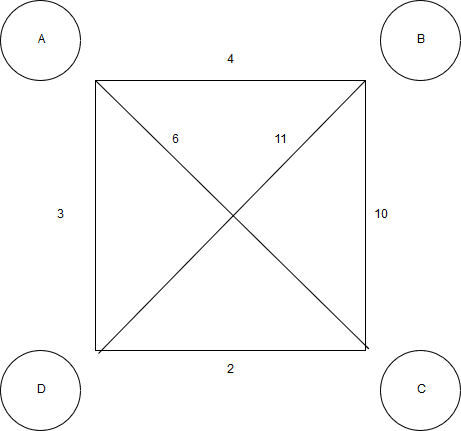
**Brute Force Algorithm**

**Description**

The Traveling Salesman Problem is not one that is easily solvable. As such, one method that will always “solve” it, albeit very slowly, is the naïve or “brute force” approach. This approach will check each distance possible, from the beginning to the end, and find the shortest distance possible after checking every possible path[4]. Quite clearly, this algorithm is very slow because it has virtually no optimization. Simply checking every distance does not make use of dynamic programming concepts which would allow us to keep track of the shortest possible distance that we have already come across. That being said, it has the benefit of eventually coming across the proper solution since we can be sure that every path will be checked[3]. That means it is an optimal solution[3].

**Example Run** [4]



This path has 4 nodes with three possible paths:

A → B → C → D → A = 19

A → B → C → D → A = 23

A → C → B → D → A = 30

The first tour is the quickest and the one that will be chosen. The important thing to note is that the algorithm finds this tour by generating each of the permutations and checking the distances against each other.

**Pseudocode**

def calc\_distance (first\_point, second\_point) //A helper function to get distance between points

dx = point1[1] - point2[1]

dy = point1[2] - point2[2]

return sqrt of dx \* dx + dy \* dy

def total\_distance(all\_points) //Function to get the total distance

total\_distance\_1 = calc\_distance(all\_points[first\_index], all\_points[length(all\_points0 – 1) //Calculate the distance between points

total\_distance\_2 = sum([calculate\_distance(point, all\_points[index + 1]) for each index in all points (from start of all\_points to the end) //Get sum of distances

total = total\_distance\_1 + total\_distance\_2 //Add together and assign to variable

return total //Return that amount

def find\_nearest\_neighbor(current, yet\_to\_visit) //Function to help traverse unvisited vertices

nearest = yet\_to\_visit[first\_index] //Assign the first unvisited neighbor to the nearest neighbor

min\_distance = calc\_distance(current, nearest) //Setup for the minimum distance for each neighbor in yet\_to\_visit:

current\_distance = calc\_distance(current, neighbor) //Get distance for the current vertex and its neighbor

if distance < min\_distance: //Update closest neighbor and min distance

nearest = neighbor

min\_distance = distance

return nearest

def brute\_force(all\_vertices)

min\_distance = infinity //Set to max possible value since we know we’re assigning a value less than this

for every number in range of 0 to length of all\_vertices:

distance = 0

beginning\_vertex = all\_vertices[current\_number]

yet\_to\_visit\_vertices = all\_vertices(entire list)

yet\_to\_visit\_vertices.remove(beginning\_vertex) //Remove this since we have already visited it by default

tour = beginning\_vertex

while length(yet\_to\_visit\_vertices) > 0

nearest = find\_nearest\_neighbor(tour, yet\_to\_visit\_vertices) //Get the nearest neighbor by passing in where we currently are with the list of unvisited vertices to our helper function

yet\_to\_visit\_vertices.remove(nearest) //We’ve visited so now we can remove it from our list

tour.append(nearest) //Now we add this vertex to the tour since we visited

distance = total\_distance(tour) //Add everything up

if distance < min\_distance //We know this will be the case since it was set to the highest possible value

min\_distance = distance //Update the distance

smallest\_tour = tour //Update the smallest route we have found

return smallest\_tour