**XXX**

**Dr. Neli P. Zlatareva**

**CS253 - Data Structures and Files**

**XXX**

**A\* Pathfinding Algorithm Implementation**

**A graphing algorithm is an algorithm that takes one or more graphs as inputs. Graphing algorithms are a core component in computer science and are used almost everywhere. In my CS253 course, our assignment was to find an algorithm that would peak our interest, research the functionality and purpose of the algorithm, and then implement it into a program that we design. Due to the countless graphing algorithms out there, the possibilities were endless; however the real challenge was finding one that was really interesting. To find the algorithm I wanted, I collected all of the algorithms I could find and then rated which one was most interesting. After rating dozens of algorithms, I came to the conclusion that the A\* search algorithm was by far the most interesting one I could find. When it came to researching this algorithm, there were several very helpful sources that I used that guided me to understanding the functionality of this algorithm. Once I researched the A\* algorithm, I then had to implement it into a program of my choice. I decided to create a two-dimensional game, which would be a more relevant implementation for this algorithms purpose. A presentation was also required for this project. This presentation consisted of information about what the algorithm is, its functionality, how we went about implementing it, and other information about other practical implementations in which the algorithm could be used. Choosing the A\* search algorithm was definitely the right choice for me, as it allowed me to learn not only about the algorithm itself, but other aspects of computer science that I wouldn’t have if I choose a more simplistic algorithm.**

**The A\* search algorithm is a graphing algorithm that determines an efficiently traversable path from one node to another. The main reasons for the algorithms popularity are its performance, accuracy, and versatility. One notable characteristic of the A\* algorithm is that it uses the elements of Dijkstra’s algorithm and Greedy best-first-search algorithm together in order to efficiently determine the shortest path from one node to another. A\* is similar to Dijkstra’s algorithm because it determines the shortest path, while it is also similar to Greedy best-first-search because its use of a heuristic. The heuristic is the distance from the current node to the destination point in the graph. The combination of these two algorithms not only exemplifies its performance, but also its accuracy. In comparison to Dijkstra’s and Greedy best-first-search, the A\* search algorithm is admissible and considers a significantly smaller amount of nodes compared to any other admissible search algorithm using an identical heuristic. A heuristic is admissible (or optimistic) if and only if it never overestimates the cost of reaching the destination point. This means that the estimated cost at the current node is never higher than the lowest possible cost to the destination point. For the A\* search algorithm, evaluating this estimation is represented as the heuristic value + movement cost (f(n) = g(n) + h(n)). The type of admissible heuristic I used for the A\* algorithm is what's known as the *Manhattan distance*. The Manhattan distance, also known as Taxicab geometry, is a type of admissible heuristic that replaces the typical distance function of** [**metric**](http://en.wikipedia.org/wiki/Metric_space)[**geometry**](http://en.wikipedia.org/wiki/Euclidean_geometry) **with a new metric in which the** [**distance**](http://en.wikipedia.org/wiki/Distance) **between two points is the sum of the** [**absolute differences**](http://en.wikipedia.org/wiki/Absolute_difference) **of their** [**Cartesian coordinates**](http://en.wikipedia.org/wiki/Cartesian_coordinate)**. The reason for using the Manhattan distance in my implementation was because a game uses the exact same coordinate system as this admissible heuristic would use. Using the Manhattan distance, calculating the heuristic is efficient and can be formulated before determining the best path from one node to another.**

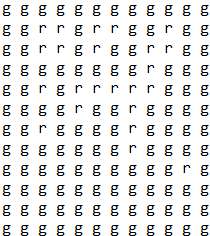
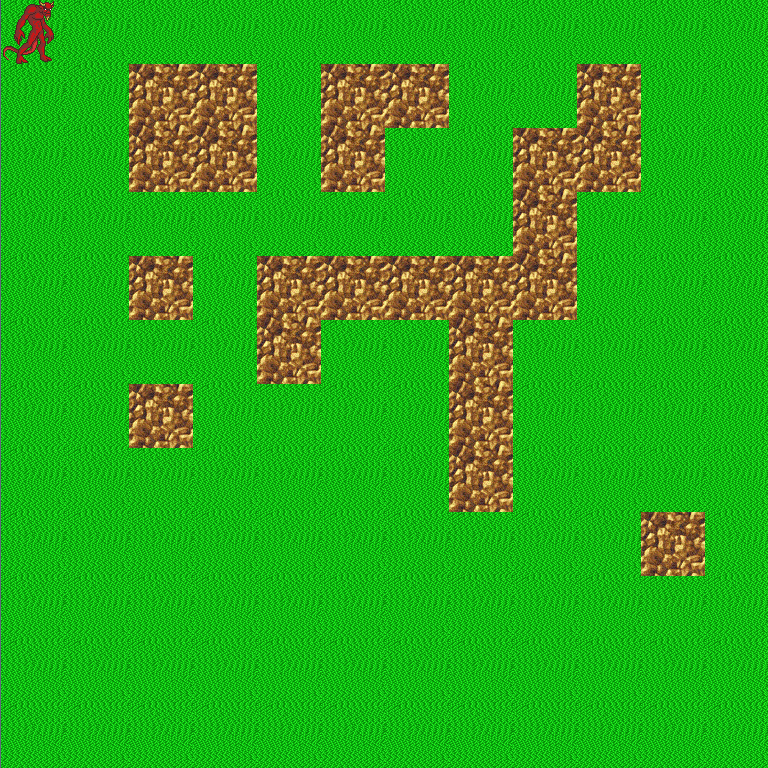
**The notable efficiency of the A\* search algorithm is another reason for its popularity. When comparing this algorithm to others such as Greedy best-first-search or Dijkstra’s, it has a more unique time complexity. Due to the use of a heuristic, A\* outperforms Dijkstra’s algorithm. Greedy best-first-search has a best of O (n), or linear, while in a worst case, it’s exponential. Similar to A\*, which also has a worst case that is O(n2), or exponential, and an average case that is polynomial. For the best case in A\*, the algorithm has an efficiency of |h(x) - h\*(x)| = O (logh\*(x)). In this case h\* is the optimal heuristic, which is the exact cost to get from x to the destination. This means that the error of h won’t grow faster than the logarithm of the perfect heuristic h\* that returns the true distance from x to the destination.**

**The A\* algorithm contains a Node that holds five four items. The first data item is the h-value. The h-value, or the heuristic, is the distance from the current node to the goal node. This is calculated using a heuristic function and is calculated before determining an actual path. This heuristic calculation however has factor that changes the value of the heuristic. This factor is whether or not you are calculating diagonal for the heuristic cost. This can heavily impact your path, especially if you are actually moving diagonal. If you are not moving diagonal, then you don’t need to account for that in your heuristic function. Not accounting for diagonals while allowing a diagonal movement can affect your path and actually create a path longer than a possible path if you were accounting for diagonals in the heuristic function. The next data item in the Node is the g-value. The g-value, or the movement cost, is a variable representing the cost of moving from the current node to a neighboring node. A conventional way of determining the movement cost is by using the values of 10 and 14. 10 would represent horizontal and vertical movements and 14 would represent diagonal movements. The reason for the usage of these values is because when you look at a 45 degree right triangle, the sides are by standard 1, 1, 1 \* square root of 2. The square root of 2 is equal to approximately 1.4. Multiply this value by 10 and the other sides by 10, and you get 10 and 14. It’s the best way to account for the distance from one node to another diagonally, in comparison to the distance from one node to another horizontally or vertically. The third data item inside is the y-value. The y-value is the value of the sum of the movement cost + the nodes heuristic value. The y-value is very important and is the referenced variable when determining the shortest path. When comparing neighbors, the neighbor of the current node with the smallest y-value becomes the child of the current node. This is then the next node that is checked and ultimately creates a path. The final data item inside the node is the Parent node. The parent node is the node in which the current node points to the next node in the path. The parent node must exist or else there would be no way to traverse through the path once reaching the goal node. Once you found the node within one node away from the goal node, you then use these parent nodes and traverse back to the original node to generate a path.**

**In order to keep track of which nodes have been checked or still need to be checked, the A\* algorithm uses two lists to record them. There is an open and closed this in which nodes are inserted/removed from as you traverse for the shortest path. Both the open list and closed this conventionally use a priority queue, however other structures can also be used for this purpose. The open list is a list of nodes that still need to be checked. Nodes are added to this list when they are the unchecked neighbors of the current node. Once the nodes on this list are checked, they are removed from the open list and are inserted into the closed list. The closed this is a list of nodes that have already been checked. The only time a node is removed from the closed list, is when the neighbor in the closed list costs less than the movement cost at the neighbor. If this is true, it is removed from the closed list and becomes the new parent.**

**For my implementation, I could have done many possibilities when it came to applying the A\* algorithm. I decided that mixing learning this algorithm and doing something I like to do would benefit me the most, and it sure has. For my implementation I decided to create a 2D game environment in which a player has free mobility around a map. The objective of this 2D environment is to give a player free movement around the map. I decided that the map would contain objects where a player would not be able to walk on. By doing so, this really exemplified the importance of the A\* algorithm and highlighted its efficiency. The A\* algorithm is used in games all the time for this exact situation. I used two textures in my game, one was a grass sprite, and the other was a rock sprite. The grass sprite was the land where the player could walk on. The rock sprite was where the player was prohibited to walk on. What the A\* algorithm did in this implementation was find the shortest path around the rocks based on where the player clicks on the map. The tile the player clicks on is where the goal node is defined.**

**Starting with the way I structured the game, I decided to use a tile based map. This means that a player can only be located on one tile at a time, and the map I constructed would consist of 144 tiles. I decided that a 12x12 map would be all that is necessary to implement my algorithm. To structure the map, I used the following format in a text file:**

**                                             **

**The reason I structured it like this was pretty simple. Building a map in a 2D environment can be tricky, but by making so each tile has an image name of 1 letter such as ‘g’ or ‘r’, I would easily be able to construct a map that I can visually see in a text file. As you can see, every place where there is a ‘g’, there is a grass sprite. As is the same with the character ‘r’. Each tile itself is a 64x64 sprite image. Each tile will represent a node in the algorithm. This way you can easily visualize what the graph would look like when traversing a path. Once constructing the map, I then used a Map object that would hold a 2D array of tiles.**

**Now implementing the actual pathfinding took me quite a lot of time because of the various ways I could have handled it. Originally, I thought for simplicity purposes I would just do horizontal and vertical movements. This way I could keep movement in general simple. I quickly decided against that, as not only would I learn more if I included diagonal movements, but it would also make my 2D game environment more dynamic. I first created a GameMap class that would hold a 2D array of visited locations, this way I can keep track of checked nodes. This class also held methods to modify this array and return a node at a given location. This class also held the blocked(tile) function which determined whether or not there was a rock on the desired tile. The pseudocode for the blocked fucntion is as follows:**

**blocked(tile)**

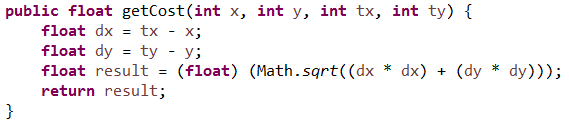
**String restricted := r**

**if(tile.getImage.getName().equals(restricted)**

**return true;**

**return false;**

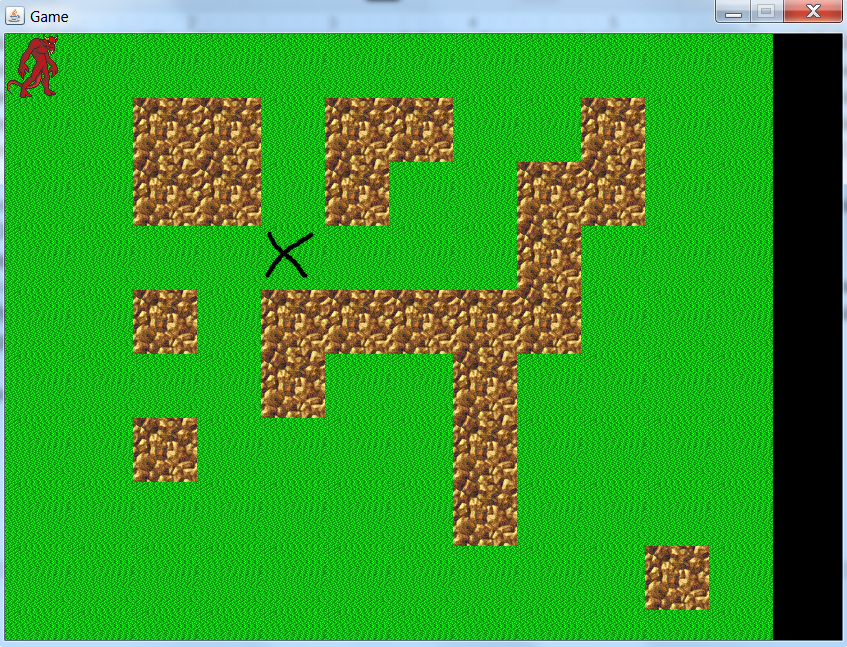
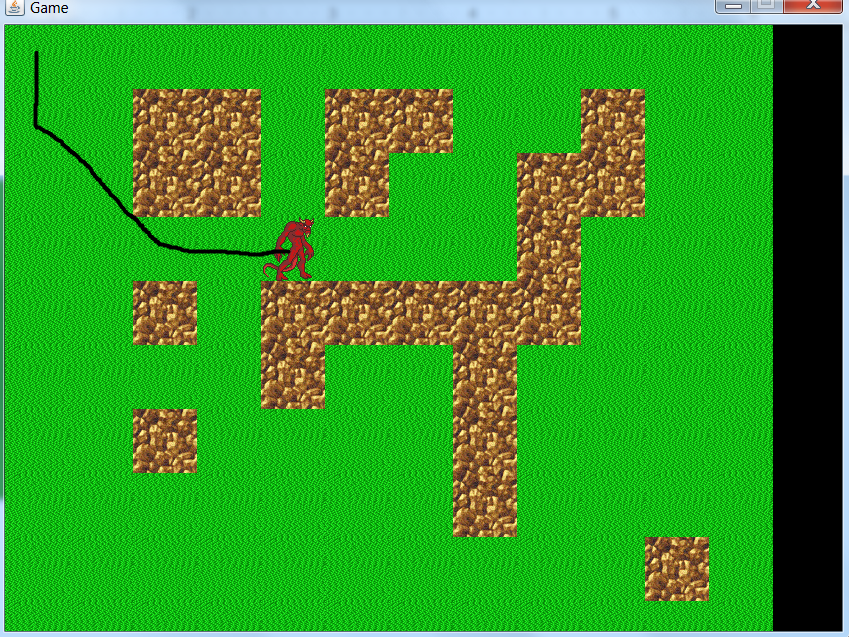
**This function is very important in the A\* algorithm because if you do not have this method, every node on the graph is available to be used inside a path. Once I established the GameMap, I needed to implement the actual Path class. The Path object would hold a list of Steps that are on the shortest path. The Path class held an inner class Step. The step would be the X, Y of the node that is next on the path. You could insert nodes at the front and tail of the Path but nowhere in between and there is no removal function for the path since it wouldn't be necessary. Once I had a GameMap and a Path, I decided to implement the heuristic function.**

****

**This is the code for the heuristic function. It takes in the parameters of the current x and y coordinates of the node, and the x and y coordinates of the goal node. This then evaluates the distance between the two. Once all of these components of the algorithm were established, I then finally needed to implement the A\* algorithm to find the shortest path. The first thing you must do is define two lists, the open and closed lists. You insert all of the neighboring nodes around the starting node into the open list, in my case the players starting location. You initially set the closed list to an empty list since we haven't checked any of the nodes yet. Then I added a check to see if the desired location was actually located on the map. I added this as a safety check so you wouldn’t be able to walk beyond the actual map. I then had to initialize all of the nodes from the game map onto a new 2D array of nodes that represent each tile. I then needed to declare a variable to keep track of the maximum death in order for it to not throw an out of bounds exception when calling an index on the 2d array of nodes. Next I declared the while loop which would continue looping until either the maximum death is >= the maximumSearchDistance, this to restrain it from searching outside the map for a path, or until the open list has been emptied. The function of this while loop is as follows. I then remove current from the open list and insert it to the closed list since now it will be checked. Next, it checks whether or not the current node is the goal node, if that is the case, then the player has no need to walk anywhere since he is already located on the goal. I then added a check for whether or not I wanted to use diagonal movements. There might be instances in the program where you do not want to allow the player to move in a diagonal direction, so this makes it so it would restrict it if you declare so in the parameters of the AStarPathFinder object. Next, you have to make sure the node is actually a valid location and isn’t a rock, which is blocked. Once it passes this check, we can then saw that we visited the node. Then we check for the next step cost. We check for this because then we can modify the parent of the node for a faster path. If this is the case, we remove from the node from the open list if it is in that list or the closed list if it is in that list. If the node is in neither those lists, we then have to add that node to the open list and calculate it’s movement cost and heuristic cost. Then this process repeats until the conditions inside the parameters of the while loop are broken. Once the while loop ends we first check if the parent node of the goal node is null. If this is the case then there is no path to the goal node and the function returns a null path. If the parent node is not null, we then have to then generate the path. This path generated is the shortest path found from the start node to the destination node. The function then returns this generated path and the shortest path has been found.**

**There will be times a user wants to walk on a grassy sprite, however this could be surrounded by rock, in which case you cannot get to this location. This is a prime example of when this algorithm will return a null path. Once the path is returned, I added a simple event system that ticks every 500ms that handles the walking. Once the player left clicks on a tile, this function is called and generates a path. The player then moves along the path every 500ms until it reaches the destination. I thought it was really cool how that once I finally got my algorithm working perfectly, I could actually see it in action. It really opens your whole mind to the possibilities of what you can actually program using this algorithm.**

**Here is an example case of the player walking:**

****

**As you can see, the player successfully moves to the desired location in the shortest path possible. Not only in my implementation, but pathfinding in general for most games is a core component of the game, and without an efficient and accurate pathfinding system, the game looks really unprofessional and sloppy. By implementing a pathfinding system, you create a stable and realistic environment for the player to play in. Now that I’ve implemented it correctly once, I can now implement it in different games that I may make in the future and it is really exciting to see what more I can do with this algorithm.**

**Moving forward with this algorithm, there was another variable I did not account for in this game environment, however it does exist in a lot of gaming environments. That is the idea of moving objects. When you having moving objects, every time the object moves you have to constantly update the blocked tiles and constantly update the location of the object so you know where those blocked locations are. Most games nowadays use moving objects and have systems designed around that. However, through my research of the A\* pathfinding algorithm, there was been instances where A\* has been used to find the shortest path on a map around an ever changing environment. No matter how the environment adapts, the algorithm will do the same and accurately find the shortest path. I am definitely going to try a simpler version of this before I would go into anything too big, however the idea of working around moving objects was definitely intriguing. Another addition I could try to use for my algorithm is different data structures to handle the open/closed list. For this situation, I just used a simple SortedList object for the open list, however, like the conventional usage of A\* algorithm, I could try and add a PriorityQueue for both the open and closed list. I would also like to try other data structures and benchmark all of them for testing purposes. It would be cool to see how fast and memory efficient I could get my algorithm to calculate all of this. Another thing I want to modify with my program is the way I handled walking. As of now, I just set the location of the player sprite to the Step of the Path every 500ms when walking the shortest path. I would like to change this so it smoothly walks the entire path so it looks more realistic and aesthetically pleasing. Right now its pretty chopping and clunky, however by adding smooth walking, it will make my 2D game environment more professional. I would also like to create a way to add following to this algorithm. Basically there would be a second player who follows behind the first player who is moving. Not only would my algorithm calculate the shortest path for the player moving, but also for X amount of players following him. This would definitely be more difficult, but it is definitely something I would want to try.**

**Overall, I loved every minute of this project. I felt that this project not only benefited my knowledge with graphing algorithms, but also a lot of other programming skills.  I knew going into researching the A\* algorithm that I would be in for a tough road, however now that I have completed the implementation and understanding the algorithm, the experience was definitely beneficial. By creating a 2D game environment, I was also able to learn a lot about game engines and different ways of displaying sprites onto the screen. I created the entire system and I am proud of what I’ve been able to implement. The A\* algorithm is definitely an interesting graphing algorithm and with my interest in game design, I might one day need to implement the same algorithm in a new game I would be making. The fact that this algorithm is so popular in the programming of games shows its efficiency and relevance in the industry. Not only did I learn a lot about game design, but the A\* algorithm itself is definitely not limited to just games. A\* is used in implementations in finding the fast freight train path to get its shipment on time. There are many real world applications for this algorithm as well, and with its phenomenal efficiency compared to other shortest path algorithms, this algorithm will remain a dominant algorithm for finding the shortest path.**

**Work Cited:**

* Introduction to A\*   From Amit’s Thoughts on Pathfinding. (n.d.). Retrieved May 8, 2015, from http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
* Introduction to A\*   from Red Blob Games. (n.d.). Retrieved May 8, 2015, from http://www.redblobgames.com/pathfinding/a-star/introduction.html
* A\* Pathfinding Tutorial. (n.d.). Retrieved May 8, 2015, from https://www.youtube.com/watch?v=KNXfSOx4eEE
* Heuristics   From Amit’s Thoughts on Pathfinding. (n.d.). Retrieved May 8, 2015, from http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html