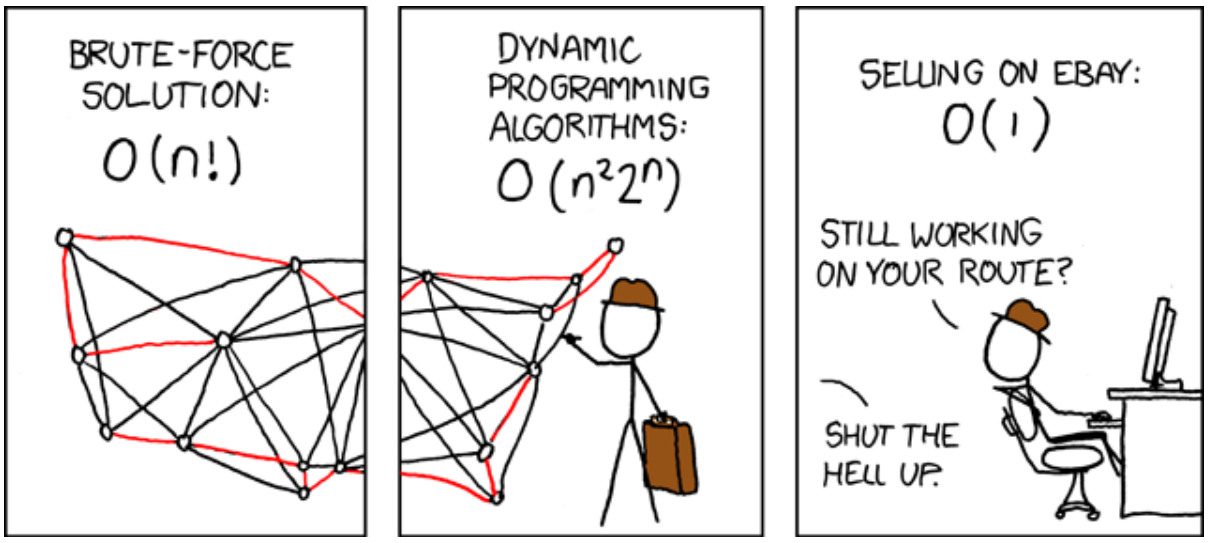
Dijkstra's Shortest Path Packet

Math-270: Discrete Mathematics
Fall of 2019

Prof. Gregory V. Bard

Contents:



Dijkstra's Algorithm for the Shortest Paths in a Graph (or Digraph)

Math-270: Discrete Mathematics

November 21, 2019

Inputs: The set of vertices, V. The weights of the edges, as a two-dimensional array, where $\texttt{weight}[i, j]$ is the weight of the edge from v_i to v_j. If no such edge exists, then denote this by $\texttt{weight}[i, j] = \infty$. Also, some $v_a \in V$ is identified as "the headquarters vertex."

Note: #V, the number of vertices is usually passed as well, though this might be obvious from the size of the set V and the dimensions of the array `weight`.

Outputs: The array `Distance`, which has the distance from each vertex to v_a. Note that $\texttt{Distance}[v_a] = 0$, and any vertices not reachable from v_a have distance ∞. Also, the array `Pred`, a list from which the shortest paths from v_a to any other vertex can be easily computed.

1. $\texttt{Distance}[v_a] \leftarrow 0$.
2. For each $v \in V - \{v_a\}$ do

   $\texttt{Distance}[v] \leftarrow \infty$.
3. $Q \leftarrow V$.
4. $D \leftarrow \{\}$.
5. While $Q \neq \{\}$ do
   (a) Select the $q \in Q$ such that $\texttt{Distance}[q]$ is the smallest.

   Note: In the first iteration, this is always $q = v_a$.

   (b) (a rarely needed step will be added here, later)
   (c) $D \leftarrow D \cup \{q\}$.
   (d) $Q \leftarrow Q - \{q\}$.
   (e) For each $v_i \in$ neighborhood(q) do
      i. $\texttt{CandidateDistance} \leftarrow \texttt{Distance}[q] + \texttt{Weight}[q, v_i]$. ⋆⋆⋆⋆⋆
      ii. If $\texttt{Distance}[v_i] > \texttt{CandidateDistance}$ then
         - $\texttt{Distance}[v_i] \leftarrow \texttt{CandidateDistance}$.
         - $\texttt{Pred}[v_i] \leftarrow q$.

Note: When while loop has terminated, we know $Q = \{\}$.

6. Return the arrays `Distance` and `Pred`.

Step 5b is given in section titled "Improving the Algorithm After You're Experienced," which also gives two other major improvements to the algorithm.

Q&A about "What's the Answer" and "Is the Answer Unique?"

Q: When this algorithm is run, what is the answer?

A: This is a real-life algorithm, so there are several outputs. "Which one is the answer" depends on the situation/application. This is not an algebra problem where you write down x=4 at the end of some equation-solving.

- **The array of distances** – This is certainly useful information, but usually you want something more than this. You cannot navigate by this, for example.
- **The predecessor array** – In some ways, this is *only* useful because it is used (in the back-tracking sub-algorithm) to generate the list of shortest paths. However, it has other advantages: in many computer languages, it is much easier to pass this simple array around rather than a much more complicated data structure. It is also much easier to grade this than the list of shortest paths, so some instructors might prefer it.
- **The list of shortest paths to each vertex** – This is the primary output. Using this, you can navigate from the origin vertex to any other vertex, confident that you are taking the shortest path to your destination.
- **The tree formed by the union of all the shortest paths** – If you take all the edges used in the shortest paths from one common origin to all possible destinations, then you get a tree. For some navigation problems, this tree would be very useful information. Ant colonies know about this aspect of shortest-path calculations, and that is why you will sometimes see "ant highways" when an ant colony has found abundant food nearby.

Q: How are the answers shown in this packet?

A: In this packet, the first set of answers will provide the predecessor array and the array of distances, as well as the shortest paths to each vertex. Next, all the spanning trees are presented by highlighting the relevant edges in light green, on the original graphs. Last, all the spanning trees are presented again, by drawing them to look like trees.

Q: How should I write the answer on an exam, a test, a quiz, or a homework assignment?

A: You have to actually read the instructions. For example, if you are asked to write the predecessor array, then you should write the predecessor array. If you are asked to make a list of shortest paths, then you should make a list of shortest paths. For this reason, make sure you know the names of all the different answer formats.

Q: Does Dijkstra's Algorithm compute the minimum-weight spanning tree?

A: No. The tree of shortest paths is a spanning tree. That means it is a tree (a connected acyclic graph) that includes all the original vertices, and it is comprised only of edges that exist in the original graph. However, there could be some other spanning tree that has a lower total weight. The last page of this packet is a counterexample to the hypothesis that the spanning tree produced by Dijkstra's algorithm is a minimum-weight spanning tree.

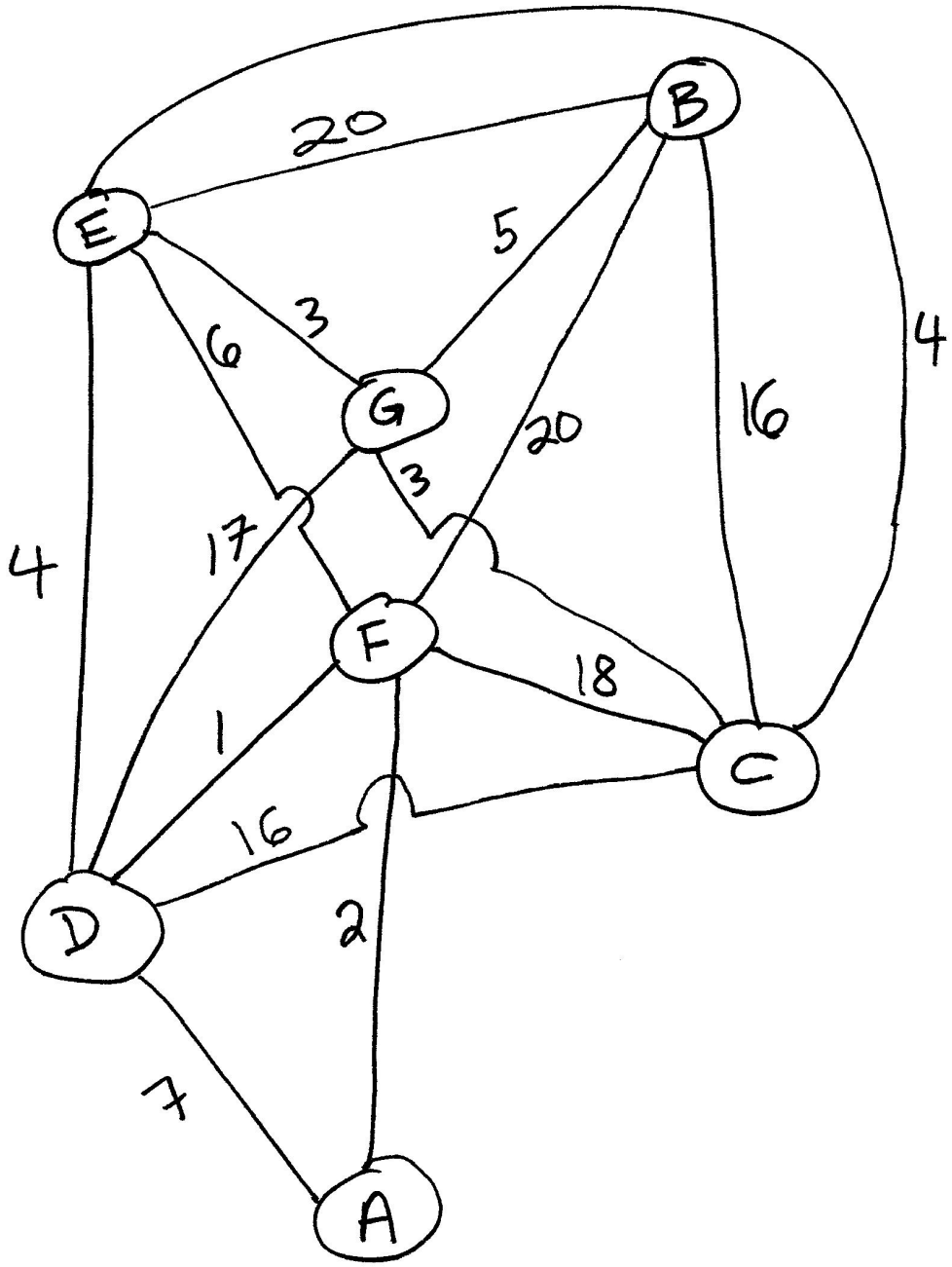
Computing a minimum-weight spanning tree is a question for which Dijkstra's algorithm is not suitable.

Q: Are the answers unique?

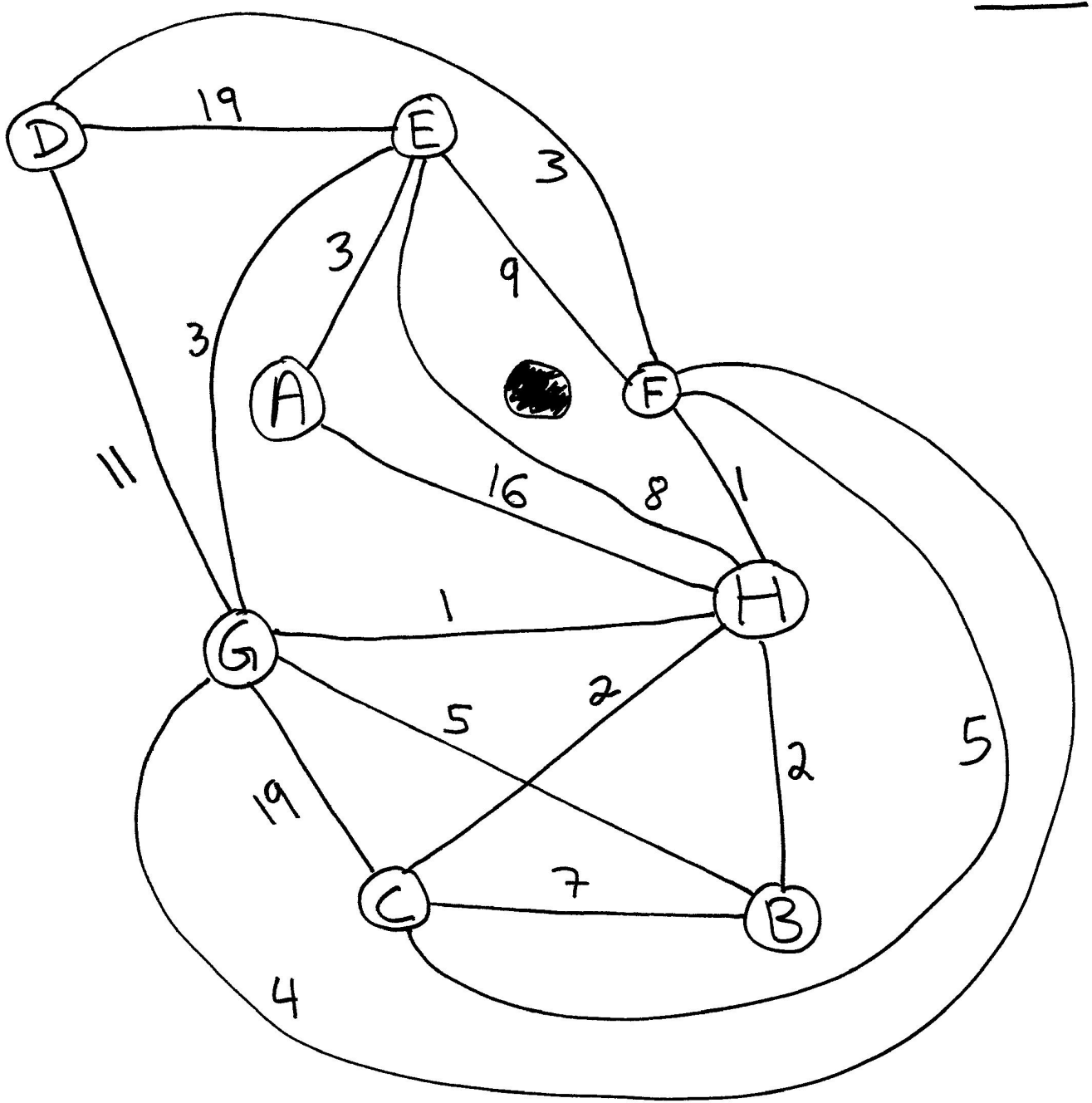
A: Questions of uniqueness (or existence & uniqueness) are favorites of pure mathematicians. As it turns out, if there are no ties in the algorithm, then all the answers are unique. By a tie, I mean the step where we "select the q from Q for which distance[q] is the smallest." If this selection never has a tie for smallest, then all the answers are unique. (By unique, we mean that there is only one correct answer, and everyone who correctly performs the algorithm will get that same answer.)

Even if there is a tie, then the array of shortest distances will always remain unique. If you and your classmate carryout the algorithm and break the ties in different ways, then you will possibly get different paths to some vertices, but the distances along those paths are the same. The spanning trees might also be different.

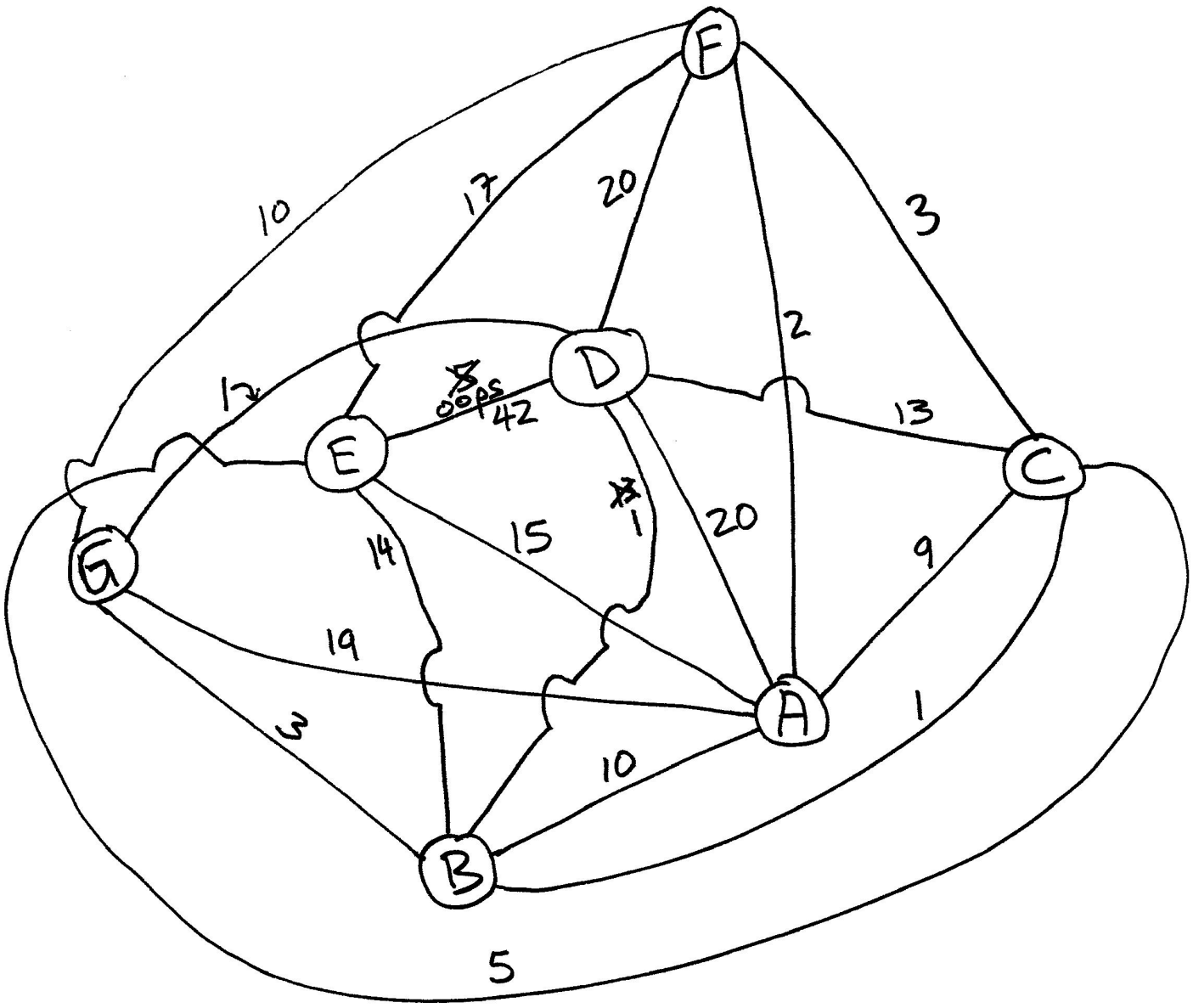
Ex 1:



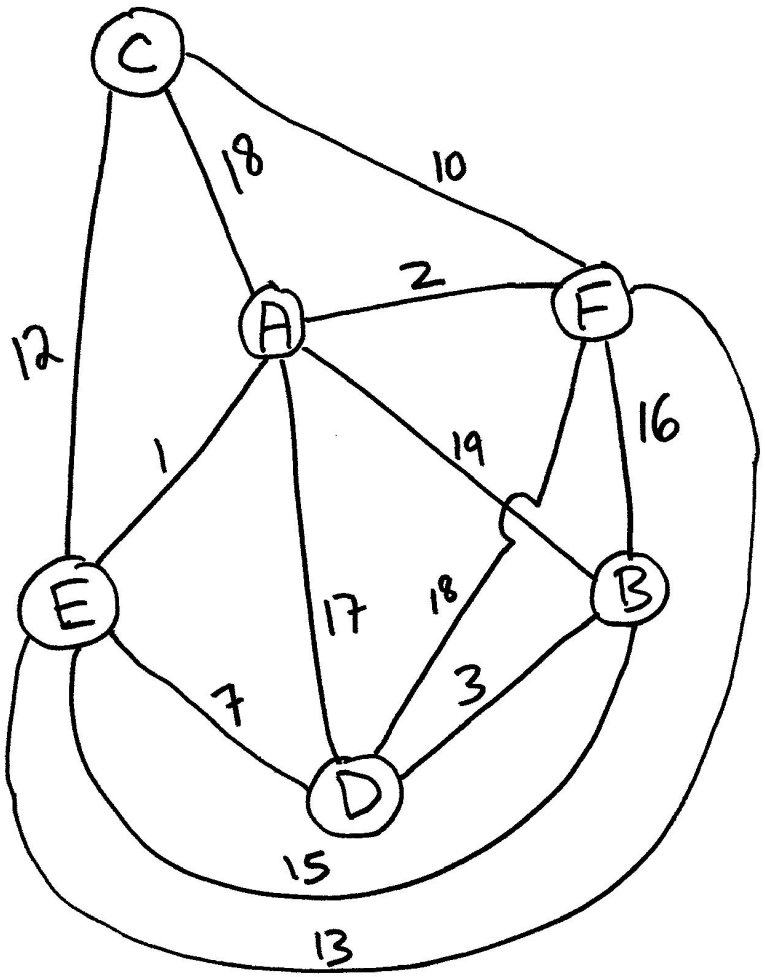
Ex2



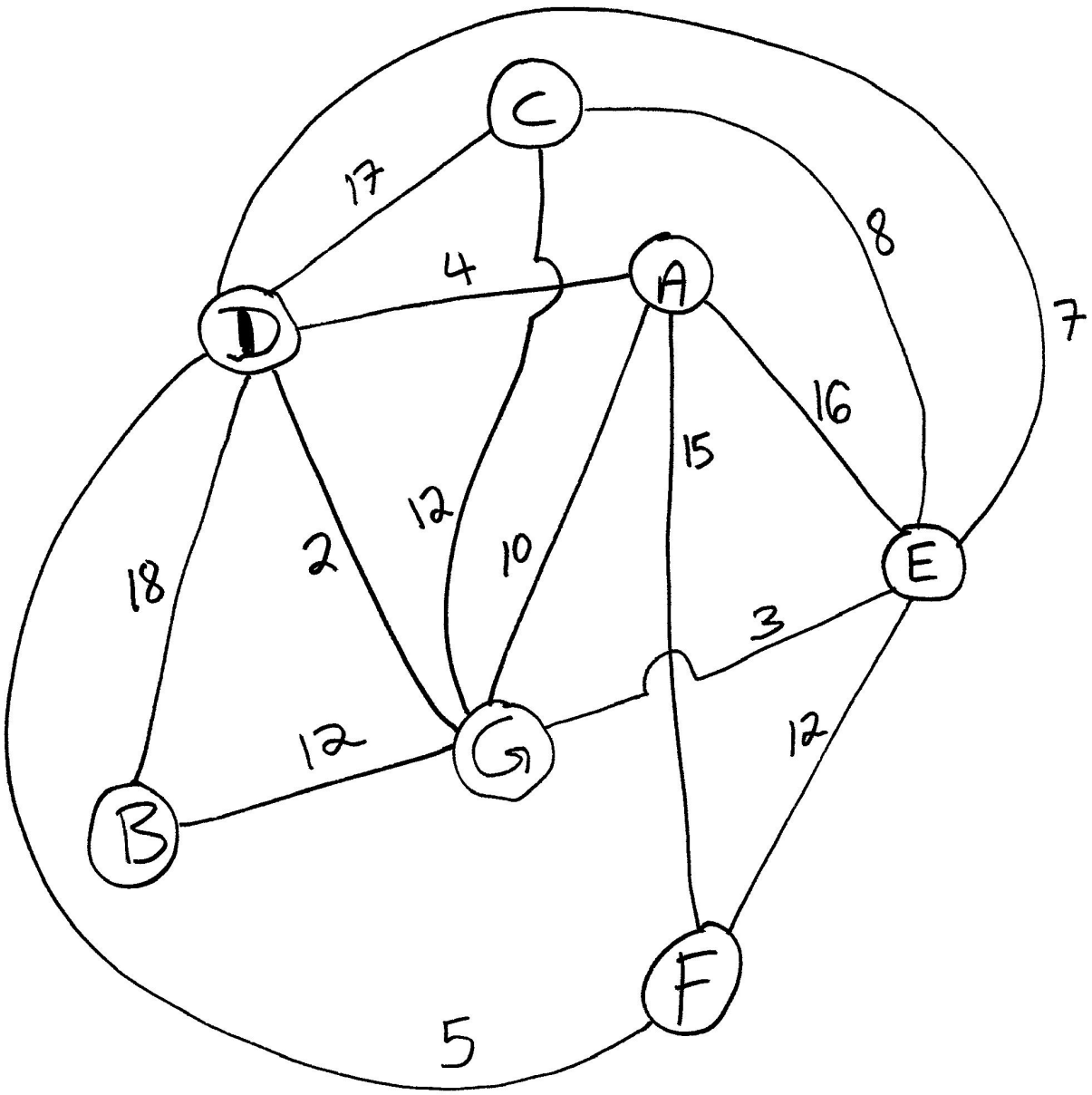
Ex3



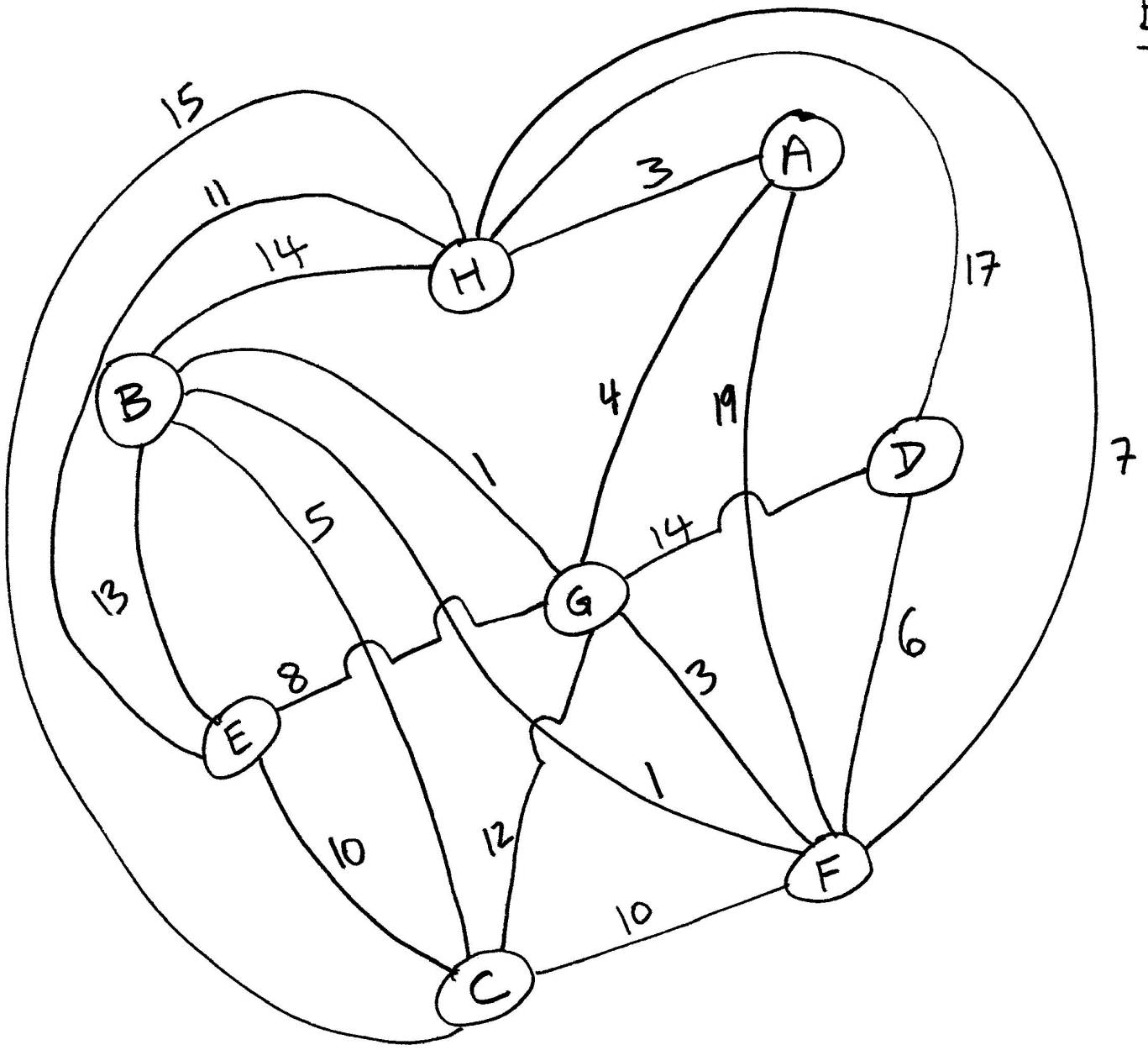
Ex 4



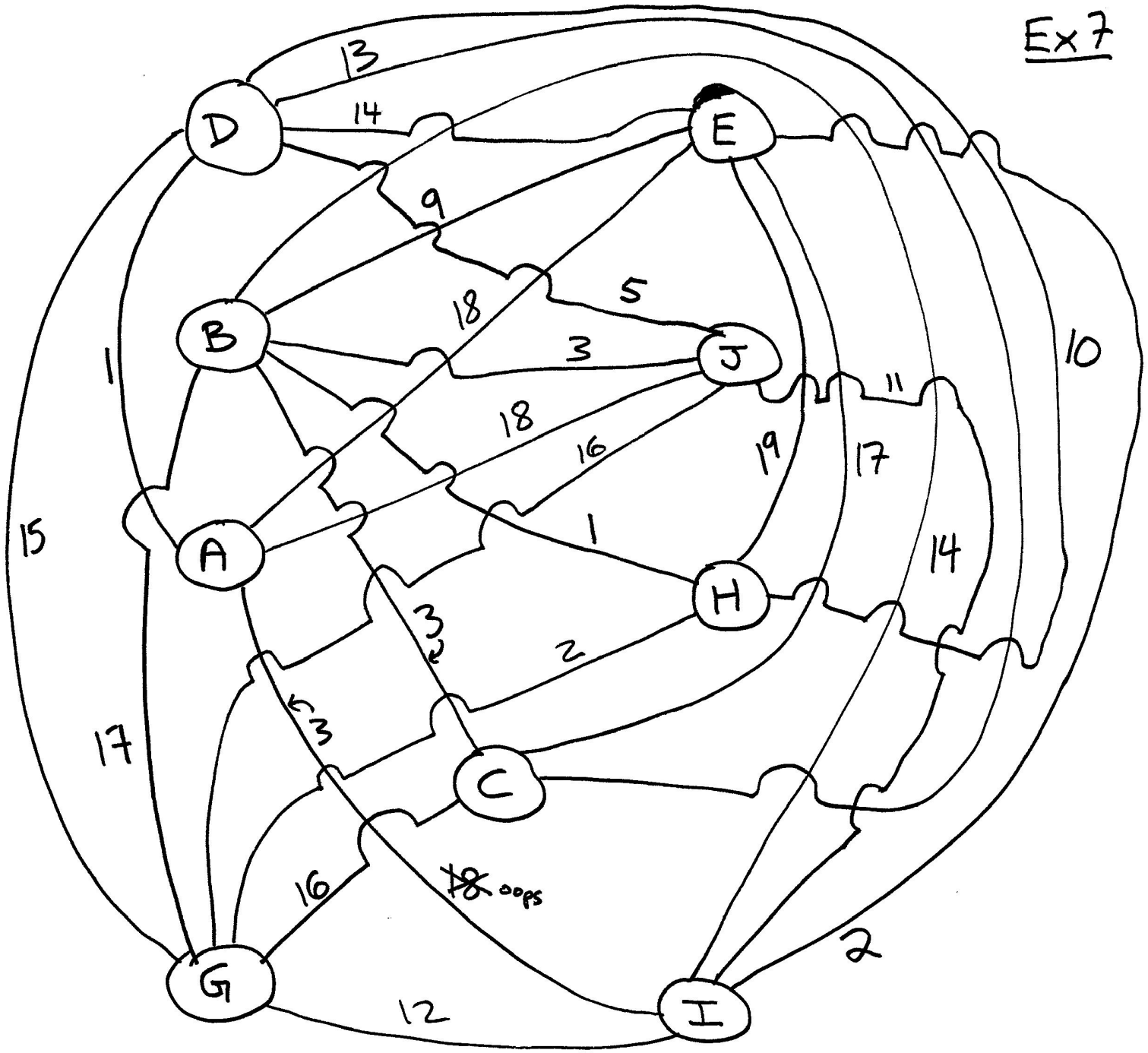
Ex5



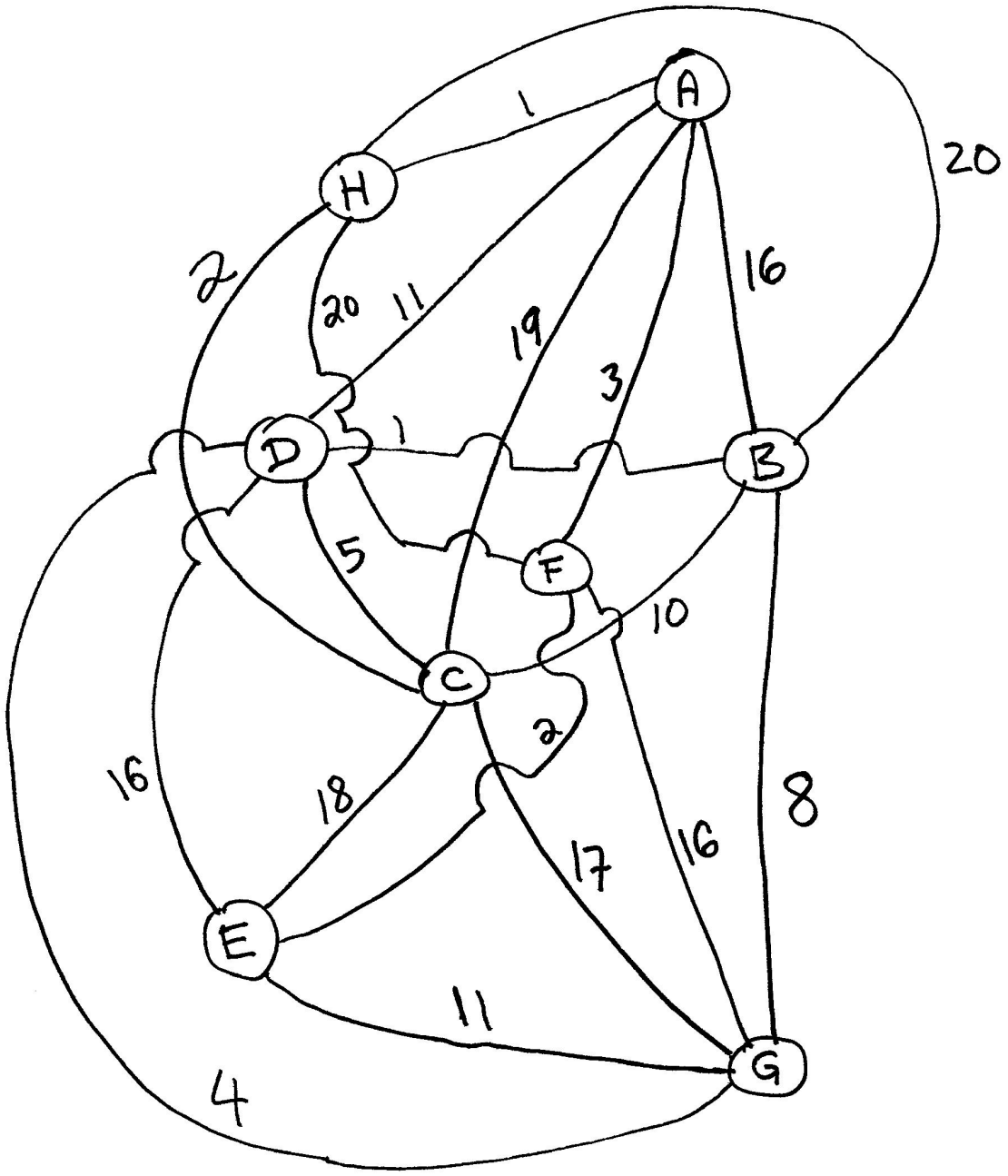
Ex6



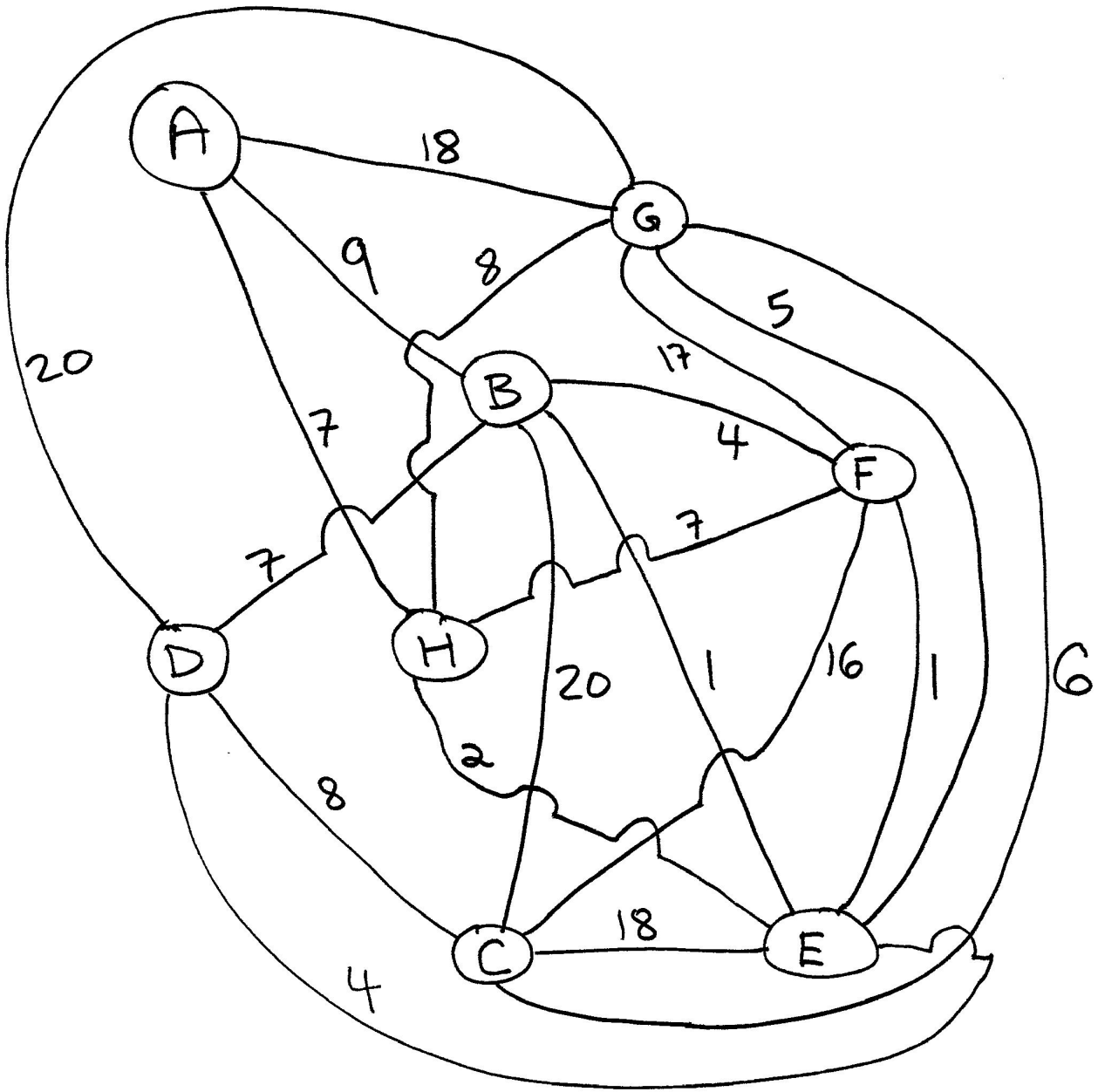
Ex 7


D
E
B
J
A
H
C
G
I
13
14
9
5
18
3
1
18
16
19
17
10
11
15
1
14
3
2
3
17
16
8 oops
2
12

Ex 8



Ex 9



A Non-Navigational Application

After Example 9, you will see Example 5A, and you might be wondering what that's about. I decided that it would be nice to have an example that is not about navigating and finding the shortest distance.

This problem represents an air-freight problem. The company, headquartered in Atlanta, has customers throughout the upper Midwest, and also in Dallas. Because airfares change suddenly and without notice, their software would have to run the Dijkstra Shortest Path algorithm extremely often---every time a fare changes.

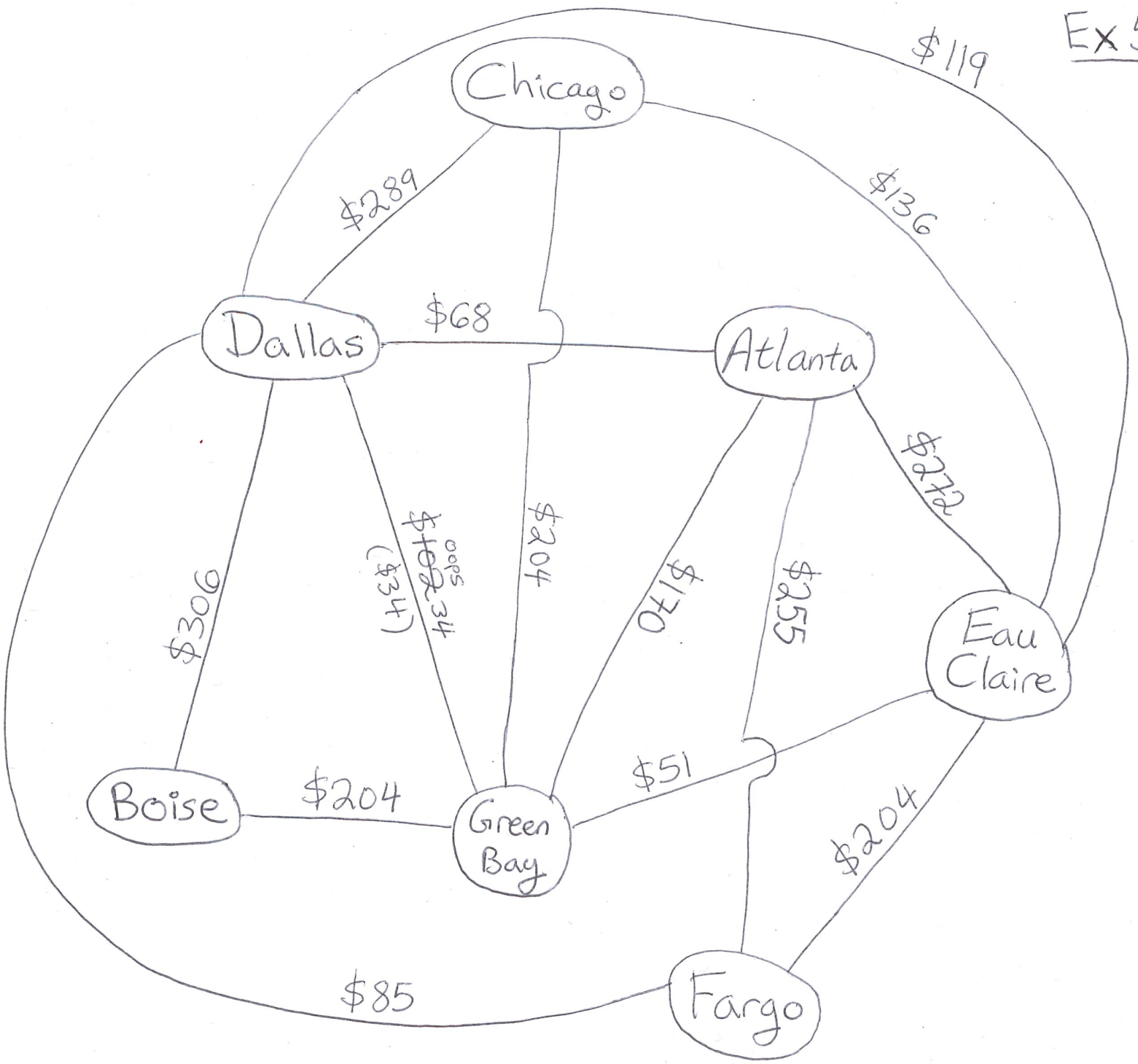
In this problem, an extremely "good deal" has come up. The fare from Dallas to Green Bay is far cheaper than expected. It is roughly 1/9 the cost of Dallas to Boise, even though the distance is roughly comparable. When you run the algorithm, you will see that this will transform Green Bay into a virtual hub for the company. All packages going to Boise, Chicago, Eau Claire, or Green Bay will be sent to or through Green Bay, at least as long as this bargain lasts. (That's 4 out of 6 possible destinations!)

Humans are not so good at realizing all the ramifications of one price change in a network of prices. That's the advantage of coding an algorithm, like Dijkstra's algorithm, that is provably correct.

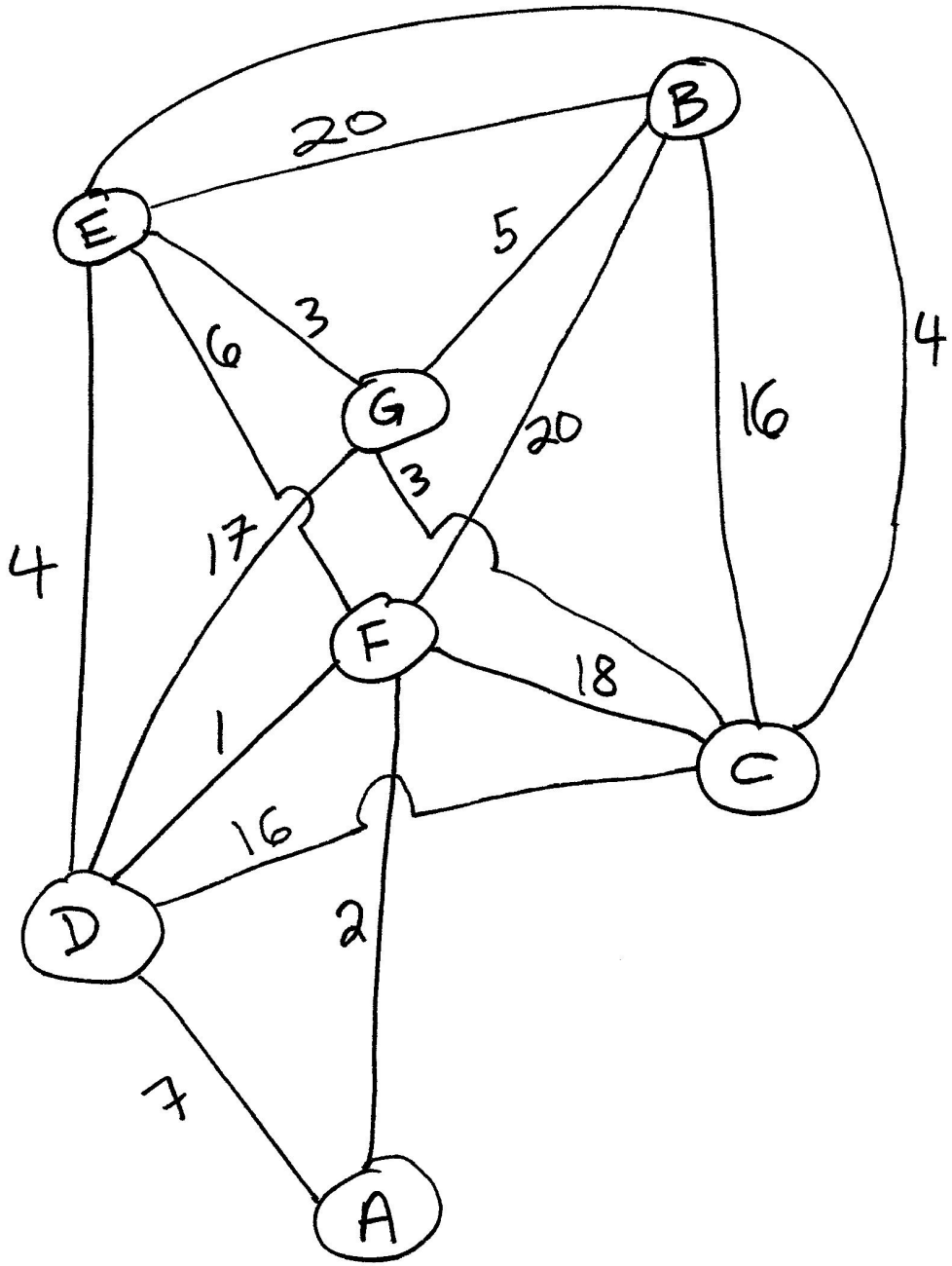
It should be noted that it is important that this is air freight and not the flight of human passengers. Usually a human would prefer a flight with 0 or 1 stops to a flight with 3 or 4 stops. If I have a possible itinerary where I have to change planes 3 times, but it is $200 cheaper than a direct flight, for sure I will choose the direct flight. That preference is hard to model mathematically, but with air freight, we would just want to know the cheapest way of sending a box of cargo from one point to another. The box of cargo does not get the chance to provide us with preferences about its flight path.

Last but not least, you might be wondering how I came up with the graph and the weights. As it turns out, I just took Example 5, and renamed the vertices. The weights are simply the weights from Example 5, but multiplied by $17 to make them look like realistic costs. This was done to emphasize that Dijkstra's algorithm solves this problem without modification. You could code an air-freight routing system for this company using what you are learning about Dijkstra's algorithm, except that a real company would have 50-150 airports that it is servicing.

Ex 5A


Chicago
$119
$289
$136
Dallas
$68
Atlanta
$306
$102 oops ($34)
$204
$170
$255
$272
Eau Claire
Boise
$204
Green Bay
$51
$204
$85
Fargo

Ex 1:



Improving the Algorithm After You're Experienced

Here are three improvements in the algorithm. Each of them alone, for some applications, will vastly improve the algorithm's performance by reducing the running time.

Which vertices in the neighborhood get updated?

Eventually, after you do a few practice problems, you will notice that whenever we consider each $v_i \in$ neighborhood(q), we never will update `Distance` or `Pred` when $v_i \in D \cap$ neighborhood(q), but we will often update `Distance` and `Pred` when $v_i \in Q \cap$ neighborhood(q). Therefore, we can cut the running time in half by replacing "$v_i \in$ neighborhood(q)" in Step 5e with "$v_i \in Q \cap$ neighborhood(q)" instead.

The list Q is the queue of vertices waiting to be processed. The list D is the "done list," which means it is the list of vertices that have been processed. Once a vertex q moves from the queue into the "done list," we know that we've found the shortest path to q. That's why it will never be updated again.

What if we only care about a few destinations?

Sometimes a user only wants to know the shortest path to one destination vertex, or a small set of destination vertices, instead of all possible destinations. Call the set of desired destinations W, for "where we want to go." If the graph is large and W is small, then we should terminate the algorithm early, after the shortest paths to all the vertices in W are known.

This can be accomplished because of the previous realization: once a vertex q moves from the queue into the "done list," we know that we've found the shortest path to q. To accommodate a destination set W, we must add W as an input to the algorithm. Instead of terminating the while loop when Q is empty, we should terminate the while loop when $Q \cap W$ is empty. At that point, the shortest paths from v_a to all vertices in W will be known, as well as their lengths. To modify the pseudocode, we should replace "While $Q \neq \{\}$ do" with "While $Q \cap W \neq \{\}$ do," in Step 5.

If V is much bigger than W, and if the furthest vertex in W from v_a is "not too far away" from v_a, then this could be a huge savings in computation. Note that for some applications, such as automobile navigation systems, it is probable that W consists of only a single vertex.

What happens on a disconnected graph?

Recall that Step 5b says "a rarely needed step will be added here, later." If `Distance`$[q] = \infty$ at that spot, then `Distance`$[v] = \infty$ for all $v \in Q$. That's because q was chosen as the member of Q such that `Distance`$[q]$ is the smallest.

This means that the original graph is disconnected. All the vertices in Q at this point are unreachable from v_a. Moreover, D is the connected component containing v_a. Viewing D as a subgraph, you've solved the problem on D, which means we know the shortest paths from v_a to all vertices in D, and their lengths. However, you can't reach any of the vertices in $Q = V - D$ from v_a. This implies that nothing additional will be learned by any later stages of the algorithm. Depending on the application, we can report an error message, throw an exception, or just return the outputs without saying anything.

Therefore, Step 5b should be...

> (5b) if `Distance`$[q] = \infty$ then the graph is disconnected. Viewing D as a subgraph, you've solved the problem on D, and you can't reach any of the vertices in $Q = V - D$ at all. Either report an error message, throw an exception, or terminate and return the outputs.

I'm sure that you can imagine an early stage of Google Maps, with all the road data for the UK, Canada, Australia and the USA. What this change will do, for someone who runs Dijkstra's Algorithm with v_a in Australia or the UK, is that the algorithm will terminate once they've processed the island containing v_a. There's no need for them to waste an enormous amount of computing time on having the algorithm read all vertices (all the road intersections) throughout the USA and Canada, because they are not relevant to the user. Therefore, the early termination would save them a ton of processor time.