2.3 Scheduling jobs on identical parallel machines

- There are \( n \) jobs to be processed, and there are \( m \) identical machines (running in parallel) to which each job may be assigned.
- Each job \( j = 1, \ldots, n \), must be processed on one of these machines for \( p_j \) time units without interruption, and each job is available for processing at time \( 0 \).
- Each machine can process at most one job at a time.

- The aim is to complete all jobs as soon as possible; that is, if job \( j \) completes at a time \( C_j \) (presuming that the schedule starts at time \( 0 \)), then we wish to minimize \( C_{\text{max}} = \max_{j=1,\ldots,n} C_j \), which is often called the makespan or length of the schedule.
- An equivalent view of the same problem is as a load-balancing problem: there are \( n \) items, each of a given weight \( p_j \), and they are to be distributed among \( m \) machines.
- The aim is to assign each item to one machine so to minimize the maximum total weight assigned to one machine.
• Local search algorithms make a set of local changes or moves that change one feasible solution to another

• Start with any schedule; consider the job \( \ell \) that finishes last; check whether or not there exists a machine to which it can be reassigned that would cause this job to finish earlier

• If so, transfer job \( \ell \) to this other machine

• We can determine whether to transfer job \( \ell \) by checking if there exists a machine that finishes its currently assigned jobs earlier than \( C_\ell - p_\ell \)

• Repeat this procedure until the last job to complete cannot be transferred
Some natural lower bounds on the length of an optimal schedule, $C_{\text{max}}$

Since each job must be processed, it follows that

$$C_{\text{max}} \geq \max_{j=1,...,n} p_j$$

On the other hand, there is, in total, $P = \sum_{j=1}^{n} p_j$ units of processing to accomplish, and only $m$ machines to do this work.

On average, a machine will be assigned $P/m$ units of work, and there must exist one machine that is assigned at least that much work

$$C_{\text{max}} \geq \sum_{j=1}^{n} p_j / m$$

Consider the solution produced by the local search algorithm

Let $\ell$ be a job that completes last in this final schedule; the completion time of job $\ell$, $C_\ell$, is equal to this solution’s objective function value.

By the fact that the algorithm terminated with this schedule, every other machine must be busy from time 0 until the start of job $\ell$ at time

$$S_\ell = C_\ell - p_\ell$$

We can partition the schedule into two disjoint time intervals, from time 0 until $S_\ell$, and the time during which job $\ell$ is being processed.
By the first lower bound, the latter interval has length at most $C_{\max}^*$.

Now consider the former time interval; we know that each machine is busy processing jobs throughout this period.

The total amount of work being processed in this interval is $mS_\ell$, which is clearly no more than the total work to be done, $\sum_{j=1}^{n} p_j$.

Hence,

$$S_\ell \leq \sum_{j=1}^{n} p_j / m$$

By combining this with the lower bound, we see that $S_\ell \leq C_{\max}^*$.

The length of the schedule before the start of job $\ell$ is at most $C_{\max}^*$, as is the length of the schedule afterward; in total, the makespan of the schedule computed is at most $2C_{\max}^*$.

The value of $C_{\max}^*$ for the sequence of schedules produced, iteration by iteration, never increases.

It can remain the same, but then the number of machines that achieve the max value decreases.

Assume that when we transfer a job to another machine, then we reassign that job to the one that is currently finishing earliest.
Let $C_{\text{min}}$ be the completion time of a machine that completes all its processing the earliest.

- $C_{\text{min}}$ never decreases and if it remains the same, then the number of machines that achieve this minimum value decreases.

- This implies that we never transfer a job twice.
  - Suppose this claim is not true, and consider the first time that a job $j$ is transferred twice, say, from machine $i$ to $i'$, and later then to $i^*$.
  - When job $j$ is reassigned from machine $i$ to $i'$, it then starts at $C_{\text{min}}$ for the current schedule.
  - Similarly, when job $j$ is reassigned from machine $i'$ to $i^*$, it then starts at the current $C'_{\text{min}}$.

- Furthermore, no change occurred to the schedule on machine $i'$ in between these two moves for $j$.
- Hence, $C'_{\text{min}}$ must be strictly smaller than $C_{\text{min}}$ (in order for the transfer to be an improving move), but this contradicts our claim that the $C_{\text{min}}$ value is non-decreasing over the iterations of the local search algorithm.

- Hence, no job is transferred twice, and after at most $n$ iterations, the algorithm must terminate.

**Theorem 2.5:** The local search procedure for scheduling jobs on identical parallel machines is a 2-approximation algorithm.
• It is not hard to see that the analysis of the approximation ratio can be refined slightly.

• In deriving the previous inequality, we included job $\kappa$ among the work to be done prior to the start of job $\ell$.

• Hence, we actually derived that

$$S_{\ell} \leq \sum_{j \neq \ell} p_j/m,$$

and hence the total length of the schedule produced is at most

$$p_{\ell} + \sum_{j \neq \ell} p_j/m = \left(1 - \frac{1}{m}\right)p_{\ell} + \sum_{j=1}^{n} p_j/m.$$

• By the two lower bounds, we see that the schedule has length at most $(2 - \frac{1}{m})C_{\text{max}}^*$.

• The difference between this bound and 2 is significant only if there are very few machines.

• Another algorithm assigns the jobs as soon as there is machine availability to process them.

• When a machine becomes idle, then one of the remaining jobs is assigned to start processing.

• This is called the list scheduling algorithm, since one can equivalently view it as first ordering the jobs in a list (arbitrarily), and the next job to be processed is the one at the top of the list.
From the load-balancing perspective, the next job on the list is assigned to the machine that is currently the least heavily loaded.

In this sense the algorithm is a greedy one.

If one uses this as the starting point for search procedure, that algorithm would immediately declare that the solution cannot be improved!

Consider a job $\ell$ that is last to complete.

Each machine is busy until $c_{\ell} - p_{\ell}$, since otherwise we would have assigned job $\ell$ to that other machine.

Hence, no transfers are possible.

Theorem 2.6: The list scheduling algorithm for the problem of minimizing the makespan on $m$ identical parallel machines is a 2-approximation algorithm.

It is natural to use an additional greedy rule that first sorts the jobs in non-increasing order.

The relative error in the length of the schedule produced in Thms 2.5 and 2.6 is entirely due to the length of the last job to finish.

If that job is short, then the error is not too big.

This greedy algorithm is called the longest processing time rule, or LPT.
Theorem 2.7: The LPT rule is a $4/3$-approximation algorithm for scheduling jobs to minimize the makespan on identical parallel machines.

Proof: Suppose that the theorem is false, and consider an input that provides a counterexample to the theorem. Assume that $p_1 \geq \cdots \geq p_n$. We can assume that the last job to complete is indeed the last (and smallest) job in the list. This follows w.l.o.g.: any counterexample for which the last job $\kappa$ to complete is not the smallest can yield a smaller counterexample, simply by omitting all of the jobs $\kappa + 1, \ldots, n$.

The length of the schedule produced is the same, and the optimal value of the reduced input can be no larger. Hence the reduced input is also a counterexample.

So the last job to complete in the schedule is job $n$. If this is a counterexample, what do we know about $p_n (= p_{\ell})$? If $p_{\ell} \leq C_{\text{max}}^*/3$, then the analysis of Theorem 2.6 implies that the schedule length is at most $(4/3) C_{\text{max}}^*$, and so this is not a counterexample. Hence, we know that in this purported counterexample, job $n$ (and therefore all of the jobs) has a processing requirement strictly greater than $C_{\text{max}}^*/3$. 
In the opt. schedule, each machine may process at most two jobs (since otherwise the total processing assigned to that machine is more than $C_{\text{max}}$).

We have reduced our assumed counterexample to the point where it simply cannot exist.

**Lemma 2.8:** For any input to the problem of minimizing the makespan on identical parallel machines for which the processing requirement of each job is more than one-third the optimal makespan, the LPT rule computes an optimal schedule.

As a consequence no counterexample to the theorem can exist, and hence it must be true. ■

---

**2.4 The traveling salesman problem**

- In TSP there is a given set of cities $\{1, 2, \ldots, n\}$, and the input consists of a symmetric $n \times n$ matrix $C = (c_{ij})$ that specifies the cost of traveling from city $i$ to city $j$.

- By convention,
  - we assume that the cost of traveling from any city to itself is equal to 0, and
  - costs are nonnegative;
  - the cost of traveling from city $i$ to city $j$ is equal to the cost of traveling from $j$ to city $i$. 

Input an undirected complete graph with a cost associated with each edge: a feasible solution (tour) is a Hamiltonian cycle in this graph.

I.e., we specify a cyclic permutation of the cities or, equivalently, a traversal of the cities in the order \(k(1), k(2), \ldots, k(n)\), where each city \(j\) is listed as a unique image \(k(i)\).

The cost of the tour is equal to

\[c_{k(n)k(1)} + \sum_{i=1}^{n-1} c_{k(i)k(i+1)}\]

Each tour has \(n\) distinct representations; it does not matter which city is the start of the tour.

There are severe limits on our ability to compute near-optimal tours.

It is NP-complete to decide whether a given undirected \(G = (V, E)\) has a Hamiltonian cycle.

An approximation algorithm for the TSP can be used to solve the Hamiltonian cycle problem:

Given a graph \(G = (V, E)\), form an input to the TSP by setting, for each pair \(i, j\), the cost \(c_{ij} = 1\) if \((i, j) \in E\), and equal to \(n + 2\) otherwise.

If there is a Hamiltonian cycle in \(G\), then there is a tour of cost \(n\), and otherwise each tour costs at least \(2n + 1\).
• If there were to exist a 2-approximation algorithm for the TSP, then we could use it to distinguish graphs with Hamiltonian cycles from those without any.

• Run the approximation algorithm on the new TSP input, and if the tour computed has cost at most \(2n\), then there exists a Hamiltonian cycle in \(G\), and otherwise there does not.

• Setting the cost for the “non-edges” to be \(an + 2\) has a similarly inflated consequence, and we obtain an input to the TSP of polynomial size provided that, for example, \(\alpha = O(2^n)\).

**Theorem 2.9:** For any \(\alpha > 1\), there does not exist an \(\alpha\)-approximation algorithm for the traveling salesman problem on \(n\) cities, provided \(P \neq NP\). In fact, the existence of an \(O(2^n)\)-approximation algorithm for the TSP would similarly imply that \(P = NP\).

• This however is not the end of the story.

• A natural assumption to make about the input to the TSP is to restrict attention to those inputs that are metric; that is, for each triple \(i, j, k \in V\) we have that the triangle inequality holds

\[ c_{ik} \leq c_{ij} + c_{jk} \]
This rules out the construction used in the reduction for the HC problem above; the non-edges can be given cost at most 2 for the triangle inequality to hold, and this is too small to yield a nontrivial nonapproximability result.

A natural greedy heuristic to consider for the TSP; this is often referred to as the nearest addition algorithm.

Find the two closest cities, say, i and j, and start by building a tour on that pair of cities consisting of going from i to j and then back to i again.

This is the first iteration.

In each subsequent iteration, we extend the tour on the current subset S by including one additional city, until we include all cities.

In each iteration, we find a pair of cities i ∈ S and j ∉ S for which the cost c_{ij} is minimum; let k be the city that follows i in the current tour on S.

We add j to S, and insert j into the current tour between i and k.
The crux of the analysis is the relationship of this algorithm to Prim’s algorithm for the minimum spanning tree (MST) in an undirected graph.

A spanning tree of a connected graph \( G = (V, E) \) is a minimal subset of edges \( F \subseteq E \) such that each pair of nodes in \( G \) is connected by a path using edges only in \( F \).

A MST is a spanning tree for which the total edge cost is minimized.

Prim’s algorithm computes a MST by iteratively constructing a set \( S \) along with a tree \( T \), starting with \( S = \{v\} \) for some (arbitrarily chosen) node \( v \in V \) and \( T = (S, F) \) with \( F = \emptyset \).

In each iteration, it determines the edge \((i, j)\) such that \( i \in S \) and \( j \notin S \) is of minimum cost, and adds the edge \((i, j)\) to \( F \).

Clearly, this is the same sequence of vertex pairs identified by the nearest addition algorithm.

Furthermore, there is another important relationship between MST problem and TSP.

**Lemma 2.10:** For any input to the traveling salesman problem, the cost of the optimal tour is at least the cost of the minimum spanning tree on the same input.
**Theorem 2.11:** The nearest addition algorithm for the metric traveling salesman problem is a 2-approximation algorithm.

**Proof:** Let \( S_2, S_3, \ldots, S_n = \{1, \ldots, n\} \) be the subsets identified at the end of each iteration of the nearest addition algorithm (where \( |S_\ell| = \ell \)), and let 

\[ F = \{(s_2, t_2), (s_3, t_3), \ldots, (s_n, t_n)\}, \]

where \((s_\ell, t_\ell)\) is the edge identified in iteration \( \ell - 1 \) (with \( s_\ell \in S_{\ell-1} \), \( \ell = 3, \ldots, n \)). We also know that \( \{1, \ldots, n\}, F \) is a MST for the original input, when viewed as a complete undirected graph with edge costs. Thus, if \( \text{OPT} \) is the optimal value for the TSP input, then

\[ \text{OPT} \leq \sum_{\ell=2}^{n} c_{s_\ell t_\ell}. \]

The cost of the tour on the first two nodes \( i_2 \) and \( j_2 \) is exactly \( 2c_{i_2 j_2} \). Consider an iteration in which a city \( j \) is inserted between cities \( i \) and \( k \) in the current tour. How much does the length of the tour increase? An easy calculation gives \( c_{ij} + c_{jk} - c_{ik} \). By the triangle inequality, we have that \( c_{jk} \leq c_{ji} + c_{ik} \) or, equivalently, \( c_{jk} - c_{ik} \leq c_{ji} \). Hence, the increase in cost in this iteration is at most \( c_{ij} + c_{ji} = 2c_{ij} \). Thus, overall, we know that the final tour has cost at most 

\[ 2 \sum_{\ell=2}^{n} c_{s_\ell t_\ell} \leq 2\text{OPT}, \]

and the theorem is proved. \( \blacksquare \)
A graph is said to be **Eulerian** if there exists a permutation of its edges of the form

\[(i_0, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_0)\]

- We will call this permutation a traversal of the edges, since it allows us to visit every edge exactly once.
- A graph is Eulerian if and only if it is connected and each node has even degree (the number of edges with \(v\) as one of its endpoints).
- Furthermore, if a graph is Eulerian, one can easily construct the required traversal of the edges, given the graph.

To find a good tour for a TSP input, suppose that we first compute a MST (e.g., by Prim’s alg.)
- Suppose that we then replace each edge by two copies of itself.
- The resulting (multi)graph has cost at most \(2 \text{OPT}\) and is Eulerian.
- We can construct a tour of the cities from the Eulerian traversal of the edges,

\[(i_0, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_0)\]

- Consider the sequence of nodes, \(i_0, i_1, \ldots, i_k\), and remove all but the first occurrence of each city in this sequence.
• This yields a tour containing each city exactly once (assuming we then return to $i_0$ at the end)
• To bound the length of this tour, consider two consecutive cities in this tour, $i_\ell$ and $i_m$
• Cities $i_{\ell+1}, \ldots, i_{m-1}$ have already been visited “earlier” in the tour
• By the triangle inequality, $c_{i_\ell, i_m}$ can be upper bounded by the total cost of the edges traversed in the Eulerian traversal between $i_\ell$ and $i_m$, i.e., the total cost of the edges $(i_\ell, i_{\ell+1}), \ldots, (i_{m-1}, i_m)$
• In total, the cost of the tour is at most the total cost of all of the edges in the Eulerian graph, which is at most 2OPT

**Theorem 2.12:** The double-tree algorithm for the metric traveling salesman problem is a 2-approximation algorithm.

• The message of the analysis of the double-tree algorithm is also quite useful
• If we can efficiently construct an Eulerian subgraph of the complete input graph, for which the total edge cost is at most $\alpha$ times the optimal value of the TSP input, then we have derived an $\alpha$-approximation algorithm as well
• This strategy can be carried out to yield a 3/2-approximation algorithm
Consider the output from the MST computation:

- This graph is not Eulerian, since any tree must have nodes of degree one, but it is possible that not many nodes have odd degree.
- $O$ is the set of odd-degree nodes in the MST.
- The sum of node degrees must be even, since each edge in the graph contributes 2 to this total.
- The total degree of the even-degree nodes must also be even, but then the total degree of the odd-degree nodes must also be even.
- I.e., we must have an even number of odd-degree nodes; $|O| = 2^k$ for some positive $k$.

Suppose that we pair up the nodes in $O$:

$$(i_1, i_2), (i_3, i_4), \ldots, (i_{2k-1}, i_{2k})$$

Such a collection of edges that contain each node in $O$ exactly once is called a perfect matching of $O$.

One of the classic results of combinatorial optimization is that given a complete graph (on an even number of nodes) with edge costs, it is possible to compute the perfect matching of minimum total cost in polynomial time.

Given the MST, we identify the set $O$ of odd-degree nodes with even cardinality, and then compute a minimum-cost perfect matching on $O$. 
If we add this set of edges to our MST, we have constructed an Eulerian graph on our original set of cities: it is connected (since the spanning tree is connected) and has even degree (since we added a new edge incident to each node of odd degree in the spanning tree).

As in the double-tree algorithm, we can shortcut this graph to produce a tour of no greater cost. This is known as Christofides' algorithm.

**Theorem 2.13:** Christofides' algorithm for the metric traveling salesman problem is a 3/2-approximation algorithm.

**Proof:** We want to show that the edges in the Eulerian graph produced by the algorithm have total cost at most $\frac{3}{2}\text{OPT}$. We know that the MST edges have total cost at most OPT. So we need only show that the perfect matching on $O$ has cost at most $\text{OPT}/2$.

First observe that there is a tour on just the nodes in $O$ of total cost at most $\text{OPT}$. This again uses the shortcutting argument. Start with the optimal tour on the entire set of cities, and if for two cities $i$ and $j$, the optimal tour between $i$ and $j$ contains only cities that are not in $O$, then include edge $(i,j)$ in the tour on $O$. 
Each edge in the tour corresponds to disjoint paths in the original tour, and hence by the triangle inequality, the total length of the tour on $O$ is no more than the length of the original tour. Now consider this “shortcut” tour on the node set $O$. Color these edges red and blue, alternating colors. This partitions the edges into two sets; each of these is a perfect matching on the node set $O$. In total, these two edge sets have cost $\leq \text{OPT}$. Thus, the cheaper of these two sets has cost $\leq \text{OPT}/2$. Hence, there is a perfect matching on $O$ of cost $\leq \text{OPT}/2$. Therefore, the algorithm must find a matching of cost at most $\text{OPT}/2$.

No better approximation algorithm for the metric TSP is known.

Substantially better algorithms might yet be found, since the strongest negative result is

**Theorem 2.14:** Unless $P = \text{NP}$, for any constant $\alpha < \frac{220}{219} \approx 1.0045$, no $\alpha$-approximation algorithm for the metric TSP exists.

It is possible to obtain a PTAS in the case that cities correspond to points in the Euclidean plane and the cost of traveling between two cities is equal to the Euclidean distance between the two points.
3 Rounding data and dynamic programming

- Dynamic programming is a standard technique in algorithm design in which an optimal solution for a problem is built up from optimal solutions for a number of subproblems, normally stored in a table or multidimensional array.
- Approximation algorithms can be designed using dynamic programming in a variety of ways, many of which involve rounding the input data in some way.

3.1 The knapsack problem

- We are given a set of $n$ items $I = \{1, \ldots , n\}$, where each item $i$ has a value $v_i$ and a size $s_i$.
- All sizes and values are positive integers.
- The knapsack has a positive integer capacity $B$.
- The goal is to find a subset of items $S \subseteq I$ that maximizes the value $\sum_{i \in S} v_i$ of items in the knapsack subject to the constraint that the total size of these items is no more than the capacity; that is, $\sum_{i \in S} s_i \leq B$. 
We consider only items that could actually fit in the knapsack, so that \( s_i \leq B \) for each \( i \in I \).

We can use dynamic programming to find the optimal solution to the knapsack problem.

We maintain an array entry \( A(j) \) for \( j = 1, \ldots, n \).

Each entry \( A(j) \) is a list of pairs \( (t, w) \).

A pair \( (t, w) \) in the list of entry \( A(j) \) indicates that there is a set \( S \) from the first \( j \) items that uses space exactly \( t \leq B \) and has value exactly \( w \); i.e., there exists a set \( S \subseteq \{1, \ldots, j\} \) s.t.

\[
\sum_{i \in S} s_i = t \leq B \quad \text{and} \quad \sum_{i \in S} v_i = w
\]

Each list keeps track of only the most efficient pairs.

To do this, we need the notion of one pair dominating another one:

\(- (t, w) \) dominates another pair \( (t', w') \) if \( t \leq t' \) and \( w \geq w' \); that is, the solution indicated by the pair \( (t, w) \) uses no more space than \( (t', w') \), but has at least as much value.

Domination is a transitive property: if \( (t, w) \) dominates \( (t', w') \) which dominates \( (t'', w'') \), then \( (t, w) \) also dominates \( (t'', w'') \).

We will ensure that in any list, no pair dominates another one.