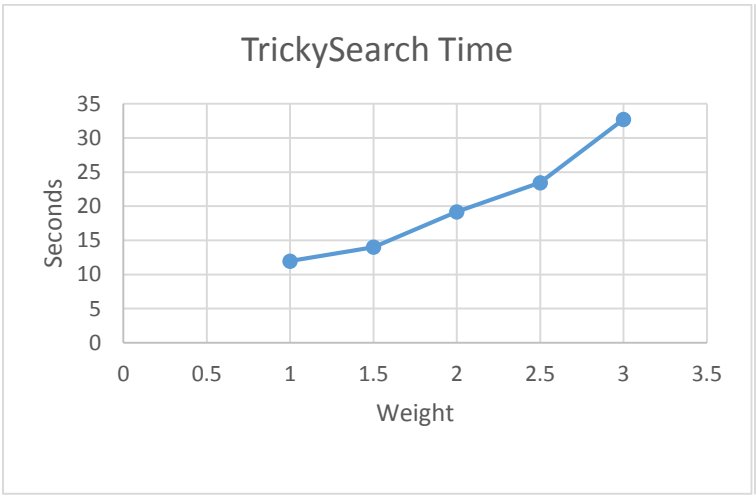
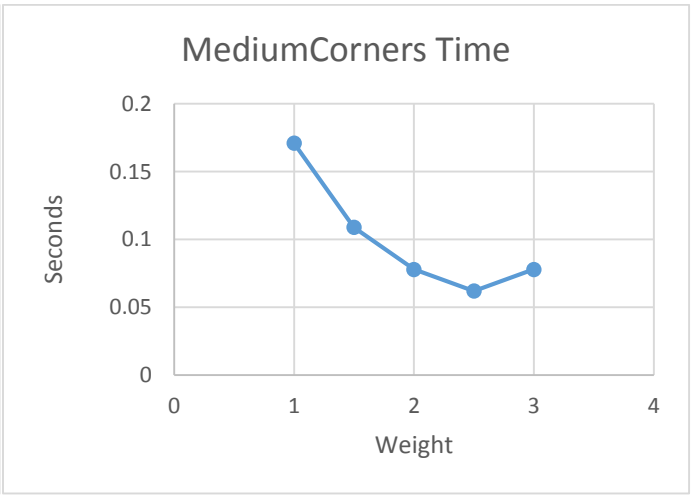
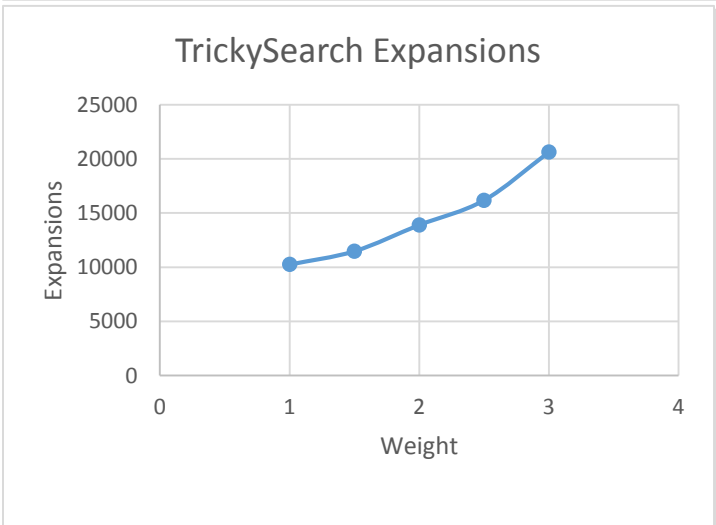
PROJECT 1 QUESTION 4

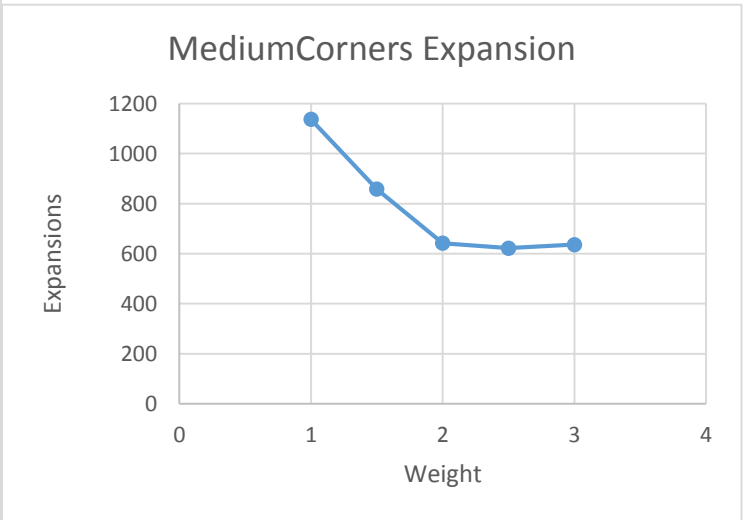
John Lucas
2-19-15
CS360

1. Before diving into my evaluation of the effectiveness of weighted A* search, it is important for me to mention how my heuristic works, as its method directly effects how effective, or not, the weight is at reducing run time. My heuristic simply assigns a node its h value be giving it the **maximum Manhattan distance to a target**. This means that if a node is being evaluated at the bottom left of a square board, and there are targets at each other corner, its h value would be the Manhattan distance to the farthest target, which would be the one in the opposite corner (this is clearly admissible because that can never overestimate). Now, I will go on to explain why, and how, weighting this heuristic changes the runtime for certain layouts significantly.









As can be seen in the above graphs, the value of weighting a heuristic is not a simple “good” or “bad” thing to do. It is also important to note that TrickySearch’s path is 60 steps, and MediumCorner’s path is 106. Additionally, TrickySearch has **13 targets** where MediumCorner has **4 targets.** After examining this same data represented above for other layouts, such as BigMaze, GreedySearch, and BigCorners, I have

come to a few conclusions about which metrics matter the most in evaluating whether using a weight is a good idea, and which weight to use.

1. **Using Higher weights will Adversely affect High Path lengths:**
   This was particularly true for BigCorners, where setting the weight at 1 completed a solution in 1.828 seconds with 4436 expansions, but 1.5 took 36.265 seconds with 9508 expansions, and anything above 2 was too long to time on my machine. The reason for this, is that the solution path for BigCorners is 2717 long, and as the algorithm traverses this path, it, obviously, must keep track of how far it has travelled when doing calculations. However, when the weight causes the heuristic to overestimate, it diminishes the role of the g value, meaning that the distance travelled matters less to the algorithm. This may not matter terribly much for layouts with shorter solutions paths, such as those above, but when g is unable to effect the overall heuristic as much as it should (due to a highly weighted h value), it can prevent the algorithm from finding the solution in a reasonable time.
2. **Using weights can sacrifice the optimal solution in favor of speed:**
   The main benefit of using a weighted approach is to develop a *decent* solution in *less time.* One example where I found this to occur was when solving BigMaze. When basic A* (weight = 1) is used, the optimal path of length 210 is found in 549 expansions, but the weight of 100 reduces this to only 467 expansions. The reason this occurs, is because the heuristic is no longer admissible, and therefore will not guarantee a solution. While, in this case, the algorithm did in fact find a solution, cases do exist where the optimal solution is quickly sacrificed for less expansions. Oftentimes, this is very acceptable, as a *good enough* solution can still be found with relative ease. However, as noted in the last observation, it is important to take into account that using a weight is not acceptable when the value found by the heuristic completely trumps the length of the path the traversal has taken so far (the g value) and essentially degrades the A* search into a form of BFS.

Conclusion:
The use of a weight is particularly important when, after the problem and the heuristic have been evaluated to be compatible with a weight, the user wants to significantly reduce memory cost of the program. By limiting expansions, memory usage, and runtime, using an effective weight in a situation can be invaluable, but only if the user is willing to sacrifice getting the *absolute optimal solution*, otherwise, simple, unweighted A* is the algorithm they must, generally, use.