Assignment 4

Gradescope Ids: 605 & 557

All group members have read and followed the guidelines for academic conduct in CPSC 320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names (and Gradescope information) away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff.

1 I'm a Lumberjack (And I'm Okay)

1.1 Root-finding

1

```
findRoot(n, e):
  min = 0
  x = n
  max = n

  while (abs(x * x - n) > e):
    x = (min + max) / 2
    if (x * x > n):
      max = x
    else:
      min = x

  return x

abs(i):
  if (i < 0):
    i =  -1 * i
  return i
```

2

With each iteration of the loop, exactly one of max or min is updated to the value of x, which is the midpoint between the two values. This cuts down our problem size by half with each iteration. A recurrence relation for the loop would be *T(n) = T(n/2) + constant*, which gives a runtime of O(lg(n)) by the second case of the Master Theorem.

1.2 Array-chopping

1

Using a 0-indexed array:

```
findC(A):
  if |A| < 2:
    c = -1    # Error case
  else if |A| == 2:
    c = (A[1] + A[0]) / 2
  else:
    c = min(A[2] - A[1], A[1] - A[0])
  return c
```

2

Once again, using a 0-indexed array.

```
findMN(A):
  c = findC(A)
  return mnHelper(A, 0, |A| - 1, c)

mnHelper(A, start, end, c):
  if start >= end:
    return infinity   # Error case

  mid = floor(start + (end - start) / 2)

  if (A[mid + 1] - A[mid]) != c:
    # Missing element is between A[mid] and A[mid + 1]
    return A[mid] + c

  else if mid > 0 and (A[mid] - A[mid - 1]) != c:
    # Missing element is between A[mid - 1] and A[mid]
    return A[mid] - c

  else if A[mid] == (A[0] + c * mid)
    # Missing element somewhere in 2nd half
    return mnHelper(A, mid + 1, end, c)

  else
    # Missing element somwehere in 1st half
    return mnHelper(A, start, mid-1, c)
```

3

As `findC` operates in constant time, the bulk of the work is done in `mnHelper`. In `mnHelper`, we have three options: recursively call `mnHelper` on indices *start* to *mid - 1*, recursively call `mnHelper` on indices *mid + 1* to *end*, or do a constant amount of work and return the answer. This effectively cuts down the sample size by half with each call until the answer is found. This gives us a recurrence relation of $T(n) = T(n/2) + constant$, and thus a runtime of O(lg(n)) by the second case of the Master Theorem.

2 Tiles and Tribulations

2.0 Warming up

1	1	3	3
1	-	4	3
2	4	4	5
2	2	5	5

2.1 Whether-proof tiles

Start, as prompted, by assuming we can cover any arbitrary 2^2x2^2 board with one cell missing. We are then able to cover any 2^3x2^3 board by placing a tile at the centre of the board to fill the 2x2 square made by a corner from each of the four 2^2x2^2 boards that make up the 2^3x2^3 board, pictured below as x's.

-	-	-	-				
-	-	-	-				
-	-	-	-				
-	-	-	-	**x**			
			x	**x**			

Placing the tile here leaves the remaining three corners each as 2^2x2^2 boards with one piece remaining, which we already know we can solve for.

This argument continues to any 2^kx2^k problem, as we can break it into four $2^{k-1}x2^{k-1}$ problems, each of which can be broken into four $2^{k-2}x2^{k-2}$ and so on until we get to our 2^2x2^2 case we know we can solve for.

2.2 A million little quadrants

We do a divide and conquer solution as follows:

Note: we assume that the "add L-piece" part of the code tags the filled spots with some unique identifier in order to be able to see the solution when it is finished

```
FillRec(square, sideLength)
```

```
// base case
if(size = 2)
    add L-piece to fill the square

// recursive case, split into 4 more quadrants and join with an L-piece
else
    create P1 to be the quadrant with the missing piece in it.
    create P2, P3, and P4 to be the other 3 quadrants

    FillRec(P1, sideLength / 2)

    add L-piece on the corner of P1 with one square in each of the remaining 3 quadrants

    FillRec(P2, sideLength / 2)
    FillRec(P3, sideLength / 2)
    FillRec(P4, sideLength / 2)
```

2.3 More whether-proofing

No. In order for a solution to be possible for a certain square with side length x, x^2-1 must be divisible by 3.

With 2kx2k, take for example k = 3. Our square is then 6x6 = 36. 36%3 = 0, so (36-1)%3 must be 2. And in fact 35 / 12 = 11 remainder 2. So it is impossible to solve this problem and so therefore we can't always solve the general problem of 2kx2k

3 Greed is Good, But Conquest is Better

3.1 BF Bank?

1.

```
init best to 0

// one based indexing
for i = 1 to n
    for j = i to n

    charge = -(j-i+1) * max{(A[i], A[i+1], ... A[j])}

    best = max(charge, best)
```

1. This solution checks all possible intervals in A for the best charge. For simplicity we call the second loop's bounds as 1 to n as well (this is also more brute force but using i to n in the original solution made the A[i] ... A[j] easier to write out), then this is n iterations of n iterations -> O(n^2)

3.2 D + C = Profit

1. Since the maximum element in A is 4, it means that the entire array A are negative numbers and the account is in

overdraft for the full time period. Therefore the charge is 4d for any interval d in the array.

2.

```
init best to 0

init count and cur_max to 0
for each a in A
    if a >= 0:
        best = max(best, cur_max * -count)
        count = cur_max = 0
    else:
        cur_max = max(cur_max, a)
        count++
```

3. This is a single iteration through the array, so worst case is O(n)

4. The average case has the same runtime, since we cannot know if we have the best solution yet without looking through the array until the end. So best case = average case = worst case = O(n)

4 Debug-and-Conquer

1a)

The best case for this algorithm is a graph which is minimally connected, for example a tree or chain. In this case, there are only *m = n - 1* edges, and so the work done within each call to subgraphs, O(m+n) is O(n). As the rest of the non-recursive work done in each call to `DC_MST` is at most linear, the work done within each node is O(n).

1b)

The worst case for this algorithm is a graph which is fully connected. In this case, there are *m = n(n-1)/2* edges, which means that the work done within each call to subgraphs, O(m+n) is $O(n^2)$. Asthe rest of the non-recursive work done in each call to `DC_MST` is at most linear, the work done within each node is $O(n^2)$.

2a)

As the size of each vertex set produced by `subgraphs()` differ by at most one, and two graphs are produced, we can say that the two subgraphs produced are each approximately half the size of the original problem. As both are recursed upon, and in the best case we have O(n) work at each node (as established in *1a)*), we have the following recurrence relation:

T(n) = 2T(n/2) + n

The Second Case of the Master Theorem states that if for some constant *k >= 0*, and recurrence relation

T(n) = aT(n/b) + f(n) where *a >= 1* and *b > 1*

$f(n) \in \Theta\ (n^c(log(n))^k)$, where $c = log_b a$, then

$T(n) \in \Theta\ (n^c(log(n))^{k+1})$

Here, we have $a = 2$, $b = 2$, $c = log_2 2 = 1$, and $f(n) = n$.

So we must show $n \in \Theta\ (n^1(log(n))^k)$. Here, choose $k = 0$. Then, $n \in \Theta\ (n)$. So, $T(n) \in \Theta\ (nlog(n))$, which also defines the tight lower bound and thus $T(n) \in \Omega\ (nlog(n))$.

2b)

By the same reasoning as in *2a)*, but with the worst-case work per node as determined in *1b)*, we have the following recurrence relation:

$T(n) = 2T(n/2) + n^2$

The Third Case of the Master Theorem states that if for some recurrence relation

$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

If $f(n) \in \Omega(n^c)$ where $c > log_b a$, and $af(n/b) <= kf(n)$ for some constant $k < 1$ and and sufficiently large n, then

$T(n) \in \Theta\ (f(n))$

We have $a = 2$, $b = 2$, $log_2 2 = 1$, and $f(n) = n^2$.

First, we have $n^2 \in \Omega\ (n^c)$ for $c > 1$. Choose $c = 2$, then we have $c > 1$, and $n^2 \in \Omega\ (n^2)$ for all n.

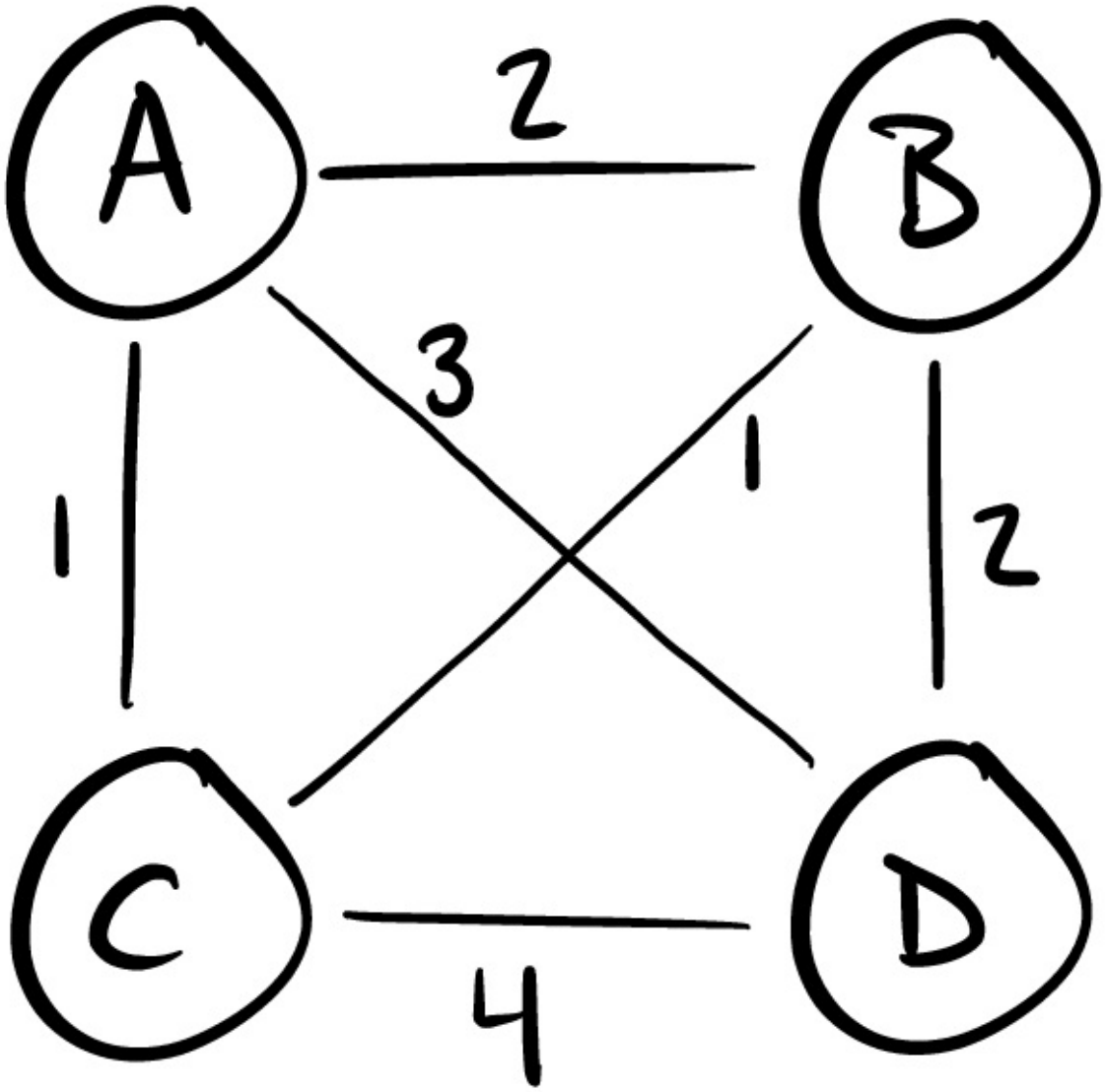
Next, substituting our values of a, b, and $f(n)$ into $af(n/b) <= kf(n)$, we have $2(n/2)^2 <= kn^2$ for $k < 1$ and sufficiently large n. Simplified, we have $n^2/2 <= kn^2$, and thus with $k = 1/2$, the expression holds for all n.

As both expressions are satisfied, we have $T(n) \in \Theta\ (n^2)$, and thus the tight upper bound $T(n) \in O(n^2)$.

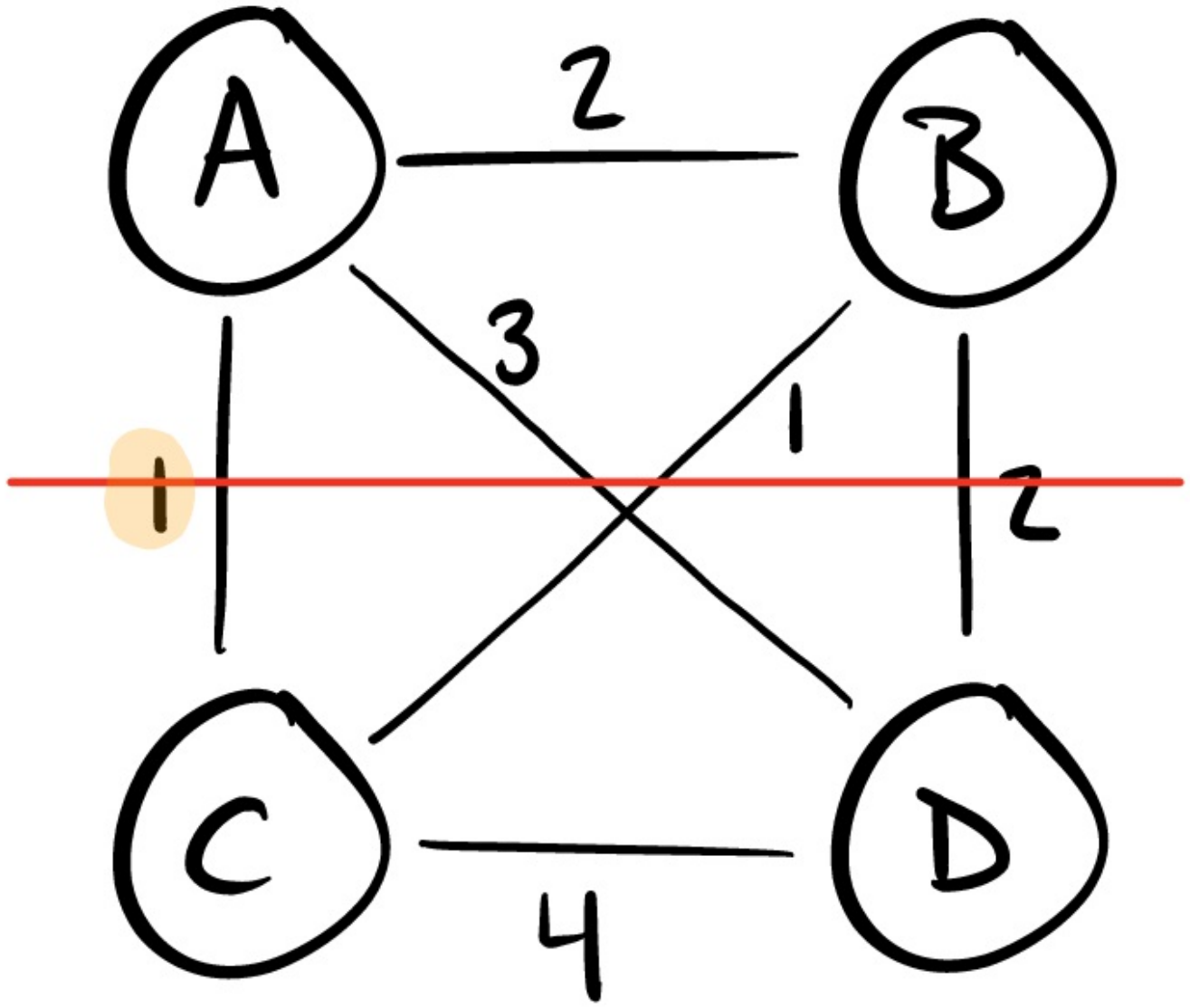
3

This algorithm does not always generate a minimum spanning tree. Its particular weaknesses lie in that `subgraphs()` does not strategically pick a cut, and it does not explore both outcomes in the case that there are two edges with the same minimum weight.

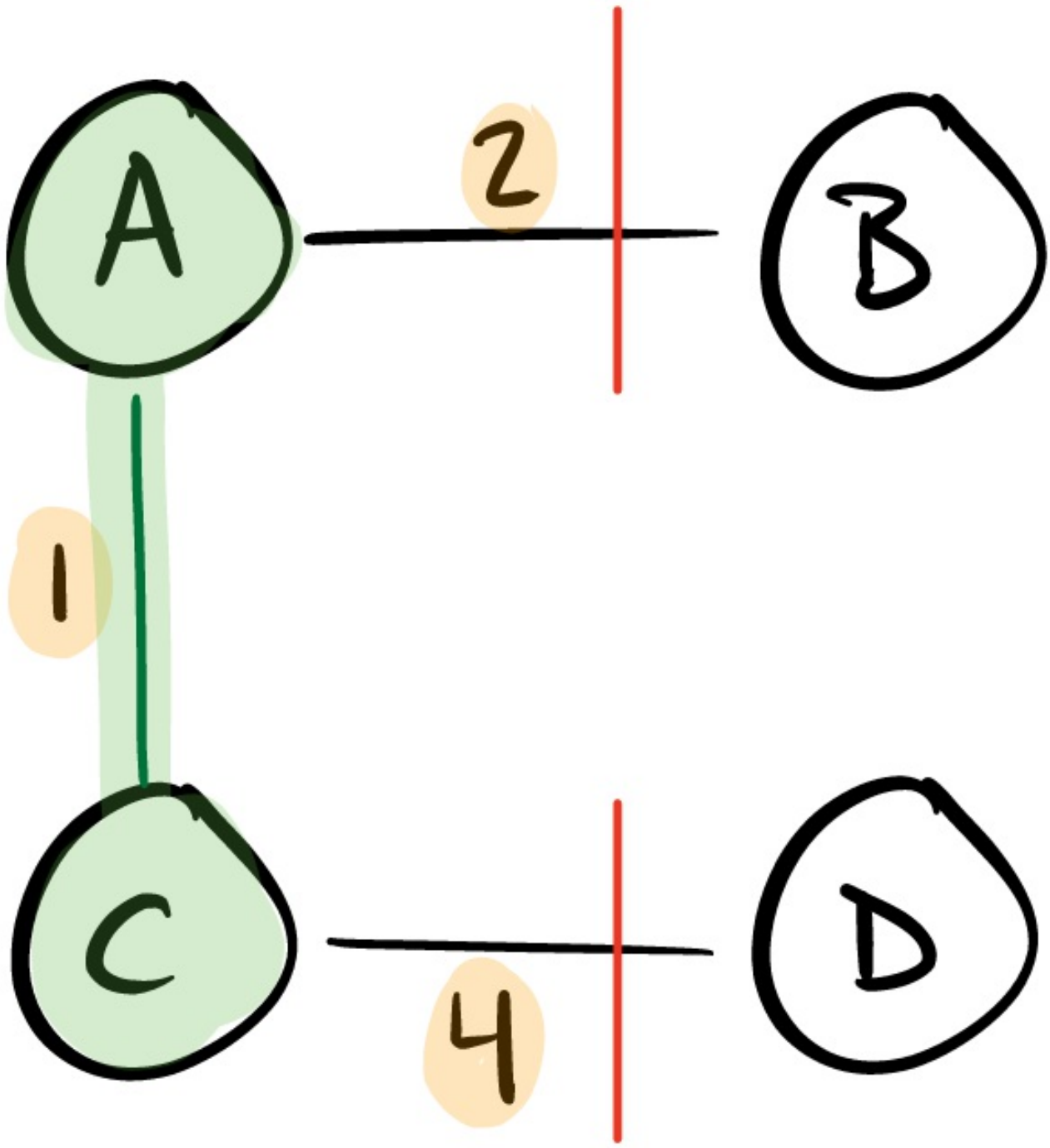
Consider the following graph G, with labelled vertices and edge weights:



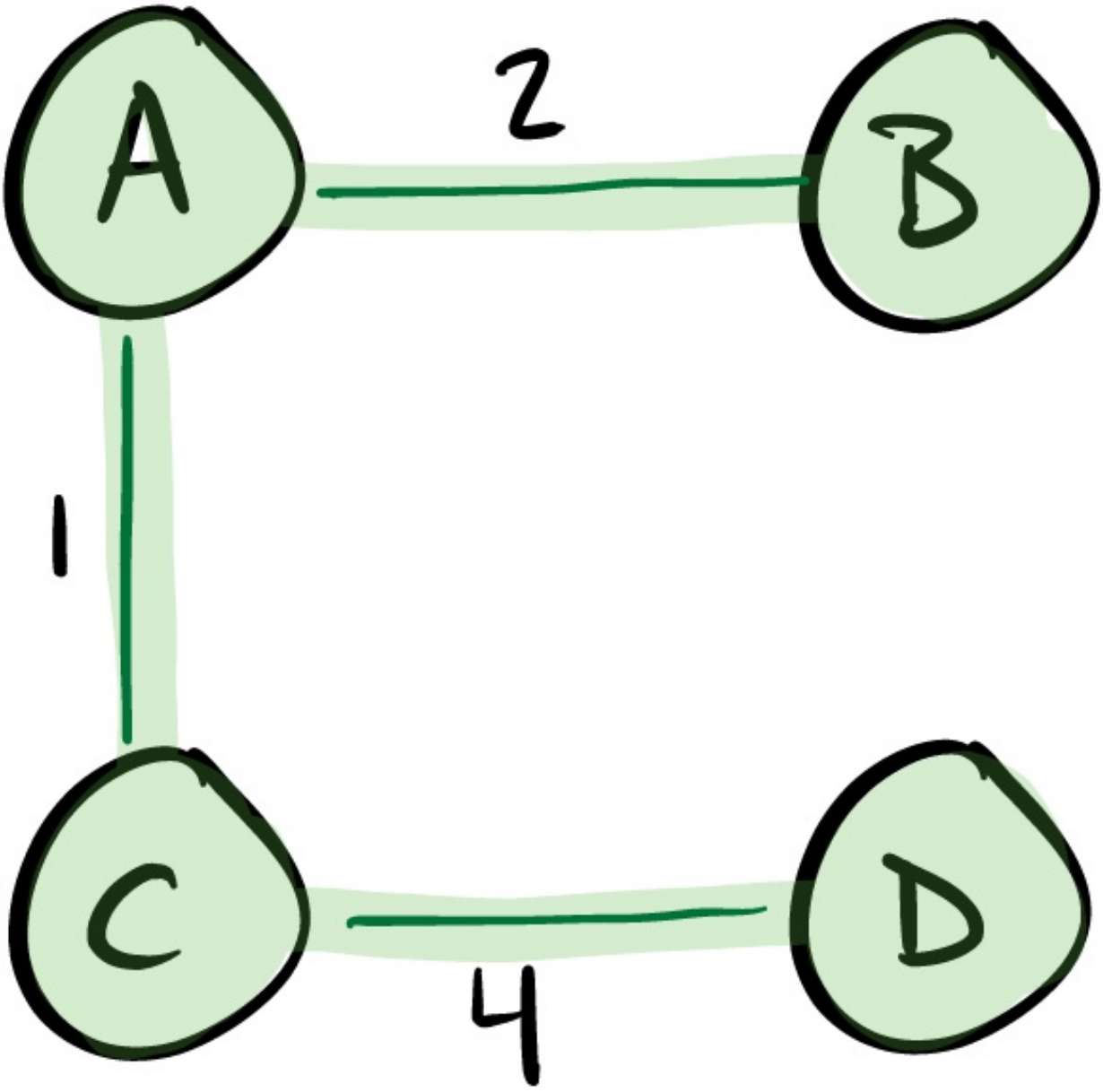
In this case, if `subgraphs(G)` partitions the graph into subgraphs H (V_H = {A, B}) and I (V_I = {C, D}) using the red-line cut, then we have two edges with minimum weight, (A, C) and (B, C). As we have no mechanism for deciding between the two, we "arbitrarily" (for the sake of argument) pick (A, C) as our minimum-weight edge, as shown below:



Now, within subgraphs H and I, we have only one edge choice each, so we choose edge (A, B) within subgraph H (weight 2) and edge (C, D) within subgraph I (weight 4):

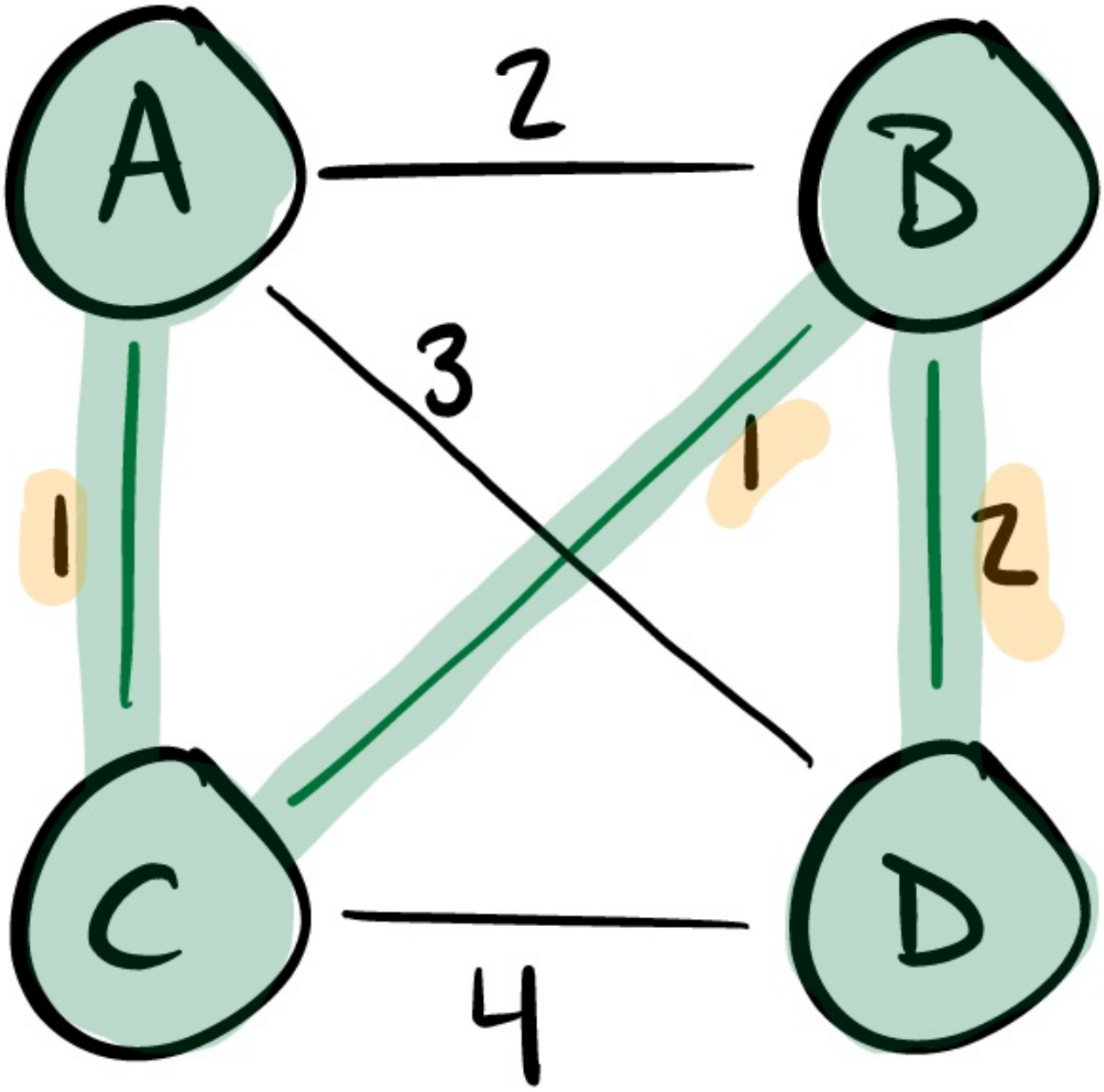


The resulting tree is shown below, with total weight 7.



This is a spanning tree, however it is not the minimum spanning tree. It is also worthy to note that had we picked edge (B, C) instead of (A, C), the result (A-B-C-D, total weight 7) would still not be the minimum spanning tree.

The minimum spanning tree in this example is shown below, with total weight 4.



This could be obtained by first cutting G into graphs J (V_J = {A}) and K (V_K = {B, C, D}), and then K into graphs L (V_L = {D}) and M (V_M = {B, C}).