St. Francis Xavier University
Department of Computer Science

CSCI 550: Approximation Algorithms
Lecture 3: Greedy Algorithms and Local Search
Fall 2021

1 Greed is (Sometimes) Good

In our introductory lecture, we discussed two algorithms for both the unweighted and weighted set cover problems that made use of the *greedy approach*. Recall that the greedy approach generally makes decisions according to the following rule: at any given step of the computation, the decision that is best *at that moment* is the one that is selected by the algorithm. In following this process, the algorithm constantly makes the best *local* choice, but this may not always result in the algorithm producing the best *global* choice at the end of its computation.

Because of the way they operate, greedy algorithms are usually known as *primal infeasible* algorithms; that is, the algorithm produces its solution over the course of its computation, instead of constantly having a feasible solution on hand at every point during the computation. However, despite this, greedy algorithms are quite popular due to how easy they are to design and implement. Even though they may not always produce the best solution, greedy algorithms do produce *a* solution, and they often do so in a modest running time.

In this section, we will consider a number of greedy algorithms for different problems, including one problem we alluded to at the very beginning of our introductory lecture: the traveling salesperson problem.

1.1 Sequential Scheduling on a Single Machine

As every student, professor, and registrar knows, creating and adhering to a schedule is quite a tough problem. A student might need to schedule their assignment due dates, a professor might need to schedule paper submissions and grant deadlines, and the registrar needs to schedule hundreds of university courses into a finite set of time blocks and lecture halls. In general, we can think of a *scheduling problem* as one where there exists a set of tasks or *jobs* to be completed and a set of *machines* capable of completing the jobs. Our goal is to optimize some aspect of the schedule: this optimization could come in the form of finishing all jobs as early as possible, or minimizing the amount of time a resource spends not working on a job, or minimizing the average time required to complete a job, or any number of other goals.

To formalize the notion of a scheduling problem, suppose we have n jobs that we need to schedule on a single machine M. Once M begins processing a job, it must continue processing that job until it is completed. For each job j, there exists an associated *processing time* p_j. Moreover, we can specify that M cannot begin working on a job j before some *release time* r_j. We assume that a schedule starts at time 0 and that each release date is nonnegative. Each job j also has an associated *due date* d_j; note that due dates are permitted to be negative. If M completes job j at time C_j, then the *lateness* of j is taken to be $L_j = C_j - d_j$.

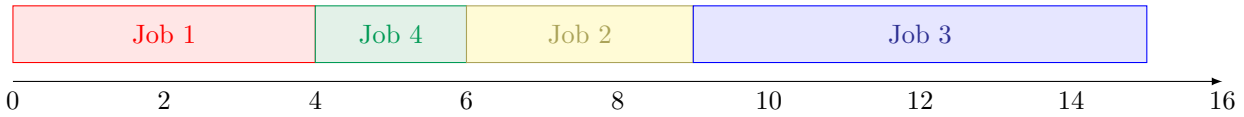
The scheduling problem on a single machine is, therefore, the problem of finding a schedule for a set of jobs that minimizes the maximum lateness of any job:

SCHEDULING-SINGLE-MACHINE
Given: a set of n jobs S and, for each job j, a release time r_j, a processing time p_j, and a due date d_j
Determine: a schedule that minimizes the maximum lateness $L_{\max} = \max_{j=\{1,\ldots,n\}} L_j$ of any job j

Example 1. Suppose we have an instance of the single-machine scheduling problem where we have four jobs to complete:

- Job 1 has release time $r_1 = 0$, processing time $p_1 = 4$, and due date $d_1 = -1$;
- Job 2 has release time $r_2 = 3$, processing time $p_2 = 3$, and due date $d_2 = 7$;
- Job 3 has release time $r_3 = 5$, processing time $p_3 = 6$, and due date $d_3 = 12$; and
- Job 4 has release time $r_4 = 4$, processing time $p_4 = 2$, and due date $d_4 = 10$.

An example of a schedule for these jobs is illustrated as follows:



For each job, we have the associated lateness values $L_1 = 4 - (-1) = 5$, $L_2 = 9 - 7 = 2$, $L_3 = 15 - 12 = 3$, and $L_4 = 6 - 10 = -4$. Therefore, for this schedule, $L_{\max} = L_1 = 5$.

The scheduling problem is NP-hard, so it seems to be a good candidate for an approximation algorithm. However, we encounter one difficulty that may prevent us from finding near-optimal solutions for a given instance of the problem: if the optimal solution to the instance is zero lateness, then an α-approximation algorithm would need to find a solution with value at most $\alpha \cdot 0 = 0$. However, since the problem is NP-hard, finding such a solution in polynomial time would imply that P = NP! Therefore, we will make one additional assumption that all due dates are negative values. In doing so, we ensure that the optimal value will always be positive, and our approximation algorithm will continue to work as expected.

Before we consider our approximation algorithm, let us establish a lower bound on the optimal value of any instance of the scheduling problem. Let S denote our set of jobs, and define analogues of our three job measures for the set of jobs: take $r(S) = \min_{j \in S} r_j$, $p(S) = \sum_{j \in S} p_j$, and $d(S) = \max_{j \in S} d_j$. Furthermore, denote the optimal value by $L^*_{\max}$. Then the optimal value will always be lower-bounded by the sum of the jobs' minimum release time and total processing times, less the maximum due date of any job.

Lemma 2. *For any set of jobs S,*

$$L^*_{\max} \geq r(S) + p(S) - d(S).$$

Proof. Consider the optimal schedule for the set of jobs S, and suppose that job j is the last job in S to be processed. Since no job in S can be processed before time $r(S)$ and all jobs require a total of $p(S)$ time to complete, we know that job j cannot be completed any earlier than time $r(S) + p(S)$. Since the due date of job j is $d(S)$ or earlier, the lateness of job j will be at least $r(S) + p(S) - d(S)$, and the lower bound on the optimal value of the solution follows. □

The rule we will use for our approximation algorithm when creating schedules should come naturally to any student: we will prioritize the job with the earliest due date. We say that a job j is *available* at a time t if $r_j \leq t$. Whenever the machine is idle (that is, not currently processing a job), our approximation algorithm will direct the machine to start processing whatever available job has the earliest due date.

Algorithm 1: Single-machine scheduling—earliest due date

```
for each t ≥ 0 do
    A ← subset of all available jobs in S at time t
    if the machine is idle at time t then
        j ← job in A with earliest due date d_j
        run job j on the machine
```

It turns out that this fairly simple heuristic for scheduling produces a decent performance ratio for our approximation algorithm.

Theorem 3. *Algorithm 1 gives a 2-approximation algorithm for the single-machine scheduling problem.*

Proof. Consider a schedule produced by the earliest due date rule, and take some job j with maximum lateness $L_{\max} = C_j - d_j$ in this schedule. Find the earliest time $t \leq C_j$ such that the machine was processing without any idle time during the interval $[t, C_j)$.

Let S be the set of jobs processed by the machine during the interval $[t, C_j)$. By our choice of time t, we know that immediately before time t, none of the jobs in S were available. Therefore, $r(S) = t$. Furthermore, since only the jobs in S were processed during this interval, we have that $p(S) = C_j - t = C_j - r(S)$. Therefore, $C_j \leq r(S) + p(S)$.

Since $d(S) < 0$ by our assumption that all due dates are negative values, we can apply Lemma 2 to get

$$\begin{aligned} L^*_{\max} &\geq r(S) + p(S) - d(S) \\ &\geq r(S) + p(S) \\ &\geq C_j. \end{aligned}$$

At the same time, if we take $S = \{j\}$, then Lemma 2 gives us

$$\begin{aligned} L^*_{\max} &\geq r_j + p_j - d_j \\ &\geq -d_j. \end{aligned}$$

Adding together both of these inequalities, we get that the maximum lateness of the schedule produced by the earliest due date rule is

$$L_{\max} = C_j - d_j \leq 2 \cdot L^*_{\max}.$$ □

There exist other heuristics for scheduling that give other outcomes, such as the shortest processing time rule, the longest processing time rule, the minimum slack time rule (where the "slack" of a job j is given by the expression $d_j - p_j - t$), and even the random scheduling rule. However, here we will leave our discussion at the earliest due date rule.

1.2 *k*-Center Problem

If someone is given a large set of data, one of the first things they typically do with it is try to identify clusters or trends within the data. If the data can be plotted on the Cartesian plane, clusters may indicate some kind of relationship between data points in either the x- or y-axis. For example, if we plot a data set showing student proximity to campus versus number of days enrolled at StFX, we may see a cluster of data points in the quadrant of the plot corresponding to "closer to campus" and "fewer days enrolled", as many first-year students opt to live in residences on campus.

Here, we will consider a problem closely related to clustering known as the *k-center problem*. In the real world, an instance of the k-center problem may be something like the following: in a town consisting of n blocks, we want to find the optimal blocks on which to place k fire stations so that firefighters don't have to travel far in case of an emergency.

To formalize our problem, we will consider an undirected complete graph G where we need to find k vertices to act as *cluster centers*. Our goal is to minimize the maximum distance of each vertex in a cluster to its cluster center.

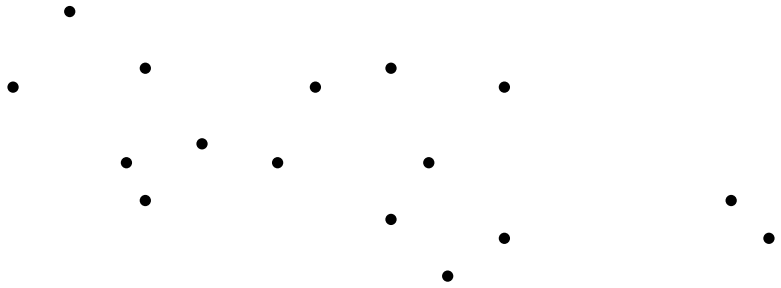
K-Center
Given: an undirected complete graph $G = (V, E)$, a distance $d_{uv} \geq 0$ for each pair of vertices $u, v \in V$, and an integer k
Determine: a subset of cluster centers $S \subseteq V$, $\lvert S \rvert \leq k$, that minimizes the maximum distance of any vertex to its cluster center

Focusing on the distance measure d, we denote the distance of a vertex v from a subset of vertices $S \subseteq V$ by $d(v, S) = \min_{u \in S} d_{vu}$. We can then say that the *radius* of S is $\max_{w \in V} d(w, S)$. Therefore, the goal of

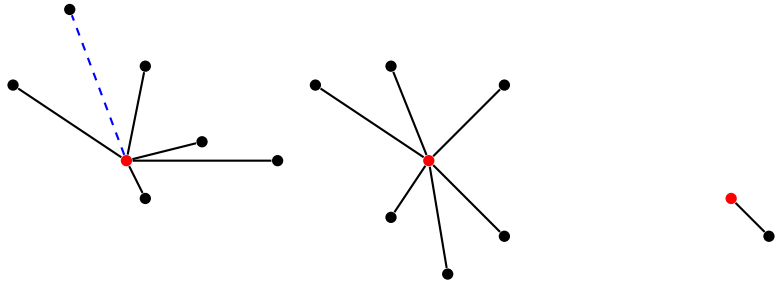
the k-center problem is to find a subset of cluster centers of size at most k that minimizes the radius of any cluster.

Note that we can make a number of assumptions with the k-center problem: we can assume that $d_{vv} = 0$ for all vertices $v \in V$, that $d_{uv} = d_{vu}$ for all vertices $u, v \in V$, and that every distance obeys the *triangle inequality*, which states that for all vertices $u, v, w \in V$, $d_{uv} + d_{vw} \geq d_{uw}$.

Example 4. Suppose we have the following vertices in a given graph (with all edges of the graph omitted for clarity):



If we want to find a solution to the k-center problem for $k = 3$ on this graph instance, we must select three vertices to act as cluster centers. We highlight these vertices in red. We then associate every other vertex to its closest cluster center, which we denote by drawing an edge between that vertex and the cluster center. This gives us the following clustering, with the maximum distance between a vertex and its cluster center denoted by the dashed blue edge:



The k-center problem is another classic example of an NP-hard problem. The greedy approximation algorithm we will develop for the k-center problem works by making the following observation: at any step after we select a vertex as a cluster center, the next cluster center we select should be as far away as possible from the others in our set.

Algorithm 2: k-center—greedy

choose an arbitrary vertex $v \in V$
$S \leftarrow \{v\}$
while $|S| \leq k$ **do**
  $u \leftarrow \arg\max_{u \in V} d(u, S)$ ▷ arg max chooses the vertex furthest from the vertices in S
  $S \leftarrow S \cup \{u\}$

Again, the greedy approach gives us a rather good approximation algorithm for the k-center problem.

Theorem 5. *Algorithm 2 gives a 2-approximation algorithm for the k-center problem.*

Proof. Consider the optimal solution $S^* = \{j_1, \ldots, j_k\}$ to the k-center problem, and let r^* denote the radius of this solution. Each pair of vertices j and j' placed in the same cluster are within a distance of at most $2r^*$ from one another (by the triangle inequality).

Consider the set of cluster centers $S \subseteq V$ selected by the greedy approach. We have two possibilities:

- If each cluster center in S is selected from some cluster of S^*, then every vertex in a cluster is clearly within a distance of at most $2r^*$ of some cluster center in S.
- If the greedy approach selects two vertices j and j', in that order, from the same cluster of S^*, then again the distance between these cluster centers is at most $2r^*$. The greedy approach selects vertex j' because it is currently the furthest from the other cluster centers already in S. Hence, all vertices are within a distance of at most $2r^*$ of some cluster center already added to S.

This remains true as the greedy approach continues to add cluster centers to S. In either case, the solution obtained by the greedy approach is within a factor of two to the optimal solution. □

1.3 Traveling Salesperson Problem

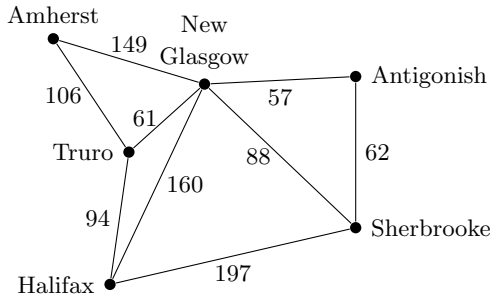
In our introductory lecture, we spoke of a problem that involved us finding the shortest path taking us through all the capital cities of the world. This problem, formally known as the *traveling salesperson problem*, is arguably one of the most famous, important, well-known, and well-studied examples of an NP-hard problem in computer science. The body of work surrounding the traveling salesperson problem is immense, and even today, work on the problem continues as researchers study methods of computing near-optimal solutions for larger and larger instances.

Here, we will revisit the traveling salesperson problem and introduce two approximation algorithms. The first algorithm we will see is greedy, while the second gives the best-known verified performance guarantee for the problem to date.

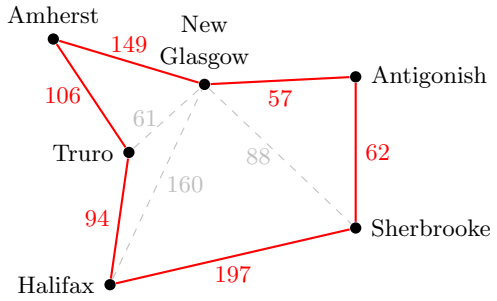
To begin, let's formalize the problem. Suppose we are given a set of n cities $C = \{1, 2, \ldots, n\}$ and, for each pair of cities c_i and c_j, there exists a cost c_{ij} to travel between the two cities. (This is quite similar to the k-center problem, where we had a set of vertices and some distance measure d between vertices.)

Traveling-Salesperson
Given: a set of cities $C = \{1, 2, \ldots, n\}$ and a cost $c_{ij} \geq 0$ for each pair of cities $i, j \in C$
Determine: a Hamiltonian cycle through all cities in C that has minimal total cost

Example 6. Consider the following map of cities around the North Shore of Nova Scotia:



Starting in Antigonish, an example of a tour (i.e., a Hamiltonian cycle) visiting all cities and returning to Antigonish is as follows, with the tour highlighted in red:



Again, as with the k-center problem, we can make a number of assumptions with the traveling salesperson problem: we can assume that $c_{ii} = 0$ for all cities $i \in C$ and that $c_{ij} = c_{ji}$ for all cities $i, j \in C$. We may further assume that all costs are nonnegative.

One of the most important assumptions we can also make is that any instance of the problem is *metric*; that is, for all cities i, j, and k, the distances between these cities abides by the triangle inequality $c_{ik} \leq c_{ij} + c_{ik}$. Indeed, if we don't assume that the instance is metric, then we run into a big, big problem: we could use an approximation algorithm for the non-metric traveling salesperson problem to solve the Hamiltonian cycle problem for graphs. Since it is NP-complete to determine whether a graph contains a Hamiltonian cycle, this would imply that there exists no α-approximation algorithm for the traveling salesperson problem unless $\mathsf{P} = \mathsf{NP}$. Therefore, the approximation algorithm we consider in this section will apply to the metric traveling salesperson problem.

Our greedy approach to the traveling salesperson problem uses an idea similar to the one we used with the k-center problem. In our greedy algorithm for the k-center problem, on each iteration, we chose the vertex furthest from any of the cluster centers we chose previously. On the other hand, our greedy algorithm for the traveling salesperson problem will choose the city *nearest* to the other cities in our tour T so far and add that city to T. This heuristic is sometimes referred to as the "nearest addition algorithm".

Algorithm 3: Traveling salesperson—nearest addition

```
select an initial city i
T ← i
E ← (i, i)
while T ≠ C do
    find cities i ∈ T, j ∉ T such that c_ij is minimal
    T ← j
    remove edge (k, i) from E          ▷ k is the city following i in the current tour
    E ← (k, j), (j, i)
```

Interestingly, the nearest addition algorithm is quite closely related to Prim's algorithm for constructing a minimum spanning tree for a given undirected graph. Prim's algorithm constructs a set S alongside its tree T by arbitrarily selecting one initial vertex, $S = \{v\}$, and iteratively determining which edge (i, j) is of minimum cost, where $i \in S$ and $j \notin S$. Then, the algorithm adds the minimum-cost edge to T.

Having noted this connection between the nearest addition algorithm and Prim's algorithm, we can make the following observation.

Lemma 7. *For any instance of the metric traveling salesperson problem, the cost of the optimal tour is lower-bounded by the cost of the minimum spanning tree on the same graph.*

Proof. For any instance of the metric traveling salesperson problem with $n \geq 2$, consider the optimal tour and delete one edge from the tour. The result is a tree that spans every city and has minimum cost. Additionally, this minimum spanning tree can have a cost no greater than the cost of the optimal tour. □

We can now analyze the performance of our greedy nearest addition algorithm for the traveling salesperson problem.

Theorem 8. *Algorithm 3 gives a 2-approximation algorithm for the metric traveling salesperson problem.*

Proof. Let $T_2, T_3, \ldots, T_n = \{1, 2, \ldots, n\}$ denote the subsets of cities selected at the end of each iteration of the algorithm. Observe that $|T_i| = i$ for all i. Furthermore, let $F = \{(i_2, j_2), (i_3, j_3), \ldots, (i_n, j_n)\}$ denote the edges selected at the end of each iteration of the algorithm; specifically, edge (i_ℓ, j_ℓ) is the edge selected in iteration $\ell - 1$.

We know that $(\{1, 2, \ldots, n\}, F)$ is a minimum spanning tree for the graph corresponding to the problem instance. Therefore, if OPT is the optimal value for the given instance of the metric traveling salesperson problem, then

$$\sum_{\ell=2}^{n} c_{i_\ell j_\ell} \leq \text{OPT}.$$

The cost of the tour on the initial two cities, i_2 and j_2, is $2 \cdot c_{i_2 j_2}$. After an iteration where we insert a new city j between existing cities i and k in the tour, the cost of the tour increases by $c_{ij} + c_{jk} - c_{ik}$. By the triangle inequality, we know that $c_{jk} \leq c_{ji} + c_{ik}$, and since this is equivalent to $c_{jk} - c_{ik} \leq c_{ji}$, we conclude that the cost of the tour increases by at most $c_{ij} + c_{ji} = 2 \cdot c_{ij}$. Hence, the final tour will have a cost at most

$$2 \sum_{\ell=2}^{n} c_{i_\ell j_\ell} \leq 2 \cdot \text{OPT}. \qquad \square$$

Can we do better than a performance guarantee of 2 for the metric traveling salesperson problem? Indeed we can, and we can still use minimum spanning trees to approximate the tour! Another method known as *Christofides' algorithm* works as follows:

1. Given an instance of the metric traveling salesperson problem, construct a minimum spanning tree T.
2. Take O to be the set of vertices in T with odd degree. By the handshake lemma, $|O|$ is even.
3. Find a minimum-weight perfect matching[1] M in the subgraph formed from the vertices in O.
4. Combine the edges of M and T to form a connected graph H in which all vertices have even degree.
5. Find an Eulerian cycle in H, and convert it to a Hamiltonian cycle by skipping repeated vertices.

While we won't get into the details of the analysis here, the following is known about Christofides' algorithm.

Theorem 9. *Christofides' algorithm gives a* $3/2$*-approximation algorithm for the metric traveling salesperson problem.*

In fact, Christofides' algorithm remains the best-known polynomial-time approximation algorithm for the metric traveling salesperson problem, though recent work continues to attempt to lower this performance guarantee to less than $3/2$.

2 Keeping It Local

In contrast to a greedy algorithm, which produces its solution over the course of its computation, a *local search algorithm* starts its computation with a feasible solution and makes a number of small tweaks to see if it can improve on that solution. These small tweaks, also called *local changes*, don't affect the overall solution much, but if an improvement can be gained, the algorithm makes the change.

In the end, when the algorithm sees no other changes that it can make, it completes its computation and produces a *locally optimal solution*. However, the straightforward approach to making local changes often results in an algorithm that doesn't run in polynomial time, so often we place conditions on the local changes in order to ensure that a locally optimal solution is produced in a reasonable amount of time.

In the same way that greedy algorithms are primal infeasible, local search algorithms are *primal feasible* due to the fact that they start with a feasible solution.

Here, we will look at a couple of problems alongside corresponding local search algorithms for those problems.

[1] A *perfect matching* is a subset of edges in a graph where every vertex of the graph is adjacent to exactly one edge in the subset.

2.1 Parallel Scheduling on Identical Machines

Previously, we considered the problem of scheduling jobs on a single machine that can complete jobs sequentially. Now, we will consider another variant of the scheduling problem where we have multiple machines that can complete jobs in parallel.

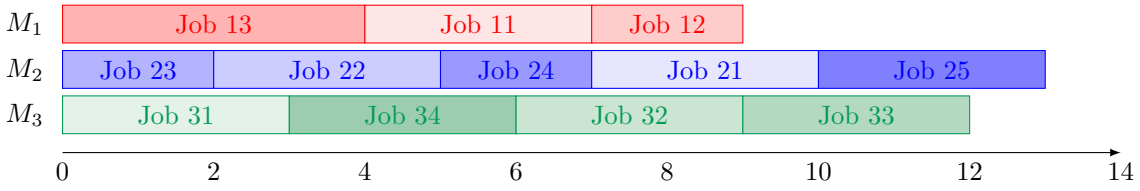
Suppose we have m identical machines and n jobs to complete on these machines. Our machines can run in parallel, so multiple machines can work on their assigned jobs without conflict. However, once a machine is assigned a job j, the machine must work on that job for p_j units of time without interruption. We will further assume that release dates do not apply in this variant of the problem; that is, all jobs are made available instantly at time 0. As before, the completion time of a job j is denoted by C_j.

The scheduling problem on identical machines is the problem of finding a schedule for a set of jobs that minimizes the time spent to complete all jobs.

SCHEDULING-MULTIPLE-IDENTICAL-MACHINES
Given: a set of m identical machines, a set of n jobs S, and, for each job j, a processing time p_j
Determine: a schedule that minimizes the total processing time $C_{\max} = \max_{j=\{1,\ldots,n\}} C_j$

The value $C_{\max}$ is sometimes referred to as the *makespan* of the schedule.

Example 10. An example of a multiple-machine schedule for a set of jobs is illustrated as follows:



Given a feasible schedule, we can create a rather naïve local search algorithm for the scheduling problem on identical machines in the following way:

Algorithm 4: Multiple identical machine scheduling—local search

find the job ℓ that finishes latest in the schedule
if there exists a machine M that is idle earlier than time $C_\ell - p_\ell$ **then**
    assign job ℓ to machine M

We can then repeatedly apply this algorithm to our given schedule until it can make no further changes. Observe that our algorithm has the property that the value $C_{\max}$ will never increase as we iteratively modify the schedule. In fact, if we make the additional assumption that we always transfer a job to the machine that is currently finishing earliest, then we can show that no job will be transferred twice and the algorithm will terminate after at most n iterations. However, we will not prove that here.

Before we get into the analysis of this local search algorithm, let's first establish some lower bounds on the value of the optimal schedule. Since any optimal schedule must complete all jobs, we naturally have that

$$\max_{j=\{1,2,\ldots,n\}} p_j \leq C^*_{\max}. \tag{1}$$

At the same time, there are $P = \sum_{j=1}^{n} p_j$ total units of processing time needed to complete all jobs, and a total of m machines available to process the jobs. Thus, each machine will perform, on average, P/m units of work. Consequently, there must exist one machine that performs at least that much work, and so we have

$$\sum_{j=1}^{n} p_j/m \leq C^*_{\max}. \tag{2}$$

We can now establish the performance guarantee for our local search algorithm.

Theorem 11. *Algorithm 4 gives a 2-approximation algorithm for the multiple identical machine scheduling problem.*

Proof. Consider a solution produced by Algorithm 4, and suppose that job ℓ finishes last in this schedule. Since the algorithm terminated after producing this schedule, it must be the case that all machines were busy from time 0 to the start of job ℓ at time $S_\ell = C_\ell - p_\ell$.

Let us partition the schedule into two disjoint time intervals: the first interval runs from time 0 to time S_ℓ, and the second interval runs from time S_ℓ to time C_ℓ. By Equation 1, we know that the interval from S_ℓ to C_ℓ has length at most $C^*_{\max}$.

Now, consider the interval from time 0 to time S_ℓ. Throughout this interval, all machines must be busy processing some job. The total amount of work performed during this interval is therefore $m \cdot S_\ell$, and this value cannot be any greater than the total work to be performed, $\sum_{j=1}^{n} p_j$. Therefore, we have that

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

and, rearranging, we get

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

Combining the above equation with Equation 2, we get that $S_\ell \leq C^*_{\max}$. However, this means that the length of the schedule before job ℓ was started is at most $C^*_{\max}$, and by our earlier observation, the length of the schedule after job ℓ was started is also at most $C^*_{\max}$. Therefore, the makespan of the schedule produced by the algorithm is at most $2 \cdot C^*_{\max}$. □

We may optionally refine our analysis to show that the schedule produced by this algorithm has a makespan of at most $(2 - \frac{1}{m}) \cdot C^*_{\max}$, which improves our performance guarantee especially if we have few identical machines available to process jobs.

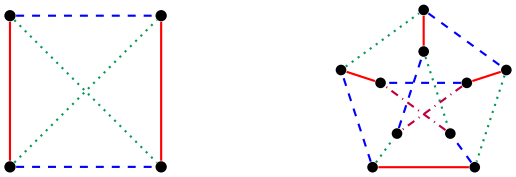
2.2 Edge Colouring

To finish this lecture, we will present an algorithm that combines elements of both the greedy approach and the local search approach. The problem we consider is that of finding a *k-edge-colouring* of a graph.

K-Edge-Colouring
Given: a simple undirected graph $G = (V, E)$ and k distinct colours
Determine: an assignment of colours to edges of G such that no two edges with the same colour share a vertex

If we are able to assign exactly one of k colours to each edge of a graph in such a way that the specifications of the problem are satisfied, then we say that the graph is k-edge-colourable.

Example 12. The graph on the left is the complete graph on four vertices, K_4. The graph on the right is the Petersen graph. While both K_4 and the Petersen graph have maximum degree 3, we see that K_4 is 3-edge-colourable while the Petersen graph requires four edge colours. This is demonstrated by the following edge colourings, where red is solid, blue is dashed, green is dotted, and purple is dash-dotted:



Colouring problems do not belong solely to the domain of graph theory. Indeed, colourings can be used to model scheduling problems, resource allocation problems, and other real-world scenarios. For example, when the university needs to schedule exams, it often aims to do so using as few time slots as possible. Since the university must also minimize the number of conflicting exams for students, it can model this scheduling problem as a graph, where vertices represent courses and an edge exists between two vertices if at least one student is taking both courses simultaneously. If the graph is k-colourable, then the university knows it will need k time slots to schedule all of the exams without conflict.[2]

Naturally, if we want to find a k-edge-colouring, then we will often want to take k to be as small as possible. If Δ is the maximum degree of any vertex in our graph, then we cannot hope to find any k-edge-colouring with $k < \Delta$, since we require at least Δ colours for each of the edges incident to the vertex of maximum degree. This observation gives us a nice lower bound on the value of k for a given graph. On the other hand, since the world is not always nice, we have the following result showing that the problem of deciding whether a Δ-edge-colouring exists in a graph with maximum degree Δ is computationally difficult:

Theorem 13. *For any graph G with $\Delta = 3$, the problem of deciding whether G is 3-edge-colourable is* NP*-complete.*

As a consequence of this result, we cannot hope to construct an efficient algorithm to find a Δ-edge-colouring, unless $\mathsf{P} = \mathsf{NP}$. However, Vizing's theorem tells us that every simple undirected graph can be edge-coloured using either Δ or $\Delta + 1$ colours, and fortunately, we can devise a procedure to find a $(\Delta + 1)$-edge-colouring!

Algorithm 5: Edge colouring—greedy/local search

```
1:  while not all edges of G are coloured do
2:      choose an uncoloured edge (u, v_0)
3:      i ← −1
4:      repeat
5:          i ← i + 1
6:          if there exists a colour c_curr not incident to both u and v_i then
7:              c_i ← c_curr
8:          else
9:              choose a colour c_i not incident to v_i
10:             v_{i+1} ← the edge (u, v_{i+1}) of colour c_i
11:     until c_i is a colour not incident to u or c_i = c_j for some j < i
12:     if colour c_i is not incident to both u and v_i then
13:         shift uncoloured edge to (u, v_i)
14:         colour edge (u, v_i) with colour c_i
15:     else
16:         choose j < i such that c_i = c_j
17:         shift uncoloured edge to (u, v_j)
18:         choose a colour c_u that is not incident to u
19:         c ← c_i
20:         E' ← all edges coloured by either c or c_u
21:         if u and v_j belong to different connected components of (V, E') then
22:             switch colours c_u and c in component containing u
23:             colour edge (u, v_j) with colour c
24:         else
25:             shift uncoloured edge to (u, v_i)
26:             switch colours c_u and c in component containing u
27:             colour edge (u, v_i) with colour c
```

[2] In our exam scheduling example, we consider k-vertex-colourings instead of k-edge-colourings, but the same underlying motivation applies.

The idea behind Algorithm 5 is to start with an uncoloured graph and, on each iteration, take some uncoloured edge of the graph (line 2) and find a colour for it (lines 3–11). We first attempt to assign a colour directly to the current edge (lines 12–14). If this can't be done, then we perform a local change to existing edge colours in the graph until we can successfully colour the current edge (lines 15–27).

As it happens, not only does this algorithm find an edge colouring that uses $\Delta + 1$ colours, but it does so in polynomial time. We won't go through the details of the algorithm's running time here, but the complete proof shows two things: first, on each iteration of the algorithm, a legal edge colouring is maintained (i.e., no two edges incident to the same vertex share a colour, and at most $\Delta + 1$ colours are used); and second, on each iteration of the algorithm, the current edge can be successfully coloured and, therefore, the algorithm terminates in time polynomial to the number of edges of the graph.

Theorem 14. *Algorithm 5 finds a $(\Delta + 1)$-edge-colouring of a graph G in polynomial time.*