Perfect mazes are mazes that have one solution. All branches in the maze lead to dead ends. A perfect maze corresponds to a spanning tree of the cells.

Braid mazes are the opposite of perfect mazes. In a braid maze, there are no dead ends. Since all paths will eventually lead to the exit, solving a braid maze is relatively simple. The difficulty comes in finding the optimal solution to the maze. People also have a hard time physically solving a braid maze from within.

For technical innovation, our project used Qt to develop the GUI and make the program cross platform. We also did the coding in C++ which none of us had any production experience using. By the end of the project we learned how to use C++ as well as Qt. Our other area of technical innovation was to build an external led display. We learned quite a bit about hardware through this adventure, but we definitely underestimated the amount of work required to build hardware. A more complete description of our experiences building the led is expressed in the conclusion of this paper.

Body

Maze Generation

Prims

Prim’s algorithm is one of the more commonly used algorithms for maze generation. In essence, it is performed by randomly selecting a cell and breaking the wall between it and a randomly selected neighbor until all of the cells in the maze are in use. In practice, it is a bit more complicated however. The following pseudo code may help to illustrate it further:

```
1. Randomly select cell, assign cell value to “IN_USE”
2. Assign cell value of all neighbors to “FRONTIER”
3. Randomly select a neighbor, set to IN_USE
4. Set new neighbors to FRONTIER
5. Select a random FRONTIER cell
6.     Randomly select a neighbor cell that is IN_USE
7.     Break wall between two cells
8.     Set all neighbor cells that aren’t IN_USE to FRONTIER
9. Goto 3 - do this while FRONTIER cells exist in maze
```

We modified step 6 from the traditional Prim’s slightly in order to always guarantee a complete maze without generating some sort of solution detector/creator at the end or merely generating random mazes until it comes up with a solvable one. By only breaking the walls between the random cell and another cell that is IN_USE, there is no possible way for it to generate an incomplete maze.

Backtracker

The backtracker algorithm for generating mazes is similar to prims, but instead of making a large number of short dead end paths this generates a small number of longer dead end paths. The paths are generated by picking a random neighbor cell and breaking the wall down between them. If a currently visited cell or a cell out of bounds is selected, the current path ends. The following pseudo code may help to illustrate it further:

```
1. Count the number of possible neighbor cells
2. If the number of possible neighbor cells is greater than zero
3.    Randomly select one of the neighbor cells
4.    Break down the wall between the two cells
5.    Add current cell to starting list
6.    Current position moves to new cell
7. If the number of possible neighbor cells is zero
8.    Remove current cell from starting list
9.    Assign current cell to cell on top of starting list
```

This process is repeated until the number of cells visited is equal to the total number of cells.

Braid

The braid algorithm for generating mazes uses the same concept as the backtracker. Once the generation is complete, all of the dead ends have a wall removed. By doing this we remove the idea of a perfect maze and create a maze with more than one solution. The following pseudo code may help to illustrate it further:

```
1. Follow pseudo code for backtracker steps 1 - 6
7. If the number of possible neighbor cells is zero
8.    Add current cell to dead end list
9.    Remove current cell from starting list
10.   Assign current cell to cell on top of starting list
11. While dead end list is not empty
12.   Current cell is equal to top cell in dead end list
13.   Pick a random wall and break it down
14.   Remove cell from dead end list
```

As you can tell, the only difference between the idea behind the backtracker and braid happens after creating a maze with the backtracker generation.

Solving

There are many different algorithms to solve mazes. This project implemented a few common algorithms and analyzed their time complexity for the mazes produced by various generation algorithms at different square sizes.

Right Hand Rule

Almost every text on mazes presents the right hand rule algorithm. It is guaranteed to find a solution to every maze that has the entrance and exit on the outside of the maze. The right hand rule works by figuratively placing a hand on the right side of the maze and following the wall. On average it will visit half of the maze, but it will backtrack and repeat cells every time it reaches a dead end. The right hand rule is not very efficient, but it is easy to visualize the progression and understand how it works.

Dead End Filler

This is a simple algorithm that will only work with perfect mazes. It works by analyzing the maze from a bird's eye view and filling in every dead end until they reach a point where a decision can be made. When the algorithm is finished, the only path remaining will lead from the start to the finish.

We did not use the dead end filler in our time analysis because it does not work for all mazes. It was included to demonstrate a unique way to solve the maze without starting from the beginning. Since we did not use it in our timings, the algorithm was implemented inefficiently.

Breadth First

The breadth first search algorithm is guaranteed to find the shortest path for the maze. It works by analyzing every path of increasing length from the start position. It analyzes almost the entire maze to find the solution, but the implementation is very fast and takes a consistent amount of time.

The implementation uses a first in first out queue to number the cells in the maze with the distance from the start. Once it locates the finish, it can work backwards through the maze to mark the correct path. The following pseudocode and diagram help explain the process.

```
1. Add start cell to queue with value 1
2. Take first cell from queue
3.   Let i = value of cell
4.   Add empty neighbors to queue and assign them value of i + 1
5.   If added neighbor is the finish, goto step 7
6. Goto 2 - Repeat for next cell in queue
7. Detect path by tracing backwards from finish
```

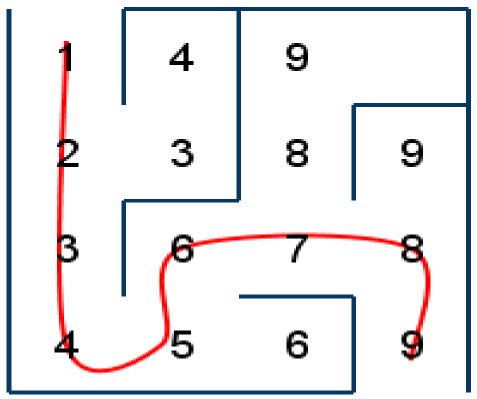


Figure 1: Breadth First Algorithm.

A*

The A* algorithm is commonly used in games to determine the best path for non-player characters to move from point to point in the environment. While it is incredibly efficient in open rooms with large search spaces, it can also be used to solve enclosed mazes.

The algorithm is a heuristic that begins at the start and guesses the next path to take based on an estimated calculation. It keeps a sorted array of all frontier cells. The ordering is based on the sum of two numbers. (1) The distance of the path travelled to reach the cell from the start and (2) the manhattan distance from the cell to the end point.

The following diagram demonstrates the calculations performed by the A* algorithm. The number in the upper left corner of each cell is the distance of the path travelled. The number in the upper right corner is the manhattan distance. The number in the bottom of the cell is the total value of the cell. The green shaded cells have been analyzed. The blue shaded cells are frontier cells that have not been analyzed. The red line is the solution, which was located by tracing backwards through the maze by decreasing the number in the upper left of each cell.

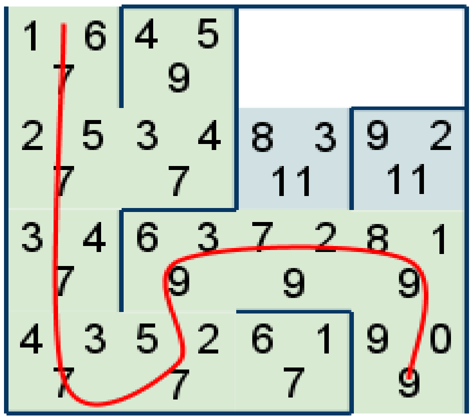


Figure 2: A Algorithm*

Graphics

The graphical user interface for this project was designed and built using the Q toolkit (Qt, pronounced ‘cute’). Qt is a cross-platform framework written in C++, primarily used for user interfaces. The choice to go with Qt was largely based on research done before the project on which toolkit to use for future cross-ptelatform graphical applications. The big candidates were GTK+, used in the GNOME desktop, and Qt, used in the K Desktop Environment. We chose Qt because it is has a longer history, is native in C++, and was in line with our development style. Qt was originally developed by Trolltech, but the company has since been acquired by Nokia and the framework has now been licensed under the LGPL.

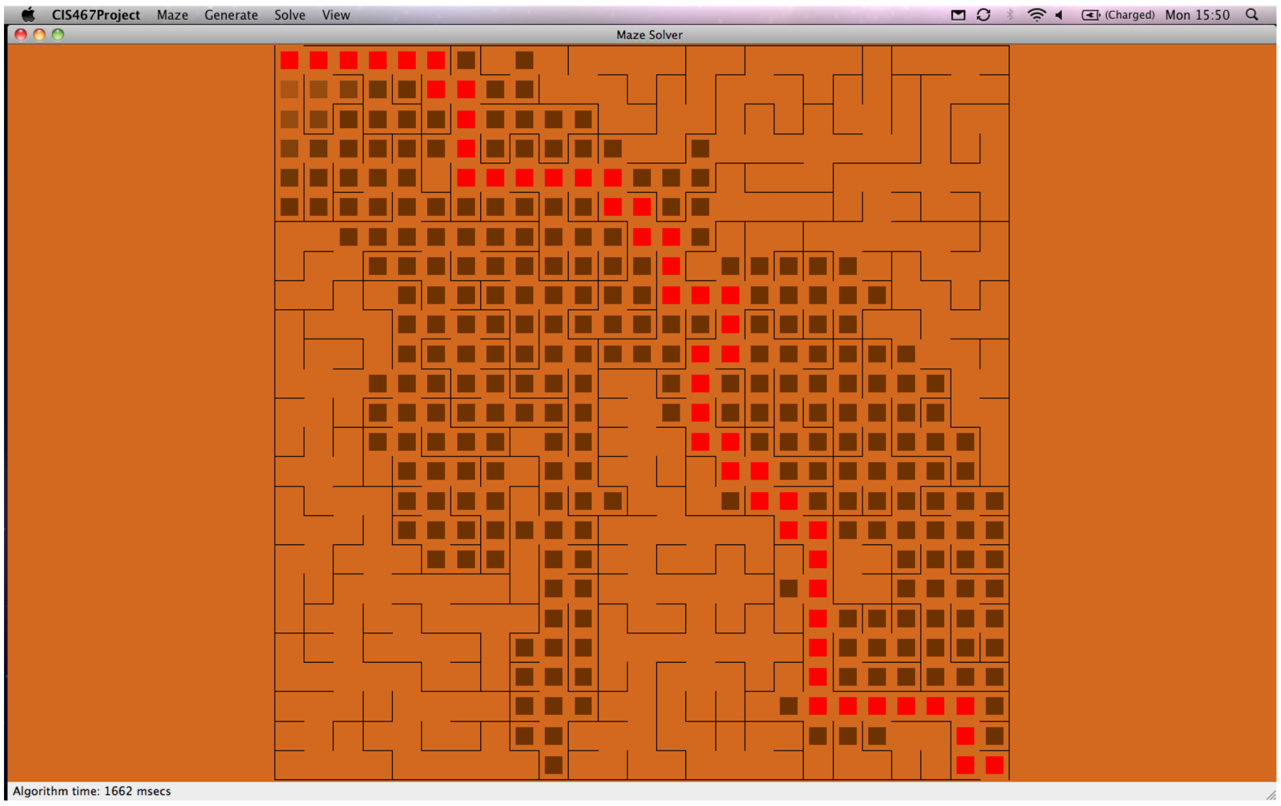
Qt provides a special mechanism called signals and slots to replace the notion of callbacks. The semantics of this mechanism are quite simple. Every class which inherits `QObject` can emit *signals*, which are used to indicate that something about the class has changed. Each class which inherits `QObject` can also have *slots*, which are connected to signals using the `connect()` method. A signal has a declaration, but no definition. Every time a signal is emitted, all corresponding slots to which it is connected are called. Signals and slots are implemented in Qt using an event loop, which is started with the call to `QApplication` in the `main()`. Qt's Meta-Object Compiler (`moc`) is responsible for compiling the non-standard signal and slots code into standard C++ so that it can be compiled by g++, msvc++, or other compilers. Signals and slots have been immensely helpful in writing this application. For example, toggling the maze animation (View -> Animation or Ctrl-A) is done by simply connecting or disconnecting the `updateCell()` signal of the `maze` object `updateCell()` slot of the `mc` (maze canvas) object.

```
// A slot in MazeCanvas
class MazeCanvas : public QGraphicsObject
{
      ...
      public slots:
            void updateCell(unsigned row, unsigned column);
      ...
}

// A signal in Maze
class Maze : public QObject
{
      ...
      signals:
            void updateCell(unsigned row, unsigned column);
      ...
}

// This function is called when the Animation menu item is activated
void Gui::toggleAnimation(bool show)
{
      if(show) // Connection of signal/slot
            connect(maze, SIGNAL(updateCell(uint,uint)), mc, \
            SLOT(updateCell(uint,uint)));
      else // Disconnection of signal/slot
            disconnect(maze, SIGNAL(updateCell(uint,uint)), mc, \
            SLOT(updateCell(uint,uint)));
}
```

Another useful characteristic of Qt is that it uses native widgets whenever possible. That is, I believe it uses pre-built widgets from Microsoft Foundation Classes, Cocoa, and X11 for Windows, Mac, and X11 (Linux, BSD), respectively. This practice allows applications written in Qt to look rather native on each platform. Here is a rendering of the application on Joseph’s MacBook:



Algorithm time: 1662 msecs

Qt also provides OpenGL acceleration, activated simply by this code:

```
view = new GraphicsView(scene);
view->setViewport(new QGLWidget(QGLFormat(QGL::SampleBuffers)));
```

The second line allows our subclass of `QGraphicsView` to use hardware accelerated rendering through OpenGL, if it is available. If it is not available, such as on Sean's laptop which does not have proper drivers, the program does not crash, the command simply does nothing.

Graphics were a very time-consuming part of this project. Learning how the library worked proved to challenging, even in the presence of decent tutorials and Qt's popularity. Although Qt is operable with STL, it provides many STL replacement classes which one is expected to use if using Qt. For example, Qt classes use `QString`s in all its classes instead of STL `string`, so there is no reason to use `string` at all. Because STL experience is considered a large part of C++ knowledge, it almost feels as though one is learning a whole new language. A lot of time was spent early in the project piecing together the models and classes we were going to use.

A very important class to both the GUI and maze solvers conceived in early development was the aptly-named `Maze` class. The `Maze` class was responsible for providing a relatively transparent interface between these two areas. Each of the `Solver` and `Generator` classes was given a pointer to an instance of the `Maze` class, from which they called appropriate `Maze` methods. `Maze` also provided the signals `updateAll()` and `updateCell(unsigned row, unsigned column)`, which were connected to `MazeCanvas` slots to update the graphics.

Another challenge in GUI/maze interaction that was addressed rather late in the project was multiplexing the calls to the maze generation and solving algorithms. Previously, clicking on their respective menu items caused a slot to be activated in either the `Generator` or `Solver` classes. This worked fine until we realized we needed to time each of the algorithms. Adding the timing code to each of the algorithms would not only repeat large portions of code, but it was nowhere close to the transparency for which we were looking. Therefore, we used a very obscure and little-known C++ feature called member function pointers. Like many C++ features on the fringe, it's really ugly. We created a multiplexing function called `solve()` which accepted pointers to member functions of `Solver`. We then passed those pointers through the solve function from the menu activation slots in `Gui`. We used the same exact method with `Solver`. Here is an example with code snippets:

```
void Gui::solveRightHandRule()
{
      solver->solve(&Solver::rightHandRule);
}
```

```
// Remove some ugliness - credit: \
http://www.parashift.com/c++-faq-lite/pointers-to-members.html#faq-
33.5

#define CALL_MEMBER_FN(object,ptrToMember) ((object).*(ptrToMember))

typedef void (Solver::*SolverAlgorithm)();

// solve() method, including timing and algorithm multiplexing
void Solver::solve(SolverAlgorithm algorithm)
{
     if(!(maze->width() && maze->height()))
     {
        emit requestMazeDimensions();
        return;
     }
```

```
    maze->resetValues();
    QTime beginTime = QTime::currentTime();
    CALL_MEMBER_FN(*this, algorithm)();
    QTime endTime = QTime::currentTime();
    emit showStatistics(beginTime.msecsTo(endTime));
    maze->update();
}
```

There may have been better ways to achieve this purpose, but we felt this method accomplished the task well without *any* performance penalties, for example versus an int-switched based approach.

On the whole, the graphics section of this project was difficult, but it provided a great learning experience and foray into Qt. Though computer science is a field with new technologies emerging literally everyday, it is still the stable, well-built systems which persist. The 25-year old C++ programming language is one of those things, still at the #3 position on the Tiobe programming language index. Qt, which is now almost 20 years old, is also one of those technologies which is useful to know and understand.

Software Engineering Code of Ethics and Professional Practice

The software engineering code of ethics and professional practice (SECEPP) is the standard code of ethics for software engineers. It was written by a joint committee from the ACM and the IEEE. Here is a list of the relevant items from the Product section of the code.

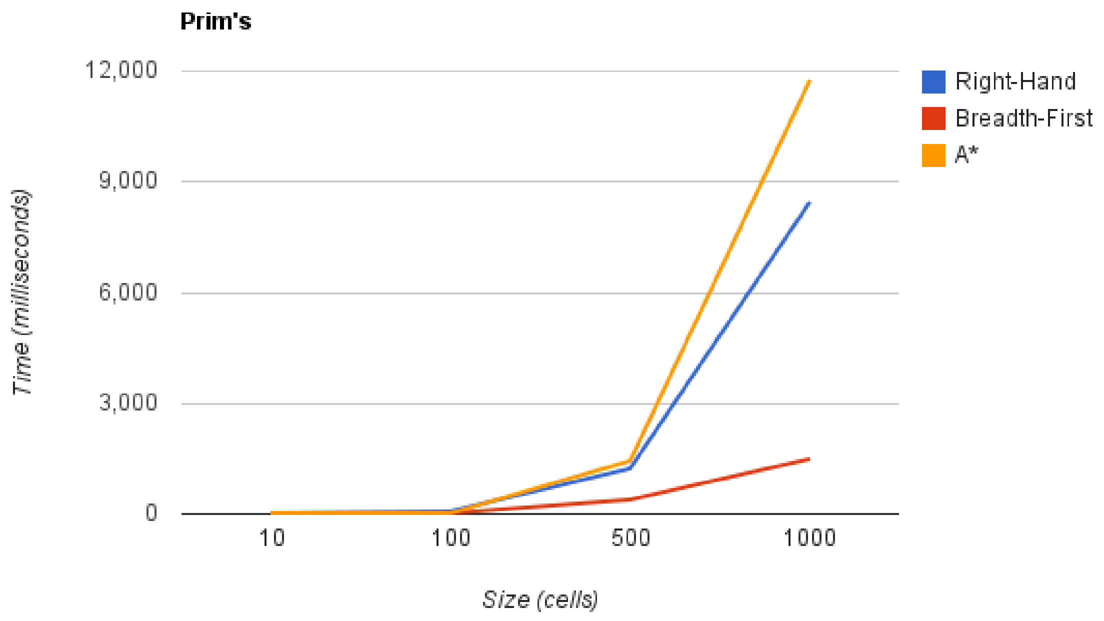
3.02. Ensure proper and achievable goals and objectives for any project on which they work or propose.
In our prospectus, we established a core set of features for our project. We also created a list of features that we hoped to accomplish if we had time. While we did not finish some of the tasks on our extra list, we did accomplish all of our core features.

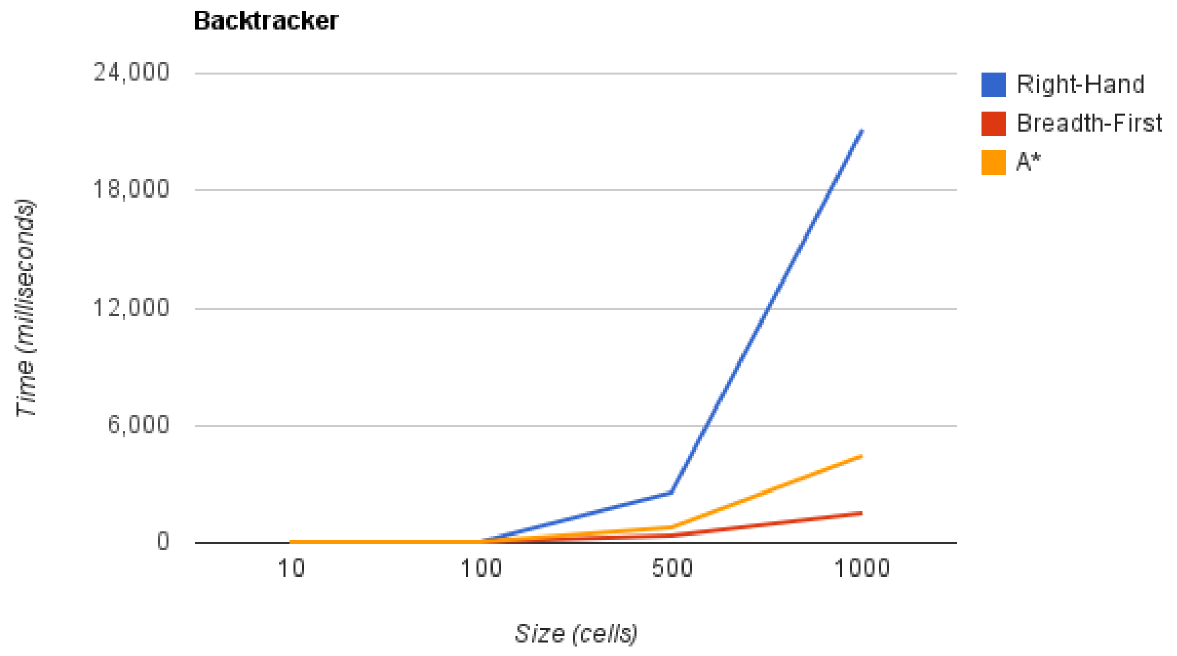
3.11. Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project on which they work.
Every few weeks, we had milestone meetings for the project. To prepare for these meetings, we created a project report detailing the work done on the project as well as any difficulties that

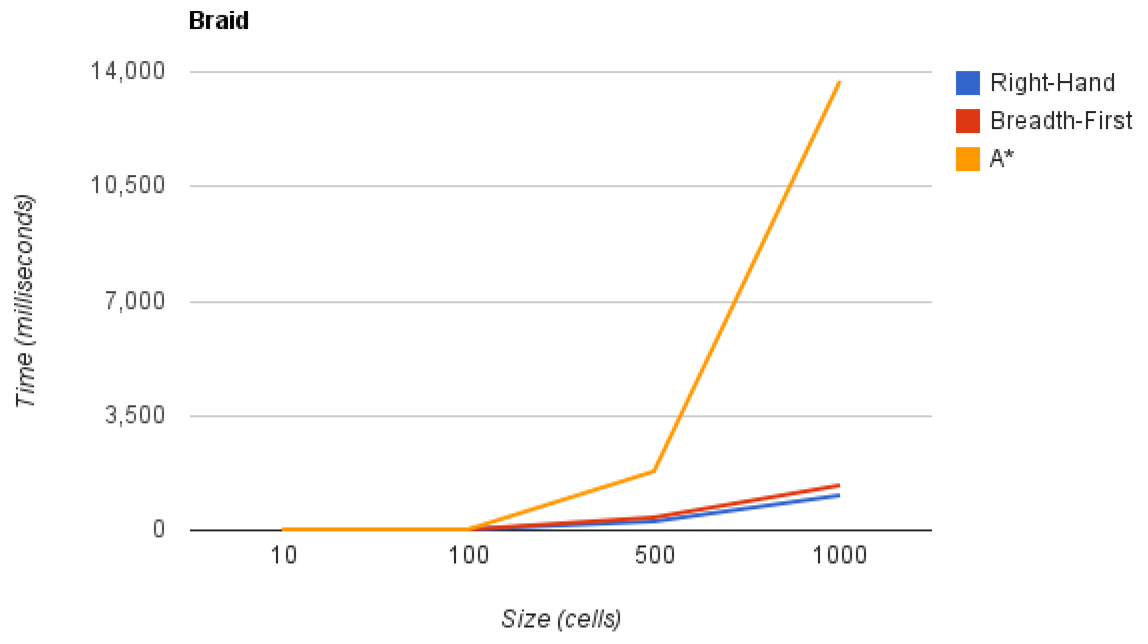
we ran into. The reports served as documentation for the process of the project.

Conclusions

Timing Results







The results from our timing of the various algorithms was not as expected. We timed each solving algorithm for the various maze generations at different sizes of square mazes. The right hand rule generally took longer as expected, and the breadth first solver was very consistent between runs. The unexpected result was the length of time that the A* algorithm took to run.

In theory the A* algorithm should take the least amount of time for most of the maze types since it does not have to analyze a high percentage of the maze. One theory into the increased time is that the A* algorithm does have to maintain a sorted array of frontier nodes. To insert a new frontier node will take log n time. Our implementation could probably be optimized to reduce the time complexity as well.

General Conclusions

One of our areas of technical innovation was a bit of a stretch for ourselves. We wanted to solder an external display out of leds. The time and knowledge required to use the computer as a controller for a matrix of leds was too much to fit into this project. The idea looked good on paper, and we were able to control a small number of leds from the parallel port. We did get exposed to building hardware which was good, but we were not able to completely build the external display.

Developing the project taught us a lot about algorithms, but it also taught us about project management. We learned that most of our features in our wish-list were unattainable in such a small period of time. We also discovered the difficulties associated with configuring 4 people's schedules and delegating work. As the project progressed, our productivity increased as we learned how to better work together.

Overall, the project was a good experience that brought together a bunch of the skills that we have learned throughout the course of our undergraduate career. It was nice to be able to synthesize our learning and build an actual project from the ground up.