Christian Fuller

Pathfinding on a Randomized Two-Dimensional Grid

Behavior and Performance of the A* Pathfinding Algorithm

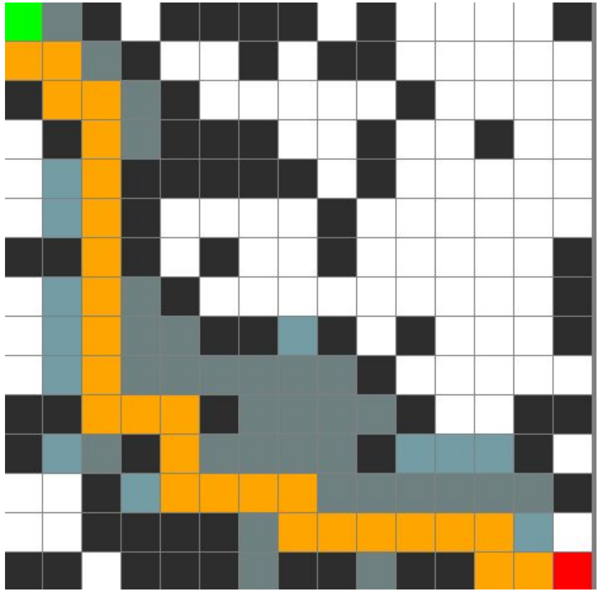
Overview

Project Goals / Scope:

- Examine the behavior and performance of the **A* algorithm**, a flexible pathfinding algorithm widely used in games and many other applications
- Look at relationship between A* and **Dijkstra's algorithm**
- Explore how A*'s accuracy and behavior can be tweaked given different parameters

Overview

- Compare the performance of the algorithms on a two-dimensional grid filled with random obstacle tiles
- Build visualizer to show algorithms working in real-time



What is pathfinding?

- Problems related to finding a path between two or more points
- Graph algorithms
- Used in games, navigation apps, etc.

What is pathfinding?

A* is widely used in game programming

What is pathfinding?

Pathfinding in games is complicated

- Moving obstacles
- Moving goals
- Three dimensions

What is pathfinding?

Pathfinding in games needs to be fast

- Usually not concerned with finding the absolute best / most optimal path: just find one that's short enough
- A* is popular in part because it can do either depending on its configuration

What is pathfinding?

For this project....

- Mainly interested in A* and Dijkstra’s ability to guarantee the (or one of the) shortest path(s) between two points
- Look at A*’s ability to quickly find a path that is “short enough”

What does the program do?

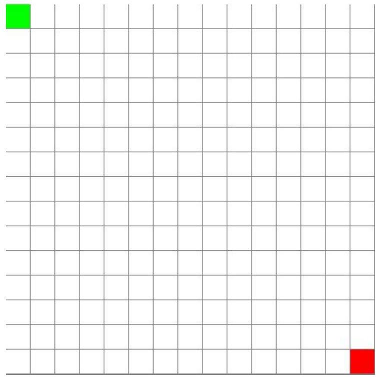
Pathfinding algorithm visualization

- Implementation of both pathfinding algorithms using **pygame**, a game library for Python
- Applied to the task of finding the (or one of the) shortest path(s) on a grid filled with random obstacle tiles

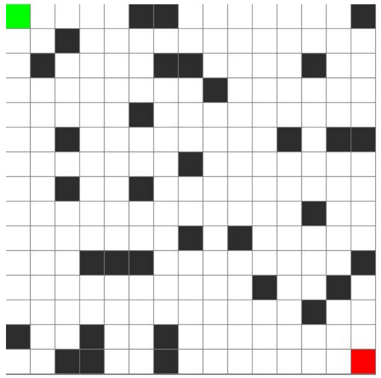
What does the program do?

Grid generation

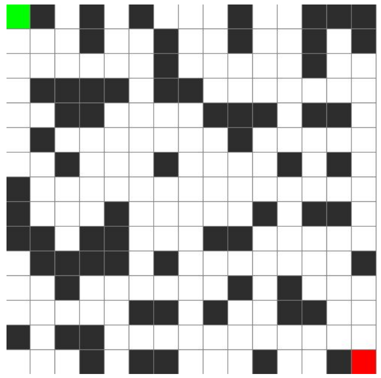
- Start and end tile locations same for all grids
- Randomly generate grids with a percentage of impassable obstacle tiles



0%



15%

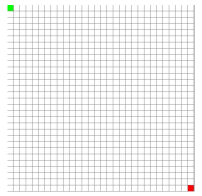
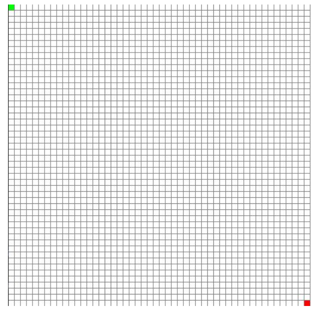
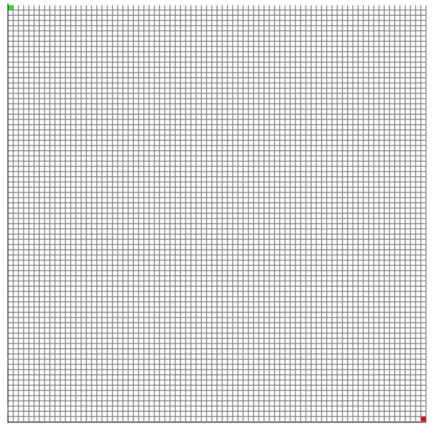


30%

What does the program do?

Performance Testing

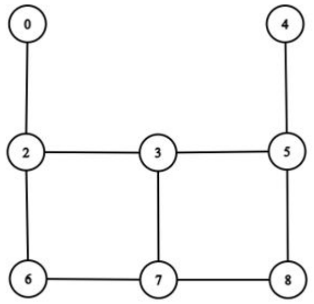
- Test runtime performance in terms of **number of tiles expanded by search**
- Series of grids of different sizes (30x30, 50x50, 80x80)
- Run each algorithm once on the same grid, throw out failed searches
- Average of 20 tests

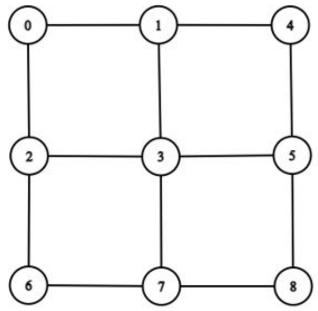




Pathfinding on a Grid

How is the grid structured?

- Tiles are nodes
- Each tile has an internal adjacency list of neighboring tiles
- The grid is an undirected graph
- Wall tiles are holes in the graph





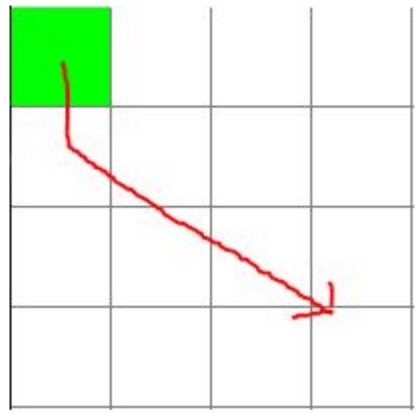
Pathfinding on a Grid

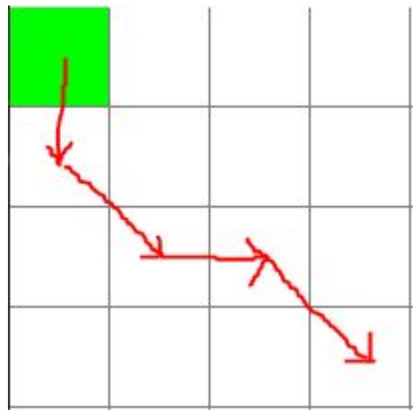
Jumping ahead a little bit....

- A* uses an estimate of the distance to the goal as part of its decision making process
- Weighted graph algorithm
- Make choices about how distance / movement cost are modeled on the grid

Pathfinding on a Grid

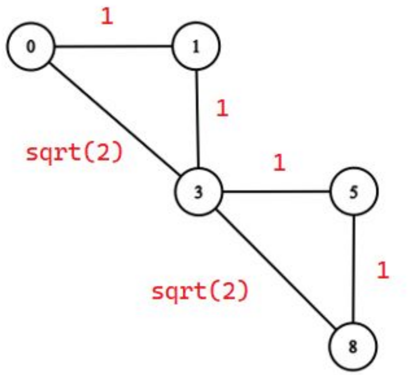
- What kind of movement is allowed?
- Game piece vs. free movement at any angle
- Any-angle movement: Euclidean distance

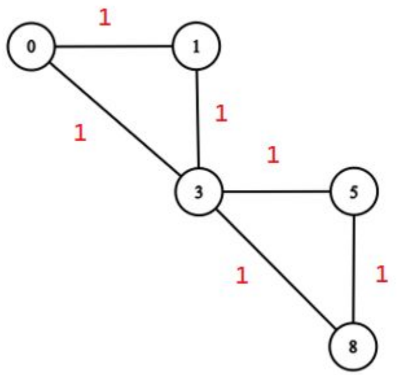




Pathfinding on a Grid

- Are diagonals allowed?
- Are we modeling real distance or just game-board movement cost?
- Real distance: octile distance
- Game-board movement w/ uniform diagonal cost: Chebyshev distance





Pathfinding on a Grid

The plan was….

- Game-board movement (simpler, more intuitive)
- Compare no diagonal movement and octile (diagonal movement allowed but more expensive)
- But… ran out of time
- Project just looks at 4-directional movement, no diagonals, uniform movement cost in each direction

Pathfinding on a Grid

Side note:

- Uniform movement cost makes Dijkstra’s perform like BFS
- Weighted graph algorithm on graph with uniform weights
- Kind of pointless! Uses a priority queue when priorities are equal: potentially expensive for no reason

Pathfinding on a Grid

Calculating distance

- City-block or **Manhattan distance**
- Each tile has an (x , y) coordinate
- Distance is just the absolute difference between the x and y values of the tile being considered to the end tile

Dijkstra’s Algorithm

History / Context

- Weighted graph search algorithm
- Conceived in 1956 by Edsger W. Dijkstra as an algorithm for finding the shortest path between two nodes
- Modern usage: finding the shortest paths from the start to multiple end nodes

Dijkstra’s Algorithm

Overview

- Prioritizes search expansion based on **running total cost from origin**
- Weighted graph algorithm, edges have associated costs
- Is considered a greedy algorithm, at every step just chooses the cheapest way outward
- Guarantees the shortest path

Dijkstra’s Algorithm

Data structures

- Table for costs
- Priority queue of nodes to expand

Implementation specific stuff

- Dictionary (hash table)
- Priority queue (binary heap)

Dijkstra’s Algorithm

1. Add start tile to queue

While the queue isn’t empty and you haven’t found the end:

2. Dequeue highest priority node
3. Examine its neighbors’ costs
4. If a neighbor hasn’t had its cost calculated yet or if it is cheaper to get to the neighbor from the current tile than from whatever way it was last examined, add the neighbor to the queue with priority = cost from the current tile
5. Update the neighbor’s cost if it was enqueued

A* Algorithm

History / Context

- Weighted graph search algorithm
- Published in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute
- Widely used in games, more sophisticated versions exist

A* Algorithm

Overview

- Can be seen as a direct extension of Dijkstra’s algorithm
- Dijkstra’s algorithm works by considering nodes by their calculated cost from the starting node
- A* builds on top of this by also incorporating the use of a **heuristic** to increase runtime performance
- Heuristic: approximation or other guiding function that helps with algorithm optimization

A* Algorithm

Overview

At each step, A* considers the node n with the smallest sum of:

> **g(n)** or g-cost : exact cost of the path from start to the node n
>
> **h(n)** or h-cost : estimated cost from n to the goal (heuristic)
>
> **g(n) + h(n) = f(n)** or f-cost

A* with h(n) = 0 is just Dijkstra's algorithm, since it will only be considering nodes by their cost from the start

A* Algorithm

Heuristic Accuracy / Effect on Performance

- More accurate heuristic function = less expanded tiles before reaching goal
- If the heuristic is exact, A*'s performance will be optimal (very fast), it won't expand unnecessarily, and it will guarantee the shortest path
- If the heuristic is an underestimate, A* will work slower but will still guarantee the shortest path
- If the heuristic is an overestimate, A* will get to the end faster, but it loses the guarantee of finding the shortest path

A* Algorithm

Data structures

- Table for costs (g-cost : cost from starting tile)
- Priority queue of nodes to expand

Implementation specific stuff

- Dictionary (hash table)
- Priority queue (binary heap)

A* Algorithm

1. Add start tile to queue

While the queue isn't empty and you haven't found the end:

2. Dequeue highest priority node
3. Examine its neighbors' g-costs
4. If a neighbor hasn't had its g-cost calculated yet or if it is cheaper to get to the neighbor from the current tile than from whatever way it was last examined…
5. Calculate the f-cost (g-cost plus the calculated h-cost from this neighbor to goal)
6. Enqueue with priority = f-cost
7. Update the neighbor's g-cost if it was enqueued

Code Demo #1

Testing Runtime Performance

Very, very rough worst-case analysis for both algorithms

Ignoring the costs associated with....

- Setting up the grid
- Building the graph (adjacency lists)
- Saving and retracing the path

.... so just the search runtime

Testing Runtime Performance

Python priority queue implementation uses a binary heap with **get()** and **put()** being worst-case **O(log(N))**

Considering runtime in terms of tiles expanded:

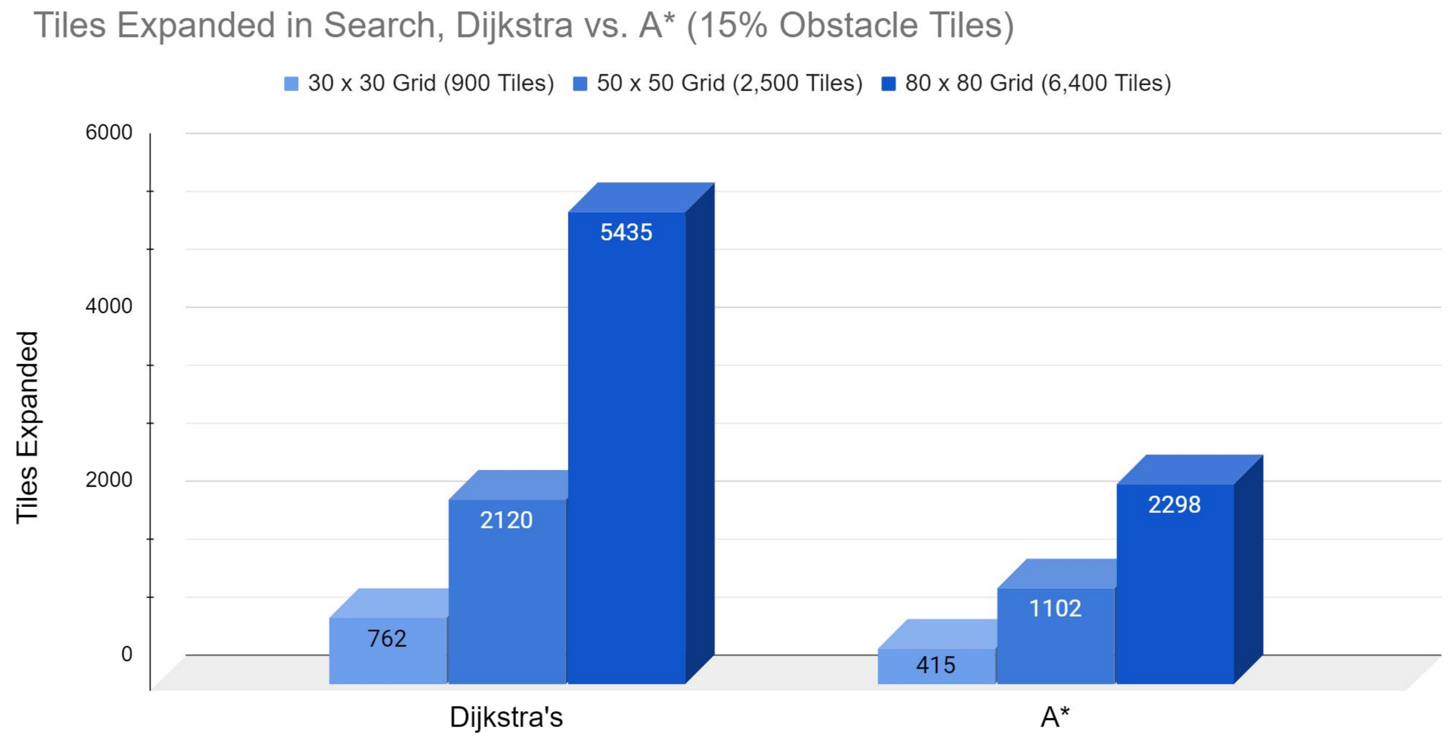
- N = number of passable tiles
- Worst-case: all tiles in grid are enqueued and dequeued
- O(N * (2log(N)))
- **O(Nlog(N))**

Testing Runtime Performance

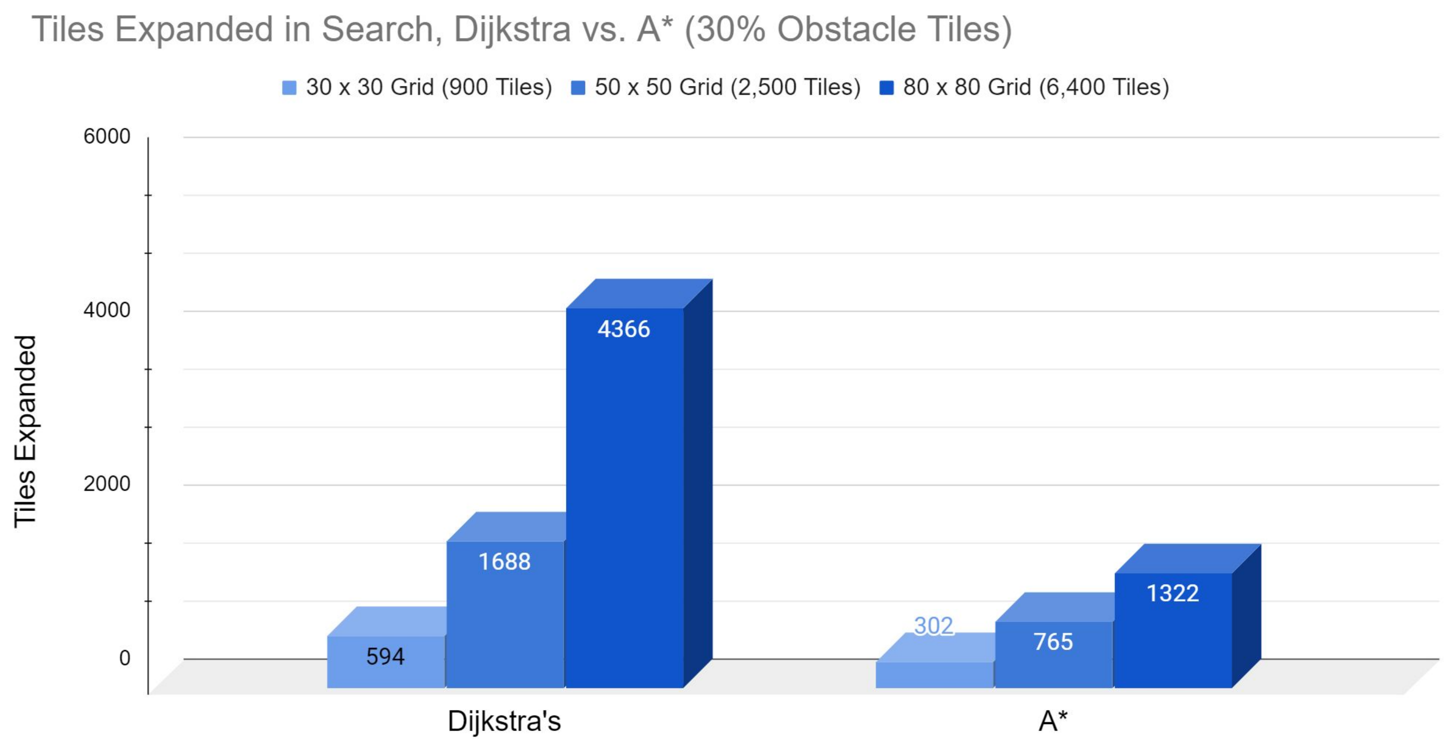
Average runtime to find shortest path

- Average of 20 tests
- Measured as number of tiles expanded in search
- Each algorithm run once on the same grid before generating a new one
- Failed searches thrown out
- Looked at grids with 15% obstacle tiles and 30% obstacle tiles at sizes 30 x 30, 50 x 50, 80 x 80

Testing Runtime Performance



Testing Runtime Performance



Testing Runtime Performance

For this problem (uniform cost grid, 4-directional movement), A* finds the end tile in less than half the time on average

- Holds for grids with 15% and 30% wall tiles
- Both algorithms perform better on graphs with more wall tiles, at least in this range (15-30%)
- More wall tiles = **more sparse graph**
- More sparse graph means less nodes to check….

Testing Runtime Performance

Testing behavior of A* depending on accuracy of heuristic

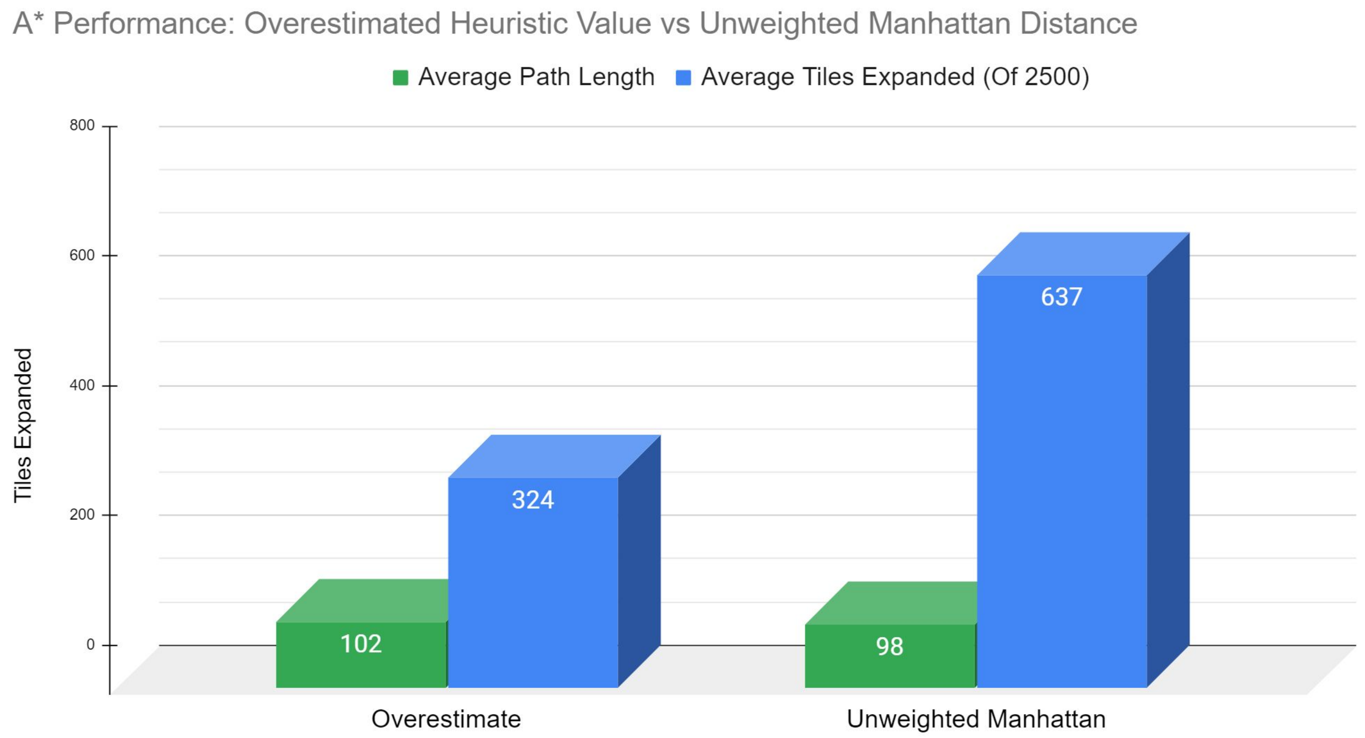
- Heuristic = Manhattan distance
- For non-empty grids, this heuristic is often a slight underestimate, but has potential to be exact
- Look at how A*'s performance changes depending on accuracy by **applying a weight to h(n)**

Testing Runtime Performance

Overestimate

- Does not guarantee shortest path, but finds “short enough” path
- Ran tests to compare average performance of the overestimate with the standard unweighted Manhattan distance heuristic
- Average of 20 tests on grids 50 x 50 (2500 tiles)
- 30% wall tiles
- **h(n) * 1.05**

Testing Runtime Performance

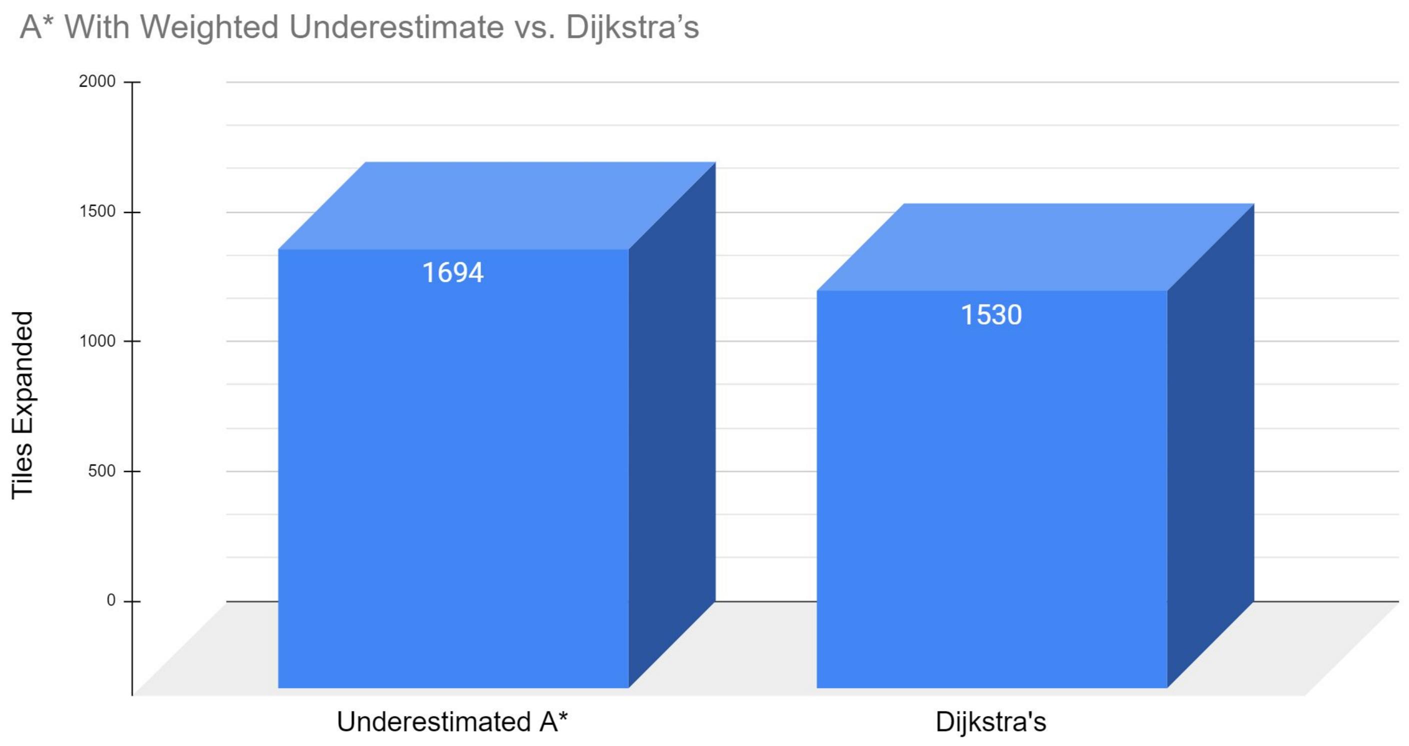


Testing Runtime Performance

Underestimate

- Guarantees shortest path, but if too far from actual distance, A* will have performance similar to Dijkstra’s
- Run tests to compare underestimate with Dijkstra’s
- Average of 20 tests on grids 50 x 50 (2500 tiles)
- 30% wall tiles
- **h(n) * 0.65**

Testing Runtime Performance



Code Demo #2