Graphs - Pathfinding

Graphs

- Revisit: Lets take a quick look at different types of graphs that we had talked about earlier.

- Directed graphs vs undirected:
  - Undirected can represent friendships.
  - Directed can represent “following”, task dependencies, preferences, road network, etc.

Graphs

- Acyclic and cyclic (directed or undirected):

  - Directed acyclic graphs aka *DAG*.
  - In certain situations, we do not want cyclic graphs.
    - For e.g.: build dependencies (or in general, task dependencies).
  - Version histories, CPU instruction scheduling, computing SQL query execution plan (for example, Apache Spark builds a DAG for queries).

  - Note: An undirected graph has a cycle if u can come back to an already visited node without traversing an edge twice (i.e., without retracing ur path).

Graphs

- Graphs with and without weights (weighted vs unweighted):
    - Weighted graphs are used for pathfinding (like the ones we use for planning our routes).
    - Unweighted graphs will give u distance in terms of “number of hops” between two nodes, or social (or professional) distance between two people in a social/professional network.
        - E.g.: 2nd degree connection in LinkedIn.

Graphs

- Trees:

  - These are a subset of graphs.
  - Are acyclic
  - All nodes are connected (by one path only).

  - Different types:

    - Binary tree

    - Ternary tree

    - And…

Graphs

- Trees:

  - These are a subset of graphs.
  - Are acyclic
  - All nodes are connected (by one path only).

  - Different types:

    - Binary tree

    - Ternary tree

    - Spanning tree (of a graph)

      - This tree contains all nodes of a graph, but not necessarily all edges.

      - Think of it as an acyclic sub graph of the original graph.

      - A given graph may have multiple spanning trees.

        - Minimum spanning tree (aka MST) is a spanning tree with the least cost(cost refers to the edge cost in the MST).

Pathfinding

- BFS
  - Breadth first search
  - Spreads out from a given source … kinda like a wave.
  - Uses:
    - ?

Pathfinding

- BFS
  - Breadth first search

  - Spreads out from a given source … kinda like a wave.

  - Uses:
    - Could use to model spread of a virus
    - Shortest path between nodes.
    - Social distance (the online kind ☺, like in Facebook or LinkedIn)
      - Aka degrees of separation.

Pathfinding

- DFS
  - Depth first search

  - Start from a given source and go deep
    - i.e., pick one neighbor and go down that path as far as u can go.
    - We will talk about this some more in a few minutes.

  - Uses:
    - ?

Pathfinding

- DFS
  - Depth first search

  - Start from a given source and go deep
    - i.e., pick one neighbor and go down that path as far as u can go.
    - We will talk about this some more in a few minutes.

  - Uses:
    - Finding a route in a maze.
    - Finding a path between a given source and a given destination vertex.

Pathfinding

- Single source shortest path (SSSP)

  - Finds the shortest path between one given source node and all other nodes.

  - Dijkstra’s (u cannot have negative edge weights)
  - Bellman-Ford (edge weights can be negative)

  - Uses:
    - ?

Pathfinding

- Single source shortest path (SSSP)

  - Finds the shortest path between one given source node and all other nodes.

  - Dijkstra’s (u cannot have negative edge weights)
  - Bellman-Ford (edge weights can be negative)

  - Uses:
    - Routes from a hospital (or some emergency center) / police station to various places in the city (incident handling).

Pathfinding

- Variation of SSSP is finding shortest path between a single source, single destination
    - A* algorithm
    - Uses:
        - ?

Pathfinding

- Variation of SSSP is finding shortest path between a single source, single destination
  - A* algorithm
  - Uses:
    - Maps used by us for finding routes.
    - Video games (where pathfinding is needed).

Pathfinding

- All sources shortest path (ASSP)

  - Finds the shortest paths between all source nodes and all other nodes.

  - Floyd-Warshall algorithm.
  - Johnson’s algorithm

  - Uses:
    - ?

Pathfinding

- All sources shortest path (ASSP)

  - Finds the shortest paths between all source nodes and all other nodes.

  - Floyd-Warshall algorithm.
  - Johnson’s algorithm

  - Uses:
    - Mileage charts.
    - Cache of alternate routes in traffic jams.

Pathfinding

- Minimum Spanning Tree (MST)

  - We talked about this.
  - This is the shortest path for visiting all nodes from a starting node.

  - Uses:
    - ?

Pathfinding

- Minimum Spanning Tree (MST)

  - We talked about this.
  - This is the shortest path for visiting all nodes from a starting node.

  - Uses:
    - Laying telecom cables.
    - Routing for mail or packages delivery.
      - e.g.: UPS or FedEx.
        - Just an example, not making any statement about either of these companies.

Pathfinding

- Pathfinding is used in

  - Maps.
    - Finding an optimal route from source to destination.

  - Games
    - Movement of game characters.

  - Robots
    - In a warehouse
    - Or I guess, wherever ☺

Traversing a graph

- Before we go further into pathfinding, lets revisit some concepts.
- We had talked searching in a graph, and it was something like the following:

- We start at a *source* vertex (can be any vertex in the graph).
- This vertex has a bunch of neighbors.
- Lets call them level 1 neighbors (N1) because they are directly connected to the source
- And lets start by looking at <u>one of those neighbors</u>, who we will refer to as neighbor N1_1.

Traversing a graph

- Before we go further into pathfinding, lets revisit some concepts.
- We had talked searching in a graph, and it was something like the following:

- We start at a *source* vertex (can be any vertex in the graph).
- This vertex has a bunch of neighbors.
- Lets call them level 1 neighbors (N1) because they are directly connected to the source
- And lets start by looking at <u>one of those neighbors</u>, who we will refer to as neighbor N1_1.

  - N1_1 will have some neighbors of its own.
    - Lets call them level 2 neighbors (N2), since they are 2 levels away from the vertex we picked as *source*

Traversing a graph

- Before we go further into pathfinding, lets revisit some concepts.
- We had talked searching in a graph, and it was something like the following:

- We start at a *source* vertex (can be any vertex in the graph).
- This vertex has a bunch of neighbors.
- Lets call them level 1 neighbors (N1) because they are directly connected to the source
- And lets start by looking at <u>one of those neighbors</u>, who we will refer to as neighbor N1_1.

  - N1_1 will have some neighbors of its own.
    - Lets call them level 2 neighbors (N2), since they are 2 levels away from the vertex we picked as *source*

  - Now, what vertices do we “know” about so far:
    - Source (L0 vertex)
    - Source’s neighbors (L1 vertices)
    - Neighbors of N1_1 (L2 vertices)

Traversing a graph

- Vertices we “know” about so far:
  - Source (L0 vertex)
  - Source’s neighbors (L1 vertices)
  - Neighbors of N1_1 (L2 vertices)

- To move further in the graph, we need to pick another vertex from one of the vertices that we “know” about so far.

- In other words, we need to pick a next vertex.

Traversing a graph

- So, when we r picking a <u>next</u> vertex to visit, the decision on <u>which vertex</u> is going to be that <u>next</u> vertex is what differentiates whether we are doing a:

  - Bread first
  - Depth first
  - Dijkstra's
  - A* , etc.

Question:

What data structures are appropriate for each of these traversals?

More info coming up next…

Traversing a graph

- So, when we r picking a <u>next</u> vertex to visit, the decision on <u>which vertex</u> is going to be that <u>next</u> vertex is what differentiates whether we are doing a:

  - bread first ← Queue
  - depth first ← Stack
  - Dijkstra’s ← Heap / Priority Queue
  - A* , etc. ← Heap / Priority Queue (with a different weight as compared to Dijkstra)

BFS

- We had talked about BFS (Breadth First Search) in course1

- We start at a source vertex.
  - This is an L0 vertex.
  - We collect all neighbors (L1)

- Then we go to <u>all</u> of source's neighbors.
  - These are L1 vertices.
  - We collect all neighbors of all L1 vertices (which are L2)

- After visiting <u>all</u> L1 vertices, we go to their neighbors(L2)… and so on…

DFS

- We had talked about DFS (Depth First Search) in course1

- We start at a source vertex.
    - This is an L0 vertex.
    - We collect all neighbors (L1)

- Then we go to <u>one</u> of source's neighbors.
    - This is an L1 vertex.
    - We collect all neighbors of this one L1 vertex (which are L2)

- Now, we visit <u>one</u> neighbor or L1 (which is an L2 vertex).

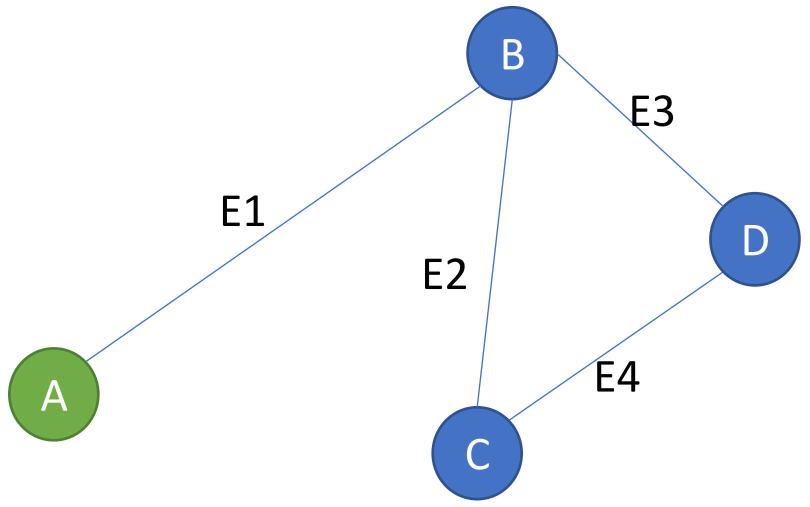
- And then we visit that one L2's neighbor (which is L3)

- And then we visit that one L3's neighbor (which is L4)
    - Lets say this L3 vertex did not have any unvisited neighbors.
    - So we don't have anywhere to go from here.
    - We look at the next L3 vertex and follow it down.
    - Once we exhaust all L3 vertices, we will look at the next L2 vertex… and so on.

BFS

- Lets go back to BFS and look at a small example.

- Distance of A from
  - B is 1
  - D is 2
  - C is 2



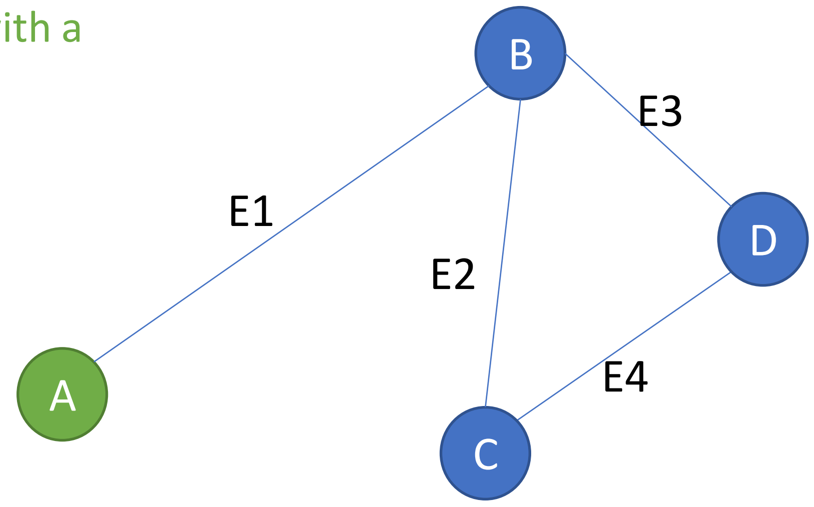
This means we r assuming the length of each edge is 1
➔ All edges are the same

Lets look at the pseudocode:

BFS

```
BFS ( Vertex  source )   {
        queue.Push( source );                    //       Lets say we start with a
        Mark  source  as  visited

        while ( ! queue.Empty() )
        {
                Vertex   v = queue.Pop();
                Print   v
                AddUnvisitedNeighbors( queue, v )
        }
}
```




```
AddUnvisitedNeighbors( Queue queue, Vertex  v )   {          // add unvisited neighbors of a
        for each  vertex   w   connected to    v             // w is a neighbor of v
                if (   not visited   w  )
                        queue.Push ( w )
                        Mark   w   as  visited
}
```

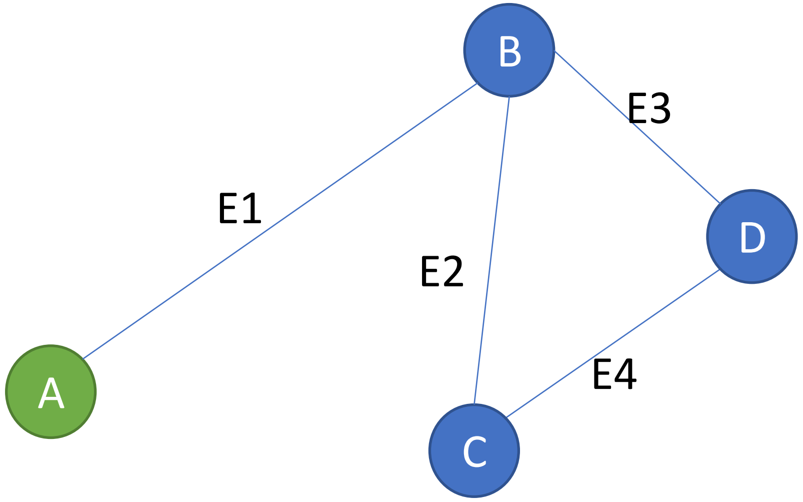
BFS

Now, if we want to know the <u>path</u> from vertex A to another vertex, say C, we will keep track of our movements

```
BFS ( Vertex   source )   {
        queue.Push(  source  );                //      Lets say we start with a
        Mark   source  as  visited
        parent [  source  ]   =  null

        while ( ! queue.Empty() )
        {
                Vertex  v = queue.Pop();
                AddUnvisitedNeighbors( queue, v )
        }
}

AddUnvisitedNeighbors( Queue   queue, Vertex   v )  {        // add unvisited neighbors of a
        for each  vertex   w   connected to    v              // w is a neighbor of v
                if (   not visited   w  )
                        queue.Push ( w )
                        Mark   w   as  visited
                        parent [ w ]   =  v
}
```



Pathfinding

Now, we print the path to vertex, say, ‘c’:

This will print the path from vertex ‘*c*’ to the *source* vertex

```
vertex  =  ‘c’
print   vertex

while (  parent[ vertex ]  !=  null  )
        vertex   =   parent [  vertex  ]
        print   vertex
```

Pathfinding

- For pathfinding, we r interested in reaching <u>one destination</u>, not all.   So, we add a check.

```
BFS ( Vertex   source,  destination  )
        queue.Push(  source  );                    //      Lets say we start with a
        Mark   source  as  visited
        parent [  source  ]   =  null

        while ( ! queue.Empty() )    {
                Vertex  a = queue.Pop();                // q:

                if (  a  ==  destination  )
                        break;

                AddUnvisitedNeighbors( queue, a )  // q:
        }
```

This should give us the shortest path from *source* to *destination*.

But, what do u notice about this path?   🡸 Next slide

Pathfinding

```
BFS ( Vertex   source,  destination  )
        queue.Push(  source  );                  //      Lets say we start with a
        Mark   source  as  visited
        parent [  source  ]   =  null

        while ( ! queue.Empty() )    {
                Vertex  a = queue.Pop();                    // q:

                if (  a  ==  destination  )
                        break;

                AddUnvisitedNeighbors( queue, a )      // q:
        }
```

Here, the shortest path from *source* to *destination* is essentially just the number of hops… because all edges have an implicit weight of 1.

What's an algorithm we have looked at where we can assign edge weights and compute shortest path?

Pathfinding

- In a real-world scenario:
  - All edges are not the same length (or weight).
  - We are interested in path to only one destination.
    - BFS and Dijkstra compute path to all destinations

Pathfinding

- In a real-world scenario:
  - All edges are not the same length (or weight).
  - We are interested in path to only one destination.
    - BFS and Dijkstra compute path to all destinations

- So, lets revisit Dijkstra's algorithm.
  - **Note**: An inference here is ➔
    - If all edge lengths (aka weights) are the same, Dijkstra's behaves like BFS.

Pathfinding

In Dijkstra’s, of all the vertices present in queue (i.e., all the ones we “know about”), we want to get the vertex **closest** to the source.

```
while ( ! queue.Empty() )    {
        Vertex  a = queue.Pop();
```

🡐 We want this to give us the “closest to source” vertex.

QQ: What are our options for the data structure?

Dijkstra’s

```
Dijkstra ( Vertex  source,  destination  )
        pQueue.Push(  source, 0  );                    //    0 is the distance from source to itself.  pQueue is a priority queue.
        Mark  source  as  visited
        parent [  source  ]  =  null;
        Initialize dist array to infinity.
        distFromSource [ source ] = 0;                 //    0 is the distance from source to itself

        while (  ! pQueue.Empty()  )
                a = pQueue.Pop();
                if (  a  ==  destination  )
                        break;
                Mark  a  as  visited
                AddUnvisitedNeighbors(  pQueue, a  )

AddUnvisitedNeighbors( pQueue, v )
        for each  unvisited vertex  w  connected to  v                                   // w is a neighbor of v
                costToNeighbor_w  =  distFromSource [ v ]  +  distanceBetween( v, w )      // calculate cost from w to source
                if  (  costToNeighbor_w  < distFromSource [ w ]  )           // if we found a shorter path to w,  update dist [ w ]
                        pQueue.Push ( w,  costToNeighbor_w )
                        distFromSource[ w ] = costToNeighbor_w
                        parent [ w ]  =  v
```

LAB

- Use a rectangular grid as a graph, or u can build a regular graph … non-grid. Using a grid may be easier.

- Pick a source vertex.
  - If using a grid, could pick (0, 0)
- Pick a destination vertex.
  - Could pick diagonally opposite end, say, (7, 7) if ur grid is of size 8 X 8.

- Now, write code for:

  - BFS traversal

  - Dijkstras algorithm, with a modification where u would stop when u reach the destination vertex.

Pathfinding

- So far we have done the following:
  - We stop as soon as we find our destination vertex.
  - We did not assume all edge weights are 1 (or the same value)

- Remember… our goal is to get to the goal ☺ (aka destination vertex).

- Any thoughts on any optimization we could potentially do (strategic optimization, not code)?

Pathfinding

- So far we have done the following:
  - We stop as soon as we find our destination vertex.
  - We did not assume all edge weights are 1 (or the same value)

- If u were to zoom in to what’s really happening:
  - We are expanding our search in <u>any</u> direction (or all).
  - This is because we look at the “distance from source” and select closest vertex as the next one.

Pathfinding

- So far we have done the following:
  - We stop as soon as we find our destination vertex.
  - We did not assume all edge weights are 1 (or the same value)

- If u were to zoom in to what's really happening:
  - We are expanding our search in <u>any</u> direction (or all).
  - This is because we look at the "distance from source" and select closest vertex as the next one.

- But, our goal is to find the shortest path to a destination (and not to all vertices).

Pathfinding

- So far we have done the following:
  - We stop as soon as we find our destination vertex.
  - We did not assume all edge weights are 1 (or the same value)

- If u were to zoom in to what's really happening:
  - We are expanding our search in <u>any</u> direction (or all).
  - This is because we look at the "distance from source" and select closest vertex as the next one.

- But, our goal is to find the shortest path to a destination (and not to all vertices).

- So, it would make sense to generally be expanding our search in the "right" direction… which means "towards the destination".
- This would make us reach the destination faster.

- Here's what we will do to make that happen.

Pathfinding

- We will add another value, say, H, to the existing “distance from source” that we were already using.
- This means, in the priority queue, we will have the **sum** of “distance from source” and H.
- And we will pick the next vertex from the queue based on this sum.

Pathfinding

- We will add another value, say, H, to the existing "distance from source" that we were already using.
- This means, in the priority queue, we will have the **sum** of "distance from source" and H.
- And we will pick the next vertex from the queue based on this sum.

- This new component that we added is a heuristic … its an estimated distance from the vertex in question to the destination.

Pathfinding

- We will add another value, say, H, to the existing “distance from source” that we were already using.
- This means, in the priority queue, we will have the **sum** of “distance from source” and H.
- And we will pick the next vertex from the queue based on this sum.

- This new component that we added is a heuristic … its an estimated distance from the vertex in question to the destination.

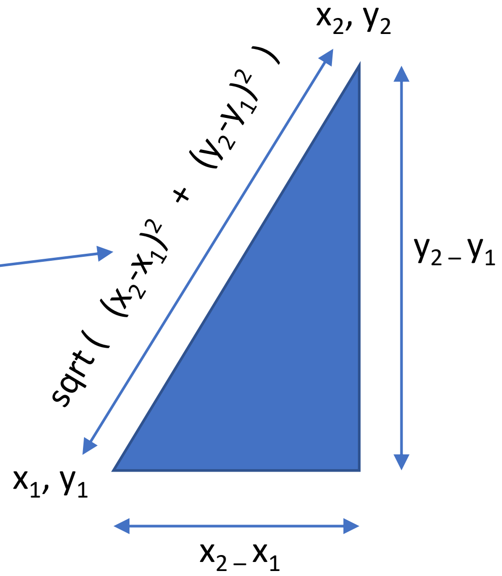
- If we use a grid as our graph, then this estimated distance could be something as simple as the Manhattan distance between the two vertices.

  - Next: Manhattan distance, Euclidean distance.

Manhattan and Euclidean

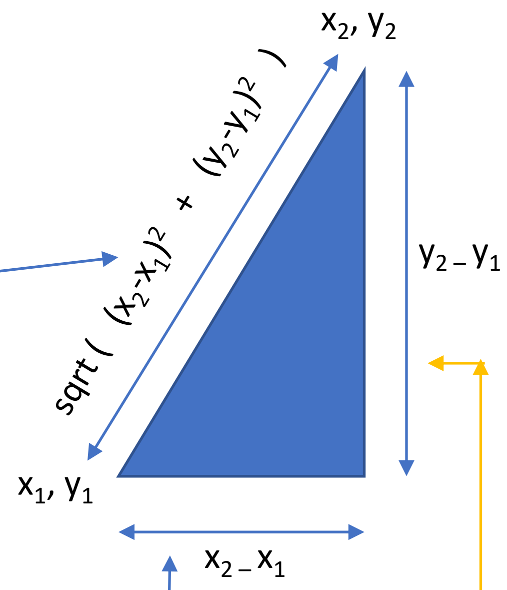
- Given:
    - Two points with coordinates (x1, y1) and (x2, y2)

    - **Euclidean** distance between these two points:
        - sqrt ($(x_2\text{-}x_1)^2$ + $(y_2\text{-}y_1)^2$)
        - From *Pythagorean* theorem.



Manhattan and Euclidean

- Given:
  - Two points with coordinates (x1, y1) and (x2, y2)
  - **Euclidean** distance between these two points:
    - sqrt ($(x_2-x_1)^2$ + $(y_2-y_1)^2$)
    - From *Pythagorean* theorem.
  - **Manhattan** distance between coordinates (x1, y1) and (x2, y2) is
    - $|x_2 - x_1| + |y_2 - y_1|$



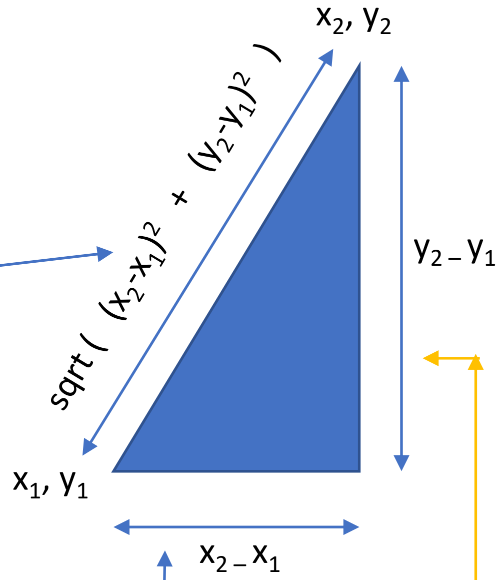
- Next: Which distance should we pick?

Manhattan and Euclidean

- Given:
  - Two points with coordinates (x1, y1) and (x2, y2)

  - **Euclidean** distance between these two points:
    - sqrt ($(x_2\text{-}x_1)^2 + (y_2\text{-}y_1)^2$)
    - From *Pythagorean* theorem.

  - **Manhattan** distance between coordinates (x1, y1) and (x2, y2) is
    - $|x_2 - x_1| + |y_2 - y_1|$



- We prefer Manhattan because computing it is cheaper … no squares or square roots to compute.
  - But u could use Euclidean distance as long as u r ok with the extra compute cost (usually wont).

- Of course, Euclidean distance is more accurate.

Pathfinding

- So, our weight component for selecting the *next* vertex   w out of the priority queue becomes:

  - Weight   =   Distance (source, vertex)
    + Distance (vertex, w)
    + Manhattan distance (w, destination)

  In the pseudocode we saw earlier, we had:
    - costToNeighbor_w = distFromSource [v] + distanceBetween(v, w)
    - Manhattan distance is the new piece we r adding now.

- First component is the actual distance computed as the search spreads out (as we saw in BFS or Dijkstra).
- Second component is the actual distance between the vertex and its neighbor w.
- Third component is the estimated distance between neighbor w and destination.

A *

```
AStar ( Vertex   source,  destination  )
        pQueue.Push(  source, 0  );          //   0 is the distance from source to itself.  pQueue is a priority queue.
        Mark  source  as  visited
        parent [  source  ]   =  null;
        Initialize dist array to infinity.
        dist [ source ] = 0;                  //   0 is the distance from source to itself

        while (  ! pQueue.Empty()  )    {
                a = pQueue.Pop();
                if (  a  ==  destination  )
                        break;
                Mark   a   as  visited
                AddUnvisitedNeighbors(  pQueue, a  )
        }
```

This is the heuristic.
We talked about using Manhattan distance as *one* potential option.

```
AddUnvisitedNeighbors( pQueue, v )
        for each  unvisited vertex   w   connected to    v                         // w is a neighbor of v
                costToNeighbor_w  =  dist [ v ]  +  distanceBetween( v, w )         // calculate cost from w to source
                if   (  costToNeighbor_w   <   dist [ w ]  )                        // if we found a shorter path to w,  update dist [ w ]
                        pQueue.Push ( w,   costToNeighbor_w  + EstimatedDistance ( w,  destination ) )
                        parent [ w ]   =  v
```

Pathfinding

- FYI

- An algorithm for finding k-shortest paths between a given source and destination is Yen's K-shortest path algorithm.

- In addition to shortest path, it also finds the 2^{nd} shortest and 3^{rd} shortest … until K^{th} shortest path.