PSET 7
CS124: Data Structures and Algorithms
Prof. Mitzenmacher

Dhilan Ramaprasad
dhilanramaprasad@college.harvard.edu

Collaborator: No one :(
Others: Advice from Phil, Benny, Marwa, and Jaylen (throughout CS124).

As an Electrical Engineering concentrator, I've made a great effort to avoid theoretical CS thoughout my short academic career (seeing as I chose the major sophomore fall). Slowness of my code implementations never scared me; after all, the solution was simple: make better transistors, improve micro-architectures, add stream buffers, more caches, lessen the cost of SRAM, introduce out-of-order processors, parallelize. The closest I ever came to an algorithm was Belady's algorithm for cache member eviction, or Tomosulo's algorithm which has nothing to do with algorithms, or that one time 'Lil Yachty said "you Harvard kids like algorithms, right" during Yard Fest 2018.

But the times have changed. In passing, I asked Eddie K., "Hey, should I switch to a joint with CS?" And he swiftly replied, "YES!" And so a new dilemma arose: (1) **take CS124 junior spring *or*** (2) **take CS20 now** and CS124 *senior spring*. The latter, of course, was not an option; after all, I wouldn't know anyone in the course :(

So I sucked it up, concurrently enrolled in a statistics class (ES150, sadly not STAT110), cried my way through the first PSET with a trip to Mitz's office hours for help on the hardest mathematical proof we did all semester, followed by a late Gradescope submission as I wasted precious hours of PSET2 time to get my matrix multiplication to work (my `Cpp` skills were crusty despite being fresh out of CS61).

Here's what I realized: **CS124 is doable.** It's not as horrifying as people make it out to be. And yes, I have submitted each of my PSETs ≥ 1 day late after going remote, but if I were on campus, I would have finished in time with no problem because I would have had the greatest team known to man by my side throughout these latter 3 PSETs.

Just like the crew of Apollo 13, when it came to **Benny Paris** (OSB, as the world knows him, the master of storytelling in Overleaf and in person, half the brains and all the wit of our operation), **Phil Nicol** (ask him about tumor generator or all things Cpp—oh, also, he's the other half of the brains and the *entirety* of the looks of our operation), and **Dhilan** (not much to contribute—no brains, no looks—but above average `.tex` formatting and the occasional mediocre code snippet)—failure was not an option—we were unstoppable.

Now, while I will never have the opportunity for Mitz to hand me a failing grade (unless I successfully cop an unsat or *UEM*), I can rest easy knowing that even if I failed, I succeeded in being member to one of the most iconic teams in human history. Thank you to Benny and Phil for everything. Thank you to Rajath for nothing. Thank you to Esther and Amy for your early guidance. Thank you to Rachel and Rose for great sections. And of course, the biggest, uh, like, thank you to an inspiration: **Mitz.**

1 Go with the Flow

Setup: We are given the maximum flow (value and flow along each edge) in a graph $G = (V, E)$ with source s, sink t, and integer capacities.

1.1 Increased by 1

Given that the capacity of one edge in our graph is increased by 1, we can easily find a linear time algorithm to compute the new maximum flow.

We first need to construct the residual graph given the original maximum flow for the unchanged (no incremented edge capacity) graph. It can be assumed that this can be done in O(|V|+|E|) time as we have the information for flow across each edge a priori.

Now, on the residual graph, we can add in our change: the increased capacity to an edge. Running Ford-Fulkerson, we run DFS in O(|V| + |E|), but seeing as we know that the max-flow has increased by **at most** 1, we only need to run DFS once to find the augmenting path (i.e., path from s to t along which we can increase flow, specifically by 1 in this case) if it exists. Naturally, if the path exists, increase the flow, add 1 to our prior maximum flow, and return the new residual graph.

It is worth noting explicitly that the maximum flow is an integer value (due to integer capacities) and it increases at most by 1. This comes naturally from our discussion in lecture and section of finding an min S-T cut which demonstrates the maximum flow of G. Of course, now one capacity may have increased by 1 along this cut, but no more than that, so the maximum flow may only increase by 1. Note that it cannot decrease.

1.2 Decreased by 1

The decrementing scenario requires two cases (quite non-trivial). Initially, I thought we could use the residual graph again and find a path from t to s this time, but I am not so sure that works.

Instead, consider the first case where the selected edge $(u, v) \in E$ for decreased capacity $c'(e) = c(e) - 1$ had flow (given in the original graph) which was less than its original capacity—i.e., $f(e) < c(e)$: **then, there is no change to the maximum flow** as there was "over-capacity" at the start (before decrementing).

In the case where original $f(e) = c(e)$ for the selected edge, e, then we must determine whether another augmenting path can compensate for the lost flow. The process is as follows: **(1)** construct the residual graph, **(2)** find a path from s to t containing the selected edge, e, and reduce flow on each edge in the found path such that now the flow to t has dropped by 1, **(3)** now seek a new augmenting path (if one exists) from s to t by running Ford-Fulkerson with DFS once (knowing that our max-flow can at most return to what it was prior—the original value—but cannot change further given it was *max* prior to our decremented capacity), **(4)** if a new augmenting path is found, we can increase flow along the path and note the new flow along edges traversed and note that the max flow value is unchanged as there was more than one unique solution to the max flow problem on the original graph; else, if no new path is found, return our adjusted graph

with decremented edges, and return the new inflow to t as our updated max flow (which is just decremented by 1).

The **runtime** is $O(|V| + |E|)$ to make the residual graph, $O(|V| + |E|)$ to find a path from s to t containing the selected edge (via BFS), and $O(|V| + |E|)$ for one run of Ford-Fulkerson $\implies O(|V| + |E|)$.

2 The War for Independence

2.1 Line on n vertices

First let's explore a few values and scenario: if we define the function $I(n)$ to return the number of independent sets given n-node line, taking the empty set to be an independent set $\implies$ when $n = 0$, $I(0) = 1$. See Table 1 for more selected values:

n	I(n)
0	1
1	2
2	3
3	5
4	8

Table 1: *Number of Independent Sets, varied size-n of line graph*

By observation, **I(n)** returns **I(n − 2)** + **I(n − 1)**. This recurrence, given its starting value (base case) $I(0) = 1$, can be represented by the requested "family of numbers" which are the **Fibonacci** numbers. Specifically, $I(n) = \text{Fib}(n+2)$

Note: My definition of *Fibonacci* numbers follows that of `mathworld.wolfram.com` wherein $\text{Fib}(1) = \text{Fib}(2) = 1$.

The proof of the recurrence comes in that adjacent nodes cannot both be in the independent set (as the set would no longer be independent), so we can isolate our nth case as follows:

1. Take node n to **not** be part of the independent set. Then node $n - 1$ (being adjacent) surely *can* be member to many independent sets without node n, and its representation is simply $I(n - 1)$.
2. Now take node n to be member to the independent set. Because in the prior step we have counted all eligible steps up to $n - 1$ nodes where node $n - 1$ may or may not be in the independent set, we must be careful not to double count sets. It is naive to say if node n is now an eligible member, node $n - 1$ is no longer an active member, so we can take $I(n - 1)$ again, as that would over-count. So we can imagine now restricting ourselves to all of the independent sets wherein node n **must** be member. This is represented by $I(n-2)$ which represents all of the independent sets wherein (1) node $n-1$ is not member and (2) node n is not member; however, given we've already accounted for all scenarios of (2), we can take $I(n-2)$ to actually represent all of the independent sets wherein (1) node $n-1$ is not member and now (2) node n is **member to all** thereby ensuring these $I(n-2)$ sets are unique—not double counts—and thereby exhaustively covering all constructible independent sets.

2.2 Cycle on n vertices

First let's explore a few values and scenario: if we define the function $K(n)$ to return the number of independent sets given n-node cycle, taking the empty set to be an independent set $\implies$ when $n = 0$, $K(0) = 1$. See Table 2 for more selected values:

n	K(n)
0	1
1	2
2	3
3	4
4	7
5	11
6	18

Table 2: *Number of Independent Sets, varied size-n of cycle graph*

After $n = 3$ (the first "standard" cycle), we can see a simple recurrence wherein $\mathbf{K(n) = K(n-1) + K(n-2)}$ which neatly matches the same recurrence as the Fibonacci numbers, but its base cases differ, so the recurrence for $n > 3$ actually maps to the family entitled **Lucas numbers**.

$$\text{for Lucas numbers, } L_1 = 1 \text{ and } L_2 = 3, \text{ and } L_n = L_{n-1} + L_{n-2} \text{ [just like Fibonacci]}$$

In the proceeding analysis, it just so happens that $\mathbf{K(n) = I(n-1) + I(n-3)}$ for all n using our prior recurrence for line graphs.

The proof of the recurrence again comes in that adjacent nodes cannot both be in the independent set (as the set would no longer be independent), so we can isolate our nth case **and reduce to the prior scenario (line graph)** as follows:

1. Take node n to **not** be part of the independent set. Then node $n-1$ and node 0 (being adjacent left and adjacent right, respectively) surely *can* be member to many independent sets without node n, and so our representation of all the number of independent sets sans node n is simply $I(n-1)$ as we have one fewer node which can be represented as being extracted from the graph leaving a line graph of size $n-1$.
2. Now take node n to be member to the independent set. Because in the prior step we have counted all eligible steps up to $n-1$ nodes where nodes 0 and $n-1$ may or may not be in the independent set, we must be careful not to double count sets. We can imagine now restricting ourselves to all of the independent sets wherein node n **must** be member. This is represented by $I(n-3)$ which represents a line graph of size $n-3$ (i.e., missing nodes 0, $n-1$, and n) which would seem to double count cases, but we now consider that node n is **member to all** of these $I(n-3)$ counted sets which makes them unique from those counted previously, exhaustively covering all constructible independent sets.

2.3 Complete binary tree

Let us begin by notating a few convenient factors and terms. First, for a complete binary tree, the number of nodes is one minus power of 2—i.e., $n = 2^k - 1$ where k conveniently represents the "height" or depth of the tree (number of "levels").

Like our prior similar question regarding *set cover* with binary trees, we know that there are two scenarios: (1) take root and grandchildren (optional in the case of constructing any independent sets—unlike in set cover) but **not** children, and (2) take children but **not** root or grandchildren.

Naturally, we see that the number of children is 2 to every parent, and the number of grandchildren is 4 to every grandparent (original root). If the parent is at level k with. $2^k - 1$ nodes, each child is a sub-tree composed of $2^{k-1} - 1$ nodes, and each grandchild now roots a sub-tree composed of $2^{k-2} - 1$ nodes.

Let's take **T(k)** to represent the number of independent sets given by a complete binary tree of height k where we have the base cases $T(0) = 1$ and $T(1) = 2$.

Suppose **we now examine the root of a tree of height k:**
using the multiplication rule in independent probability/counting (from statistics), we know that for children of the first node there are $T(k-1) \times T(k-1)$ independent set options (given there are **2** children each of whom are roots to a sub-tree of height $k - 1$).

Similarly, for grandchildren, $\exists\ T(k-2) \times T(k-2) \times T(k-2) \times T(k-2)$ independent set options (given there are **4** grandchildren each of whom are roots to a sub-tree of height $k - 2$).
Let's now follow a similar strategy as before (line and circle graphs) wherein we **take *or* exclude** the root (node n) and have a two-term sum recurrence. The term of the recurrence which represents inclusion of the root *is* counting correctly because, like before, it mandates inclusion of the root (node n) in **all** independent sets counted by the term, which prohibits any double counting—i.e., counted sets are unique.

We will sum the cases where (1) we include the root (and thereby grandchildren are eligible, though not required) which prohibits any children being included, and (2) the case where we do **not** include the root, thereby, we are left with the children subtrees and are free to create independent sets on the smaller trees whose total count will multiply as stated before:

$$\implies T(k) = \underbrace{T(k-1) \times T(k-1)}_{\text{children}} + \underbrace{T(k-2) \times T(k-2) \times T(k-2) \times T(k-2)}_{\text{grandchildren}}$$

From the Online Encyclopedia of Integer Sequences (OEIS), the result could be found, but a Python script was used to confirm: for 127 nodes ($k = 7$), the number of independent sets is **133453460314446328414276439 06**

Code Exerpt:

```
def T(n):
    if n == 0:
      return 1
    elif n == 1:
      return 2
    else:
      return T(n-1)*T(n-1) + T(n-2)*T(n-2)*T(n-2)*T(n-2)
```

3 MAX-k-CUT

3.1 Randomized Algorithm

Taking our randomized algorithm for MAX CUT, the generalization here, is simple. Suppose we have a k-sided die. Roll the die to place each vertex into a set numbered $(1, 2, \dots, k)$.

The probability a vertex, $v \in V$, is in set k is naturally $P(v \in S_k) = \frac{1}{k}$. For a given edge, $(u, v) \in E$, the probability it crosses between sets (any two distinct sets) is the probability that u and v are not in the same set:

$$P((u, v) \in E \text{ crosses between sets}) = P(v \notin S_k \mid u \in S_k) = 1 - P(v \in S_k \mid u \in S_k)$$
$$\implies P((u, v) \in E \text{ crosses between sets}) = 1 - \frac{1}{k}$$

Given the above, we now find the expectation (i.e., the expected number of edges crossing between sets):

$$\mathbf{E}[(u, v) \text{ crosses between sets } \forall (u, v) \in E] = |E| \cdot \left(1 - \frac{1}{k}\right)$$

The upper bound on edges crossing any of k-cuts is O(|E|), so our randomized algorithm has an approximation factor of $\left(1 - \frac{1}{k}\right)$.

3.2 Local Search

We can begin to approach the algorithm as we did in lecture:

WHILE $\exists$ a vertex ($v \in V$) that increases the MAX-k-CUT by moving to a new disjoint set k, move the vertex v.

First, we should show termination of this algorithm:
We know that there is an upper bound on the cut size (as seen when examining the randomized algorithm), so at some point, there will definitively **not** be a moveable vertex which will increase the MAX-k-CUT beyond its intrinsic limit.

Now, let's examine the division of a vertex and its neighbors given termination of the algorithm:
Just as in lecture, we can bound the number of neighbors a vertex has in each set—i.e., for each vertex, no more than $\frac{1}{k}$ of its neighbors are in its final assigned set; else, we'd move the vertex to a new set such that we could capitalize on an increase in number of edges crossing the cuts. In other words, where $\delta(v)$ represents the neighbors of a vertex, v, we know that $\left(1 - \frac{1}{k}\right) \cdot \delta(v)$ connecting vertices are in other sets $\implies \geq \left(1 - \frac{1}{k}\right) \cdot \delta(v)$ edges crossing between sets per vertex.

Finally, let's perform some *adapted* accounting to prove performance bound:

1. Take any vertex $v \in S_i$
2. For every vertex $w \in S_j, j \neq i$, add $\frac{1}{2}$ to a running sum.
3. Repeat for each vertex in each set—i.e. $\forall v \in S_i, i = \{1, 2, \dots, k\}$

In mathematical notation, where C is the number of crossing edges:

$$C = \frac{1}{2}\left(\sum_{v\in S_1}\left|\{w\colon (v,w)\in E, w\in S_j, j\neq 1\}\right| + \cdots + \sum_{v\in S_k}\left|\{w\colon (v,w)\in E, w\in S_j, j\neq k\}\right|\right) \tag{1}$$

Knowing a lower bound for the edges crossing between sets per vertex as delineated above to be 1 - $\frac{1}{k}$. We can rewrite C as follows:

$$C \geq \frac{1}{2}\left(\sum_{v\in S_1}\left(1-\frac{1}{k}\right)\delta(v) + \cdots + \sum_{v\in S_k}\left(1-\frac{1}{k}\right)\delta(v)\right) \tag{2}$$

$$C \geq \left(\frac{1}{2}\right)\left(1-\frac{1}{k}\right)\left(\sum_{v\in S_1}\delta(v) + \cdots + \sum_{v\in S_k}\delta(v)\right) \tag{3}$$

$$C \geq \left(\frac{1}{2}\right)\left(1-\frac{1}{k}\right)\underbrace{\sum_{v\in V}\delta(v)}_{\leq 2|E|} \tag{4}$$

$$\implies C \geq \left(\frac{1}{2}\right)\left(1-\frac{1}{k}\right)2|E| \tag{5}$$

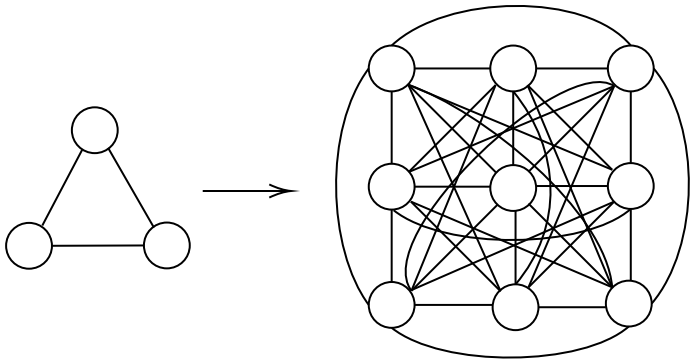
We have shown, then that the number of crossing edges is bounded below by $C \geq \left(1-\frac{1}{k}\right)|E|$. The upper bound on edges crossing any of k-cuts is O(|E|), so our randomized algorithm has an approximation factor of $\left(1-\frac{1}{k}\right)$.

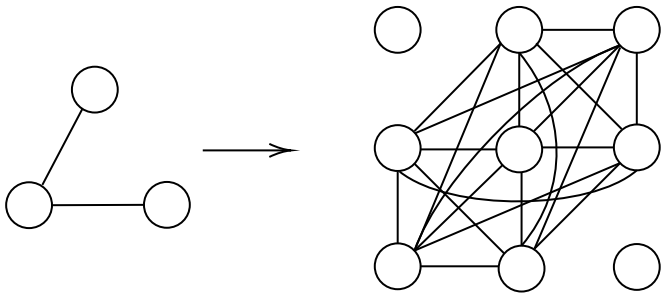
4 It never really *clique'd*

I'd like to preface this solution with an apology. I spent too much time on the other problems, leaving this for last, and naturally, my solution demonstrates what the kids often call *a weak showing*. I would say, "I will do better next time," but seeing as this is the last pset—well, uh, like—I'm sorry. Nevertheless, I hope some of what I have said merits partial credit :)

4.1 Examples and clique size in G′:

Let's explore two simple examples:





G → G'

$$G' = G \times G = (V', E')$$

$$V' = V \times V$$

$$\{(u, v), (w, x)\} \in E' \textbf{ iff } [\{(u, w)\} \in E \textbf{ or } u = w] \textbf{ and } [\{(v, x)\} \in E \textbf{ or } v = x]$$

By construction, it can be easily seen, then that for a k-maximum clique in G, $\exists$ a k^2 clique in G'.

As we move toward higher dimensionality (i.e, $G'', G''', \ldots, G^n$) we see that for a given k-maximum clique in G^i, $\exists$ a k^2 maximum clique in $G^{(i+1)}$ and likewise $2k^2$ edges connecting the clique, again by construction and given the constraint for edge placement in the Cartesian product graph. Proving the other direction (i.e., $\exists$ a $\sqrt{k}$ maximal clique in G given a k-maximum clique in G') is not so simple. I will return to this if time permits, but considering

our example shown above, let us take, for now, that $\exists$ a $\lfloor \sqrt{k} \rfloor$-maximum clique in G for a given k-maximum clique in G'.

4.2 Poly-time ϵ-approximation algorithm:

Knowing there exists a 2-approximation algorithm to find the k-maximum clique; thereby, the approximation algorithm would return between $\frac{k}{2}$ and $2k$ for the size of the maximum clique in G (**note:** Eric Zhang explained on Piazza that the 2-approximation does not go above the bound of the actual k, but it will not affect our analysis). Similarly, given G' will have a maximum clique of size k^2, the 2-approximation would return between $\frac{k^2}{2}$ and $2k^2$ for $G' \implies$ an approximation within the bounds of $\left\lfloor \frac{k}{\sqrt{2}} \right\rfloor$ and $\lfloor \sqrt{2}k \rfloor$ in G.

As we generalize to repeated construction of our Cartesian product graph such that we make G^n (which—given a k-maximum clique in G—will have a corresponding k^{2^n}-maximum-clique), we can recognize that our bounds will be further constrained to returning for G a k-maximum clique within the range $\left\lfloor \frac{k}{\sqrt[2^n]{2}} \right\rfloor$ and $\lfloor \sqrt[2^n]{2}k \rfloor$.

It naturally follows, then, that some arbitrarily large value for n can be chosen such that we satisfy $1 + \epsilon > \sqrt[2^n]{2}$, where $\epsilon > 0$.

Given a polynomial-time algorithm for the 2-approximation, it will be run on n graphs, where n only changes with respect to ϵ, not $|V|$ or $|E|$ or $k \implies$ we retain a polynomial run-time algorithm.

5 Local search for *makespan* minimization

Considering the local search algorithm in which we repeatedly swap a single job from one machine to another **if** the swap will *strictly reduce* total completion time. The algorithm continues until the jobs are in a steady state such that a swap is not possible. Though the steady state is not necessarily optimal, we shall show that the algorithm always terminates in a steady state and that the completion time is within a factor of $\frac{4}{3}$ of the optimal (i.e., a $\frac{4}{3}$-approximation algorithm).

5.1 Termination

Naturally, the algorithm must terminate, specifically given (1) the operational fact that swaps are only made if completion time is *strictly* reduced, and given (2) the assertion that there does exist an optimal completion time which serves as our lower bound $\implies$ our algorithm cannot exceed said optimal (i.e. have a completion time lower than the optimal). This argument is similar to that which we made for the local search algorithm for approximating MAX CUT wherein there exists an upper bound on cut size $\implies$ the algorithm must terminate (cannot continue increasing). Analogously, here $\exists$ lower bound on completion time at which the algorithm must stop, for $\nexists$ swap which would present lower completion time.

5.2 4/3-approximation

Let's first establish some terminology. Suppose we have two machines, M_1 and M_2 which have jobs (j_i) with corresponding run-times (r_i) assigned to each: $(j_{1,i}\,, r_{1,i}) \in M_1$ and $(j_{2,i}\,, r_{2,i}) \in M_2$. The maximum load ($L$) of either machine (where $L_1 = \sum_{j_{1,i}} r_{1,i}$ and $L_2 = \sum_{j_{2,i}} r_{2,i}$) is entitled the *makespan* or completion time which we will represent as C. Take C^* to represent the optimal completion time for a set of n jobs—given by distributing jobs onto each machine such that their respective loads are as similar as possible, thereby parallelizing work. The optimal *makespan* given n jobs is bounded below by $C^* \geq \frac{1}{2}\sum_i^n r_i = \frac{L_1+L_2}{2}$ and is bounded below by the longest job: $C^* \geq r_{max}$.

As suggested by the provided hint, let us **suppose** that our local search algorithm terminates (i.e., reaches stable state wherein no transfer of jobs lessens the max load of the machines) with completion time C_{ls} greater than $\frac{4}{3}C^*$.

We can take that there exists a heavier and lighter-loaded machine (L_{light} and L_{heavy}) as even in the best case optimal *makespan*—where the machines are loaded equally—given our $\frac{4}{3}$-factor, the local search algorithm could *still* load one machine more heavily thereby allowing completion time $C_{ls} \geq \frac{4}{3}C^*$. L_{light} must be bounded above by C^* by conservation of workload (one load must be below average, one above, and C^* is at best average) $\implies L_{heavy} \geq C^*$.

In the case of our supposition wherein the algorithm's completion time is not within a factor of $\frac{4}{3}$ of the optimal, we can state the following regarding the load on the machines, given we are at steady state:

$$\underbrace{L_{light}}_{<C^*} + r_i > L_{heavy} \qquad \forall (j_i, r_i) \in M_{heavy} \tag{6}$$

$$L_{heavy} > \frac{4}{3}C^* \tag{7}$$

The above is true because if there existed **any** job on M_{heavy} (even the shortest) which could be placed on the lighter-

loaded machine and not exceed L_{heavy}, we would not be at steady state and would have found a better distribution thereby lowering our total completion time (i.e., possibly bringing C_{ls} *within* a bound of $\frac{4}{3}C^*$).

Isolating segments of the inequality shown in Equation 6, we can construct the following:

$$r_i > L_{heavy} - L_{light} \quad \forall (j_i, r_i) \in M_{heavy} \tag{8}$$

Knowing $C^* \geq \frac{1}{2}\sum_i^n r_i = \frac{L_1+L_2}{2} = \mu$ (average of total workload), we can substitute for the inequality in Equation 7:

$$L_{heavy} > \frac{4}{3} \cdot \frac{L_{light} + L_{heavy}}{2} \implies \frac{1}{2}L_{heavy} > L_{light} \tag{9}$$

Returning to Equation 8, we find the following while remembering Equation 8 held for all jobs on M_{heavy}, so in particular (and without loss of generality), we'll select the lightest (smallest/shortest) job (r^*) which will still meet the following inequalities and their conversions through substitution—namely using what we found in Equation 9:

$$r^* > L_{heavy} - L_{light} \tag{10}$$

$$r^* > L_{heavy} - \frac{1}{2}L_{heavy} \implies r^* > \frac{1}{2}L_{heavy} \implies \boxed{r^* = L_{heavy}} \tag{11}$$

Herein lies our contradiction . We stated that r^* represented the smallest job on M_{heavy}, and since in Equation 11 we state $r^* > \frac{1}{2}L_{heavy}$, it must be the sole job on M_{heavy} as it is $> \frac{1}{2}$ the total load on the heavier machine, yet the smallest job on the machine $\implies$ no larger job would allow L_{heavy} to meet said bound where the smallest job is *still* greater than half of the load, so naturally M_{heavy} then has only a single job with run-time $r^* = L_{heavy} = C_{ls}$ as L_{heavy} is defined to be the maximum load and thereby defines the *makespan* of the local search algorithm's result.

We previously also stated that the optimal *makespan* is bounded below by the longest job: $C^* \geq r_{max}$, and given all of the jobs in our trial and the fact that M_{heavy} is a single-job machine and also the determining factor for completion-time C_{ls}, then we know $\mathbf{r^* = r_{max}}$ because any longer jobs longer than r^* existing on M_{light} would conflict with our definition of (M_{heavy}, L_{heavy}).

So, $r^* = L_{heavy} = C_{ls} = r_{max}$ and $C^* \geq r_{max}$.

We initially assumed $C_{ls} \geq \frac{4}{3}C^*$, but now we find $C_{ls} = r_{max} \not\geq \frac{4}{3}C^*$ as $\boxed{\mathbf{r^* \not\geq \frac{4}{3}r^*}}$

Therefore, by contradiction, we have shown that the local search algorithm will not produce a completion time C_{ls} which is beyond a 4/3 factor of optimal. ■