CS 491 I Approximation Algorithms
Lecture Notes

David Owen

March 22; April 3,5

1 Network Structures

Given a graph, we would like to identify a structure within the graph (a tree or a cycle, for example) that is optimized for some objective function. We consider the following network structure problems: Minimum Steiner Tree, Traveling Salesman.

For graph problems we will generally assume:

1. The graph G is complete.
2. Edge weights $w_{ij} + w_{jk} \geq w_{ik}$ $\forall i, j, k$ (called the "Triangle Inequality Condition").

1.1 Minimum Steiner Tree Problem

STP (Minimum Steiner Tree Problem):

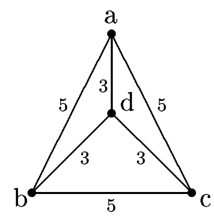
Given a complete graph $G = (V, E)$ with edge weights w_i and a set of vertices $V' \subseteq V$, find a Steiner Tree (a subtree of G including all vertices in V') with minimum $\sum_{i=1}^{n} w_i$.

We observe that, when $V' = V$, STP (the Steiner Tree Problem) is identical to the MST (The Minimum Spanning Tree Problem) for V, which is comparatively easy to solve. Also, if $|V'| = 2$, STP is the Shortest Path Problem for the two vertices in V' and the graph G. But in the general case (when $|V'|$ is unrestriced), STP is NP-complete.

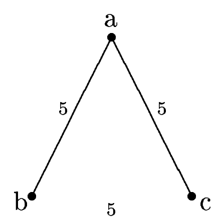
Definitions:

1. *Steiner Vertices* are those vertices in $V - V'$ (in V but not included in the Steiner Tree).
2. Euler Tour: A tour going through every edge of a graph exactly once (and returning to its starting point).
3. Eulerian Graph: A graph with an Euler Tour (note: a graph is Eulerian if and only if all of its vertices have even degree).

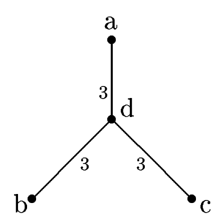
Consider the following graph, with $V = \{a, b, c, d\}$ and $V' = \{a, b, c\}$:



Perhaps we can approximate the STP on V by finding MST on V'. MST on V' will be a feasible Steiner Tree, since it must include all vertices in V'. But it may not be the *minimum* Steiner Tree, because a Steiner Tree may include vertices in $V - V'$, while MST on V' may only include vertices in V'. In this case MST on V' has greater cost than STP:

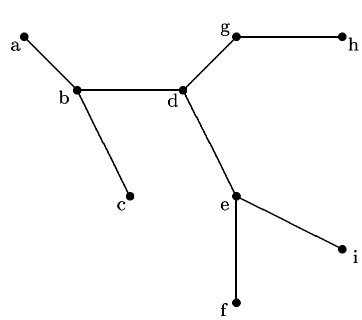


MST on V' (cost $= 10$).

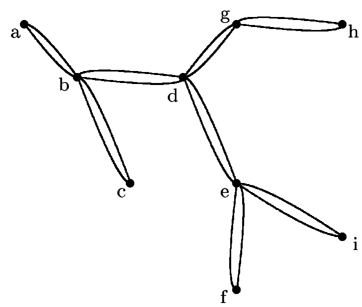


STP (cost $= 9$).

So MST on V' is not necessarily the optimal solution, but how bad is MST on V' as an approximation for STP? We would like to show that any Steiner Tree for V will be a Spanning Tree for V', and we want to argue some bound for the cost of that Spanning Tree compared to MST for V'. Consider an optimal Steiner Tree with cost $= OPT$ and $a, c, f, h, i \in V'$ (there may be more to the graph of V than is shown here; this is just the optimal Steiner Tree):

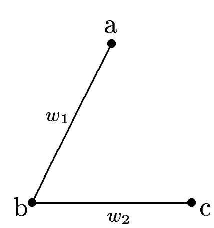


Consider the Eulerian Graph formed by replacing each edge with two edges (I have not drawn edge weights in this graph; the two new edges would each be assigned the same weight as the single edge they replace):

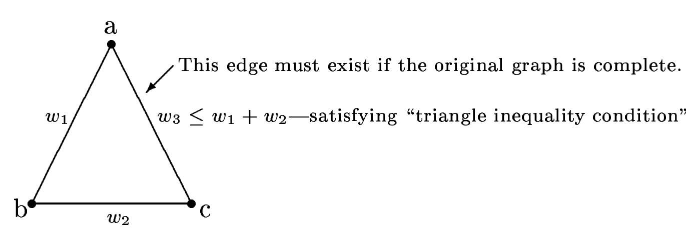


Since we have doubled all the graph's edges, we know that all vertices must now have even degree. As stated above, a graph is Eulerian if and only if all its vertices have even degree. So our graph must now be Eulerian, which means that it must have an Euler Tour. The Euler Tour of the graph goes through all of its edges. Since the new edges have each been given a weight equal to the single edge they replaced and the cost of the original Steiner Tree $= OPT$, the cost of the Euler Tour will be $2 \cdot OPT$.

We assume (as stated above) that our graph is complete and that the "triangle inequality condition" holds. This means that if in our original graph we have nodes a, b, and c—with a connected to b and b to c—we can draw an edge from a to c bypassing b (this edge must exist because our graph is complete) and be assured that the weight (or cost) of the new edge is $\leq$ the sum of the weight of the two old edges from a to b and from b to c (because of the "triangle inequality condition"). The pictures below will hopefully clarify the last few sentences:



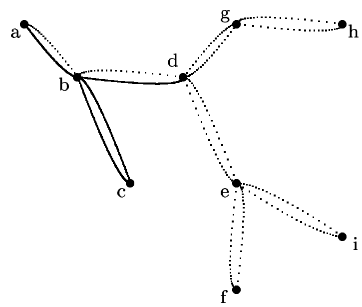
A graph with edges from a to b (weight $= w_1$) and b to c (weight $= w_2$).



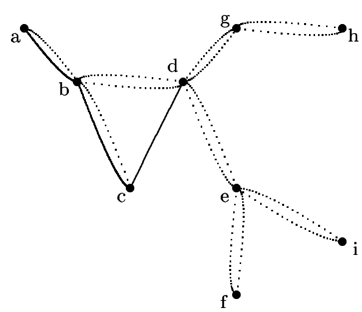
A new graph with an edge drawn directly from a to c bypassing b.

We again consider our doubled-edge Euler Tour graph. It is possible to construct a Hamiltonian Cycle from the Euler Tour. The Euler Tour, in order to cover all the graph's edges, must repeat certain vertices. A Hamiltonian Cycle need only cover each vertex once. So when we construct a Hamiltonian Cycle from our Euler Tour, we can "shortcut" repeated vertices in the same way we drew the edge from a to c bypassing b in the example above (because we have assumed a complete graph and the "triangle inequality condition").

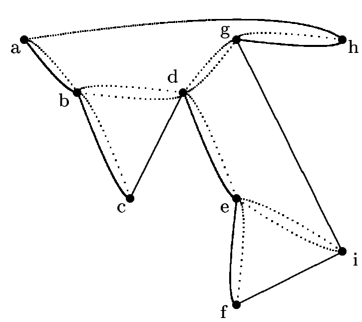
For example, our Euler Tour goes from a to b, from b to c, back to b, and then to d:



We can replace the edges from c back to b and from b to d by a single-edge "shortcut" directly from c to d. This "shortcut" edge must exist, because the graph is complete; and its weight must be $\leq$ the sum of the weights of the two edges it replaces, because of the "triangle inequality condition":



If we work our way through the entire Euler Tour, bypassing repeated vertices with new "shortcut" edges, we will get a Hamiltonian Cycle:



Because the "shortcut" edges must have weight $\leq$ the sum of the edges they replace (the "triangle inequality condition"), the total Hamiltonian Cycle cost $\leq$ Euler Tour cost $= 2 \cdot OPT$.

If we simply remove one edge from the Hamiltonian Cycle, we have a spanning tree ST' for V', the cost of which must be $\geq MST$ (the cost of the a minimum spanning tree on V' can not be greater than the cost of this particular spanning tree). And ST' must also have cost $\leq$ the Hamiltonian Cycle from which it was constructed, since it has one fewer edges. Therefore:

$$MST \leq \text{cost}(ST') \leq \text{our Hamiltionian Cycle} \leq 2 \cdot OPT$$

$$MST \leq 2 \cdot OPT$$

We can 2-approximate STP (the Minimal Steiner Tree Problem by finding MST (the Minimum Spanning Tree).

Summary of the Analysis:

1. Consider Minimum Steiner Tree on some V'; double the edges to assure that every vertex will have even degree—the new graph will be Eulerian.
2. Construct an Euler Path (through every edge). This must be possible since the graph is Eulerian. The cost of the Euler Path will be $2 \cdot OPT$.
3. Construct a Hamiltonian Cycle by "shortcutting" repeated edges in the Euler Path. The cost of this Hamiltonian Cycle must be $\leq 2 \cdot OPT$.
4. Delete any one edge from the Hamiltonian Cycle to make a spanning tree. The cost of this tree must be $\geq$ the cost of the Minimum Spanning Tree. Therefore MST on $V' \leq 2 \cdot OPT$.

1.2 Traveling Salesman Problem

TSP (Minimum Traveling Salesman Problem):

Given a weighted graph G, find the minimum total cost tour through all vertices (and back to the starting vertex).

1.2.1 General TSP

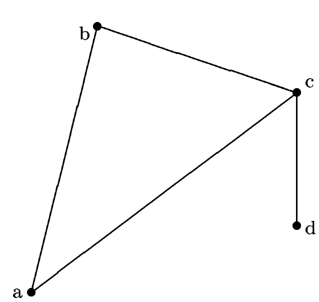
Theorem:

Unless $P = NP$, there is no c-approximation for general TSP (by *general* we mean allowing for graphs that may not be complete or for which the "triangle inequality condition" does not hold).

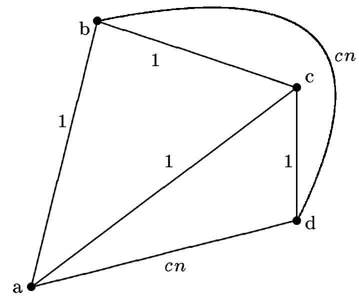
Proof: if there is an algorithm that c-approximates general TSP, then (we show that) the Hamiltonian Cycle Problem can be solved by the same algorithm. In terms of a reduction relationship:

$$\text{Hamiltonian Cycle Problem} \leq c\text{-approximate solution to general } TSP$$

Consider some unweighted graph G:



Construct G' by assigning weight $= 1$ to all edges of G, and then drawing as many new edges as is required to make the graph complete. Assign these new edges weight $= cn$ (note that G' is complete but does *not* satisfy the "triangle inequality condition" for $c \geq \frac{2}{n}$):



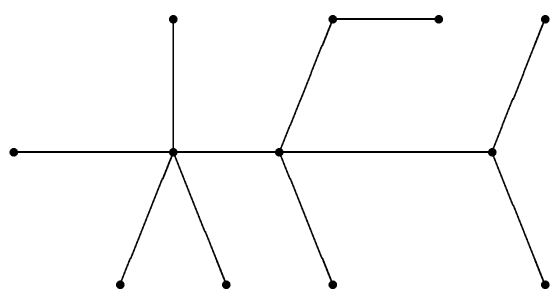
If G had a Hamiltonian Cycle (in this case G does not), G' would have the same Hamiltonian Cycle, and since it would be made up of edges originally in G, all of its edges would have weight $= 1$, so that its total cost would be $= n$. In fact this cycle would be the Minimum (optimal) TSP solution for G'. If we had an algorithm that could c-approximate TSP for G', we could get an approximate solution for TSP of G' between n and cn.

But in this case, since G has no Hamiltonian Cycle, we will need to include at least one of the new cn-weight edges in any feasible TSP solution for G'. Therefore our c-approximation algorithm (if we had one) would have to return a solution with cost $\geq cn$. We could use this algorithm (again, if it existed) to determine whether any arbitrary graph has a Hamiltonian Cycle. We would simply assign weight $= 1$ to all the edges in the graph, construct a new complete graph from that one by adding cn-weight edges, and then run our algorithm on the new graph. If we get an approximate TSP solution $\geq cn$, we know our original graph has no Hamiltonian Cycle (if we get an approximate TSP solution $\leq cn$, we know our original graph does have a Hamiltonian Cycle). So the theorem is proved: unless $P = NP$, there is no c- approximation for general TSP.

1.2.2 TSP Restricted to Complete Graphs Satisfying "Triangle Inequality Condition"

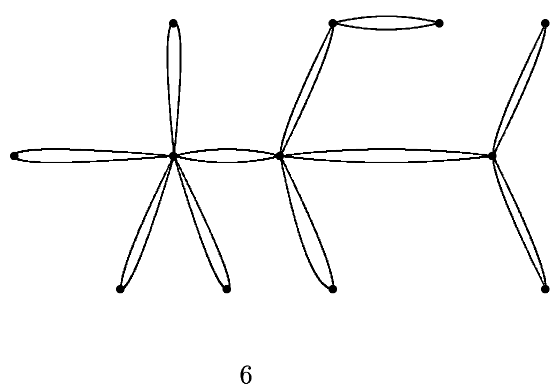
Now let us consider TSP restricted to graphs matching our earlier assumptions (G is complete; G satisfies "triangle inequality condition").

Consider the Minimum Spanning Tree of some graph G:

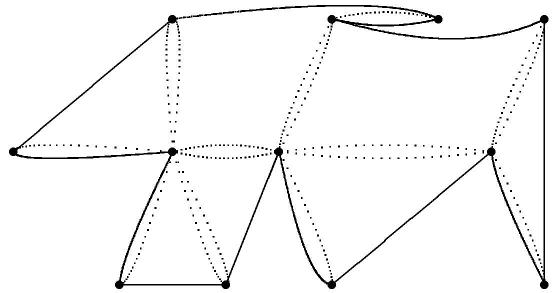


Because G is complete and satisfies the "triangle inequality condition," we can use the same procedure followed above for the Steiner Tree Problem.

1. Double the edges, assigning to each of the new edges the weight of the single edge they replace (the Euler Tour through all these edges will have cost $2 \cdot MST$):



2. "Shortcut" repeated vertices to make a Hamiltonian Cycle (with cost $\leq 2 \cdot MST$):



We can convert an arbitray graph to an Eulerian Graph by doubling the edges. And the Euler Tour for this new graph can be found in polynomial time. We have shown above that in complete graphs for which the "triangle inequality condition" holds, the Euler Tour may be converted into a Hamiltonian Cycle with equal or lower cost. This Hamiltonian Cycle must have cost $\leq 2 \cdot MST$, and because MST must be $\leq OPT$ (optimal TSP), our Hamiltonian Cycle must have cost $\leq 2 \cdot OPT$:

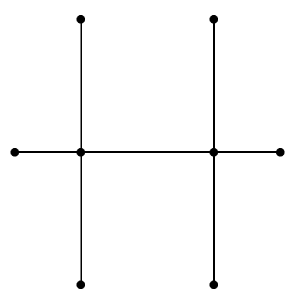
$$MST \leq OPT$$

$$\text{cost(the Hamiltonian Cycle we have constructed)} \leq 2 \cdot MST \leq 2 \cdot OPT$$

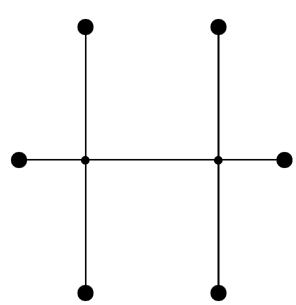
1.2.3 A Better Bound for Complete Graphs Satisfying "Triangle Inequality Condition"

In the strategy above, we double all edges in order to make sure all vertices have even degree. But is it really necessary to double all the edges? It is possible that all vertices *already* have even degree, or perhaps only some of the edges need to be doubled. In the following discussion, we will consider which vertices, in general, must be doubled in order to ensure that the resulting graph is Eulerian.

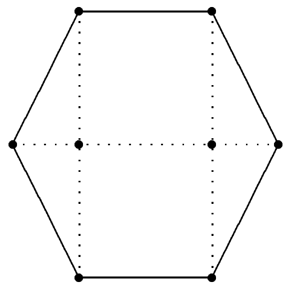
Consider the minimum spanning tree of a complete graph G satisfying "triangle inequality condition":



Let $V' \in V$ be the set of odd-degree vertices in the tree:



Consider the optimal tour restricted to the odd-degree vertices (we can draw this tour because the original graph G is complete):



This tour's cost $t_0 \leq OPT$, since OPT must include vertices that, in the minimum spanning tree, had even degree (vertices in $V - V'$).

Definitions (Matchings):

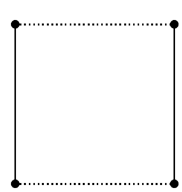
1. Matching - A collection of disjoint edges.
2. Perfect Matching - A matching which does not leave out any of the graph's vertices
3. If a graph has a perfect matching, it must have an even number of vertices—this is a necessary but not sufficient condition.
4. If a graph is complete and the number of vertices is even, then a perfect matching must exist.
5. Weighted Matching - A matching in a weighted graph; the cost of the matching is the total cost of the weights of all the egdes included in it.
6. $MWPM$ (Minimum Weight Perfect Matching) - if one or more perfect matchings exist in a graph, this is one with minimum weight.

In general, a graph must have an even number of odd-degree vertices. Each edge in a graph must be connected to two vertices. If a particular edge was removed from a graph, the degree of each of the vertices formerly connected to that edge would be decreased by 1, which means that the sum of the degrees of those two vertices would be decreased by 2. Since every edge is connected to exactly two vertices, the sum of the degrees of all vertices in a graph must be equal to 2·(the number of edges), and:

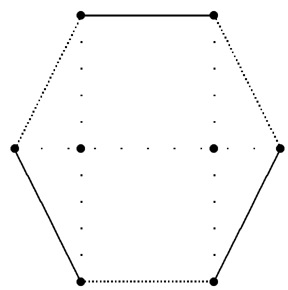
$$(\sum(\text{degrees of odd-degree vertices})) + (\sum(\text{degrees of even-degree vertices})) = 2\cdot(\text{the number of edges})$$

The RHS of this equality must be even, and the ($\sum$(degrees of even-degree vertices)) must be even. Therefore the ($\sum$(degrees of odd-degree vertices)) must be even, which means there must be an even number of odd-degree vertices.

In any cycle with an even number of vertices, at least 2 perfect matchings exist. For example, the following cycle with 4 vertices has a perfect matching represented by solid lines and a perfect matching represented by (the darker) dashed lines:



If we again consider the hexagonal graph for which we marked odd-degree vertices above, we see there are in fact an even number of odd-degree vertices (6). So 2 perfect matchings must exist in the cycle restricted to odd-degree vertices. They are shown here as solid lines and (darker) dashed lines:



For TSP, we are dealing with weighted graphs. For the hexagonal graph above, if it is a weighted graph, one of the two odd-degree vertex matchings must have minimum weight—it is the $MWPM$ (Minimum Weight Perfect Matching).

We stated above that the cost of the tour t_o through all odd-degree vertices must be $\leq OPT$, where OPT is the minimum (Traveling Salesman) tour through all vertices in the graph. Clearly the cost of the tour t_o is equal to the sum of the cost of the (darker) dashed matching and solid line matching pictured above:

$$\text{(cost of (darker) dashed matching)} + \text{(cost of solid line matching)} = t_o$$

So the cost of the $MWPM$ (whether it is the (darker) dashed or solid line matching) must be $\leq \frac{1}{2}t_o$, since the $MWPM \leq$ (cost of the other matching).

With all this in mind we revise our TSP approximation algorithm in order to get a better bound:

1. Find MST.
2. Let $V' \subseteq V$ be the set of odd-degree vertices in the MST.
3. Find $MWPM$ on V'.
4. Impose $MWPM$ on original graph, in the same way we doubled edges above to assure the resulting graph would be Eulerian.
5. Find Euler Path... construct Hamiltonian Path by "shortcutting" wherever possible (as before).

Previously we had a bound of approx-$TSP \leq 2 \cdot MST \leq 2 \cdot OPT$. The "2" comes from the fact that all edges were doubled to get an Eulerian Graph. But now instead of doubling the edges we add only the edges in the $MWPM$, the total cost of which is $\leq \frac{1}{2}t_o \leq \frac{1}{2} \cdot OPT$. Our new and better bound is therefore:

$$\text{approx-}TSP \leq MST + \tfrac{1}{2} \cdot OPT \leq \tfrac{3}{2} \cdot OPT$$

In practice the algorithm may do a bit better than $\frac{3}{2} \cdot OPT$, because when the Hamiltonian Path is constructed from the Euler Path we will likely "shortcut" several vertices, making the cost of the Hamiltonian Path significantly less than that of the Euler Path we are using as a bound.

2 Multiprocessor Scheduling

Suppose we have a several processors, and we need to schedule a set of non-preemptive jobs to be done by those processors.

Processors	Time→
1	j_1 | j_2 |
2	j_3 | j_4 |
$\vdots$	$\vdots$
m	j_{n-1} | j_n |

A possible schedule for jobs $j_1, j_2 \ldots j_n$ with processing times $p_1, p_2 \ldots p_n$.

Several different objective functions may be used to measure the quality of the output. Often we attempt to minimize the time required to complete all jobs. We call this objective function C_{max}:

$$\text{minimize } C_{max} = max(\text{completion time for any processor})$$

If we have only one processor, there is only one possible solution. But with even two processors the problem becomes NP-complete [GJ79].

2.1 List Scheduling Algorithm

We will show that a relatively simple greedy approach gives us a 2-approximation for m processors.

List Scheduling Algorithm:

> Go through the list of jobs one at a time. Assign job j_i to whichever processor m_j has the least amount of job-processing time already assigned to it, breaking ties arbitrarily.

A good lower bound for C_{max} would be the sum of the processing times divided by the number of jobs (this is the result we would get if we could split up jobs and assign some fraction of a job to a processor—OPT can never be better):

$$C_{max} \leq \frac{\sum_{i=1}^{n} p_i}{m} \leq OPT$$

Let W be equal to the total time required for all jobs ($\sum_{i=1}^{n} p_i$):

$$C_{max} \leq \frac{W}{m} \leq OPT$$

Let C_{max}^{LS} be the maximum time required by any processor in the solution resulting from our List Scheduling algorithm. Let j_n be assigned to processor m_k. The time consumed by all other processors (m_j, where $j \neq k$) $\geq C_{max}^{LS} - p_n$ (p_n is the time required for job j_n). And the total time (sum of all the processors' completeion time) is therefore:

$$W \text{ (the total time) } = \sum_{i=1}^{n} p_i \geq m(C_{max}^{LS} - p_n) + p_n$$

$$W - p_n \geq m(C_{max}^{LS} - p_n)$$

$$\frac{W - p_n}{m} \geq C_{max}^{LS} - p_n$$

$$\frac{W}{m} - \frac{p_n}{m} + p_n \geq C_{max}^{LS}$$

$$\frac{W}{m} + p_n(1 - \frac{1}{m}) \geq C_{max}^{LS}$$

As stated above, $\frac{W}{m} \leq OPT$. And OPT must be $\geq p_n$ (OPT is the best possible finish time; it can't be less than the time required for a particular job p_n), we can substitute without changing the inequality:

$$C_{max}^{LS} \leq OPT + OPT(1 - \frac{1}{m})$$

$$C_{max}^{LS} \leq OPT(2 - \frac{1}{m})$$

$$C_{max}^{LS} \leq 2 \cdot OPT$$

So the List Scheduling Algorithm gives us a 2-approximation for m processors. The following exmample gives us $C_{max}^{LS} \leq OPT(2 - \frac{1}{m})$, showing we can not do any better than that bound.

2.2 Worst-Case Example for List Scheduling Algorithm

Suppose we must schedule $m(m-1)$ jobs of length $p_n = 1$ and a single job, which is placed last in line, of length m (as before, m is the number of processors). The List Scheduling algorithm will go through all the shorter jobs first, eventually assigning $m-1$ jobs to all m processors. The last large job will then be assigned to the first processor, so that it will require $m + (m-1)$ $(= 2m-1)$ time to complete its jobs.

The optimal solution would be to assign m short jobs to the first $m-1$ processors and then give the last long job a processor of its own, making $OPT = m$. The List Scheduling algorithm gives us:

$$C_{max}^{LS} = m + (m-1) = m + m(1 - \tfrac{1}{m})$$

$$C_{max}^{LS} = m(2 - \tfrac{1}{m})$$

Substituting OPT for only the first m gives us the expression derived previously:

$$C_{max}^{LS} = OPT(2 - \tfrac{1}{m})$$

2.3 LPT (Longest Processing Time) Algorithm

We can improve the List Scheduling algorithm to a $\frac{4}{3}$-approximation by simply sorting jobs in decreasing order and then applying the List Scheduling algorithm:

LPT (Longest Processing Time) Algorithm:

1. Sort jobs $j_1 \ldots j_n$ in decreasing order, so that $p_1 \geq p_2 \geq p_3 \ldots \geq p_{n-1} \geq p_n$.
2. Apply List Scheduling algorithm defined above.

If we apply LPT to the worst-case example from the previous section, we will get the optimal solution. LPT will sort the jobs first, putting the long job at the beginning, so that it is scheduled first. It will be given its own processor, and the other $m(m-1)$ jobs will be distributed evenly among the other $m-1$ processors, with m jobs going to each, so that $C_{max}^{LPT} = m$.

2.3.1 Background for LPT Analysis

For the analysis of LPT, we need two lemmas:

Lemma 1:

If all processors have at most one job in the optimal solution, LPT gives us that optimal solution.

If all processors have at most one job, n (the number of jobs) must be $\leq m$ (the number of processors). LPT will sort jobs and then assign a single job to each processor, and C_{max}^{LPT} will be equal to the length of the first (and longest) job. Clearly OPT can not be less than the length of the longest job.

Lemma 2:

If all processors have at most two jobs in the optimal solution, LPT gives us that optimal solution.

To prove Lemma 2, we first consider an LPT scheduling solution in which no processor is assigned more than two jobs (to simplify the argument, we assume that all jobs have a unique processing time and that no ties will need to be broken; this is a safe assumption because in a practical problem jobs would have unique names and ties could be broken by lexical ordering of the job names):

Processors	Time$\rightarrow$
1	$j_a \qquad \vert j_y \vert$
2	$j_b \qquad \vert j_x \quad \vert$
$\vdots$	$\vdots$
m	$j_h \quad \vert j_i \quad \vert$

Clearly job j_a has the greatest processing time (p_a) of all jobs, since LPT has assigned it first. $p_b \geq p_h$ since LPT has assigned j_b before j_h, and $p_h \geq p_i$, since p_h was assigned first. If p_y were $\geq p_x$, LPT would assign j_y before j_x. But we know that $p_b \leq p_a$; therefore LPT must have assigned p_x before p_y (because LPT follows the List Scheduling algorithm in making assignments—jobs are assigned to the processor with the least amount of time already scheduled). We can conclude that $p_a \geq p_b \geq p_h \geq p_i \geq p_x \geq p_y$.

Now let us consider the optimal solution, which, for the sake of argument, we say $\neq LPT$:

Processors	Time$\rightarrow$
1	$j_1 \qquad \vert j_5 \quad \vert$
2	$j_4 \qquad \vert$
$\vdots$	$\vdots$
m	$j_6 \quad \vert j_7 \quad \vert$

We can switch the assignments for any two processors without changing C_{max}^{OPT} (in the example below, assignments for processor m_1 and m_2 are switched):

Processors	Time$\rightarrow$
1	$j_4 \qquad \vert$
2	$j_1 \qquad \vert j_5 \quad \vert$
$\vdots$	$\vdots$
m	$j_6 \quad \vert j_7 \quad \vert$

We can also switch the order of two jobs assigned to a particular processor wihout affecting C_{max}^{OPT} (in the example below, jobs j_6 and j_7 are switched):

Processors	Time$\rightarrow$
1	$j_4 \qquad \vert$
2	$j_1 \qquad \vert j_5 \quad \vert$
$\vdots$	$\vdots$
m	$j_7 \quad \vert j_6 \quad \vert$

Now we consider two processor assignments from OPT with two jobs each:

Processors	Time$\rightarrow$
$\vdots$	$\vdots$
m_i	$j_q \qquad \vert j_j \quad \vert$
m_{i+1}	$j_r \quad \vert j_k \quad \vert$
$\vdots$	$\vdots$

From the discussion above, we know we can rearrange processors and jobs within a processor without affecting C_{max}^{OPT}. Assuming the necessary rearranging has already been done, we know that $p_q \geq p_j$, $p_q \geq p_r$ and $p_r \geq p_k$. But what about p_j and p_k? If this were an LPT solution, we could be sure $p_k \geq p_j$, because the fact that $p_q \geq p_r$ tells us that p_k would have been assigned first, and LPT always schedules longer jobs first.

Suppose that in OPT, however, $p_j \geq p_k$. C_{max} for these two processors is either $p_q + p_j$ (completion time for the first processor) or $p_r + p_k$ (completion time for the second). Since $p_q \geq p_r$, our best C_{max} for these two processors would be achieved by switching j_j and j_k (if p_j is in fact $\geq p_k$)—because the longest first-column job ought to be matched with the shortest second-column job.

When we assumed that $p_j \geq p_k$, we found that OPT could only be improved by switching j_j and j_k. This means we can switch second-column jobs so that the shortest are paired with the longest first-column jobs—without affecting C_{max}^{OPT}. We now have all the tools we need to transform any optimal solution into LPT without affecting C_{max}^{OPT}. For example, the following procedure could be used:

1. Arrange the jobs assigned to each processor in OPT so that the longer job is first.
2. Interchange OPT's second-column jobs so that all jobs in LPT's first column are, in OPT, assigned to a unique processor (this is possible because the longest jobs are in LPT's first column; so if two jobs from LPT's first column are assigned to the same processor m_a in OPT, there will always be a shorter job in some other processor's second column to exchange with the job in the second column of m_a).
3. Repeat step 1., then rearrange processors in OPT so that its first column matches the first column of LPT.
4. Interchange OPT's second column jobs so that the shortest are paired with the longest first-column jobs (if there are processors assigned only one job, second-column jobs should be interchanged so that the longest jobs are not paired with any second-column job; the longest of the remaining first-column jobs would then be paired with the shortest second-column jobs).

So, for the case in which all processors are assigned two or fewer jobs, we can get LPT from OPT without making C_{max}^{OPT} any worse, which means that in this case $LPT = OPT$.

2.3.2 LPT Analysis

To get the approximation bound for LPT, we consider two cases:

1. p_n (the last job after sorting and therefore the shortest) $\leq \frac{OPT}{3}$

   In this case we recall from our discussion of the List Scheduling Algorithm:

   $$C_{max}^{LS} \leq \frac{W}{m} + (1 - \frac{1}{m})p_n$$

   Because $\frac{OPT}{3} \geq p_n$ we can substitute without changing the inequality:

   $$C_{max}^{LS} \leq \frac{W}{m} + (1 - \frac{1}{m})\frac{OPT}{3}$$
   $$C_{max}^{LS} \leq OPT(\frac{4}{3} - \frac{1}{3m})$$
   $$C_{max}^{LS} \leq \frac{4}{3} \cdot OPT$$

2. $p_n > \frac{OPT}{3}$

   If p_n (the smallest job) has length $> \frac{OPT}{3}$, there can be at most 2 jobs assigned to any processor (if 3 jobs, each with length $> \frac{OPT}{3}$, were assigned to the same processor, its time to completion would exceed OPT, which is not possible).

   By Lemma 2 above: if at most 2 jobs are assigned to any processor, LPT gives us the optimal solution.

References

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, 1979.