For finding the shortest path between two points, the algorithm of choice is usually Dijkstra’s algorithm. Invented by Edsger Dijkstra in 1956, it is guaranteed to find the optimal path between two points [INSERT REFERENCE]. Given a source node, *s,* and a destination node, *d,* Dijkstra’s algorithm works as follows:

1. First, we assign each node, *n*, an approximate distance from *s* to *n*. If *n* is *s*, then this distance will be 0; otherwise, it will be ∞.
2. Add each node to a list of *unvisited nodes*, which maintains which nodes have not yet been considered, and the approximate distance from *s* to each node.
3. Select the node, *n,* with the shortest approximate distance from *s* to *n*. For each node, *p*, that is connected to *n*, calculate the approximate distance from *s* to *p* by going through *n*. That is, if *e* is the length of the edge from *n* to *p*, then the distance from *s* to *p* is the distance from *s* to *n* plus *e*. If the approximate distance from *s* to *p* is less than the old approximation from *s* to *p*, replace it.
4. Remove *n* from the list of *unvisited nodes*.
5. Repeat steps 3-5 until one of two conditions is met: *d* has been removed from the *unvisited nodes* or the minimum approximate distance in the *unvisited nodes* is ∞ (indicating there is no path from *s* to *d*).

The only downside, however, is that Dijksta’s Algorithm runs in O(|E| + |V|log |V|) time. In our implementation, we represent the possible positions as an *n* by *m* grid, with each element constituting a node in our graph. Furthermore, each element has up to 8 connections with its neighboring cells, which represent the edges of our graph. Thus:

|V| = *n* x *m*

|E| ≈ 8|V|

O(8|V| + |V|log|V|).

If we have a 500 x 500 grid, it has an approximate time complexity of

8|500 x 500| + |500 x 500|log |500 x 500| = 6,482,892

Since our goal is to have multiple moving objects, it’s clear that we need a faster algorithm. That being said, it means we unfortunately have to trade optimality for speed. In many applications, however, such as video games, this is a fair and reasonable trade-off.

Thus, we turn our attention to A\*, a variant of Dijkstra’s algorithm that uses a heuristic to quickly find a good, but perhaps suboptimal, path from *s* to *d*. Like as is common with many heuristics, A\* uses a cost evaluation function in the form f(x) = g(x) + h(x), where f(x) is the estimated cost to get from *s* to *d* using the current path, g(x) is the cost from *s* to the current node, *n*, and h(x) is the estimated movement cost to get from *n* to *d*. Put in simpler terms, f(x) is how far you’ve traveled from the goal plus how far away from the goal you estimate that you are. The idea is instead of searching every node, you only search those which are most “promising”, a measure based on f(x).

[TODO: INSERT SECTION ON THE DIFFERENCE BETWEEN DIJKSTA’S AND A\*]

At first, we considered simply running A\* multiple times. However, we found that this technique wasn’t sufficient to avoid collisions because the objects may attempt to reach the same position at the same time. Thus, we modified our algorithm to have the objects avoid each other’s paths entirely. This had the unintentional result of rendering some paths impossible because the paths effectively become walls that cannot be crossed. Our next attempt proved to be more successful because instead of completely avoiding the paths, we instead took into account the time at which each object is at a particular point on its path. A path is defined as a series of points p[0… *t*], where each element in *p* is the position of the object at time *t*. Two objects will collide if their paths *p1* and *p2* share the same value at time *t*. Thus, if object will intersect the path of another object, we check if they will collide. If they don’t, it becomes a valid potential point on the path. If not, it is simply ignored, but may be reconsidered at a different value of *t*.

**The Algorithm**

Inputs: a grid of size *w* x *h*, a list of points to find paths

1. We need two priority queues – the *open list* (nodes which still need to be considered) and the *closed list* (nodes which do not need to be considered). Add the starting node, *start* to the open list. Set f(x), g(x), h(x), and *time* to be 0 for *start*.
2. If *open list* is empty, return – there is no path from *start* to *dest*
3. Remove the node, *n*, from *open list* that has the lowest cost (i.e. lowest f(x) value). We want to select the most promising node.
4. For each node, *q* that has an edge from *n* to *q* (i.e. the surrounding 8 nodes):
   1. If *q* is not a valid location, continue
   2. If *q* is on the *closed list*, continue
   3. If *q* is not walkable (meaning there is a wall there), continue
   4. Set *t* for *q* to be *t* for *n* +1 (since it’s one step forward in time)
   5. For each path *p* that runs through the position *q*
      1. If *q.t* is within the time interval of