The Branch and Bound Algorithm Design Technique

Now we want to study a powerful technique that is a combination of brute force—creating a tree of possibilities—combined with clever ways to determine that some nodes of the tree are "non-promising" and need not be pursued further.

As we will see, this technique can sometimes be useful when the best known algorithm is the brute force algorithm, but by "pruning" non-promising branches, we can sometimes find solutions to instances that we could not expect to solve in a reasonable amount of time by the brute force method. This material has a strong "heuristic" flavor—we try to do things that are likely to solve a given instance, but we must always realize that there is a possibility that the algorithm will run, for some instances, for as long as the brute force method.

As an overview, note that optimization problems—and other problems, too—involve decisions. For many problems involving decisions, we can view the brute force method as constructing the tree of all possible decisions. Then the heuristic element of the "branch and bound" technique means that we can often determine that there is no point in creating the subtree rooted at a certain node because we can see that the decisions made up to that point prevent us from doing better than some known result, no matter what decisions we make going on from that node.

To design an algorithm using the possibilities tree (that's the "branch" part of the technique), we need to clearly specify our notation—what decision is being made at each level of the tree, and what the various branches mean. Sometimes we must label the branches, and other times we can use a convention and avoid that work. Also, it is important to store the appropriate information in each node, and to clearly say what that information means. Ideally, we should put enough information in the nodes that we don't have to label the branches, or even literally build the tree.

We can create an algorithm based on a possibilities tree, with or without bounding, in various ways. We sometimes use recursive calls, where the tree is not actually constructed, each node is simply a recursive call, and the necessary information is passed as arguments to the call. Or, we can literally construct the nodes, as objects that store the necessary information in instance variables of the node class.

Branch and Bound Applied to the 0-1 Knapsack Problem

Let's apply these ideas to the 0-1 knapsack problem.

We will store this information in each node:

> The node number, where nodes are numbered consecutively in order of creation.
>
> The items that have been chosen and rejected at this point (rejected items are shown crossed out).
>
> The profit achieved by the chosen items.

The total weight of the chosen items.

A cleverly computed bound that says no matter how we accept or reject the remaining items, we can't get a better profit than this bound.

Let's explore the branch and bound technique for this problem with this sample instance:

Let $W = 16$ and use this item data, where we have added the profit per weight data as an aid to computing bounds, and sorted the items according to it:

i	p_i	w_i	$\frac{p_i}{w_i}$
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

⇒ You might want to watch the video `branchAndBoundKnapsack.mp4` which works through this example in detail.

We will use a *best first* approach to building the possibilities tree, where the node with the best bound is explored next. Also, we will monitor the best actual profit in any node seen so far, and prune any node whose bound is worse than that profit. We will also, of course, prune any node whose total weight exceeds the allowed capacity.

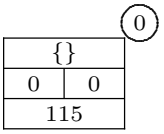
The computation of the bound is the only even slightly difficult part of this algorithm—everything else is quite simple and natural. The bound for a node is computed by saying "what profit could we achieve, by choosing or rejecting any desired *fractional part* of the remaining items?" We also sort the items in order of decreasing profit per weight, which implies that we should use all of item $m + 1$ that fits, until we've used it all, and then use whatever part of item $m + 2$ that fits, and so on.

We will consider a node non-promising if its total weight exceeds W, of course.

In the video we will draw the tree as we work through this example, but the following is the actual way the algorithm should be implemented.

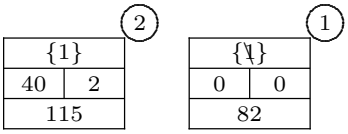
In the following example, instead of literally drawing the tree, we add and remove nodes to and from a *priority queue*. We won't detail the efficient heap data structure based way to implement a priority queue, but will instead add nodes where they belong in sorted order according to bounds.

We start by generating the root node, which is added to the priority queue:



This is node 0, with no items chosen (or rejected), for a profit of 0, and a weight of 0. To figure the bound, we add items—or a fraction of the last item used—in order until we reach of weight of 16 (or run out of items). So, all of item 1 fits, giving profit of 40 and weight 2. Then all of item 2 fits, for a profit of $40+30=70$, and a weight of $2+5=7$. Then only 9 units of item 3 fit, for a weight of 16, and a profit of $70+9\cdot 5=115$.

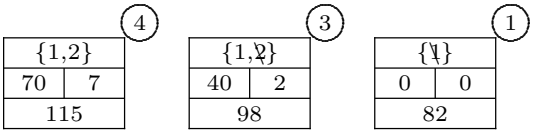
Now we remove the item with the best bound, which is of course node 0 (since it is the only node in the priority queue), and add its children, obtained by rejecting item 1 (node 1) and accepting item 1 (node 2), obtaining this priority queue:



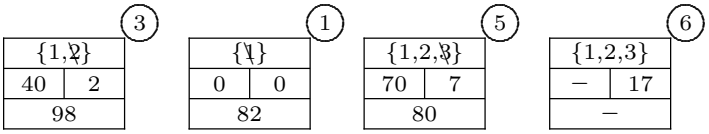
Note that we are arbitrarily creating the left child—the one that rejects the next item—first, and then the right child—the one that accepts the next item.

For node 1 the bound was computed by using all of item 2, for a profit of 30 and a weight of 5, then using all of item 3 for a total profit of 80 and a total weight of 15, and then using 1 weight unit of item 4, which brings the total profit up to 82 and the total weight up to 16.

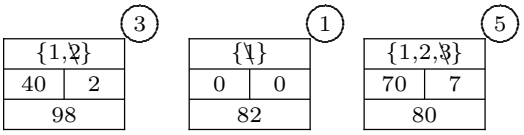
Next, the algorithm removes node 2, because its bound (115) beats the bound 82 of node 1, and adds nodes 3 and 4, yielding (in sorted order)



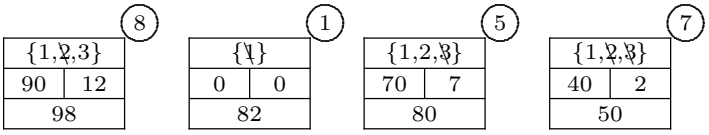
Next we remove node 4 and add its children, nodes 5 (reject item 3) and 6 (accept item 3), giving:



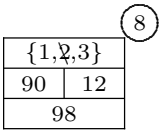
Then node 6 is pruned, because it is too heavy (we probably wouldn't even put it in the priority queue in the first place, but we did here to show it so we could see why it was pruned), giving:



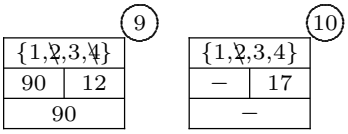
Next node 3 is pruned, and nodes 7 and 8 are added, giving:



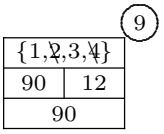
Now something exciting happens: since node 8 actually achieves a profit of 90, we prune all the nodes that have a bound less than or equal to 90, giving:



Next we remove node 8 and add nodes 9 and 10, giving:



Then we prune node 10, since it is too heavy, giving:



At this point we stop and declare node 9 the best, realizing that there are no more items to consider adding.

The Tree Is Usually Implicit

As mentioned earlier, note that the possibilities tree used in thinking about branch and bound algorithms is mostly a conceptual tool. When it comes down to implementing such an algorithm, the nodes can turn into method calls, with the necessary information for each node passed in as arguments to the method calls, or can be stored as separate nodes with no actual tree structure. Partly this is helpful because it is easier to not have to literally build a tree, but mostly it is to save memory space. For example, with our branch and bound 0-1 knapsack algorithm, only the nodes that have been created but not yet explored have to be stored at any given time. In the worst case this could still be $\Theta(2^n)$ nodes, but we can reasonably hope that far fewer nodes ever need to be stored at one time.

Efficiency Analyses of Two Approaches to 0-1 Knapsack Problem

The basic (fill in the whole chart) dynamic programming algorithm for the 0-1 knapsack problem takes $O(nW)$ time, since the table has n rows and W columns. If we do the "only compute the value of a cell if it is needed" version we need at most 2^j cells on row $n - j$, requiring $O(2^n)$ time in the worst-case. Thus, this version takes $O(\min(nW, 2^n))$ time in the worst-case.

The branch and bound method has worst-case efficiency in $O(2^n)$, in the case that the entire possibilities tree has to be built. The heuristic element of this approach is that very often far less work will need to be done, because many nodes will be pruned as non-promising.

Exercise 25 (branch and bound for 0-1 knapsack)

Perform the branch and bound algorithm for the 0-1 Knapsack problem similarly to the previous example, on the instance given below.

For convenience, you probably will want to actually draw the tree, noting that at any moment the nodes that have no children are the ones that would be in the priority queue, and the node with the best bound can be found by simply scanning those nodes.

Here's the instance to solve:

$W = 15$ and

i	p_i	w_i	p_i/w_i
1	64	4	16
2	90	6	15
3	91	7	13
4	48	4	12
5	33	3	11
6	10	1	10

Project 9 (implementing 0-1 knapsack branch and bound)

Your job on this project is to write code in Java to implement the branch and bound algorithm for the 0-1 knapsack problem.

Create a Java application named `Project9` that will ask the user for the name of a data file, read the information from the data file, and then solve that instance of the 0-1 knapsack problem using the branch and bound method, following exactly the algorithm demonstrated in the previous example—without an explicit tree.

The data file must contain the capacity of the knapsack, followed by the number of items, followed by the profit and weight information (on one line) for each item. All these values are integers. You may assume that the items are sorted in order from most profitable per unit of weight to least profitable.

As a small but important requirement to avoid irritating me, you *must* label the items starting with 1, rather than 0.

Here is a sample data file:

```
12
6
100 4
120 5
88 4
80 4
54 3
80 5
```

Here is a sample run on this data file, showing the output your program must produce (within cosmetic differences), and, implicitly, the algorithm it must use:

```
Capacity of knapsack is 12
Items are:
1: 100 4
2: 120 5
3: 88 4
4: 80 4
5: 54 3
6: 80 5

Begin exploration of the possibilities tree:

Exploring <Node 1:   items: [] level: 0 profit: 0 weight: 0 bound: 286.0>
      Left child is <Node 2:   items: [] level: 1 profit: 0 weight: 0 bound: 268.0>
         explore further
      Right child is <Node 3:   items: [1] level: 1 profit: 100 weight: 4 bound: 286.0>
         explore further
         note achievable profit of 100

Exploring <Node 3:   items: [1] level: 1 profit: 100 weight: 4 bound: 286.0>
      Left child is <Node 4:   items: [1] level: 2 profit: 100 weight: 4 bound: 268.0>
         explore further
      Right child is <Node 5:   items: [1, 2] level: 2 profit: 220 weight: 9 bound: 286.0>
         explore further
         note achievable profit of 220

Exploring <Node 5:   items: [1, 2] level: 2 profit: 220 weight: 9 bound: 286.0>
      Left child is <Node 6:   items: [1, 2] level: 3 profit: 220 weight: 9 bound: 280.0>
         explore further
      Right child is <Node 7:   items: [1, 2, 3] level: 3 profit: 308 weight: 13 bound: 308.0>
         pruned because too heavy

Exploring <Node 6:   items: [1, 2] level: 3 profit: 220 weight: 9 bound: 280.0>
      Left child is <Node 8:   items: [1, 2] level: 4 profit: 220 weight: 9 bound: 274.0>
         explore further
      Right child is <Node 9:   items: [1, 2, 4] level: 4 profit: 300 weight: 13 bound: 300.0>
         pruned because too heavy

Exploring <Node 8:   items: [1, 2] level: 4 profit: 220 weight: 9 bound: 274.0>
      Left child is <Node 10:   items: [1, 2] level: 5 profit: 220 weight: 9 bound: 268.0>
         explore further
      Right child is <Node 11:   items: [1, 2, 5] level: 5 profit: 274 weight: 12 bound: 274.0>
         hit capacity exactly so don't explore further
         note achievable profit of 274

Exploring <Node 2:   items: [] level: 1 profit: 0 weight: 0 bound: 268.0>
```

```
      pruned, don't explore children because bound 268.0 is smaller than known achievable profit 274

Exploring <Node 4:   items: [1] level: 2 profit: 100 weight: 4 bound: 268.0>
      pruned, don't explore children because bound 268.0 is smaller than known achievable profit 274

Exploring <Node 10:   items: [1, 2] level: 5 profit: 220 weight: 9 bound: 268.0>
      pruned, don't explore children because bound 268.0 is smaller than known achievable profit 274

 Best node: <Node 11:   items: [1, 2, 5] level: 5 profit: 274 weight: 12 bound: 274.0>
```

Your application must display on screen each node when it is first reached, showing the items selected at that node, the total profit for those items, the total weight for those items, and the bound on that node.

Note that your algorithm must use a priority queue to replace manually looking at all the nodes in the current tree to determine which has the best bound. Whenever a node is explored, it should either be pruned, or should be removed from the priority queue with both its children added to the priority queue.

You may implement the priority queue without worrying about efficiency (if you use a heap data structure removing the highest priority item and adding an item will both have efficiency in $\Theta(\log n)$, but it is okay to just use a list of some kind and have removing the highest priority—best bound—node take $\Theta(n)$, with adding being in $\Theta(1)$.

Be sure to test your algorithm thoroughly.

To submit your work, send me an email with the single file `project9.zip` attached, containing all the source code for your project, in portable form (no meaningless `package` statements or hard-coded file names).

Branch and Bound Approach to ETSP

We have already seen the traveling salesperson problem and noted that the brute force approach to this problem—simply generating all the permutations of the n vertices and computing the total weight of each corresponding tour—has time efficiency in $\Theta((n-1)!)$, and that the dynamic programming-based algorithm for this problem has time efficiency in $\Theta(n^2 2^n)$. Since neither of those efficiencies are practical when n is large, we now want to consider a branch and bound approach to this problem.

This approach will lead to a *heuristic algorithm*, which will typically solve an instance of TSP in a reasonable amount of time, but gives no guarantee that the time will not blow up to exponential time for some instances. However, our approach will have the nice property that it will include proof that the tour we find is optimal, if our approach ever finishes.

We will actually work with the Euclidean TSP, henceforth to be named ETSP, where the vertices are viewed as points in the plane and the cost of traveling between vertices is the ordinary geometry (Euclidean) distance between them. This saves us a lot of bother coming up with weights—the weight on the edge between two vertices is the geometrical distance between them. Note that this makes the graph undirected, whereas our earlier weights could be different whether going from v_j to v_k or v_k to v_j.

We will design our heuristic approach by using the branch and bound technique, with the bound provided by approximating the ETSP by a linear programming problem.

Attempting to Formulate ETSP as an LP

We begin by attempting to model ETSP as a linear programming problem. We can come tantalizingly close, but if we could actually do this, we would have proven that P=NP (it turns out that ETSP, even though apparently simpler than general TSP, where the weights can be any positive numbers, is still in NP-complete).

> In our brief study of computational complexity coming up soon, we will learn exactly what this means. The statement above uses the fact, which we will not prove, that LP is in P, namely that there are algorithms (one first discovered by Khachiyan and collaborators in 1979, with a more practical algorithm popularized by Karmarkar in 1984 (actually previously discovered in the 1960's by other researchers)) that have polynomial-time worst-case efficiency. In our following discussion, we will use the simplex method to solve instances of LP, even though it was shown long ago that in the worst-case the simplex method has exponential running time—on average it runs in polynomial time.

So, let's try to write down a linear programming problem that corresponds to a given ETSP instance. We begin by making the fairly obvious choice that our decision variables should be x_{jk}, representing the *intensity* of the connection between vertices j and k, where $x_{jk} = 0$ means that there is no edge at all between vertex j and vertex k, and $x_{jk} = 1$ means that there is an edge between those two vertices. We would like x_{jk} to only have one or the other of those values, but we will see that this, unfortunately, does not always happen.

Note that since the edges in our graph are undirected (because the distance from A to B is the same as the distance from B to A), we only need to use decision variables x_{jk} where $j < k$.

Using these x_{jk} decision variables, if we let c_{jk} be the Euclidean distance between vertices j and k, then the objective function, which we want to minimize, can be taken as

$$z = \sum_{j<k} c_{jk} x_{jk}.$$

We like the standard restriction feature of the simplex method, namely that the obvious restrictions $x_{kj} \geq 0$ are automatic. Now we need to design constraints that will somehow say that the choice of values for these decision variables corresponds to a tour.

We want all x_{jk} to be either 0 or 1, but the closest we can come to saying that in an LP instance is to say that $x_{jk} \geq 0$, which is automatic, and that $x_{jk} \leq 1$, which is an inequality constraint that has to be converted to an equality constraint by adding a slack variable s_{jk} and changing the constraint to $x_{jk} + s_{jk} = 1$.

> These last constraints are unfortunate, because they double the number of variables we have. But, they are essential, because otherwise the optimal solution to the LP can easily have edges with weight 2. This occurs naturally when all the points are paired up and far away from each other.

The trickier constraints try to say that the choice of intensities form a tour. We note that in any tour, exactly two of the edges connected to each vertex have intensity 1 and the others have intensity 0. We can't say exactly that in an LP, but we can say that all the intensities of edges connected to each vertex add up to 2.

For example, if we have 6 vertices, then for example for vertex 3, we have the constraint

$$x_{13} + x_{23} + x_{34} + x_{35} + x_{36} = 2.$$

Let's try out these ideas in a small LP instance.

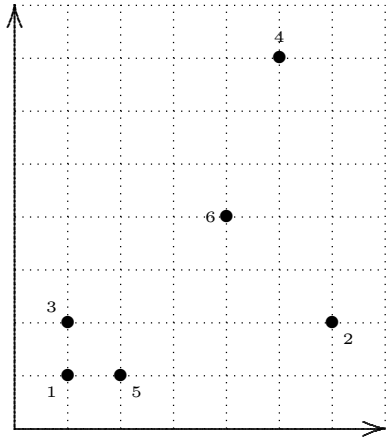
Example of an LP Instance that Models ETSP

⇒ You may want to watch the video `ETSPexample1.mp4`, available at the course web site. It goes through the following example.

Let's tackle the ETSP instance given by these 6 points:

$v_1 = (10, 10)$, $v_2 = (60, 20)$, $v_3 = (10, 20)$, $v_4 = (50, 70)$, $v_5 = (20, 10)$, $v_6 = (40, 40)$

Here is a graph of these points (where the grid lines are at multiples of 10):



Here are the weights (distances) for this problem, namely the values of c_{kj} (rounded to 2 decimal places):

	2	3	4	5	6
1	50.99	10	72.11	10	42.43
2		50	50.99	41.23	28.28
3			64.03	14.14	36.06
4				67.08	31.62
5					36.06

For example,

$$c_{25} = \sqrt{(60-20)^2 + (20-10)^2} = \sqrt{40^2 + 10^2} = \sqrt{1700} = 41.23105625617661 \approx 41.23.$$

Now we can write down the LP instance corresponding to this ETSP instance using the constraints we have figured out:

$$\begin{aligned}\min\ c_{12}x_{12} + c_{13}x_{13} + c_{14}x_{14} + c_{15}x_{15} + c_{16}x_{16} + \\ c_{23}x_{23} + c_{24}x_{24} + c_{25}x_{25} + c_{26}x_{26} + \\ c_{34}x_{34} + c_{35}x_{35} + c_{36}x_{36} + \\ c_{45}x_{45} + c_{46}x_{46} + \\ c_{56}x_{56}\end{aligned}$$

subject to the constraints

$$\begin{aligned}x_{12} + x_{13} + x_{14} + x_{15} + x_{16} &= 2 \\ x_{12} + x_{23} + x_{24} + x_{25} + x_{26} &= 2 \\ x_{13} + x_{23} + x_{34} + x_{35} + x_{36} &= 2 \\ x_{14} + x_{24} + x_{34} + x_{45} + x_{46} &= 2 \\ x_{15} + x_{25} + x_{35} + x_{45} + x_{56} &= 2 \\ x_{16} + x_{26} + x_{36} + x_{46} + x_{56} &= 2 \\ x_{12} + s_{12} &= 1 \\ x_{13} + s_{13} &= 1 \\ x_{14} + s_{14} &= 1 \\ x_{15} + s_{15} &= 1 \\ x_{16} + s_{16} &= 1 \\ x_{23} + s_{23} &= 1 \\ x_{24} + s_{24} &= 1 \\ x_{25} + s_{25} &= 1 \\ x_{26} + s_{26} &= 1 \\ x_{34} + s_{34} &= 1 \\ x_{35} + s_{35} &= 1 \\ x_{36} + s_{36} &= 1 \\ x_{45} + s_{45} &= 1 \\ x_{46} + s_{46} &= 1 \\ x_{56} + s_{56} &= 1\end{aligned}$$

$$x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{23}, x_{24}, x_{25}, x_{26}, x_{34}, x_{35}, x_{36}, x_{45}, x_{46}, x_{56},$$

$$s_{12}, s_{13}, s_{14}, s_{15}, s_{16}, s_{23}, s_{24}, s_{25}, s_{26}, s_{34}, s_{35}, s_{36}, s_{45}, s_{46}, s_{56} \geq 0$$

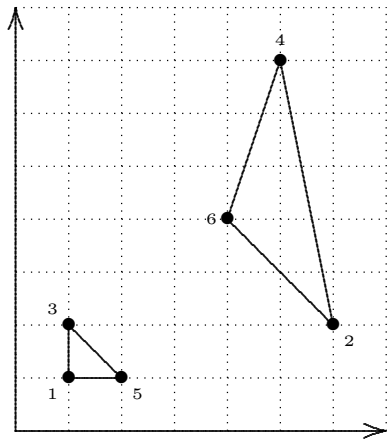
Then we create, in the file `Code/ManualSimplex/etsp1`, the Phase 1 tableau for this LP instance, recognizing that we need an artificial variable for each of the first kind of constraints.

> The slack variables can serve as initial basic variables for all the $x_{jk}+s_{jk}=2$ constraints.

Here is this starting tableau:

	z	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{23}	x_{24}	x_{25}	x_{26}	x_{34}	x_{35}	x_{36}	x_{45}	x_{46}	x_{56}	s_{12}	s_{13}	s_{14}	s_{15}	s_{16}	s_{23}	s_{24}	s_{25}	s_{26}	s_{34}	s_{35}	s_{36}	s_{45}	s_{46}	s_{56}	a_1	a_2	a_3	a_4	a_5	a_6	rhs
z	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0
a_1	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	2
a_2	0	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
a_3	0	0	1	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	2
a_4	0	0	0	1	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	2
a_5	0	0	0	0	1	0	0	0	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2
a_6	0	0	0	0	0	1	0	0	0	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2
s_{12}	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
s_{13}	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
s_{14}	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
s_{15}	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
s_{16}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
s_{23}	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
s_{24}	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
s_{25}	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
s_{26}	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1
s_{34}	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
s_{35}	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
s_{36}	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
s_{45}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1
s_{46}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
s_{56}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1

After doing Phase 1 and Phase 2, we obtain the optimal point with $z = 145.03$, and basic original decision variables $x_{24} = 1$, $x_{46} = 1$, $x_{26} = 1$, $x_{35} = 1$, $x_{13} = 1$, $x_{15} = 1$, and $x_{23} = 0$ (basic but at 0). We can draw the edges of intensity 1 to see this optimal solution:



Note that this is the optimal solution to the LP that is trying to model ETSP, but it is clearly not the optimal solution to ETSP—it is not even valid. Basically it is cheating, from the perspective of ETSP, getting a lower score than any possible tour, but not being a tour.

This example motivates us to introduce another kind of constraint.

Cuts

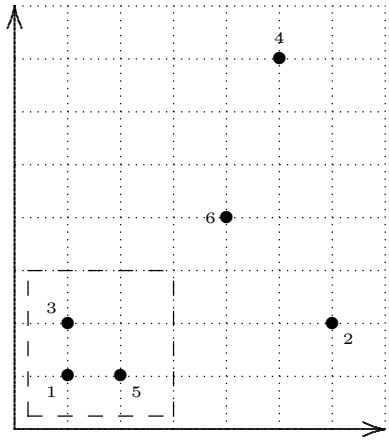
It turns out we can sort of solve (again, it's a heuristic approach—often when we try this technique, it works, but we have no guarantee that it will always work) the disconnected sub-tours problem by a clever idea, known as a "cut."

The idea is a generalization of the earlier idea that the sum of all the edges coming into a vertex must be 2.

If we consider any non-empty subset S of the set A consisting of all the vertices, then we know that for any tour the number of edges crossing the boundary between S and $A - S$ must be at least 2. This is a little hard to describe carefully, but is actually pretty obvious.

So, we can add the corresponding constraint, and change our LP to include this constraint that forces any feasible point of the LP to satisfy this property.

To apply this idea to our instance, let's take $S = \{1, 3, 5\}$:



the corresponding cut constraint is

$$x_{12} + x_{14} + x_{16} + x_{23} + x_{34} + x_{36} + x_{25} + x_{45} + x_{56} \geq 2.$$

So, our current idea is to start with an LP instance that uses the original constraints (which are really cut constraints with $S = \{v_j\}$), along with the $x_{jk} \leq 1$ constraints, and do the simplex method to obtain the optimal tableau. Then, if we observe that we don't have a single connected tour, we add an appropriate cut constraint and solve the new LP instance that has one more constraint.

> Note that if we added all such constraints—one for every subset of A—we would have an LP instance for which the only feasible points were tours. The problem with this approach, of course, is that there are 2^n such subsets, so we'd be adding 2^n constraints, which would make the LP far larger and more costly to compute than the good old dynamic programming chart. So, in a heuristic spirit, we are adding just a few constraints from cuts as needed, we hope, to somehow get an optimal point for the LP which happens to be a tour.

Continuing the Example

Now we could add the additional cut constraint described above. Adding this constraint would unfortunately require adding a surplus variable, say just named s_1, and another artificial variable, a_7, to be the basic variable for the new row at the start of Phase 1.

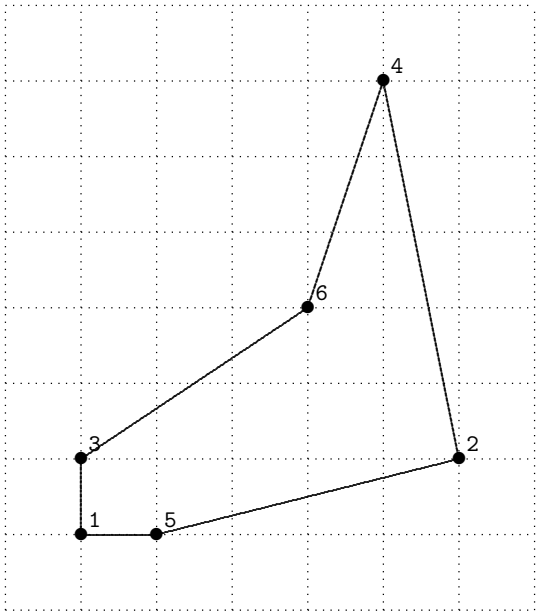
But, somewhere around here this whole process becomes too cumbersome to be worth it! Instead, I took some time and created a Java application `HeuristicTSP` that takes the data for the points, constructs the corresponding tableaux that tries to model ETSP for those points, and then automatically does the simplex method to find the optimal point, graphically displaying the results.

If we make the file `Code/ETSP/points1` and run `HeuristicTSP` on it, we see the solution we saw before, with two sub-tours.

But, `HeuristicTSP` also has the ability to add any number of additional "cut" constraints. We simply put in the data file at the bottom the desired cut, namely

```
cut 1 3 5
```

The application adds the cut constraints to the tableaux and solves it. If we do this for `points1`, we get this optimal point:



This is in fact the optimal tour—the solution to the ETSP instance. The objective function value is 179.9, which worse—bigger—than the earlier optimal point for the LP. This makes sense because without the cut constraint, the LP solution could cheat and have two sub-tours.

We can prove that this is the optimal point for ETSP, because every tour is a feasible point for the LP. The problem is that there are lots of feasible points for the LP that

aren't tours. So, if we stumble across a tour that is the best feasible point for the LP, then it is of course the best tour.

> If we were feeling skeptical we could run `DynProgTSP` on this instance to see if it thinks this is the optimal tour.

Exercise 26 (modeling ETSP by LP)
Write down in symbolic form (not a matrix and don't convert inequalities to equalities) all the constraints for any ETSP instance with 5 points and a cut that encloses vertices 1 and 3.

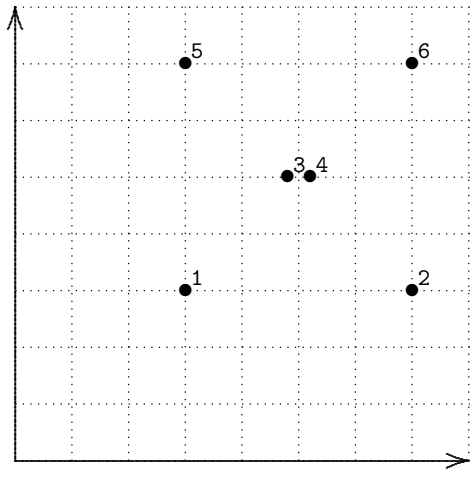
So, now we are feeling pretty good about the whole business, maybe, but the next little example will disabuse us of that optimism.

Another ETSP Example to Try

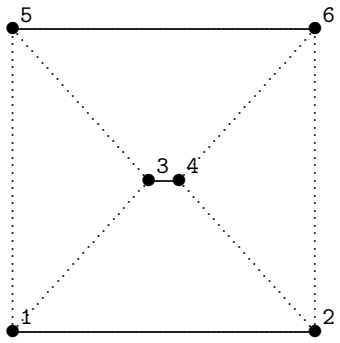
Consider these points (discovered by Randy Sorensen, a student in CS 4050 in 2015):

```
30 30
70 30
48 50
52 50
30 70
70 70
```

Here is a picture of these points:



These points are in the file `Code/ETSP/sorensen`. If we run `HeuristicTSP` on this data, we get this optimal point for the LP:



where the dotted edges all have intensities of $\frac{1}{2}$. The solid edges, as usual, have intensity 1.

This example shows the final and most devastating way in which the LP can cheat—can find an optimal point that is not a tour, despite our best efforts to put in constraints that force a tour. And, sadly, we have no way to effectively deal with this by simply adding more constraints.

> If we imagine drawing a border around any subset of the vertices, we see that the total intensity crossing that border is at least 2. If we think of cutting vertices 5 and 6, for example, we see that they are connected to the other 4 vertices by four edges with intensity $\frac{1}{2}$, for a total of 2. If we cut vertices 1, 3, and 5, we see that they are connected to vertices 2, 4, and 6 by three edges of intensity 1, for a total of 3. The cut with vertices 2 and 5 gives connection intensities totaling 4.

Branch and Bound Algorithm for ETSP

We have finally reached our main goal, namely seeing how branch and bound ideas can be applied to the ETSP.

The idea is very simple:

> when we have an LP, including any cuts (note that all tours satisfy all cuts, so adding a cut constraint will never eliminate any tour from the feasible set, so we can add cuts freely), whose optimal solution includes one or more of these nasty edges with non-integer intensity, corresponding to some variable x_{kj} having a value strictly between 0 and 1 (for example $\frac{1}{2}$), we can *branch* by adding a constraint $x_{kj} = 0$ or $x_{kj} = 1$. Then we can use our usual ideas for branch and bound, eventually, we hope, reaching an LP whose solution is a tour, and eventually one whose solution must be optimal.

First, note that it is easy to add constraints $x_{kj} = 0$ or $x_{kj} = 1$ to an LP, without actually adding stuff.

> After creating a tableau with any desired cuts added as constraints, we can then go through and for each `zero k j` command (for `HeuristicTSP`) simply delete the column

for x_{kj}, the column for s_{kj}, and the constraint row $x_{kj} + s_{kj} = 1$. For each `one k j` command, we do those same things, except before deleting the column for x_{kj}, we subtract all its components from the corresponding right hand side vector.

Actually, this is the approach taken by the version of `Tableau` in the `AutoHeuristicTSP` sub-folder—the one in `ETSP` is slightly older (and the newer one is slightly incompatible with `HeuristicTSP`).

Here is a heuristic algorithm, using the branch and bound technique, for solving, perhaps, any given ETSP instance:

Get the root of the branch and bound tree by solving the original version of the related LP. If the optimal solution gives disconnected sub-tours, add in `cut` constraints until one connected graph is obtained.

If the graph is a tour, it is the optimal tour, and we can stop with the solution to the ETSP.

Otherwise, look for a "red" edge (intensity not 0 or 1). Branch on the choice of that edge being forced to be either present ($x_{kj} = 1$) or not ($x_{kj} = 0$).

Repeat in the usual branch and bound way, pursuing a depth-first search for a tour (to reach a tour more quickly for bounding purposes), or using a best-first search (to pursue the most promising branches first). When a tour is obtained, it is the optimal tour for the original problem plus the specific choices made along the branch to reach it. It therefore provides a bound on all the other nodes—if any node has a worse score than this tour, then that node can be pruned.

For each node reached, we can freely add cut constraints until a single connected graph is obtained. The scores will get worse, because we are adding constraints, but we will never be eliminating tours by adding these constraints.

A Small Example of the Branch and Bound Algorithm for TSP

Consider the Sorensen problem. We apply `HeuristicTSP` to the base problem, obtaining the optimal solution with $z = 177.8145$ (rounding to 4th decimal place, and ignoring the $-$ introduced by doing minimization instead of maximization).

Next we pick a variable, say x_{15}, on which to branch. We add `zero 1 5` to the `sorensen` data file and obtain an optimal solution to the corresponding LP with $z = 180.6394$. Since this solution is a tour, we prune the tree below that node.

Then we consider the other possibility, and replace the $x_{15} = 0$ constraint with $x_{15} = 1$. This gives an optimal solution to the corresponding LP with $z = 177.8145$, which is a tour.

The optimal choice, resulting in the optimal tour, is obviously the node produced by using $x_{15} = 1$.

A Bigger Example of the Branch and Bound Algorithm for TSP

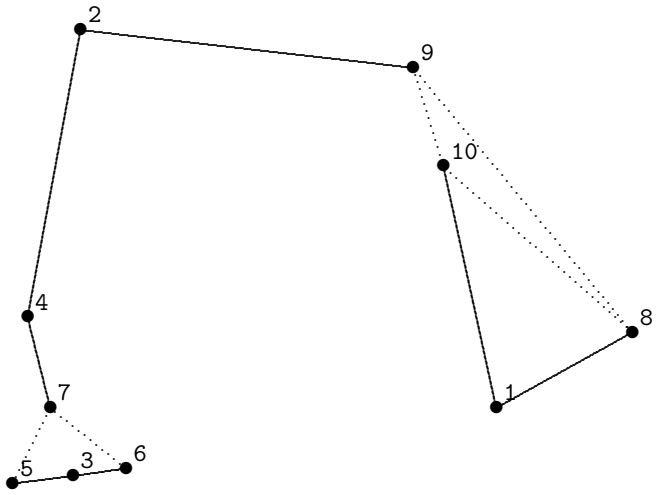
Consider these 10 points giving a ETSP instance (in the data file `page165`)

```
74 17
19 67
18 8
12 29
10 7
25 9
15 17
92 27
63 62
67 49
```

We will make various choices in the following in order to illustrate the branch and bound ideas. Later we will see that we can solve this problem in an easier way by making different choices.

As we go through this example, note that we should be writing on scratch paper a sketch of our branch and bound tree, showing the branches we have made with choices to require some x_{jk} to be either 0 or 1, with each node containing its score, which is the objective function value for the corresponding LP at that node.

If we run `HeuristicTSP` on this problem, we obtain this optimal point for the LP, where solid lines represent decision variable values of 1 (blue edges), and dotted lines represent decision variable values of $\frac{1}{2}$ (red edges):

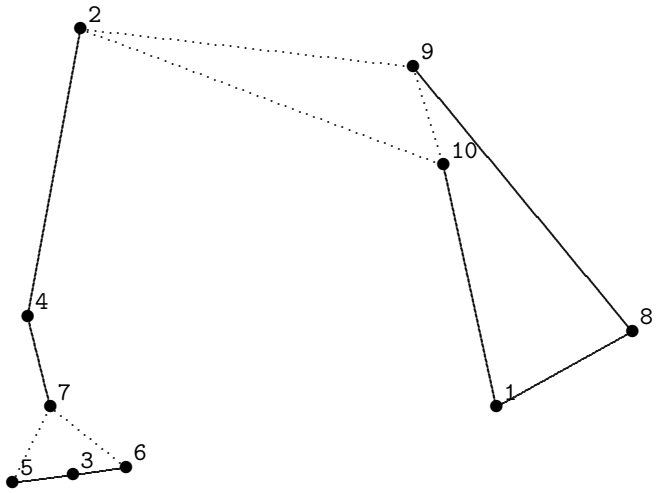


Since the graph is connected, we don't use a cut (we could do `cut 2 3 4 5 6 7` to good effect, but we want to demonstrate the branch and bound algorithm, so we won't), but instead pick a red/dotted edge on which to branch. Let's use $x_{8,9}$, because it looks like it might or might not want to be in the optimal tour (picking a good edge on which to branch is a heuristic element of this algorithm).

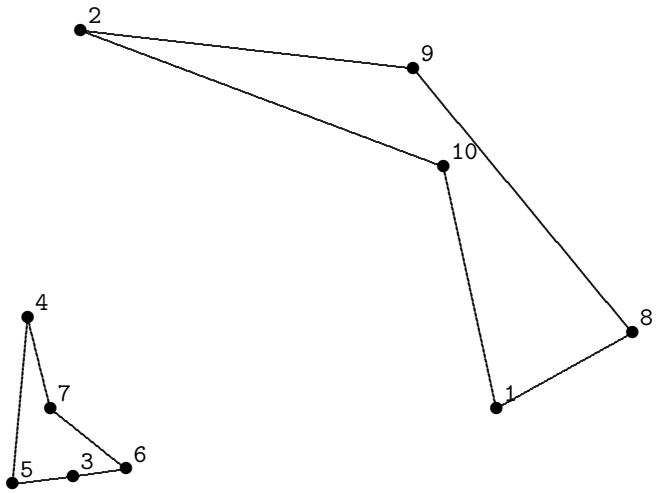
So, we add

`one 8 9`

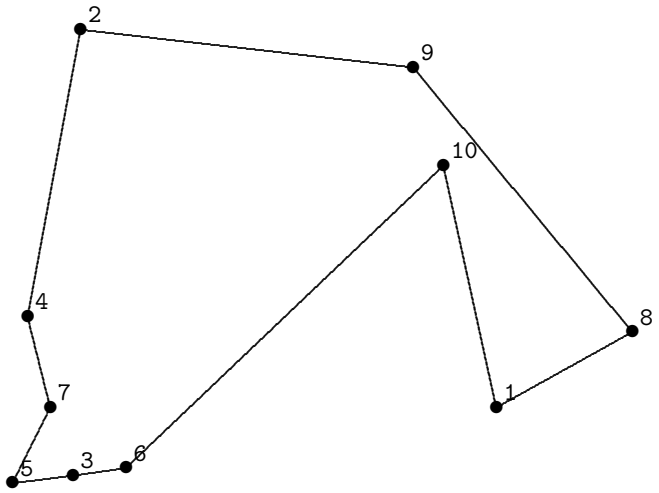
to our data file and run the application again, obtaining this graph:



Again we have a connected graph with some red/dotted edges, so we branch on x_{29}. If we use $x_{29} = 1$, we obtain

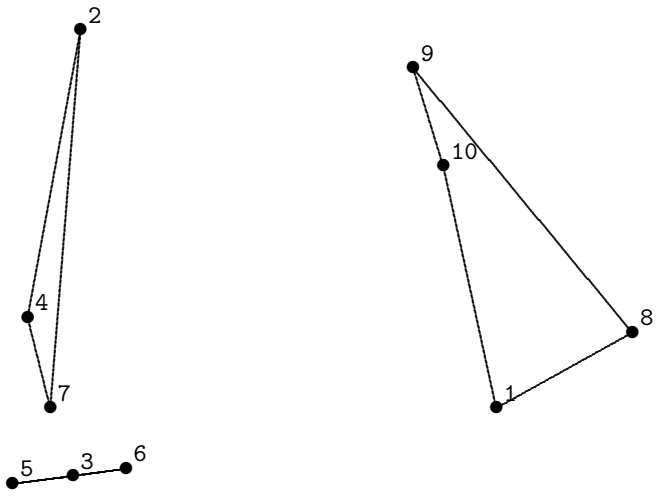


Now we add `cut 3 4 5 6 7` to try to eliminate the two sub-tours, and we obtain

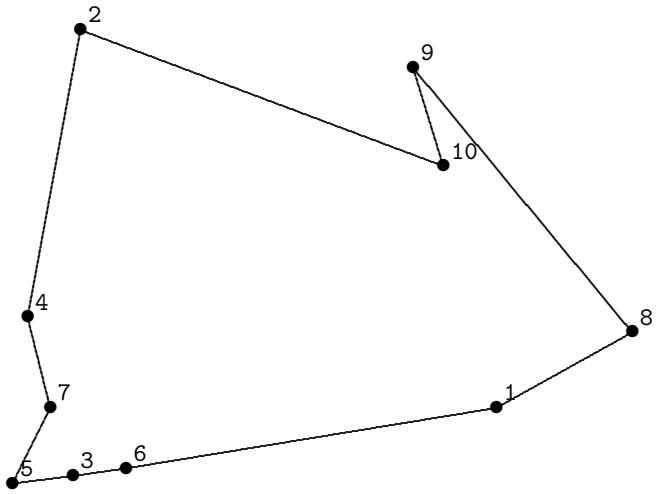


with a score (total weight) of 278.40673. We are now very excited because we've found a tour, which means that if we find any node with score bigger than 278.40673, we can prune it.

Now we have one unresolved node, so we use the branching choices $x_{8,9} = 1$ and $x_{2,9} = 0$ and obtain

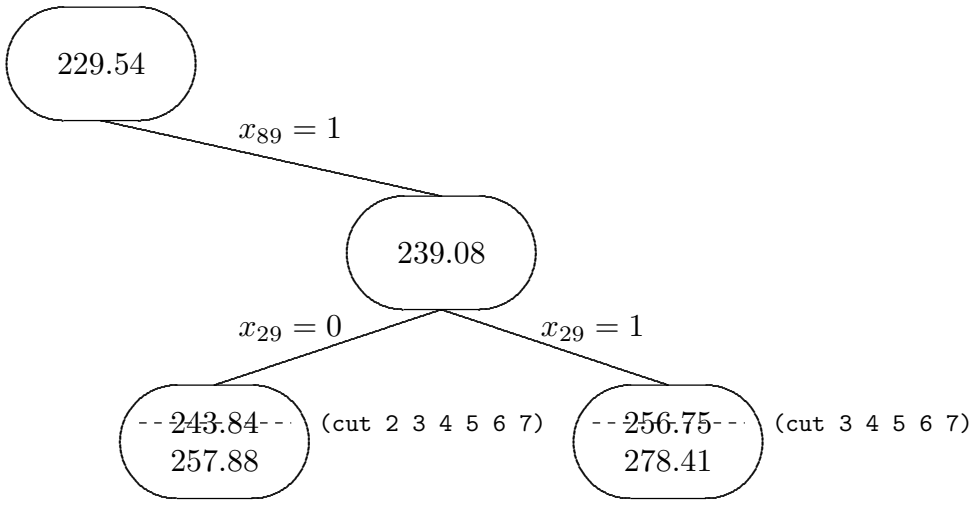


with a score of 243.83717. Since this score is better than our bound, we can't prune this node. But, since it is disconnected, we can add a cut, say `cut 2 3 4 5 6 7` yielding

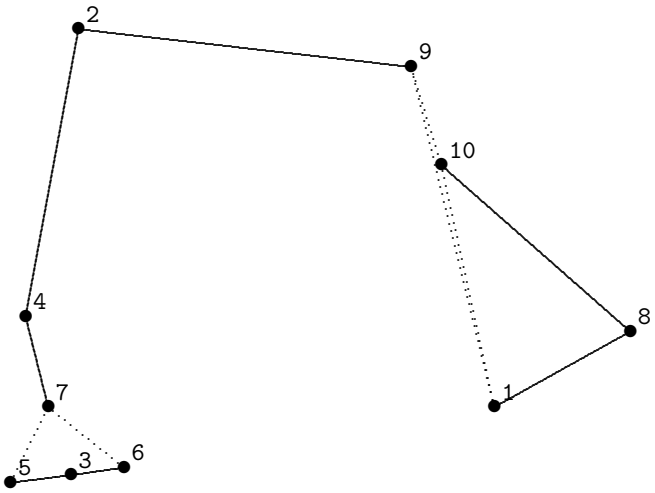


with a score of 257.88113. Now we stop, since it's a tour, and notice that we have found a tour with a better score than our previous best (only, actually) tour, so we update our bound and continue.

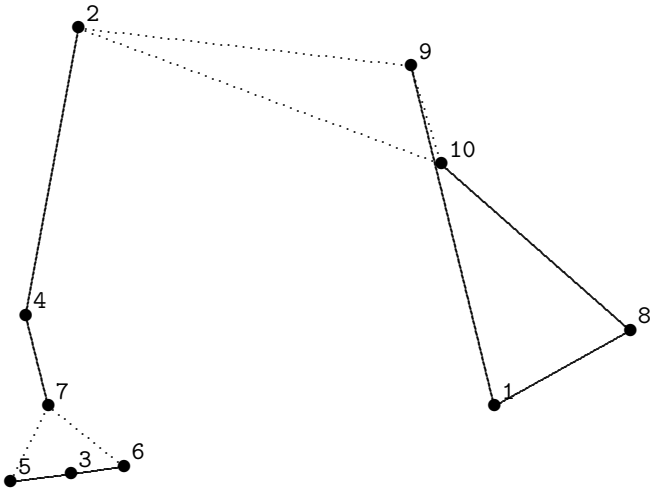
At this point our hand-drawn sketch of the branch and bound tree looks like this:



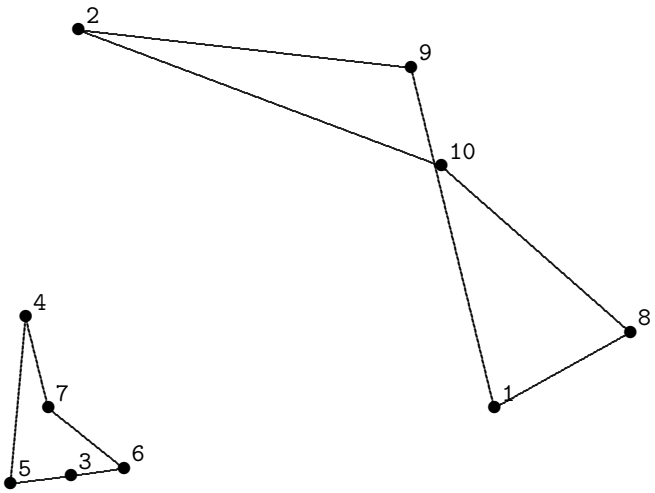
Now we pursue the possibility, back at the root node, that x_{89} should be 0, obtaining



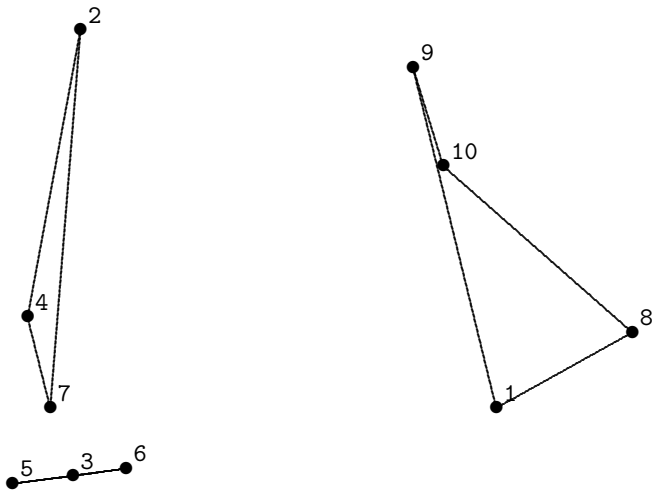
with a score of 230.21930. We decide to branch from here on $x_{1,9}$. If we pursue $x_{1,9} = 1$, we obtain



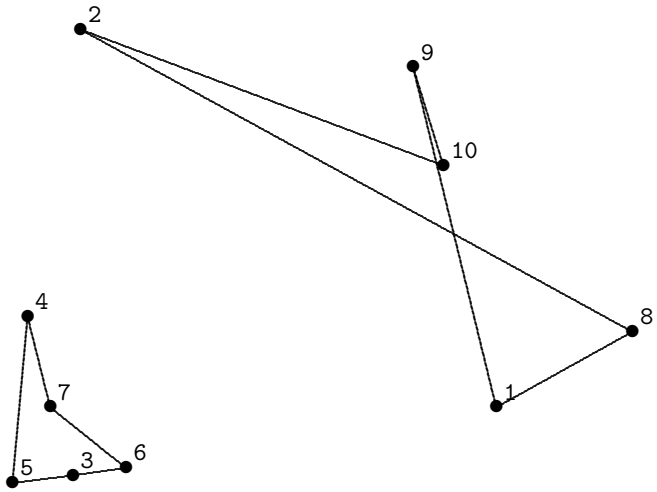
Now we branch from here, first using $x_{2,9} = 1$ and obtaining



with a score of 258.16466. Since this is worse than our current best known tour score (257.88), we prune this node (since a cut will only make its objective function value worse, we don't bother). On the other branch, with $x_{2,9} = 0$, we obtain

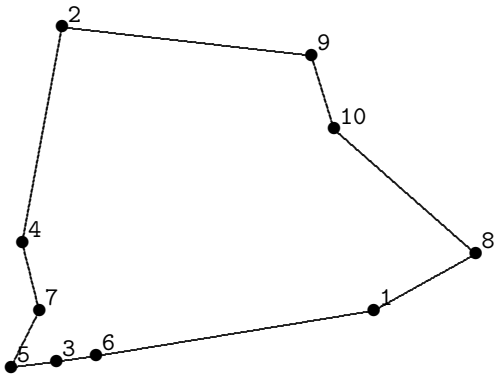


with a score of 245.25381. This means, sadly, that we should pursue this node further. Since it is disconnected, we'll do `cut 2 3 4 5 6 7`, and obtain



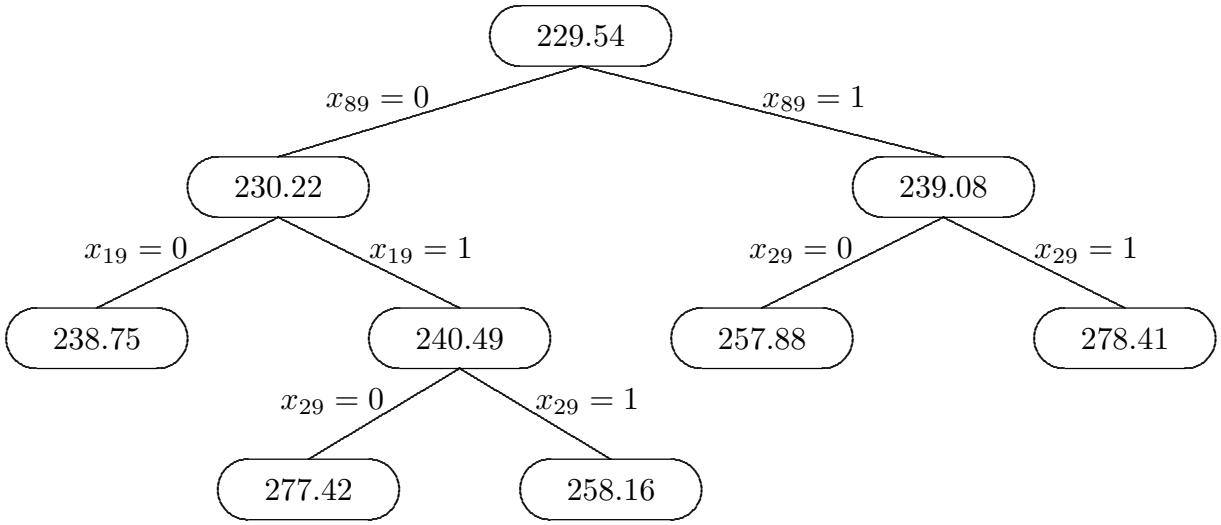
with a score of 277.42192, which allows us, thankfully, to prune this node.

Now we go back to the only unresolved node, with $x_{8,9} = 0$ and $x_{1,9} = 0$, obtaining



with a score of 238.74866. This is, obviously, the optimal tour, and we are done!

Here is the possibilities tree that we created through this process (with objective function values rounded to two decimal places):



Note that if we had carefully pursued nodes with better scores first, we would have found this much sooner (but I've rigged the choices in this example to show more action). This is a heuristic sort of thing, using the node with better score first, even though the scores don't mean anything provable unless the corresponding graph is a tour.

Also, it turns out that we can profitably make cuts when a whole "peninsula" of vertices is only connected to the others by two red edges.

This works because such a cut constraint says that all the edges crossing from the cut points to the other points have to sum to at least 2, which forces more connections than the total of 1 resulting from the 2 red edges at 1/2 each. This seemed to save a lot of branching compared to tackling red edges by branching on them.

Semi-Automated Branch and Bound Algorithm for ETSP

⇒ You should watch the video `ETSPexample2.mp4` for a demonstration of using `AutoHeuristicTSP` to solve an instance of ETSP by the branch and bound approach (this is what you need to understand for Exercise 27).

In the folder `AutoHeuristicTSP` you will find the application `AutoHeuristicTSP` that somewhat automates doing the best-first version of the branch and bound algorithm. It eliminates the need to manually edit the input file by providing these handy commands:

Key	Mnemonic	Action
c	cut	enter a sequence of vertex numbers separated by spaces and terminated by `enter`, and can hit `backspace` as desired, specifying a cut constraint to tbe added
b	branch	enter a pair of vertex numbers specifying a desired branch—nodes setting that variable to both zero and one are added to the priority queue
t	tour	inform the application that the current node gives a tour so it can update its knowledge of the best tour found so far

After solving each LP instance corresponding to a node in the branch and bound tree, the result is shown graphically, with intensity 1 edges shown in blue, intensity 0 edges not drawn, and intensity $\frac{1}{2}$ edges shown in red (edges can have other intensities—they are set to different, other colors). Edges that have been set to 0 in the current node will be shown in white, and edges that have been set to 1 will be shown in black.

Also, as you can see by examining the output in the console window, this application maintains a priority queue of unexplored nodes and always explores the node with the best score, putting it on display so you can cut it repeatedly, and then branch as needed.

Note that in order to fully automate this semi-automatic "algorithm," we would need a way to identify sub-tours so we could add cut constraints. An algorithm such as Prim's or Kruskal's algorithms for finding minimum spanning trees could be used to detect disconnected sub-graphs to be used with cuts. A harder thing would be to spot the "two red edge peninsulas." Also, of course, if we automated this algorithm, we would need a heuristic for determining which node to explore next. An obvious one would be to explore the node with the smallest score. Probably the most difficult work to fully automate this algorithm would be to come up with a good heuristic for selecting the non-integer intensity edge on which to branch.

Exercise 27 (solving ETSP by branch and bound)
Your job on this Exercise is to solve some fairly large ($n = 30$) instances of ETSP using a best-first branch-and-bound algorithm that uses an approximating LP to provide the bounds.

> For an instance of ETSP with $n = 30$, the dynamic programming approach would take $\Theta(n^2 2^{n-1})$ or so steps, which is some multiple of 900 billion steps, requiring a table with roughly a billion rows and 30 columns).
>
> So, it should be fairly impressive that our new approach will be able to solve such problems quite easily!

To do this, just run `AutoHeuristicTSP` with the desired data file as input, and interactively do cuts and branches until the optimal tour is found.

First solve the instance `bad1`, drawing by hand the branch and bound tree following the style used on page 171 (`AutoHeuristicTSP` keeps track of everything for you, so you don't really need to draw the tree to solve the instance, but I'm requiring it to force you to practice what you'll need for Test 3).

Then, solve the instance `bad2`.

For both instances, report the optimal tour length and the branching choices that led to the node for this optimal tour in the branch and bound tree.

> I came up with `bad1` and `bad2` by using `RandomTestingHeuristicTSP` with $n = 30$ to generate random test instances with 30 points and picking ones that looked hard.