Designing Algorithms to Approximately Solve a Problem

In this chapter we will introduce the idea of not designing an algorithm to solve a given problem, but instead to design an algorithm to only *approximately* solve a problem. The idea is that, as we will see in the next chapter, some problems appear to be hopelessly difficult, so why don't we settle for only getting within some percentage of the optimal solution? This can be quite useful in practical terms, especially if we can prove that the approximation algorithm will produce an approximation that is within some known percentage of the best possible.

This sounds ridiculous—how can we know we are within some percentage of the optimal solution without being able to find that optimal solution? But, we will see that it can sometimes be done.

We will look at two such approximation algorithms for ETSP.

In addition to introducing the idea of approximation algorithms, this material will provide a nice wrap-up for the course, as we will use a greedy algorithm (Prim's Algorithm), a branch and bound algorithm (to solve the minimum weight matching problem), and again use linear programming to provide a bound for a problem.

Approximately Solving ETSP by Finding a Minimum Spanning Tree

The first algorithm begins by finding a minimum spanning tree (MST) for the graph. We can use either Prim's or Kruskal's algorithms to do this in polynomial time.

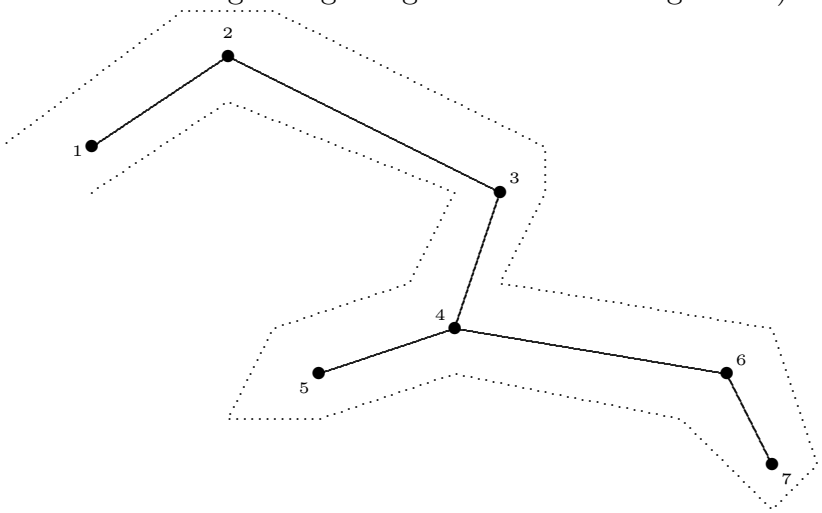
Let's denote our minimum spanning tree by T.

To relate a minimum spanning tree to a tour, suppose that we have any tour of our graph. If we remove any one edge from this tour, then the remaining edges give a spanning tree for the graph. Thus, the total cost of the edges in a minimum spanning tree must be less than the total cost of any tour, including an optimal tour.

Now, we can view T as a physical thing, and starting at vertex 1, we can travel along the edges of T, keeping those edges to our left as we go, and eventually get back to vertex 1. In so doing, we will travel along each edge of T twice—once in each direction.

Example

Given the 7 points shown, we construct a spanning tree, indicated by the solid edges between vertices (this might not actually be minimal, but it is clearly a spanning tree, and that's enough for getting the idea of the algorithm):



As we follow the dotted path, we go along these edges:

$$1 \to 2 \to 3 \to 4 \to 5 \to 4 \to 6 \to 7 \to 6 \to 4 \to 3 \to 2 \to 1.$$

It is easy to see that this path goes along each edge in T twice. So, the total cost of this path is equal to twice the cost of the edges in T, which is less than or equal to twice the cost of the optimal tour (since we saw earlier that the edges in T add up to less than or equal to the optimal tour edges).

Thus, the cost of the dotted path is less than or equal to twice the cost of an optimal tour.

Now we improve the dotted path by taking short-cuts, meaning instead of following some parts of the path that involve several edges, we go immediately from the first vertex to the last. Because the triangle inequality holds, short-cuts will be better than original paths.

And, we take short-cuts whenever we need to in order to avoid duplicate points and thus produce a tour.

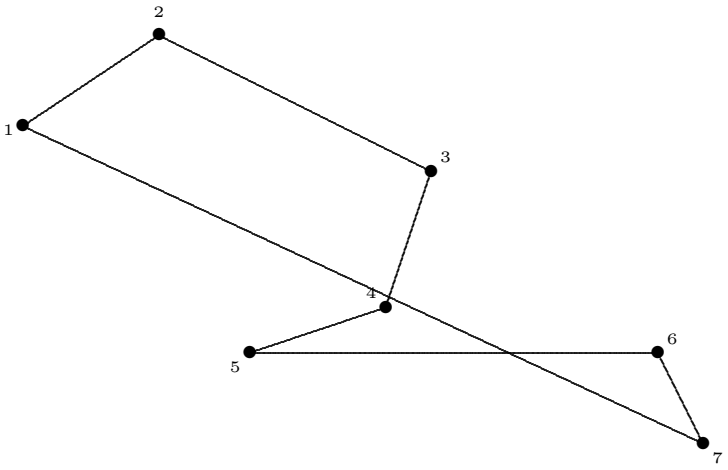
In the example, we go from 1 to 2 to 3 to 4 to 5, but then instead of going from 5 to 4 to 6, we take the short-cut from 5 to 6. So, where the dotted path has the costs of going from 5 to 4 and from 4 to 6, the short-cut only has the cost of going from 5 to 6.

Continuing similarly, we go from 6 to 7, but then instead of going on in the dotted path, we eliminate the second visits to 6, 4, 3, and 2, and immediately short-cut to 1.

So, we end up following the edges

$$1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 1,$$

like this:



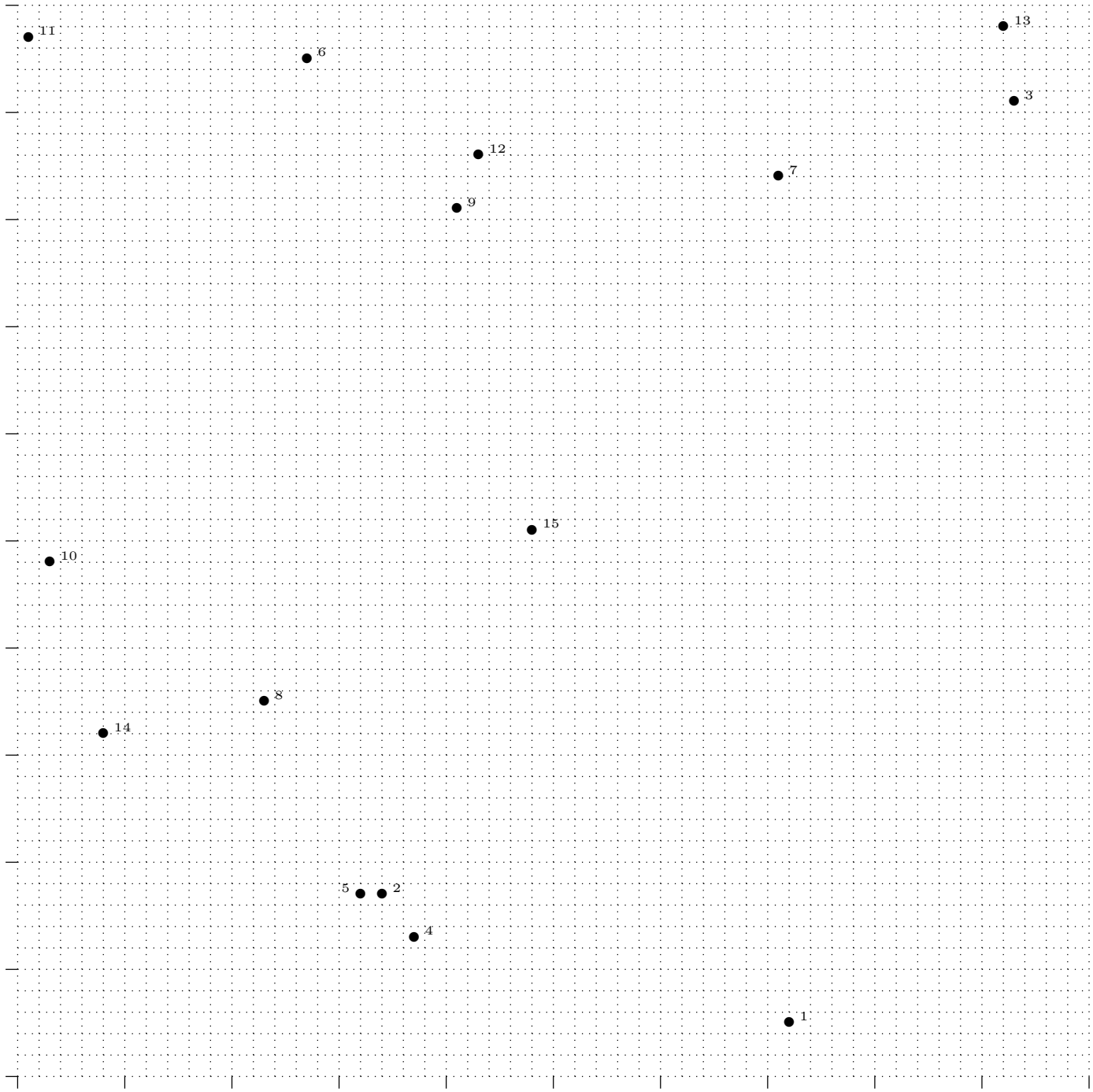
This tour has cost less than or equal to twice the cost of T, which is less than or equal to twice the optimal tour cost.

So, since we can find a minimum spanning tree for a giving graph in polynomial time, then the previous discussion shows how we can find a tour that is at worst twice as costly as an optimal tour.

It is important to note that there are a lot of methods for finding tours that probably produce better tours than the one we are considering, but our method has a *guaranteed* goodness that purely heuristic methods lack.

A Bigger Example

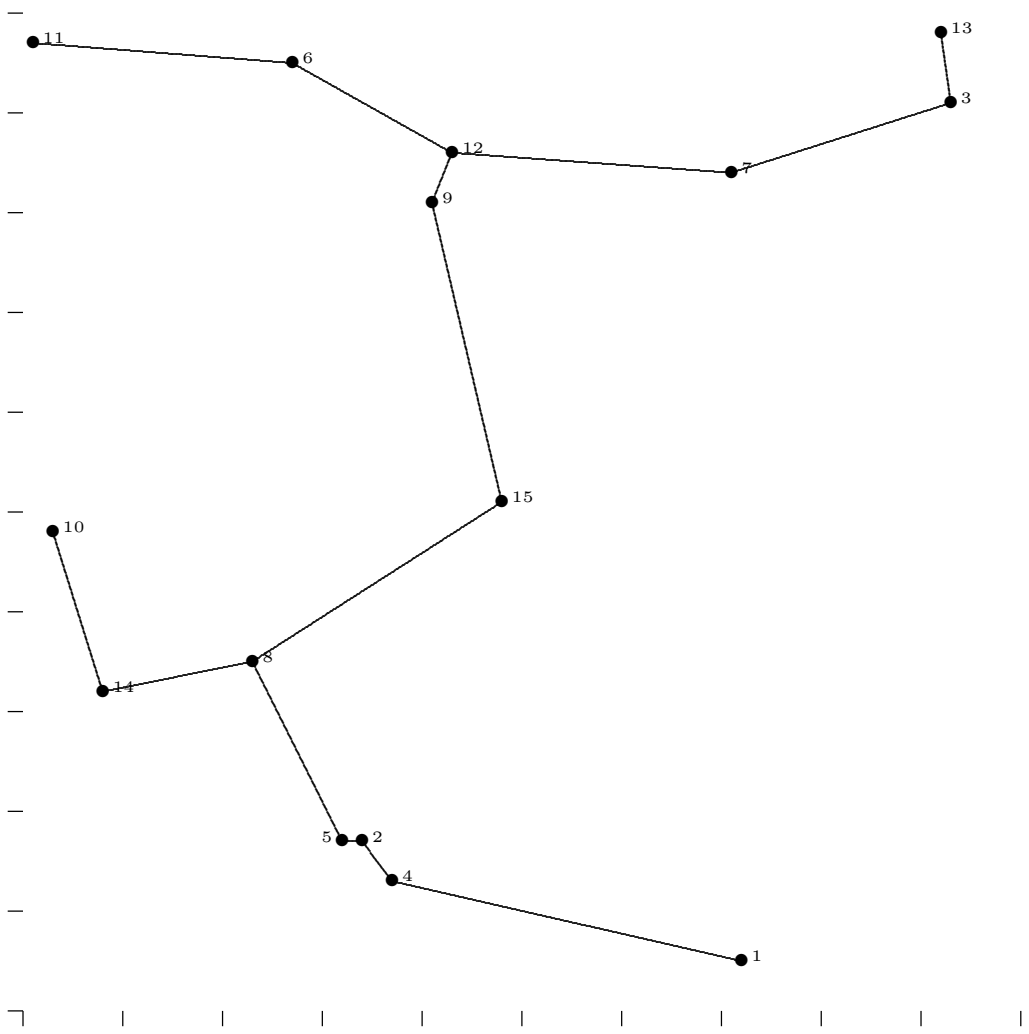
Here is an instance of ETSP:



⇒ Let's perform Prim's algorithm visually, which means we start with vertex 1, find the closest vertex to it, which is 4, connect them with an edge, then find the closest unconnected vertex to 1 or 4, which is obviously vertex 2, and so on.

In case it is too hard to tell which distances are smallest, without using a ruler, here are the actual distances:

	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	39.85	88.53	35.90	41.76	100.62	79.01	57.45	82.08	81.30	116.21	86.03	95.13	69.46	51.88
2		94.64	5.00	2.00	78.31	76.54	21.10	64.38	43.84	86.54	69.58	99.62	30.02	36.77
3			96.02	95.90	66.12	23.09	89.64	52.95	99.74	92.20	50.25	7.07	103.47	60.21
4				6.40	82.61	78.72	26.08	68.12	48.80	91.39	73.25	101.24	34.67	39.56
5					78.16	77.52	20.12	64.63	42.45	85.80	69.87	100.80	28.30	37.58
6						45.35	60.13	19.80	52.77	26.08	18.36	65.07	65.80	48.75
7							68.59	30.15	76.94	71.20	28.07	25.24	81.69	40.22
8								49.40	23.85	65.79	54.78	93.43	15.30	29.68
9									50.33	43.08	5.39	53.76	59.08	30.81
10										49.04	55.17	102.08	16.76	45.10
11											43.42	91.01	65.38	65.76
12												50.45	64.35	35.36
13													106.83	64.38
14														44.28



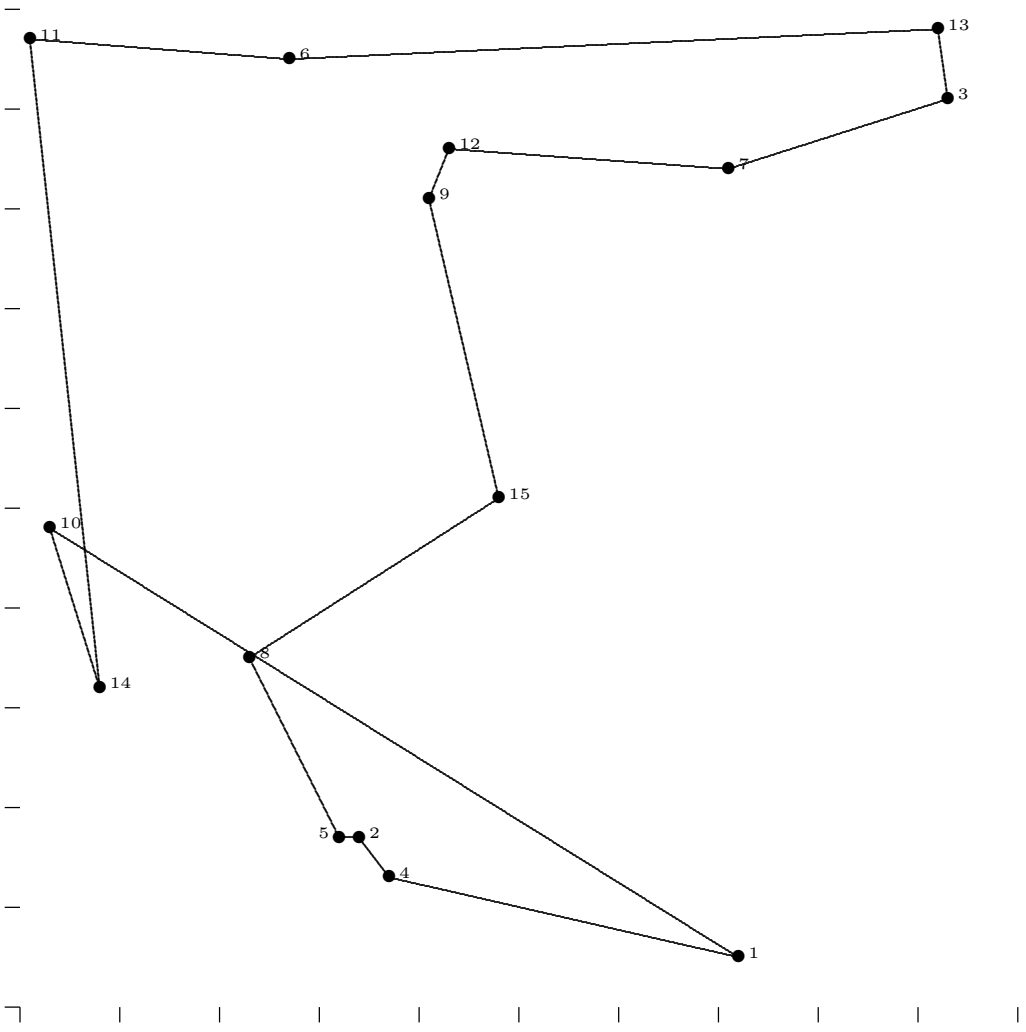
If we traverse this tree, keeping it to our left, starting out heading from vertex 1 to vertex 4, we hit vertices in the order

$$1, 4, 2, 5, 8, 15, 9, 12, 7, 3, 13, 3, 7, 12, 6, 11, 6, 12, 9, 15, 8, 14, 10, 14, 8, 5, 2, 4, 1.$$

Now we scan these vertices, and whenever we hit one we have already seen, we skip it, producing

$$1, 4, 2, 5, 8, 15, 9, 12, 7, 3, 13, 6, 11, 14, 10, 1,$$

which is the tour shown below, with cost 441.72:



Using `DynProgTSP`, with the data file named `christofides`, we find that the optimal tour is $1, 15, 3, 13, 7, 12, 9, 6, 11, 10, 14, 8, 5, 2, 4, 1,$ with a cost of 367.862.

Christofides' Algorithm

Now let's look at another approximation algorithm, published by Christofides in 1976. It produces, in polynomial time, a tour which has a cost that is no more than 1.5 times the cost of an optimal tour.

> According to Wikipedia, as of 2017 this algorithm gives the best factor—1.5—of any known algorithm for TSP instances that satisfy *metric space* properties (symmetry—the cost of the edge from vertex j to vertex k is the same as the cost of the edge from vertex k to vertex j—and the triangle inequality).

Here is Christofides' Algorithm, according to Wikipedia:

- Find a minimum spanning tree T of the graph G (since we're doing ETSP, this graph has all possible edges among the n points).

  > We have seen how to do this in polynomial-time using Prim's algorithm.

- Let O be the set of vertices in T that have an odd number of edges connected to them. It is pretty easy to see that O will have an *even* number of vertices.

  > Let $deg(v)$ (for "degree") in an undirected graph be the number of edges connected to the vertex v. For any graph with vertices V and edges E, $\sum_{v \in V} deg(v) = 2|E|$, because each edge gets counted twice—once for each end.
  >
  > Thus, the sum of degrees of all the vertices in the graph is even, so if we subtract the degrees of all the vertices of even degree, the remaining sum—the sum of all the odd degrees—is still even. But, to add up a bunch of odd numbers and get an even number, there must be an even number of those odd numbers, hence the number of odd vertices must be even.

- Find a *minimum-weight perfect matching* M in the subgraph of G consisting of the vertices in O and only the edges that connect two vertices in O. We will consider below how to do this.

- Combine the edges in T and M to form a multi-graph (allowing more than one edge between a pair of vertices) in which each vertex has even degree.

- Find an *Eulerian circuit* in the multi-graph.

  > An Eulerian circuit crosses each edge in a graph exactly once, and can hit vertices one or more times (as opposed to a Hamiltonian circuit, which we have been calling a tour, which visits each vertex exactly once before returning to the start, and only uses some of the edges).

- Do short-cuts to eliminate repeated vertices, producing a tour.

This description leaves out two key sub-algorithms, namely finding a minimum-weight perfect matching and finding an Eulerian circuit. Of these, which are blithely mentioned in the Wikipedia page, the second is easy and the first is very difficult.

In 1965 Jack Edmonds published the *blossom algorithm*, which, with some further ideas from linear programming, can produce, in polynomial-time, a minimum-weight perfect matching in a graph with an even number of vertices.

We can observe, however, that if we really want our own way to find min-weight perfect matching in a graph with an even number of vertices, we can do a branch-and-bound approach, very similar to what we did for ETSP. This is detailed in the folder `Code/MWM`, where I took our code for ETSP and changed it to work for MWM. The only differences are that the constraints just say, for each vertex, that the total intensities attached to it add up to 1, instead of 2 for ETSP, and there is no need to put in the constraints of the form $x_{jk} + s_{jk} = 1$. Thus, the MWM tableaux are quite a bit easier to form, and smaller, than for ETSP.

To find an Eulerian circuit on a multi-graph where every vertex has even degree, we can use *Hierholzer's Algorithm* from 1873. Start at vertex 1 and randomly follow as-yet unused edges until we return to vertex 1. Since all vertices have an even number of edges attached, if we can find an unused edge that goes into a vertex, then there must be an unused edge going out. If there are any vertices in this sub-tour that have unused edges, we start at such an edge and follow unused edges until we get back to it. This is repeated until all edges have been used. The Eulerian circuit is obtained by following edges in a fairly obvious way.

Now we want to prove that Christofides' algorithm produces a tour whose cost is less than 1.5 times the optimal cost:

As before, the total cost of the edges in the minimum spanning tree is less than or equal to the optimal cost, since every tour gives a spanning tree by removing any edge.

The new thing here is the claim that the total cost of the edges in the matching M is less than half the cost of an optimal tour. To see this, suppose we have an optimal tour N of just the vertices O—the ones in the MST of odd degree. Then N has an even number of edges, and if we take every other edge both ways, we get two sets of edges, say N_1 and N_2, each of which gives a perfect matching of all the vertices in O.

For example, suppose $O = \{a, b, c, d, e, f\}$, and suppose N is the 6 edges $a \to e$, $e \to c$, $c \to b$, $b \to f$, $f \to d$, $d \to a$. Then N_1 is the 3 edges $a \to e$, $c \to b$, $f \to d$, and N_2 is the 3 edges $e \to c$, $b \to f$, $d \to a$. Thus, both N_1 and N_2 are perfect matchings of the 6 vertices in O.

Now, since M is the *minimum-weight* perfect matching of the vertices in O, the cost of M is less than both the cost of N_1 and the cost of N_2. Since the cost of N is the sum of the cost of N_1 and the cost of N_2, the cost of N is

greater than or equal to half the minimum of these two costs, so the cost of M is less than or equal to half the cost of N. But, since N is an optimal tour of just some of the vertices in the original graph, clearly its cost is less than or equal to the cost of an optimal tour. So, the cost of M is less than or equal to half the cost of an optimal tour of the original graph.

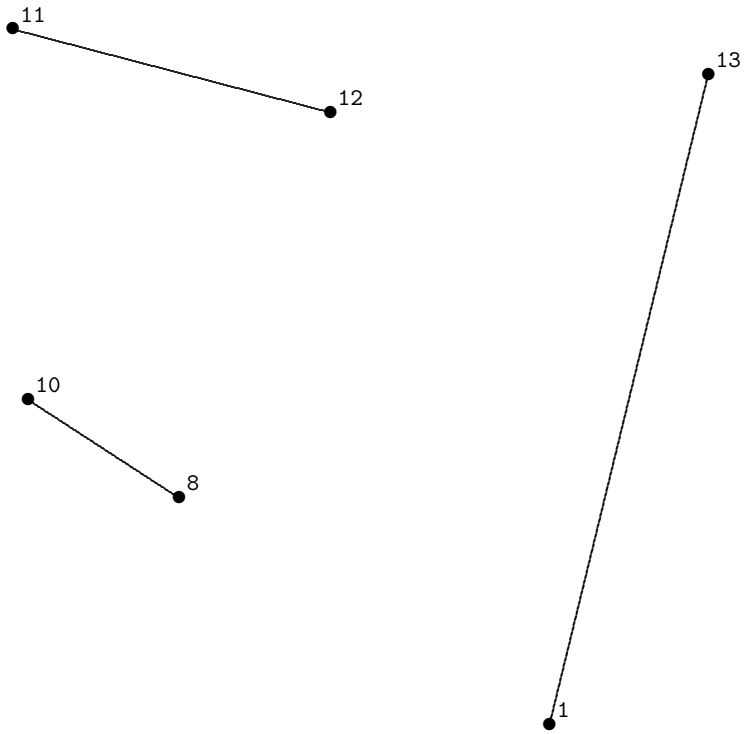
So, the tour obtained by this process has cost less than or equal to the cost of T plus the cost of M, which is less than or equal to the cost of the optimal tour plus half the cost of the optimal tour, which is 1.5 times the cost of the optimal tour.

Example of Christofides' Algorithm

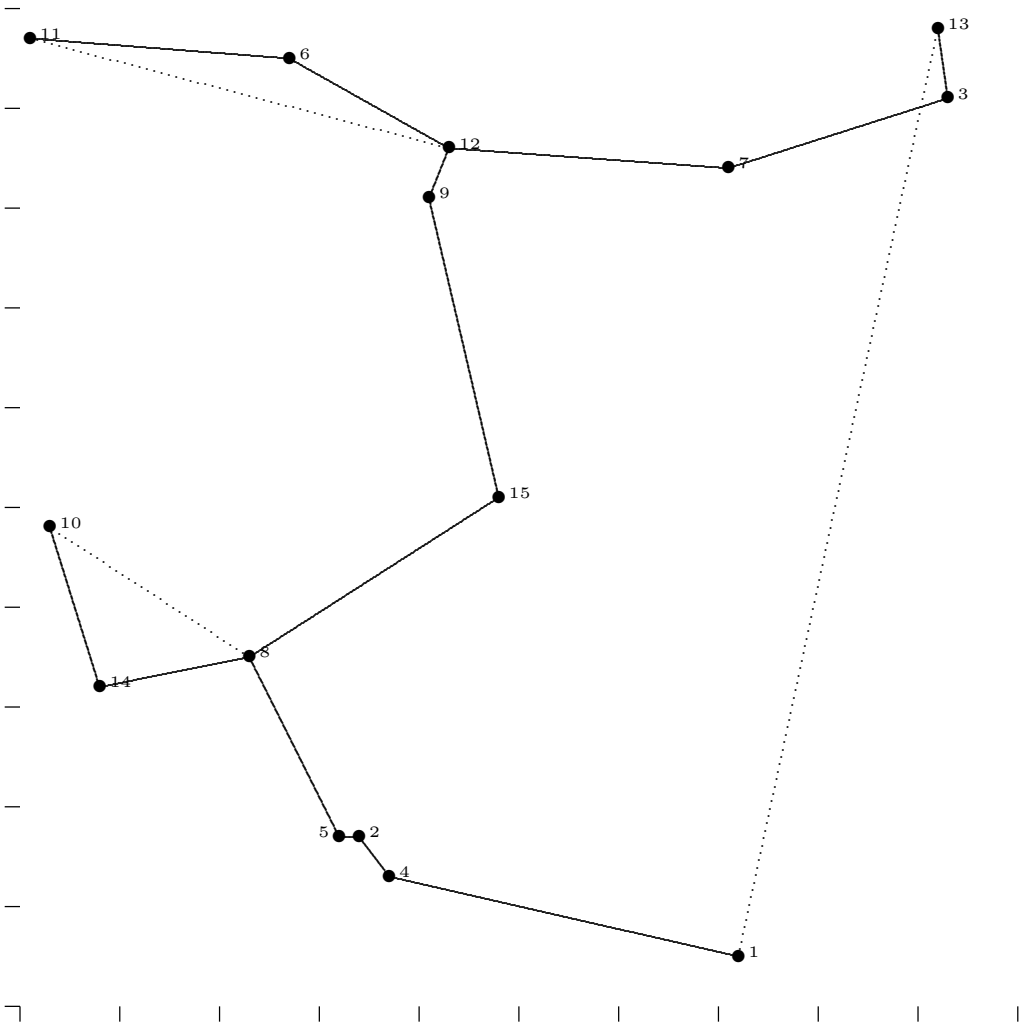
Consider the 15 points given earlier. After finding a minimum spanning tree for those points, shown back on page 156, we see that the vertices of odd degree are

$O = \{1, 3, 8, 10, 11, 12\}$ (these include leaf vertices and some interior vertices).

In the folder `Code/MWM` we have the data file `christofidesOdd` that contains just the points in O, and when we run `HeuristicMWM` on that data file, we get this minimum-weight perfect matching (fortunately, the LP solution has no non-zero basic variables that are not equal to 1—i.e., no infamous "red edges"):



Now if we combine the edges in the minimum spanning tree on page 156 and these three matching edges (shown dotted), we have this graph where all vertices have even degree:



Now we find an Eulerian circuit (cheating a little to make all the edges fit into one original tour):

$$1, 4, 2, 5, 8, 14, 10, 8, 15, 9, 12, 6, 11, 12, 7, 3, 13, 1.$$

Then we short-cut this sequence to the tour

$$1, 4, 2, 5, 8, 14, 10, 15, 9, 12, 6, 11, 7, 3, 13, 1$$

(which you are welcome to draw in yourself on the graph above)
which is guaranteed to have total cost less than or equal to 1.5 times the optimal cost of 367.862, and, in fact, the total cost (consulting the distances chart given on page 156) is

$$35.90 + 5 + 2 + 20.12 + 15.30 + 16.76 + 45.10 + 30.81 +$$
$$5.39 + 18.36 + 26.08 + 71.20 + 23.09 + 7.07 + 95.13$$
$$= 417.31.$$

Exercise 33 [4 points] (target due date: Monday, November 30)

Your job on this Exercise is to demonstrate Christofides' algorithm on the instance of ETSP with these 12 points:

1: $(64, 65)$
2: $(7, 71)$
3: $(8, 55)$
4: $(51, 57)$
5: $(55, 56)$
6: $(80, 75)$
7: $(3, 84)$
8: $(17, 84)$
9: $(72, 72)$
10: $(65, 31)$
11: $(58, 10)$
12: $(8, 30)$

Your demonstration will be mostly graphical, drawing on the diagram on the next page the edges of the minimum spanning tree T obtained by Prim's algorithm, the set O of vertices with an odd number of edges of T connected to them, the edges in the minimum weight matching M of the points in O, a list of vertices that form an Eulerian traversal of the vertices, and finally the list of vertices for the tour obtained by this algorithm.

You must also compute the total length of this tour, and compare it to the optimal tour (obtained by using one of our earlier algorithms). Verify that this tour has total length less than or equal to 1.5 times the optimal tour's total length.

For convenience, here are all the distances between points:

	2	3	4	5	6	7	8	9	10	11	12
1:	57.31	56.89	15.26	12.73	18.87	63.89	50.70	10.63	34.01	55.33	66.04
2:		16.03	46.17	50.29	73.11	13.60	16.40	65.01	70.46	79.51	41.01
3:			43.05	47.01	74.73	29.43	30.36	66.22	61.85	67.27	25.00
4:				4.12	34.13	55.07	43.42	25.81	29.53	47.52	50.77
5:					31.40	59.06	47.20	23.35	26.93	46.10	53.71
6:						77.52	63.64	8.54	46.49	68.62	84.91
7:							14.00	70.04	81.57	92.20	54.23
8:								56.29	71.51	84.60	54.74
9:									41.59	63.56	76.55
10:										22.14	57.01
11:											53.85

You will need to use the code to solve the minimum weight matching problem, in the folder `Code/MWM` at the OneDrive folder for the course.

Draw your edges (ideally somehow distinguish the edges in T and the edges in M) directly on this diagram (or on your own version of this diagram):