PSET 3
CS124: Data Structures and Algorithms
Prof. Mitzenmacher

Dhilan Ramaprasad
dhilanramaprasad@college.harvard.edu

Collaborator: Benny Paris
Others: Esther/Amy

1 Risk-Free Currency Exchange

1.1 Assumptions:

— We seek to yield a profit by returning to a **start currency**. If no such return path exists, our relative profit is harder to quantify and could be better expressed by starting from a different currency if there is profit to be made.

— By the last item, I assert that in our problem we seek a scenario where if we start at currency $i = 1$, we return to it, so our exchange rates will go $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot ... \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1}$.

1.2 Goals:

1. Seek out backedges (i.e., find a loop if any exists)
2. Check to see if any origin paths from a currency node which is the destination of a backedge result in $r_{i,j} \cdot r_{j,i} > 1$.
3. The **Bellman-Ford Algorithm** finds *negative* cycles. Perhaps transform our "graph" (wherein nodes represent a currency type and edges represent exchange rates) to a setup where traversal of paths need not require multiplication of exchange rates, $r_{i,j}$, but instead addition (summing) of rates $\implies$ log transform.
4. Simply return whether a risk-free exchange *exists* (not necessarily finding it).

1.3 Algorithm:

Let us set up our currency exchange problem by representing all currencies as nodes on a directed graph. Exchange rates, if they exist, will populate our graph as directed edges.

---

1. Convert edge-weights to their log equivalents (e.g., substitute $log_2(r_{i,j})$ for $r_{i,j}$).
2. Negate the entire graph (i.e., change all edge weights to their negative value) and run Bellman-Ford at each vertex to return if there exists a negative cycle (which would reflect a positive cycle in the original graph).
3. To be clear, Bellman-Ford needs to be run such that it would indicate a negative cycle, so—given the algorithm—its "relaxation" process needs to **iterate** $|\mathbf{V}|$ **times** *each time* the algorithm is run, rather than $(|V| - 1)$ times as assumed in Bellman-Ford when seeking a general shortest path.

1.4 Proof:

1.4.1 The Log Transform:

First, by direct proof, I show that a log transform is valid. Consider the following rule of logarithms:

$$log_2(k) < 0, k < 1 \tag{1}$$

$$log_2(k) = 0, k = 1 \tag{2}$$

$$log_2(k) > 0, k > 1 \tag{3}$$

Given the problem's original specification:

$$r_{i,j} \cdot r_{j,i} > 1 \tag{4}$$

$$\therefore log_2(r_{i,j} \cdot r_{j,i}) > 0 \tag{5}$$

So, using the logarithm product rule (which holds for additional inserts of exchange rates, r by converting a logarithmic expression's argument (expressed as a product of values) into a summation of individual logarithmic terms of each value, respectively) we can express our desired path:

$$log_2(r_{i,j}) + log_2(r_{j,i}) > 0 \tag{6}$$

Naturally, if we negate our edge weights, we can synonymously multiply the above, equation 6, by -1 to get:

$$-log_2(r_{i,j}) + -log_2(r_{j,i}) < 0 \tag{7}$$

Given the validity of the prior steps, we have reduced this problem into a situation where we need to apply the Bellman-Ford algorithm to find a negative cycle (i.e., the sum of edge weights results in some path yields a distance less than 0 as seen in equation 7). Bellman-Ford is proven (in course notes) to be a correct method to seek negative cycles. ■

1.5 Run-time Analysis:

Our initial transformation of edge-weights to their log-2 equivalents takes a constant time per edge $\implies O(|E|)$. Then, the Bellman-Ford algorithm is applied with run-time $O(|V| \cdot |E|)$ at every vertex.

$$O(|E|) + O(|V|^2 \cdot |E|)$$

$$\approx O(|V|^2 \cdot |E|)$$

---

2 New MST Search

2.1 Given + Assumptions:

- We are given the original minimum spanning tree, T
- We are given the whole graph, G

2.2 Algorithm:

First, we shall scan simultaneously the original MST and the adjusted tree to find the changed edge. This can be performed in $O(|E|)$ time. Next, when the changed edge is identified, remove it from the MST. Now we have split our graph into sets, let's call them S and $V - S$, seeing as all vertices are contained in the two sets, and we have split the adjusted tree into at least one segment X, where X is a subset of some minimum spanning tree, and S has no edge in X crossing between S and $V - S$. We need to now apply the Cut Property to search for a minimum weight edge crossing between S and $V - S$. In this scenario, we can take the proof as shown in Lecture 7 to show that we will return a minimum spanning tree of the adjusted graph.

2.3 Satisfy Cut Property Requirements?

We must show that we satisfy the cut property requirements, namely that our section X is a portion of a minimum spanning tree of the graph. We can prove by contradiction: suppose that X is not part of the minimum spanning tree of G, then there exists a method by which to connect the vertices in S in a better fashion than currently done in X implying that our original MST was not an MST, after all, which contradicts the fact that it was an MST. So let us now show that we have actually found a way to split and select S and $S - V$ such that no edge crosses. Suppose the preceding was untrue, thereby an edge crosses S and $S - V$ despite removing an edge from our adjusted tree (specifically, the changed weight edge). If this is the case, there would be a cycle in the original MST, negating its position as an MST $\implies$ contradiction. So we can see that we satisfy cut property requirements and can reduce this problem without further proof.

2.4 Run-time analysis:

The run-time is composed of two elements: first the finding of the augmented edge in the MST $O(|E|)$, then the iterations to find the lightest edge between our cut which requires visiting all vertices in one cut and checking edges which connect to the other cut segment $O(|V|)$.

$$\implies O(|V| + |E|)$$

---

3 Set Cover Family

Take a family of sets such that the number of elements can be represented in the following way:

$$n = k \cdot 2^a$$

where $k = 3$ for our initial scenario.

Say there exist an optimal set cover in which $\frac{1}{3}$ of the total elements are covered in subsets O_1, O_2, O_3. Now take an additional grouping of subsets wherein the first S_0 contains $\frac{1}{2}$ of each O_1, O_2, O_3 thereby comprising $\frac{1}{2}$ of the entire set, thus selected first by a greedy algorithm (S_0 is of size $k \cdot 2^{a-1}$. S_1 contains $\frac{1}{2}$ of each O_1, O_2, O_3 not already covered by S_0 (i.e., $\frac{1}{2}$ of all remaining uncovered elements which is larger than those covered by O_1, O_2, O_3). This process repeats until there exists 1 element in each O_1, O_2, O_3 which remain uncovered by any set S_i (where S_i is of size $k \cdot 2^{a-i}$, for integer $i \geq 1$). Thus we must collect the sets O_1, O_2, O_3 which would have been optimal at the start if we collected them first despite their lower coverage percentage of uncovered sets. (See Figure 1 for an example where $k = 3$.)

We collect, then, $a - 1 + k$ sets which is larger than $log(k \cdot 2^a)$ by logarithmic rules:

$$a - 1 + k > log(k) + a \tag{8}$$

We have shown that for the example above and fully generalizable, the set cover returned by the greedy algorithm is bounded below by log(n).

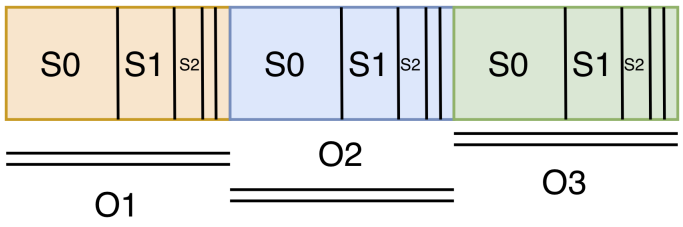

Figure 1: $k = 3$ *Set Cover example*
Note the divisibility once by three afforded by the k term in $n = k \cdot 2^a$
and then the repeated divisibility by 2 after which point given by the power of 2.

4 Scheduling Jobs

Given a set of jobs $\{j_1, j_2, j_3, ..., j_n\}$ with respective run-times, r_i, for sequential processing—but distributable to 2 equivalent machines—we must show that a greedy algorithm of placement of each job successively on the machine with the lower current load yields at worst a $\frac{3}{2}$ performance ratio when compared to the optimal (i.e., minimal run-time) placement of jobs on the two machines.

4.1 Lower Bounds of Optimal Run-time:

Before continuing further, let's consider the lower bounds of an optimal solution.

1. I hold that optimal completion time $:= t_{optimal} \geq r_{max}$ where r_{max} is the largest run-time of a job, j_i. This is clearly true as *some* machine **must** run the longest job.
2. Additionally, I assert that $t_{optimal} \geq \frac{1}{2} \sum_i r_i$ because the machine with the heavier load **must** do at least half of the total job load by definition of it being the "heavier load" machine $\implies t_{optimal} = t_{(completion\ time\ heavy)}$.

4.2 Upper Bounds of Greedy Run-Time

First, I hold that the two machines can, at worst, differ by r_{max} using the greedy allocation algorithm. Let us show this via induction:

1. In our base case, on the first iteration, no jobs have been placed. The first job can be at most with of run-time load $r_{max} \implies$ the machine loads can differ at most by r_{max}.
2. Suppose for all iterations $n < (N-1)$ our machine loads differ at most by r_{max} on the Nth iteration, then, the following scenarios can occur:
   (a) The longest job has already been placed, and—given our greedy algorithm—any next job will go on the machine with lighter load, and seeing as this job, j_N has run-time $r_N < r_{max}$, the load-gap between the machines **can only narrow**, as the only way the machine with the lighter load can exceed the load of the heavier machine (given their difference at $(N-1)$th iteration is $\leq r_{max}$) would be by adding a load greater than r_{max}.
   (b) The longest job has not yet been placed (or there exist multiple replicas of this longest job—one of which has not been placed), by our induction hypothesis, the machines' difference at the $(N-1)$th iteration is $\leq r_{max}$, and by our greedy algorithm, we add r_{max} to the machine with the lighter load again which **can only narrow** our machines' load gap (irrespective of which machine then takes the lead in heaviness of load) if not nullify the gap entirely

   $\implies$ for all iterations, using the greedy allocation, our machines differ at most by r_{max}.

Now, let's establish some bounds on machine loads at the $(n-1)$th iteration using an average work metric similar to our lower bound of optimal run-time from Section 4.1:

1. By definition of its being the lighter-loaded machine (must be below average):

$$load_{light} \leq \frac{\sum^{n-1} r_i}{2} \implies load_{light} \leq \frac{r_{total} - r_n}{2} \tag{9}$$

2. By definition of its being the heavier-loaded machine (i.e., above average) and by what was shown above that

the machines can differ at most by r_{max}:

$$load_{heavy} \leq \frac{\sum^{n-1} r_i}{2} + r_{max} \implies load_{heavy} \leq \frac{r_{total} - r_n}{2} + r_{max} \tag{10}$$

Unfortunately, though, this bounding is not tight enough; after all, if we consider the lighter-loaded machine being at its upper bound (i.e., the average of the (n-1) assigned jobs), the heavier machine cannot be r_{max} above this average simply because we have upended the essence of the "average" now. To reiterate, the upper bound of the heavier machine, given our loose bounds in Equation 10, is r_{max} above the average of the (n-1) assigned jobs, and if the machines can only differ by r_{max}, the lighter machine **must** be at the "average", but then our *average* no longer remains the so-called average as **one machine cannot be above average without the other being strictly below average**.

Let's make a tighter bound, then wherein we *implicitly* take that the machines can maximally differ by r_{max} but **explicitly** state tight upper ("worst-case" bounds):

1. The lighter loaded machine:

$$load_{light} \leq \frac{\sum^{n-1} r_i}{2} - \frac{r_{max}}{2} \implies load_{light} \leq \frac{r_{total} - r_n}{2} - \frac{r_{max}}{2} \tag{11}$$

2. By definition of its being the heavier-loaded machine (i.e., above average) and by what was shown above that the machines can differ at most by r_{max}:

$$load_{heavy} \leq \frac{\sum^{n-1} r_i}{2} + \frac{r_{max}}{2} \implies load_{heavy} \leq \frac{r_{total} - r_n}{2} + \frac{r_{max}}{2} \tag{12}$$

Consider, now, the *n*th iteration (i.e., the final job placement). There exist two cases:

1. $r_n = r_{max}$ thereby equation 11 transforms (by substitution of $r_n = r_{max}$) to:

$$load_{(light\ previous\ iteration)} \leq \frac{r_{total}}{2} \tag{13}$$

   and equation 12 can be written similarly:

$$load_{(heavy\ previous\ iteration)} \leq \frac{r_{total}}{2} \tag{14}$$

   such that the machines can, now, be of equal load share at the upper end of this bound—this is an uninteresting case.

2. $r_n < r_{max}$ thereby equation 11 transforms to:

$$load_{(light\ previous\ iteration)} \leq \frac{r_{total} - r_{max}}{2} + \frac{r_n}{2} \tag{15}$$

   and equation 12 can be written similarly:

$$load_{(heavy\ previous\ iteration)} \leq \frac{r_{total} + r_{max}}{2} - \frac{r_n}{2} \tag{16}$$

Equation 16 is our largest upper bound (i.e., the worst case), then, and it comes in the scenario where the heaviest machine remains the heaviest load (and thereby determines the greedy run-time) after adding the final job.

To write this bound without reference r_n, we can loosen slightly without any loss of generality (given we're loosening):

$$\textbf{greedy run time} \leq \frac{r_{total} + r_{max}}{2} \tag{17}$$

4.3 Reconciling Worst Case Bounds

$$t_{optimal} = max\left(\frac{1}{2}\sum_i r_i, r_{max}\right) \tag{18}$$

$$t_{greedy} = \frac{r_{total} + r_{max}}{2} \tag{19}$$

Let's consider three comprehensive cases, now:

1. $r_{max} > \frac{1}{2}\sum_i r_i$

$$\frac{t_{greedy}}{t_{optimal}} = \frac{r_{total}}{2r_{max}} + \frac{1}{2} \tag{20}$$

The expression above is definitively less than 1 ($\implies$ less than $\frac{3}{2}$) because the first term on the right hand side of equation 20 is strictly less than $\frac{1}{2}$ by the supposition $r_{max} > \frac{1}{2}\sum_i r_i = \frac{r_{total}}{2}$.

2. $r_{max} < \frac{1}{2}\sum_i r_i$

$$\frac{t_{greedy}}{t_{optimal}} = 1 + \frac{r_{max}}{r_{total}} \tag{21}$$

The expression above is definitively less than $\frac{3}{2}$ because the second term on the right hand side of equation 21 is strictly less than $\frac{1}{2}$ by the supposition $r_{max} < \frac{1}{2}\sum_i r_i = \frac{r_{total}}{2}$.

3. $r_{max} = \frac{1}{2}\sum_i r_i$

$$\frac{t_{greedy}}{t_{optimal}} = 1 + \frac{r_{max}}{r_{total}} \tag{22}$$

The expression above is equal to $\frac{3}{2}$ because the second term on the right hand side of equation 22 is $\frac{1}{2}$ by the supposition $r_{max} = \frac{1}{2}\sum_i r_i = \frac{r_{total}}{2}$.

$\implies$ The greedy strategy yields a completion time at worst still within a factor of $\frac{3}{2}$ of the best possible placement of jobs. ■

4.4 Example:

Suppose we have two machines and three jobs where the first two jobs are of length 1 and the third of length 2. In the best placement, the third job (of length 2) is placed on one machine and the other two jobs (of length 1 each) are placed on the remaining machine $\implies$ run-time of 2. In the greedy algorithm, if the first two jobs in to be placed are the shorter ones, the machines are loaded each with one of the 1-length jobs, then the 2-length job is placed on a machine $\implies$ run-time of 3; thereby, the factor of $\frac{3}{2}$ is achieved.

4.5 m-machine Generalization

4.5.1 Lower Bounds of Optimal, Upper Bounds Greedy:

Following our analysis from Sections 4.1 and 4.2:

$$t_{optimal} = max\left(\frac{1}{m}\sum_i r_i, r_{max}\right) \tag{23}$$

$$t_{greedy} = \frac{r_{total}}{m} - \frac{r_{max}}{m} + r_{max} \tag{24}$$

To clarify, equation 24 comes from our earlier analysis of the sole interesting case where $r_n < r_{max}$ (see equation 17 above). Following the "three comprehensive cases" analysis performed in equations 20, 21, and 22, I will show (because it shows the strictest bound the interesting case wherein $r_{max} = \frac{1}{m}\sum_i r_i = \frac{1}{m} r_{total}$:

$$\frac{t_{greedy}}{t_{optimal}} = 1 + \frac{r_{total}}{m \cdot r_{max}} - \frac{1}{m} \tag{25}$$

$$= 2 - \frac{1}{m} \tag{26}$$

4.5.2 Family of examples:

Suppose, with our greedy allocation, we place on every of m machines $(m - 1)$ jobs of length 1 and have a single remaining job of length m. The greedy allocation would place the last job such that total run-time is $2m+1$; whereas, a best case placement would spread one of the machine's $(m - 1)$ length-1 jobs to each other machine, one per remaining machine, before then placing the long m-length job on a machine by itself $\implies$ run-time is m. The ratio, then, reaches our maximum bound shown above:

$$\frac{t_{greedy}}{t_{optimal}} = \frac{2m - 1}{m} = 2 - \frac{1}{m}$$

5 Three-Split n-digit Multiplication

5.1 Assumptions:

— From Piazza @348, "If n is not divisible by 3, you could imagine left padding the number with 1 or 2 zeros which doesn't change the asymptotics. So for simplicity, feel free to assume that **n is divisible by 3**."

— From Piazza @369, addition and bit-shift operations can be summarized in a single $O(n)$ term.

5.2 Setup:

Let's split our two n-digit numbers (where n is a multiple of 3) into thirds as seen in Figure 2:

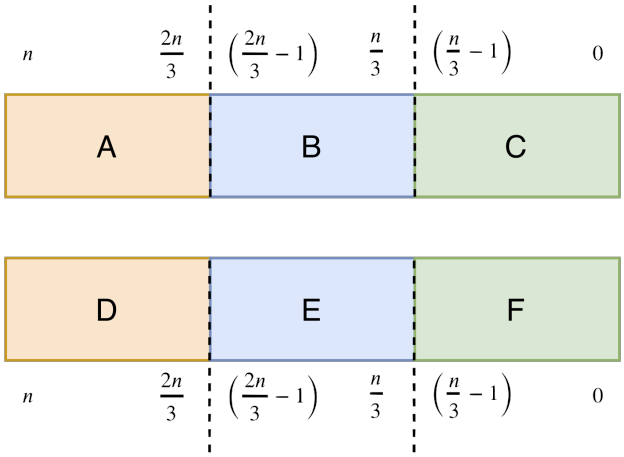

Figure 2: *n-digit numbers split into thirds*

Let $x := \frac{n}{3}$.

5.3 Multiply!

We can express our multiplication, now:

$$(a \cdot 10^{2x} + b \cdot 10^{x} + c) \cdot (d \cdot 10^{2x} + e \cdot 10^{x} + f) \tag{27}$$

Distributing terms of equation 27:

$$(ad \cdot 10^{4x} + ae \cdot 10^{3x} + af \cdot 10^{2x} + bd \cdot 10^{3x} + be \cdot 10^{2x} + bf \cdot 10^{x} + cd \cdot 10^{2x} + ce \cdot 10^{x} + cf) \tag{28}$$

$\implies$ non-ideal 9 multiplications! We can resolve by combining like terms of equation 28:

$$ad \cdot 10^{4x} + (ae + bd) \cdot 10^{3x} + (af + be + cd) \cdot 10^{2x} + (bf + ce) \cdot 10^{x} + cf \tag{29}$$

Equation 29 can be re-written in a manner which minimizes the number of unique products needed:

$$ad \cdot 10^{4x} + [(a + b)(e + d) - ad - be] \cdot 10^{3x} + [(a + c)(d + f) - ad - cf + be] \cdot 10^{2x} + [(b + c)(e + f) - be - cf] \cdot 10^{x} + cf$$

5.4 Reiterating the Algorithm

Split the n-digit value as in Section 5.2. Then add/multiply as follows:

$$ad \cdot 10^{4x} + [(a+b)(e+d) - ad - be] \cdot 10^{3x} + [(a+c)(d+f) - ad - cf + be] \cdot 10^{2x} + [(b+c)(e+f) - be - cf] \cdot 10^{x} + cf$$

Let's now consider the unique multiplications required:

1. $a \cdot d$
2. $b \cdot e$
3. $c \cdot f$
4. $(a+b) \cdot (e+d)$
5. $(b+c) \cdot (e+f)$
6. $(a+c) \cdot (d+f)$

Hooray! It's just 6 multiplies!

5.5 Run-time Analysis (Part B):

We can create a recurrence for this multiplication technique as follows:

$$T(n) = 6T\left(\frac{n}{3}\right) + O(n)$$

Using Master Theorem, we can approximate:

$$O(n^{log_3(6)})$$

Since this is not better than the two-split multiplication $O(n^{log_2(3)})$, I prefer to split into two.

5.6 Five Multiplications:

Supposing we could complete the integer multiplication using five unique multiplications, our recurrence could be re-written:

$$T(n) = 5T\left(\frac{n}{3}\right) + O(n)$$

Using Master Theorem, we can approximate:

$$O(n^{log_3(5)})$$

and since this is better than the two-part split, I would rather split into three if I could reduce to 5 multiplications.