Using Dijkstra's Algorithm for Score Calculation in a Tile-based Game

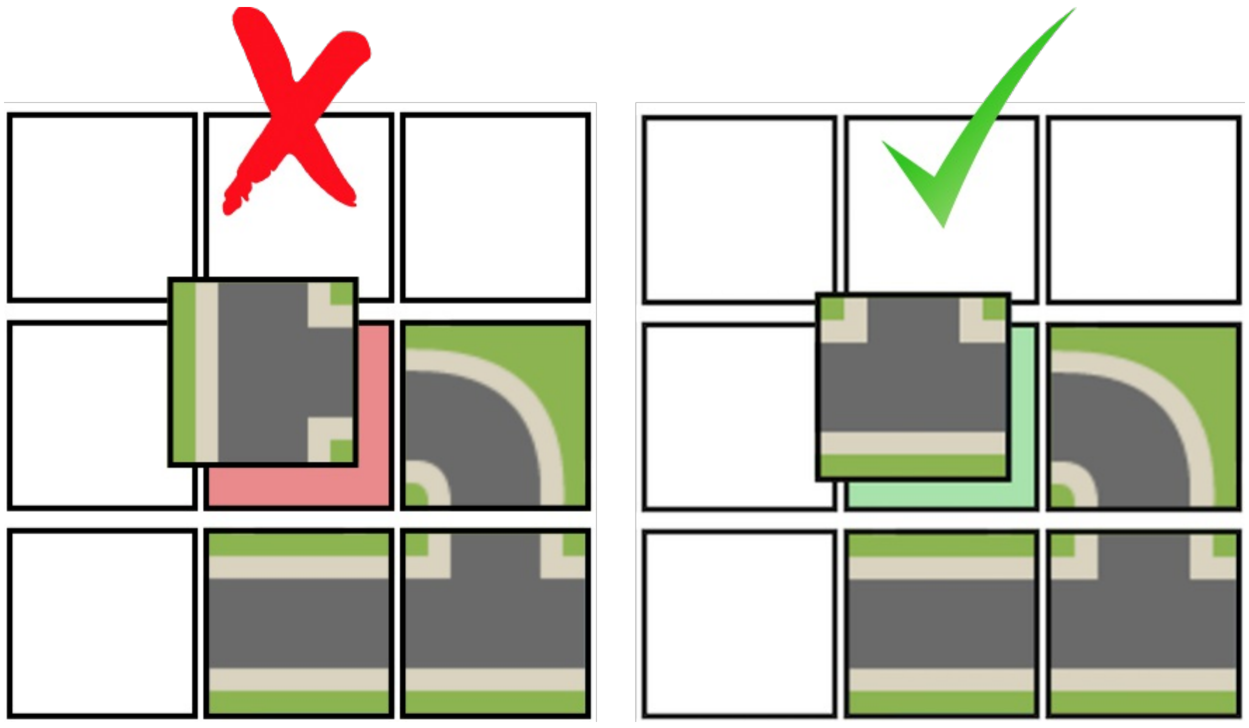
Recently, I've been building a multiplayer tile-laying game inspired by games like Carcassonne and Settlers of Catan. I'll soon have some other blog posts about the game and some of the technologies and architectures I'm building it with, but today I'm going to walkthrough and illustrate how I used Dijkstra's Algorithm to implement the scoring component of the game. Since there are already so many good resources explaining what Dijkstra's algorithm is and how it works, I'm writing this for somebody who already understands the basic concepts involved, and who would like to see a real-life implementation of it in a tile-based Javascript game.

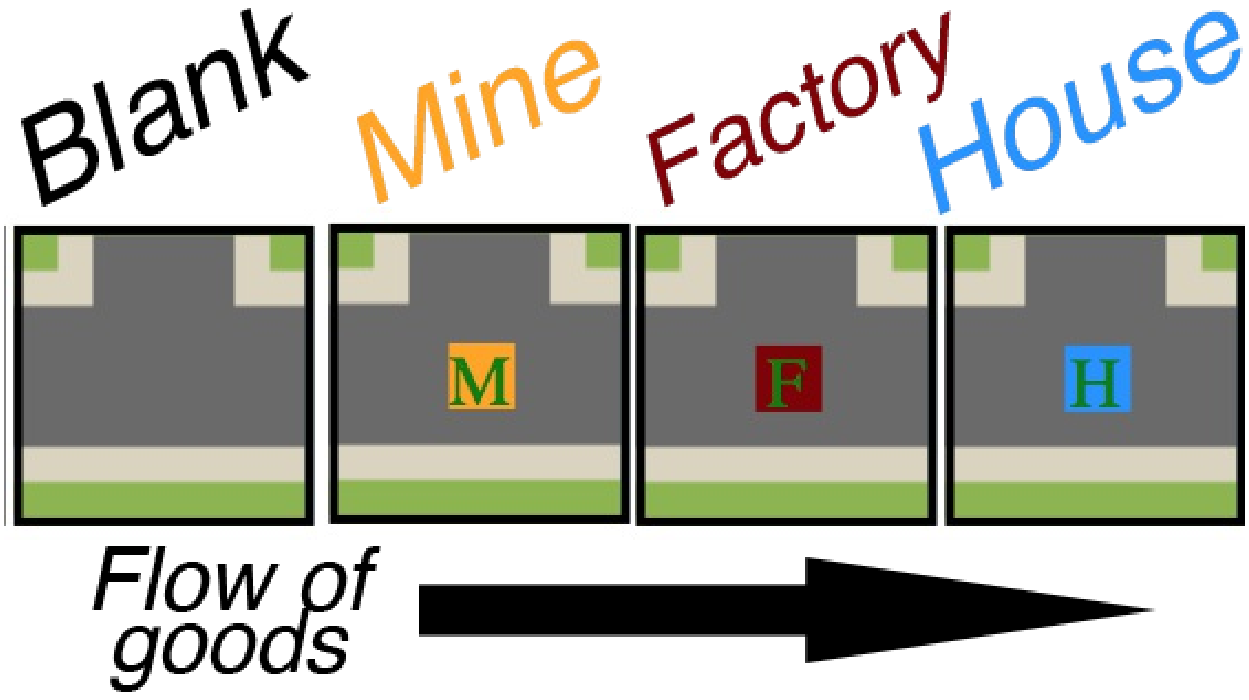
The Game — Rules and Objectives

To begin, I'm going to explain the game, so as to motivate why I even needed a pathfinding algorithm in the first place. If all you want to see is my implementation, feel free to skip to the next section.

Turns and Tile Placement

Like Carcassonne, the game begins with a single tile on the board. When it is each players' turn, they draw a random tile and must place it somewhere on the board. In order to place the tile, all of its edges must match with those surrounding it. Streets have to match up to streets, and grass has to match up with grass. Tiles can be rotated but not swapped out. So what a player will do on their turn is largely dependent on what tile they draw.

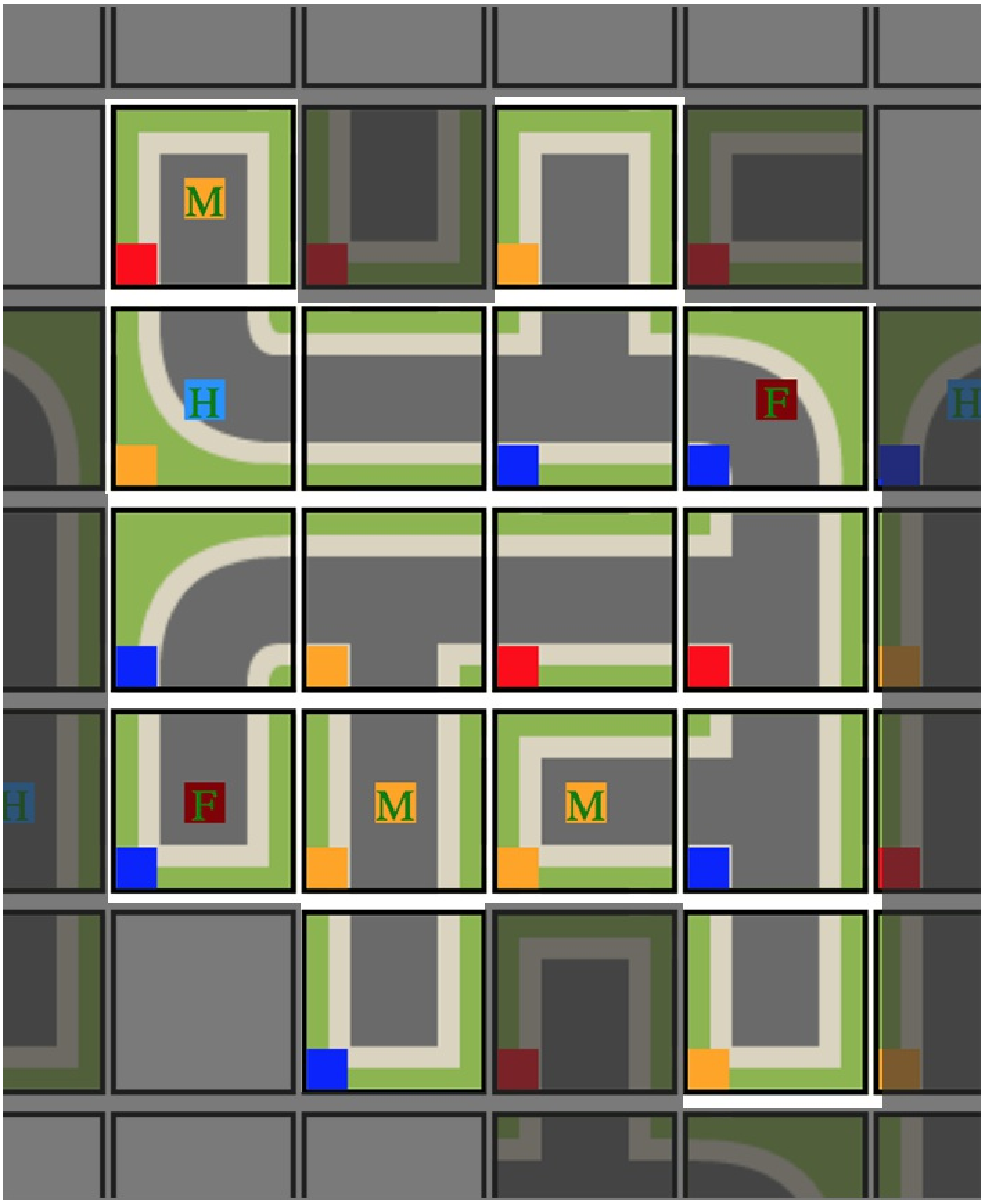




Types of Tiles

Tiles can either be a simple road tiles which are used to connect other tiles, or they can be *special tiles* which includes one of three game pieces on them: a mine, a factory, or a house. The basic logic of the game is as follows: **Mines provide factories with materials, factories turn these materials into goods, which they then ship to houses for consumption.**

Players *own* the tiles that they place, including any special tiles. This is indicated (at least right now in this mock-up stage) by little badges with the player's color on the bottom left of the placed tile. (See next image for example.) As tiles are laid and the board expands, paths are formed and eventually completed. Only after a path is completed is it scored, and players receive points based on the value of the tiles that they own.



Scoring the Game

A path is scored whenever it gets *completed*, meaning that there is no possibility for other tiles to be added to it: it becomes a closed system. Upon completion, this closed system is analyzed to determine the value of each of the tiles within it according to the following rules:

- Each tile on a completed path is automatically worth `1` point.
- Each mine *which is connected to a factory* is worth a number of points equal to the length of the shortest possible path between it and the factories that it supplies. If it supplies multiple factories, the distances of these paths are summed. A mine which is not connected to at least one factory is worth the default `1` point.
- Each factory *which is supplied by a mine* ***and*** *which is connected to a house* is worth a number of points equal to the length of the shortest possible path between it and the houses that it supplies **multiplied by** `2`. If a factory is neither supplied by a mine nor connected to at least one house, it is only worth the default `1` point.

Since the majority of the scoring is ultimately based on the length of the paths between the special tiles, it is important that these paths be calculated thoroughly. A player should not receive extra points for their path just because a random-walk algorithm was used and took the long-route to get there. This, along with the complexity of some of the rules of calculating the paths that we will see in the next section, underscore my decision to use a shortest path finding algorithm in implementing the scoring logic of the game.

But why Dijkstra and not A* or other popular shortest distance algorithms? I'll come back to this decision after filling out some details of what exactly I wanted the algorithm to do in my game.

Example

For this example, I am going to take the completed path that I already showed you above and walk through exactly what I want to happen when the values of the tiles are calculated as the game goes through the three rules I outlined above. As we step through the rules, I'll show illustrations and keep a tally. (For convenience, I'm not going to list the non-special tiles, because their values won't change after the 1st rule is applied, but keep in mind that they are there.)

Step 1

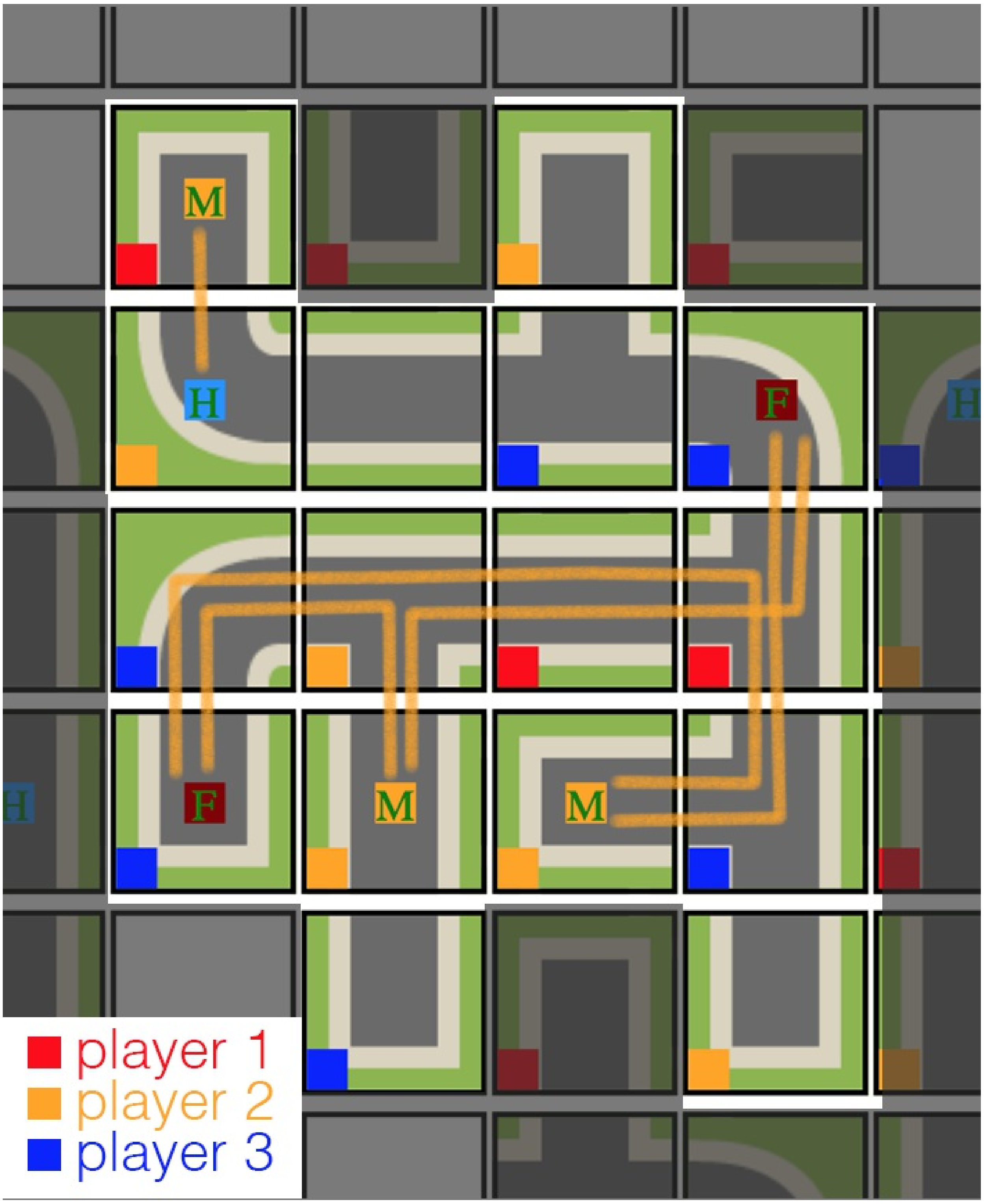
Step 1 is easy. Loop through all of the tiles on the completed path and assign a value of 1. This rewards players who used their tiles to help build and complete the path, even if they aren't reaping benefits of industry.

```
// special tiles from top left
// to bottom left of the image
values = {
  mine1 = 1, //owned by red
```

```
    house1 = 1, //owned by yellow
    factory1 = 1, //owned by blue
    factory2 = 1, // owned by blue
    mine2 = 1, // owned by yellow
    mine3 = 1 // owned by yellow
    // all other tiles = 1
  }
```

Step 2

Now we are coming to the pathfinding. We need to find the optimal valid paths from each mine to each factory that it can possibly supply. The following image shows what we would like our pathfinding algorithm to do. The orange lines are leading from the mines to the factories.



What's happening, in prose: Yellow owns two mines (`mine2` and `mine3`), and each mine has a free path to each of the two factories owned by blue (`factory1` and `factory2`). Good for Yellow! The mine on the right (`mine3`) has two valid paths, one of length `3` and one of length `6`. Keeping in mind that this mine has an initial value of `1` from the first scoring rule, this means that the new tile value for `mine3` is `1 + 3 + 6 = 10`. A pretty nice bonus! Similarly, Yellow's mine of the left also has two valid paths to factories, one of length `3` and one of length `4`, bringing this mine tile's total to `1 + 3 + 4 = 8`.

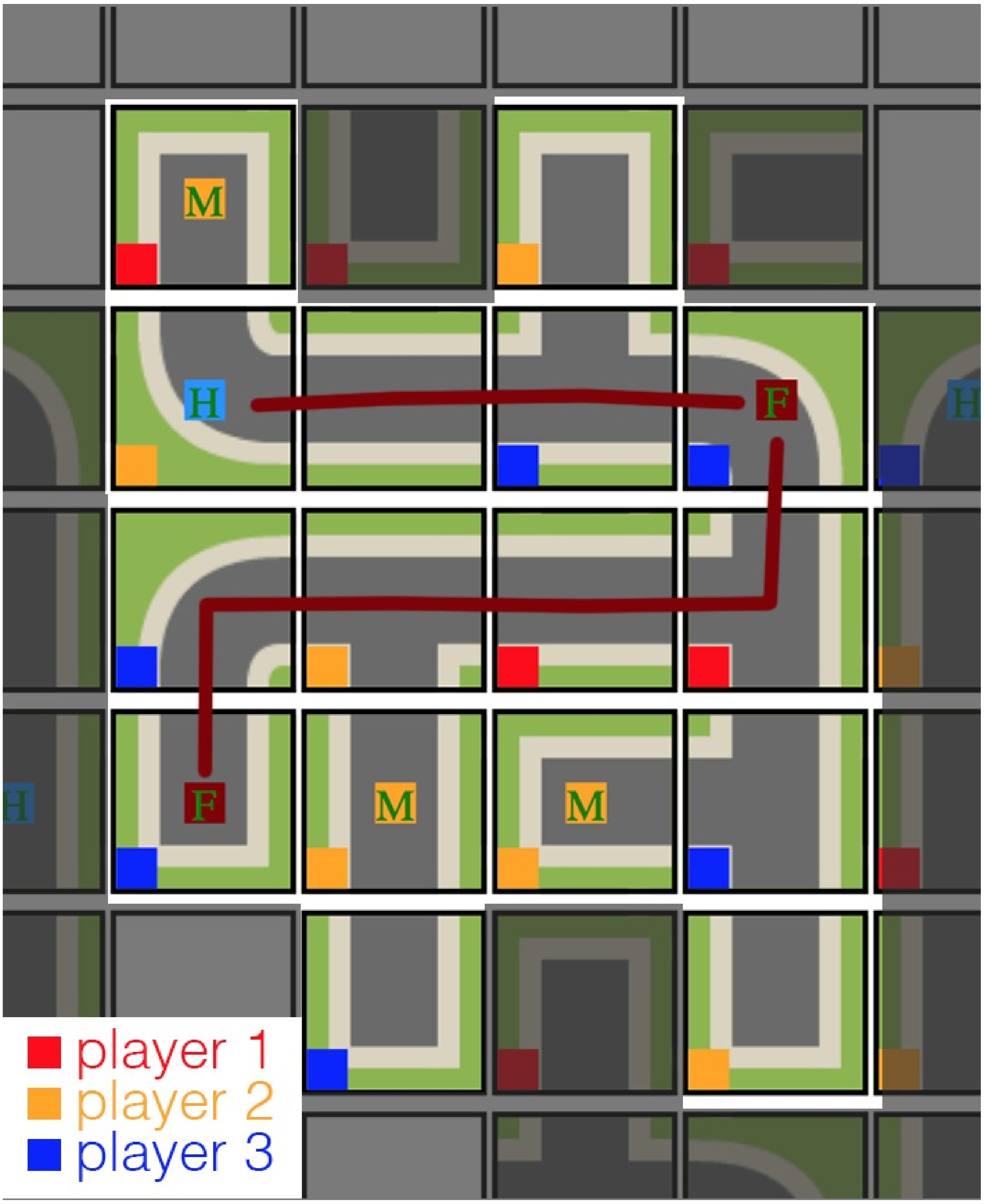
So while it seems that Yellow is reaping a lot of points this turn, we notice that red also owns a mine in the upper left of the path (`mine1`). Unfortunately for red, however, Yellow's house (`house1`) is blocking it from connecting to a factory, causing Red's mine to only be worth the default amount of `1` point. In order to add an element of strategic placement and defensive blocking, we want special tiles to only be able to serve as end points and for it to be invalid for paths to pass through them. Sorry Red…

So after the calculations from step 2, the tile values look like this:

```
// after step 2: mine->factory
values = {
  mine1 = 1, //owned by red
  house1 = 1, //owned by yellow
  factory1 = 1, //owned by blue
  factory2 = 1, // owned by blue
  mine2 = 1 + 3 + 4, // total 8; owned by yellow
  mine3 = 1 + 3 + 6 // total 10; owned by yellow
  // all other tiles = 1
}
```

Step 3

After all of the paths from the mines are calculated and scored, the game then moves on to the factories. Remember, only factories which have been supplied by mines are even eligible for shipping their goods to houses. In our example, however, both factories have been supplied, so both will attempt to draw paths to houses on path. Lucky for Blue, they own both of the factories, and with the 2x multiplier, they should expect a big payout. Let's see what happens:



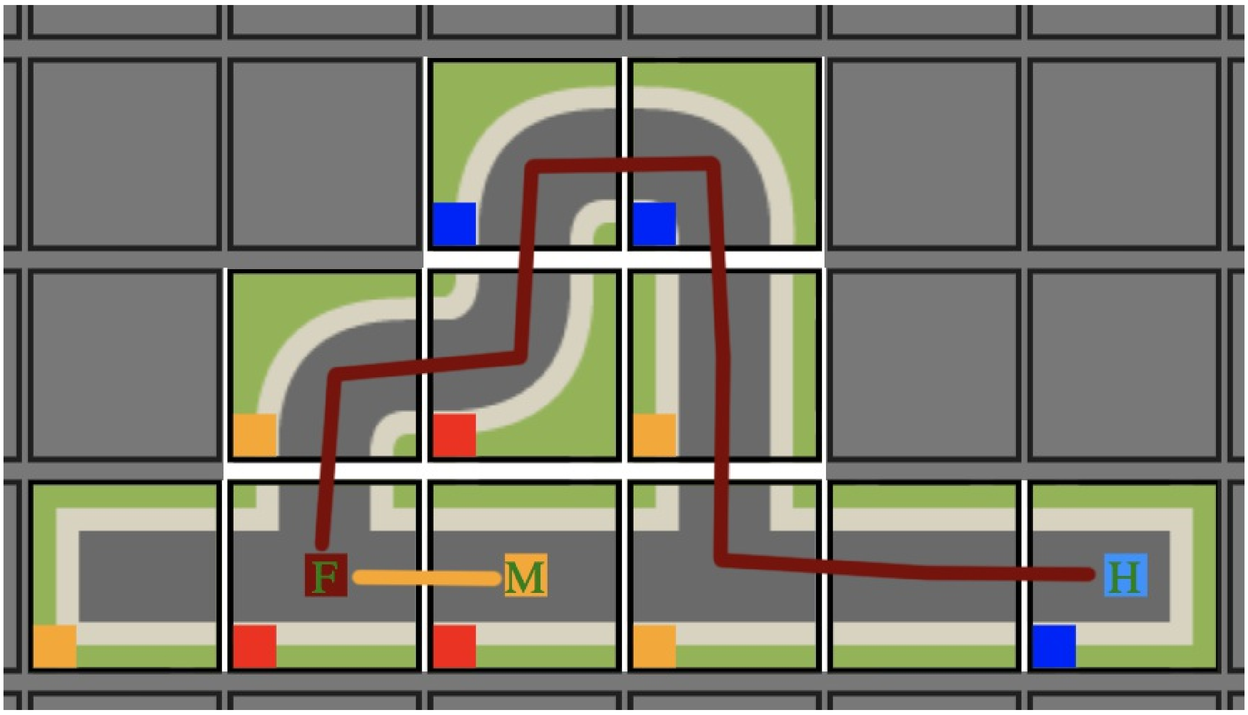
The upper factory (`factory1`) has a nice straight little jot to `house1` with a path length of `3` , making the value of `factory1` `1 + (3*2) = 7` . Unfortunately, Blue didn't choose the best layout, because `factory1` is blocking the only possible path from `factory2` to `house1` , and thus misses quite a few points. Better luck next time, Blue!

So after Step 3:

```
// after step 3: factory->house
values = {
  mine1 = 1, //owned by red
  house1 = 1, //owned by yellow
  factory1 = 1 + (3*2), // total 7; owned by blue
  factory2 = 1, // owned by blue
  mine2 = 1 + 3 + 4, // total 8; owned by yellow
  mine3 = 1 + 3 + 6 // total 10; owned by yellow
  // all other tiles = 1
}
```

If it seems that the restriction on movement through other special tiles is frustrating, one only need to look at one of the strategic possibilities depicted in the next image to understand the positive complexity they add to the game. Players must think ahead about how they want to plan their routes, all while competing with the plans of other players and against the luck of the tile draw.

A clever layout to maximize points (17 for the factory!):



So now that I've shown how the game's scoring mechanic relies on a pathfinding algorithm, I'll explain my choice to use Dijkstra's Algorithm over the several other popular shortest path finding algorithms available.

Why Dijkstra and not A*?

Quick Note on Terminology: In the literature about pathfinding, the concepts of **graphs**,

nodes, and **edges** are used. A graph is an abstract data structure used to analyze the connections between objects. A graph consists of nodes which are connected to each other by edges. In the example of our game, the closed system of a completed path is a graph, each tile within it is a node, and the connections between the tiles are edges.

From the previous section, we saw that the pathfinding algorithm should be able to accomplish the following things:

- Find the shortest possible paths from **one** starting node to **all** possible ending nodes. (e.g. from one mine to all factories).
- Ensure that special tiles only serve as endpoints on paths and cannot be traveled through.

In looking through the possible shortest path finding algorithms which solve these problems, two stuck out as good candidates: the A* Search algorithm, and Dijkstra's algorithm.

A* seems to be a crowd favorite, at least for certain use-cases. If you poke around on game-development forums and blogs, you see A* touted everywhere, due mainly to its increased performance over Dijkstra in searching for the shortest path between two given nodes. Whereas Dijkstra's algorithm needs to analyze every node on a graph, A* uses heuristics (basically, educated guesses) to find the shortest path between a beginning and an end node without needing to check every possible node on on entire graph. One stackexchange user writes: "When it comes to pathfinding, A* is pretty much the golden ticket that everyone uses."

A* is basically an extended version of Dijkstra's algorithm that uses a system of educated guesses to get to an end node while analyzing as few nodes on the graph as possible, and thus taking less time. This is useful for situations, like pathfinding in a RTS, where one has a clear start node and end node. In our example, however, where one mine might connect to several different factories, we would need to run the A* algorithm several times from the same starting node, which could cause to the algorithm to retrace its steps several times—an inefficiency we would like to avoid if possible.

In debating between using A* and Dijkstra, I opted for Dijkstra because I knew that I would need to reach multiple end nodes from one starting node, and that if I were using A*, I would need to run the algorithm multiple times in order to accomplish this, potentially calculating parts of the same route over and over again. Because Dijkstra's algorithm—at least, in my implementation of it—can calculate the shortest distance to every node in a graph without needing to have a target or goal node up-front, it seems to make the most sense for my game, wherein one starting node can (and likely will) have multiple semi-overlapping paths to multiple end nodes.

The Code — Implementation in Javascript

Ok, now that we understand how the game works, what goals we want the algorithm to accomplish, and why I choose Dijkstra's algorithm to accomplish these goals, we can start to look at the code. Everything is written in Javascript (ECMAScript 2016). (Note: Throughout the post, I've hidden a lot of the project-specific implementation details or simplified things in cases where it seemed superfluous or confusing. If you want to see the full code all in one place, you can find it here.)

The `nodesOnPath` Object

The first thing to understand is the information the rest of the game provides to the pathfinding/scoring mechanism. Every time a player places a tile and ends their turn, the game checks to see if placed tile is on a completed path. (The specifics of this will be for another blog post.) If that path is completed, the game passes an object containing an array called `nodesOnPath` to the scoring mechanism. This array is populated with objects representing each tile on the completed path. A typical `nodesOnPath` array looks like this:

```
// nodesOnPath - array of tileObjects
[
  // tileObject
  {
    id: 10, // unique id of tile
    neighbors: [7, 11, 4], // unique ids of neighboring tiles
    type: "factory", // tile type "mine", "factory", "house", or null
    playedBy: 1, // unique id of player who owns tile
    x: 4, // x coord
    y: 7 // y coord
  },
  // another tileObject
  {
    id: 7,
    neighbors: [8, 10],
    type: null,
    // ...
  }
]
```

This array is passed as the only argument to a function called `calcCompletedPathScore` which is responsible for taking the information contained in this array, and turning it into point amounts awarded to each player who owns any tiles on the newly completed path. This is accomplished in three steps, each corresponding to one of the steps outlined in Scoring Rules section above.

```
calcCompletedPathScore: function(nodesOnPath) {
  // calculates the value of a completed path
  // for the various players
  // step 1: prepare nodesOnPath and set value = 1 for each tile
  // ...

  // steps 2 + 3: use Dijkstra to increase value of special tiles
  // ...

  // final: give player points equal to the value of each tile they own
  // ...
}
```

Scoring: Step 1

Before doing anything related to pathfinding and Dijkstra, the function does a little set-up and sets some initial values. The way that points will be awarded to each player is simple: each tile is given a `value` property which is an integer number of points that the player who owns the tile will receive at the end of the scoring calculation. In accordance with Step 1 of our scoring rules, each tile's default `value` property is set to `1`, so that each tile on the path is automatically worth `1` point.

```
calcCompletedPathScore: function(nodesOnPath) {
  // calculates the value of a completed path
  // for the various players

  // used to keep track of which special tiles,
  // if any, are on the path
  const specialTiles = {};

  // for each node on path...
  for (let i = 0; i < nodesOnPath.length; i++) {

    // add initial value of tile for score
    nodesOnPath[i].value = 1;

    // if tile is a special tile
    if (nodesOnPath[i].type) {

      // set loaded and supplying properties
      nodesOnPath[i].loaded = false;
      nodesOnPath[i].supplying = false;

      // populate specialTiles object
      if (!specialTiles[nodesOnPath[i].type]) {
        specialTiles[nodesOnPath[i].type] = [];
      }
      specialTiles[nodesOnPath[i].type].push(nodesOnPath[i]);
    }
  }

  // steps 2 + 3: use Dijkstra to increase value of special tiles
  // ...

  // final: give player points equal to the value of each tile they own
  // ...
}
```

We also create a `specialTiles` object to be used for keeping track of which special tiles we need to pay attention to on this path, as well as set the `loaded` and `supplying` properties of each tile, which are used later in Step 3 of the scoring rules, when factories must be 'loaded' by mines before they can provide houses with goods.

If the completed path were to contain no special tiles, the scoring process would end here. The final step of the `calcCompletedPathScore` function loops through the tiles objects in `nodesOnPath` and adds the `value` property to the score of whichever player's id corresponds with the `playedBy` property. Without any special tiles, this would always been `1` and each player would receive the same number of points as the number of tiles they own on the newly completed path.

```
calcCompletedPathScore: function(nodesOnPath) {

  // step 1: prepare nodesOnPath and set value = 1 for each tile
  //...

  // steps 2 + 3: use Dijkstra to increase value of special tiles
  //...

  // for each node on the path that belongs to player,
  // give that player the value of the tile
  for (let i = 0; i < nodesOnPath.length; i++) {
    // playerObject has a method addPoints(int)
    this.players[nodesOnPath[i].playedBy].addPoints(nodesOnPath[i].value);
  }

}
```

It's the stuff that happens in Steps 2 and 3 of this process which are really of interest to us here in this article, however, so let's start taking a look at the code which actually implements Dijkstra's algorithm and assigns value to the special mine, factory, and house tiles.

Steps 2 and 3: Pathfinding

(Note: My implementation of Dijkstra's algorithm was heavily inspired by that of Stella Chung's, which she documents in a blogpost here.)

The following is the component of the `calcCompletedPathScore` function which deals with pathfinding. I left out most of the nitty-gritty details of validity checking for clarity, but the structure should be clear. You'll notice that there are two main functions which this component relies on which we haven't looked at yet: `getDijkstraTree` and `getOptimalPathFromTree`. We'll look at the actual functions for each of these more in a second, but for now, just pay attention to where they are being called in the `calcCompletedPathScore` function.

```
calcCompletedPathScore: function(nodesOnPath) {

  // step 1: prepare nodesOnPath and set value = 1 for each tile
  const specialTiles = {};
  // populate specialTiles...
  // ...

  // steps 2 + 3: use Dijkstra to increase value of special tiles
  const calcSpecialTileBonus = function(specialTiles, startType, endType, bonusAmou

    // check if there is at least one of each startType and endType on path...
    // if startType must be loaded, check if it is...
    // for each instance of startType on path...

    let startId = // id of tile to be start node
    let dijkstraTree = getDijkstraTree(nodesOnPath, startId);

    // for each valid endType on path...

    let finishId = // id of tile to be end node
    let optimalPath = getOptimalPathFromTree(dijkstraTree, finishId);

    // check path validity...

    // if valid:
    // add path length to value of tile
    nodeOnPath[startId].value += optimalPath.length * bonusAmount;

    // if invalid:
    // no valid path - unscored
    return
  }

  // mine->factory, bonus: 1, mustBeLoaded = false
  calcSpecialTileBonus(specialTiles, 'mine', 'factory', 1, false);
  // factory->house, bonus: 2, mustBeLoaded = true
  calcSpecialTileBonus(specialTiles, 'factory', 'house', 2, true);


  // final: give player points equal to the value of each tile they own
  // ...

}
```

For each valid start node, we call a function `getDijkstraTree` which will return a data structure that contains the shortest possible paths from the start node to every node on the graph. You'll notice that we only call this function *once* per start node. This function is where all the heavy lifting is done, so we want to call it as few times as possible. Then, once the `dijkstraTree` has been built, we call a function `getOptimalPathFromTree` for each valid end node. This function simply looks up the path between the start node and the given end node in the already built `dijkstraTree` object, and stores it as `optimalPath`. After checking for validity, the length of `optimalPath` is then used to set the value of the tile, which will later be used to assign points to the player.

We wrap all of this up into a function called `calcSpecialTileBonus` and then call it twice, once with `'mine'` and `'factory'` as the startTypes and endTypes, and once with `'factory'` and `'house'`. The second time, you'll notice that we include in the arguments a flag that tells us to check if `loaded == true` and the modifier `2` for calculating the bonus. It seemed like a good idea to abstract this rather than write it twice because 90% of the code would be shared between the two cases.

Implementing Dijkstra's Algorithm

Let's turn now to the actual implementation of the algorithm and look at both functions `getDijkstraTree`, `getOptimalPathFromTree`, as well as the helper functions which support them.

For Dijkstra's algorithm, we usually want to keep track of three things:

- the cost of getting to each node from the start node
- the chain of previous or parent nodes used in traversing the graph
- a list of unvisited or unprocessed nodes

My implementation, following Stella Chung, uses a list of processed nodes instead of a list of unprocessed nodes, but the end result is the same.

Following the algorithm definition on the Wikipedia page, we first create our lists, set the cost of the start node to `0` and the unknown costs of the rest of the nodes on the graph to `Infinity`:

```
const getDijkstraTree = function(path, startId) {
  // build a tree of costs and parents from startId
  // to every other node on the path

  // create initial objects and lists
  const costs = {};
  const parents = {};
  const processed = [];
  // set start cost = 0
  costs[startId] = 0;

  // set tentative cost for all other nodes to Infinity
  for (let i = 0; i < path.length; i++) {
    if (path[i].id != startId) {
      costs[path[i].id] = Infinity;
    }
  }

  // currentNode = start node
  let currentNode = getLowestNode(costs, processed);

  //...
};
```

The helper function `getLowestNode` on the last line above returns the lowest cost, unprocessed node in the graph. This way, the algorithm is always choosing the optimal node as the next one to process. The first time it is run, it will return the start node, of course, because it is the only non-Infinity node in the `costs` object.

```
const getLowestNode = function(costs, processed) {
  return Object.keys(costs).reduce((lowest, node) => {
    if (lowest === null || costs[node] < costs[lowest]) {
      if (!processed.includes(node)) {
        lowest = node;
      }
    }
    return lowest;
  }, null);
};
```

In the rest of `getDijkstraTree`, we perform steps 3–7 of Wikipedia's algorithm definition.

For each node, we get its current cost (which will be the total thus-far cost required to reach it from the start node) and the costs of its neighbors. Since our graph is just a grid of connected tiles, we use `1` for the edge cost for connections to blank tiles and `1001` for connections to special tiles. (The game has a limited number of tiles and turns and always will end before a blank path longer than 1000 tiles can be created, so this number is safe.) This ensures that the lowest cost path will always be through blank tiles, and thus implements the rule that paths are not allowed to travel through special tiles.

```
const getDijkstraTree = function(path, startId) {
  //...
```

```
    // currentNode = start node
    let currentNode = getLowestNode(costs, processed);

    // as long as there are nodes which are not in processed...
    while (currentNode) {
      let cost = costs[currentNode]; // cost of current node
      let children = getNodeById(path, currentNode).neighbors;
      let childrenCosts = getCostsOfNeighbors(path, children); // 1 or 1001

      for (let n in childrenCosts) {
        let newCost = cost + childrenCosts[n]; // cost of current node + travel
        if (costs[n] > newCost) {
          costs[n] = newCost;
          parents[n] = currentNode;
        }
      }

      // add currentNode to processed list after processing
      processed.push(currentNode);

      // get lowest, unprocessed node
      currentNode = getLowestNode(costs, processed);
    }

    // return path, costs, and parents for use with getOptimalPathFromTree
    return {path, costs, parents};
  };
```

(Note: `getNodeById` and `getCostsOfNeighbors` are just helper functions which respectively use the node's unique id to access its `neighbors` property and assign a cost of either `1` or `1001` depending on whether the neighbor is a blank or special tile.)

Once we have the costs of the current node and all of its neighbors, we tentatively add the cost of the current node to the cost of each of its neighbors, and check whether the currently stored lowest cost for each neighbor node is higher than the tentative one we just calculated. If the already stored cost *is* higher, that means we just found a less-costly way to reach this node from the start node, so we store that instead and record the current node as the 'parent' of the neighbor node we just updated.

Once all the neighbors of the current node have been checked out, we then mark the current node as processed, meaning that it will never be visited again, and get a new current node: whichever has the lowest cost from start and has yet to be processed. This loop is completed until every node on the graph has been processed. What we are left with are two objects: `costs` and `parents`, which we return and designate as `dijkstraTree` in our `calcCompletedPathScore` function.

Now that we have a calculated tree containing the costs to each node from the start node, we can use it to find the optimal path to each of the end nodes that we are interested in by using `getOptimalPathFromTree`. We pass it the tree that we calculated in `getDijkstraTree` as well as the unique id of the end node that we are interested in finding the optimal path to.

```
const getOptimalPathFromTree = function(tree, finishId) {

  // return array of actual nodes on the optimal path
  let optimalPath = [getNodeById(tree.path, finishId)];
  let parent = tree.parents[finishId];
  while (parent) {
    optimalPath.push(getNodeById(tree.path, parent));
    parent = tree.parents[parent];
  }
  optimalPath.reverse();

  // return int of the target Score
  // used to check validity. if >2000,
  // then invalid
  const targetScore = tree.costs[finishId];

  return {optimalPath, targetScore};
};
```

The function starts with the end node, gets its parent node, and pushes it to an an array `optimalPath`. It then gets the parent node of this node, and so forth, until it retraces its way back to the start node, which has no parent. In doing this, it recreates the path taken from the start node to the end node. It returns this path, along with the final score of this final node, designated as `targetScore`. Because we set connections to special tiles to be `1001`, any targetScore higher than `2000` must indicate a path that passes through another special tile, and thus is invalid. We check for this in `calcCompletedPathScore`.

```
const calcSpecialTileBonus = function(specialTiles, startType, endType, bonusAmount
  // ...

  // check if there is at least one of each startType and endType on path...
  // if startType must be loaded, check if it is...
  // for each instance of startType on path...

  let startId = ;// id of tile to be start node
  let dijkstraTree = getDijkstraTree(nodesOnPath, startId);

  // for each valid endType on path...

  let finishId = // id of tile to be end node
  let optimalPath = getOptimalPathFromTree(dijkstraTree, finishId);

  // check path validity...

  // if greater than 2000, then invalid
```

```
      if (optimalPath.targetScore < 2000) {
      // add path length to value of tile
      nodeOnPath[startId].value += optimalPath.length * bonusAmount;
    }

    //...
  };
```

If the optimal path returned by `getOptimalPathFromTree` is valid, we then use the length of this path (along with the `*2` modifier in the case of factories and houses) to change the value of the special tiles on the completed path, and thus to allow proper score calculation based on the rules we set out to implement.

Conclusion

I hope that you were able to see from this example what it looks like to implement Dijkstra's (or any other shortest path finding algorithm) in a grid- or tile-based game like the one I've been developing as well as some of the design decisions that went into choosing this algorithm. If I were building a more dynamic, performance intensive game like a real-time strategy or tower defense game, I would probably use A* or some version of it.

Although I tried to keep my mind bent towards efficiency while building this, I'm sure there are plenty of places where it could be improved. Many suggest that Dijkstra's algorithm should be implemented with a Fibonacci heap, but due to the relatively small sizes of the graphs that my game is capable of generating, it seemed to be far more trouble than it would be worth to implement this. But perhaps for a future blog post!

If there is anything that I missed or could improve, please reach out to me at jbierfeldt@gmail.com and let me know!

Inspiration and Resources:

- https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- https://en.wikipedia.org/wiki/A-star_search_algorithm
- https://hackernoon.com/how-to-implement-dijkstras-algorithm-in-javascript-abdfd1702d04
- https://en.wikipedia.org/wiki/Maze_solving_algorithm
- https://gamedev.stackexchange.com/questions/1/what-path-finding-algorithms-are-there
- https://mitpress.mit.edu/books/introduction-algorithms