CSCI 3104 Spring 2017 Problem Set 7
Benton, Gregory (09/20)
Chen, Amy (06/20)
Sidrow, Evan (05/30)
References: CSCI3104 lecture notes, CLRS textbook, and APPM3170 (Discrete Mathematics) notes.

---

1. *Hermione needs your help with her wizardly homework. She's trying to come up with an example of a directed $G = (V, E)$, a source vertex $s \in V$ and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running a breadth-first search on G, no matter how the vertices are ordered in each adjacency list. Include an explanation of why your example satisfies the requirements.*

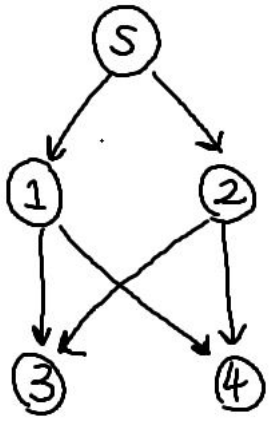
   First, recall how breadth-first search works. Simplistically (without getting into the black-grey-white method), once the breadth-first search discovers a node it will not uncover another tree edge to that nodes. Therefore, we constructed a graph that has multiple paths of the same length to the ending nodes (nodes that have an out-degree of zero) and a solution can be found when breadth-first search can return two trees with different edges by creating E_π from edges that are only found in different searches. Remembering that breadth-first search is first-in-last-out from the queue, we are able to find a set of edges E_π that cannot be produced by breadth-first search regardless of the order of the vertices in the adjacency list.

   The directed graph that fulfills these requirements is:

$$V = s, 1, 2, 3, 4$$

$$E = (s, 1), (s, 2), (1, 3), (1, 4), (2, 3), (2, 4)$$

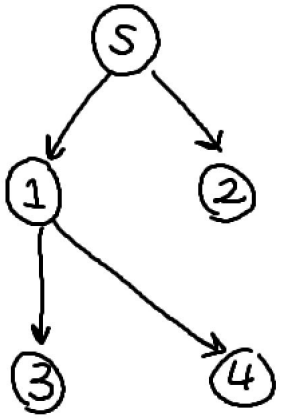
   and looks like this:



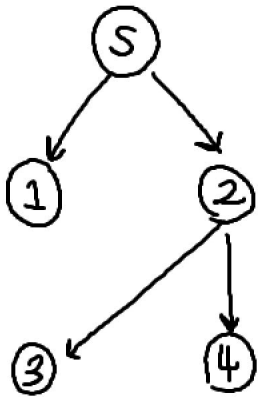
The adjacency list looks like this:
s: 1, 2 (or 2, 1)
1: 3, 4 (or 4, 3)
2: 3, 4 (or 4, 3)
3: -
4: -

Note that when we run breadth-first search on the tree, only the order that 1 and 2 are put in the queue matter (since discovering either will then discover 3 and 4, so the order of 3 and 4 in the adjacency list is arbitrary). Thus, the breadth-first search will create a tree in one of two ways:

(a) If 1 comes before 2 in the adjacency list, then 1 will leave the queue first and discover 3 and 4. Thus, we will have edges $E_1 = (s, 1), (s, 2), (1, 3), (1, 4)$.



(b) If 2 comes before 1 in the adjacency list, then 2 will leave the queue first and discover 3 and 4. Thus, we will have edges $E_2 = (s,1),(s,2),(2,3),(2,4)$.



Therefore, a breadth-first search of $G = (V, E)$ where

$$V = s, 1, 2, 3, 4$$

$$E = (s,1),(s,2),(1,3),(1,4),(2,3),(2,4)$$

can only produce one of two sets of edges:

$$E_1 = (s,1),(s,2),(1,3),(1,4)$$

$$E_2 = (s,1),(s,2),(2,3),(2,4)$$

Thus, we can construct E_π by taking one of each the possible edges to 3 and 4 that can be created in each scenario, guaranteeing that a breadth-first search will never return the set of edges:

$$E_\pi = (s, 1), (s, 2), (1, 3), (2, 4)$$

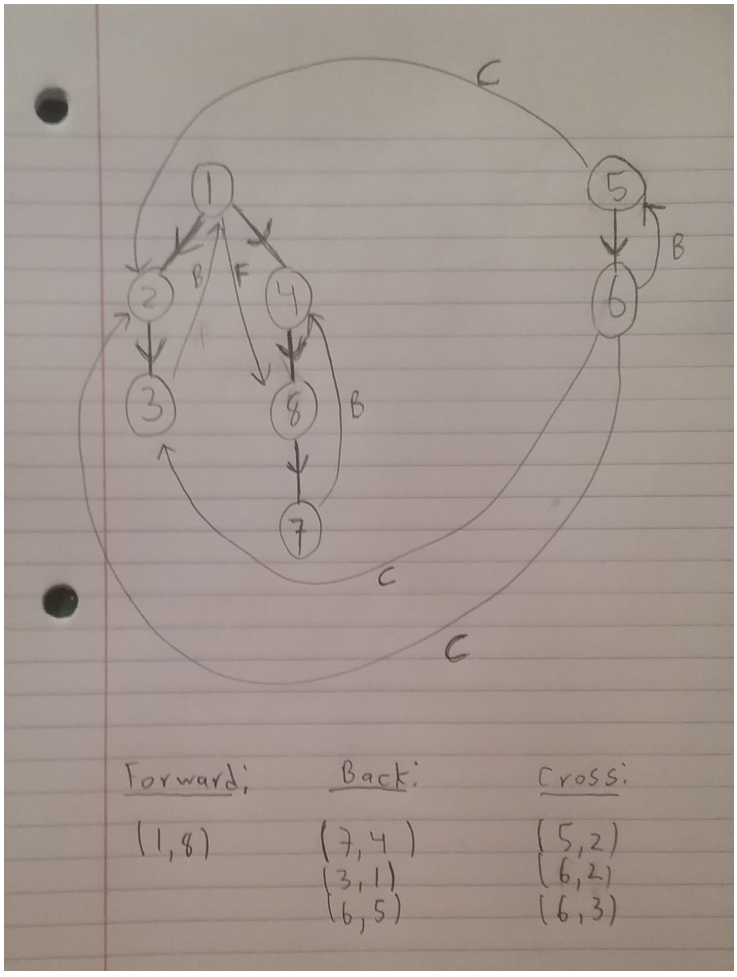
because if 1 comes before 2 in the adjacency list, the edges to 3 and 4 must both come from 1 and if 2 comes before 1 in the adjacency list, the edges to 3 and 4 must both come from 2. Breadth-first search does not allow for the edges to 3 ad 4 to come from different vertices by construction of the original graph G.

2. *Review the material on depth-first spanning forests in Chapter 22.3 of the textbook. Then, consider the directed graph G defined by the edge list*

$$G = (1,2), (1,4), (1,8), (2,3), (3,1), (4,8), (5,2), (5,6), (6,2), (6,3), (6,5), (7,4), (8,7)$$

(a) *Draw the depth-first spanning forest, including and identifying all tree, back, forward, and cross edges. (If you prefer, you can identify the forward, back, and cross edges in separate lists instead of trying to draw and label them.)*

After doing the depth-first search by hand and picking each of the vertices in order if a vertex has multiple neighbors (1 before 2, 2 before 3, etc.), we get the following spanning tree:



(b) *List all the strong components in G.*

The strong components of G are: {4,8,7}, {1,2,3}, and {5,6}. This is clear from the depth-first spanning tree shown above. Each of these strong components are cycles.

3. *Professor Snape claimes that when an adjacency matrix representation is used, graph algorithms take $\Omega(V^2)$ time, but there are some exceptions. One such exception, Snape claims, is determining whether a directed graph G contains a black hole, which is defined as a vertex with in-degree $|V|-1$ (i.e. every other vertex points to this one) and out-degree 0. Snape sourly claims that this determination can be made in time O(V), given an adjacency matrix for G. Prove that Snape is correct. Hint: this is a "proof by algorithm" question. That is, describe and analyze an algorithm that yields the correct answer on every type of input ("yes" when a black hole exists and "no" when it does not.) Think about boundary cases, where it is most difficult to distinguish a correct "yes" from a correct "no".*

The first thing to note is that there can only be one black hole vertex. This is because a black hole vertex must be pointed to by every other vertex, so every other vertex has an out degree of at least one, so each of these vertices cannot be a black hole. Now, we also consider what the adjacency matrix looks like when there exists a black hole. if vertex i is a black hole, then the i^{th} row of the matrix must be zeros (since the node points nowhere), and the i^{th} column of the matrix must be ones, except for the (i,i) entry, which is a zero (since the black hole doesn't point to itself), with this in mind, we implement the following algorithm:

Start by setting $i = j = 0$, assuming we index starting from zero. then, while i and j are both less than $|V|$, increment i by one if the G[i,j] == 1 and increment j by one if G[i,j] == 0.

Now, check the i^{th} row. If it contains any ones, return "no". Otherwise, check the i^{th} column. If it contains any zeros except for the i^{th} row, return "no". Otherwise, return "yes".

The psuedocode to find a "black hole" vertex for this algorithm is given below.

```
find_black_hole(G){
```

```
    i = 0
    j = 0

    while ((i < |V|) and (j < |V|)){
        if (G[i,j] >= 1){
            i = i+1
        }
        else{
            j = j+1
        }
    }

    if (i == |V|){
        return "no"
    }
    for (k = 0 to |V| - 1){
        if (G[i,k] >= 1){
            return "no"
        }
    }
    for (k = 0 to |V| - 1){
        if (G[k,i] == 0 and k ~= i){
            return "no"
        }
    }
    return "yes"
}
```

First, note that this algorithm is indeed $\mathcal{O}(|V|)$. We can see that we first run through a while loop which indexes i and j by one every time, but i and j must remain below $|V|$, so this loop will run no more than $2|V|$ times, and every line inside the loop takes $\mathcal{O}(1)$ time. Then, we run through two for loops which iterate $|V|$ times, and every line inside the loop also takes $\mathcal{O}(1)$ time.

Now, we prove the correctness of the algorithm. We prove this by considering two cases:

- **There is a black hole.** If there is a black hole, then the adjacency matrix will

look like this:

$$\begin{bmatrix} \ldots\ldots\ldots\ldots & \geq 1 & \ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots & \geq 1 & \ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots & \geq 1 & \ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ 0 \;\; 0 \;\; \ldots \;\; 0 \;\; 0 \;\; 0 \;\; \ldots \;\; 0 \;\; 0 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots & \geq 1 & \ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots & \geq 1 & \ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots & \geq 1 & \ldots\ldots\ldots\ldots \end{bmatrix}$$

Where if the k^{th} vertex is a black hole, the column of ones occurs in the k^{th} column and the row of zeros occurs in the k^{th} row.

Now, using this algorithm, either i or j will eventually equal k, since $k \leq |V| - 1$. If $j = k$ first, then the algorithm will increment i until $i = k$, since the k^{th} column of G is made up of entries greater than zero.

Once we have $i = k$, then, because the i^{th} row is made up completely of zeros, the algorithm will increment j until $j = |V|$, at which point we exit the while loop.

Now, since $i = k$, the algorithm simply checks that the i^{th} row is made up of zeros in the first for loop (which it is) and then checks that the i^{th} column is made up of non-zero entries except for entry (i, i) (which it is). Therefore, The algorithm will reach the final return statement and correctly return "yes".

- **There is not a black hole.** If there is not a black hole, then there will exist no structure in G such as the matrix shown above. This is equivalent to saying that there is no number k such that the k^{th} row is made up of zeros and the k^{th} column is made up of non-zero entries except for the (k, k) entry.

  Therefore, after the while loop, i will be some number. If it is —V—, then we correctly return "no". If it is not, then i indexes some row and column in G. We then check to see if i^{th} row is made up of zeros in the first for loop and check if the i^{th} column is made up of non-zero entries except for the (i, i) entry. However, since we already no that no such vertex exists, one of the if statements in our for loops must be satisfied, so the algorithm will enter the if statement and correctly return "no".

Since we have found that the algorithm returns the correct output both if there is and isn't a black hole, we must have that it always returns the correct output. Further, since it takes $\mathcal{O}(|V|)$ time, we have proven Snape's claim.

4. *Deep in the heart of the Hogwarts School of Witchcraft and Wizardry, there lies a magical Sphinx that demands that any challenger efficiently convert directed multigraphs into undirected simple graphs. If the wizard can correctly solve a series of arbitrary instances of this problem, the Sphinx will unlock a secret passageway.*

   Let G = (E,V) denote a directed multigraph. An undirected simple graph is a G' = (V,E') such that E' is derived from the edges in E so that (i) every directed multi-edge, e.g., (u,v), (u,v) or even simply (u,v), has been replaced by a single pair of undirected edges (u,v), (v,u) and (ii) all self-loops (u,u) have been removed. Describe and analyze an algorithm (explain how it works, give pseudocode if necessary, derive its running time and space usage, and prove its correctness) that takes O(V+E) time and space to convert G into G', and thereby will solve any of the Sphinx's questions. Assumer both G and G' are stored as adjacency lists.

   Hermione's hints: Don't assume adjacencies `Adj[u]` *are ordered in any particular way, and remember that you can add edges to the list and remove ones that you don't need.*

   One way to do this that is within the space and time complexity limits is to take the adjacency list we are given and scan through it adding each edge to a new, blank adjacency list as both a forward and backward edge (i.e. if we see an edge from a to b in the given list, add a to b and b to a to the new list). This will induce duplicates in many situations, so all we need to do then is go through our newly made list and delete duplicates. This can be done by creating a third and final adjacency list, and importing all edges from our intermediate list into the final list while avoiding duplicates.

   The duplicate avoidance problem is tricky, but there are a few clever solutions. One of these is to allocate a row vector of size $|V|$, indexed by each of the vertices in the graph. In this vector we will keep track of whether or not an edge has already been copied from our second intermediate adjacency list to the final adjacency list that has all duplicates removed. This will work by storing a flag in each index of the row vector that will indicate whether or not the edge has already been copied. By making the flag the source vertex of the edge we never need to clear out the elements in this vector, saving running time.

Pseudocode for this algorithm is as follows

```
Sphinx_Graph(G){
    // adj list to copy everything to including duplicates
    initialize intermed_adj_list of size |V|

    //adj list to copy to w/o duplicates
    initialize final_adj_list of size |V|

    // copy all edges to intermediate adjacency list
    for each vertex v in G{
        for each edge (v,u) from v{
            insert u into intermed_adj_list[v]
            insert v into intermed adj_list[u]
        }
    }

    // vector to determine if item copied already
    intialize copy_vector of size |V| to all -1's

    // copy from intermed_adj_list to final_adj_list
    for each vertex v in intermed_adj_list{
        for each edge (v,u) from v{

            if copy_vector[u] != v { // haven't copied yet
                insert u into final_adj_list[v]
                // flag that u has been copied now
                copy_vector[u] = v
            }
        }
    }

    return final_adj_list

}
```

We will now show three things: the algorithm has $O(V + E)$ spatial complexity, the algorithm has $O(V + E)$ time complexity, and that the algorithm is correct.

To show the algorithm is $O(V + E)$ in spatial complexity we need to look at all of the data structures used. There are a few simple variables used to create the loops used, but these are ultimately unimportant in the upper bound as they only contribute constants. The real cost is driven by storing the `intermed_adj_list` and `final_adj_list` variables. The `intermed_adj_list` variable is an adjacency list with $|V|$ vertices and $2|E|$ edges, since each edge from the original graph is added to this list twice, as a forward and backward edge. This contributes $V + 2E$ spatial complexity. The `final_adj_list` has $|V|$ vertices and can, at most, contain all of the edges from `intermed_adj_list`, giving a spatial complexity that is $O(V + 2E)$. So in total this algorithm uses $O(2V + 4E + c)$ space (where c is some constant) or just $O(V + E)$ space.

To show the algorithm has running time that is $O(V+E)$ we need to consider 3 things, the initializations and the two sets of nested for loops. In total the initializations account for a running time that is $O(3V)$, since each empty adjacency list initialized and the `copy_vector` variable are all of size $|V|$. The first set of nested for loops has a running time that is $O(V + E)$ as has been seen in class many times. It loops through all of the vertices and all of the edges. The second set of nested for loops has a running time that is $O(V + 2E) = O(V + E)$ since it loops through all the vertices in `intermed_adj_list` which has $|V|$ vertices and $2|E|$ edges. Thus we have running time $O(5V + 3E) = O(V + E)$.

To show this algorithm is correct we need to show that all edges in G appear in both directions in G' and that G' is free of duplicate edges. We see from the first set of for loops that each edge (v, u) in G is copied into `intermed_adj_list` as edges (v, u) and (u, v), now assuming that the method employed to copy the edges from `intermed_adj_list` to `final_adj_list` works correctly (shown below) this guarantees the first claim: each edge in G appears in G' in both the forward and backward directions.

To see that the copy routine from `intermed_adj_list` to `final_adj_list` functions correctly we need only examine the use of the `copy_vector` array. This array starts as all -1's, showing that no edges have been copied. Then as we are copying edges originating at v we will overwrite the appropriate elements in `copy_vector` to contain a v, that is if we are determining whether the edge (v, u) has been copied yet we would check `copy_vector[u]` and see if it contains a v, if not then this edge hasn't been copied, so we would copy the edge and overwrite that element of the array to contain a v, then as we continue looping through edges originating from v, we will know that

one edge from v to u has already been copied. In that way we avoid duplicates, thus ensuring that the final graph G' is indeed a simple graph.

These two properties ensure that we have correctly solved the Sphinx's problem and we are going to get to see that awesome secret passageway.

5. *Crabbe and Goyle are at it again. Crabbe things that min-heaps are for muggles and instead wants to use a max-heap to run the SSSP algorithm. He writes the following code on the whiteboard and bounces up and down with excitement, because he thinks he's going to get an algorithm named after himself.*

```
CrabbeSSSP(G, s) :
    # G is an adjacency list with n nodes and m directed edges.
    H = empty max-heap
    for each node i {
        d[i] - INF
        insert i into H with key d[i]
    }
    d[s] = 0
    insert s into H with key d[s]
    while (H is empty) {
        v = extractMin(H)
        for each edge (v,u) {
        if (v,u) is tense {
            Relax(v,u)
            insert v into H with key d[v]
        }
        }
    }
```

Hermione snorts, saying this is hardly an efficient algorithm, and besides it has one or more bugs in it (she won't say which).

(a) *Identify the bug(s) in Crabbe's algorithm, explain why they would lead to an incorrect algorithm and give the correct pseudocode.*

There are a number of issues with Crabbe's algorithm. Working our way through his code we see that the first issues come with the lack of a list of predecessors. Without this list reconstructing the shortest paths found becomes impossible, and his algorithm is useless. Thus we will need to add a list `pred()` where `pred(v)` returns the predecessor of `v` in the shortest path.

Next we note that in the loop `for each node i` we really should only be looping through the nodes `i` $\neq$ `s`. This will prevent `s` from being in teh max-heap twice. Furthermore, in this loop we don't want to add of these edges to the heap `H`, the

idea of this algorithm is to expand out from the source relaxing tense edges, if we add all of the vertices in the graph directly into the heap the main purpose of this algorithm gets broken down.

In the main while-loop of the given algorithm there is a run while `H is empty` this just doesn't make sense, and clearly needs to be `H is NOT empty`.

Nested in the while-loop is the `for all edges (v,u)` loop. This functions correctly except that when the edge `(v, u)` is relaxed, it is not `v` that needs to be inserted into the max-heap, but vertex `u` since this is the vertex to which the path distances could now be different. Thus the max-heap insertion needs to be altered in Crabbe's algorithm.

The corrected algorithm is

```
CrabbeSSSP(G, s) :
    # G is an adjacency list with n nodes and m directed edges.
    H = empty max-heap
    for each node i != s {
        d[i] = INF
        pred[i] = NULL
    }
    d[s] = 0
    pred[s] = NULL
    insert s into H with key d[s]
    while (H not empty) {
        v = extractMin(H)
        for each edge (v,u) {
        if (v,u) is tense {
            Relax(v,u) // Assume pred[] and d[u] are updated here
            insert u into H with key d[u]
        }
        }
    }
```

(b) *Analyze the corrected version of Crabbe's algorithm to determine its running time, and comment on why Crabbe will probably no get an algorithm named after him.*

This is the same as Dijkstra's algorithm, except a max-heap is used instead of a min-heap. Working off of the running time for Dijkstra's, which has running time $O((V+E)\log V)$. The $\log V$ term comes from the cost of extracting the minimum

element from the min-heap. In Crabbe's algorithm a max-heap is employed, and since there is no way of guaranteeing the location of the minimum element in a max-heap we will need to search the entire heap each time `extractMin(H)` is called. The max-heap could theoretically contain each vertex, thus an upper bound on the cost of this operation is now $O(V)$, giving the final running time of `CrabbeSSSP` to be $O((V+E)V)$ which is much worse than using an algorithm such as Dijkstra's, so it is very unlikely that this one will go down in the textbooks.

6. *Professor Snape has provided the young wizard Hermione with three magical batteries whose sizes are 12, 7, and 6 morts respectively. (A mort is a unit of wizard energy.) The 7-mort and 6-mort batteries are fully charged (containing 7 morts and 6 morts of energy, respectively), while the 12-mort battery is empty, with 0 morts. Snape says that Hermione is only allowed to use, repeatedly, if necessary, the mort transfer spell when working with these batteries. This spell transfers all the morts in one battery to another battery and it halts the transfer when either the source battery has no morts remaining or when the destination battery is fully charged.*

   Snape condescendingly challenges Hermione to determine whether there exists a sequence of mort-transfer spells that leaves exactly 2 morts either in the 7-mort or in the 6-mort battery.

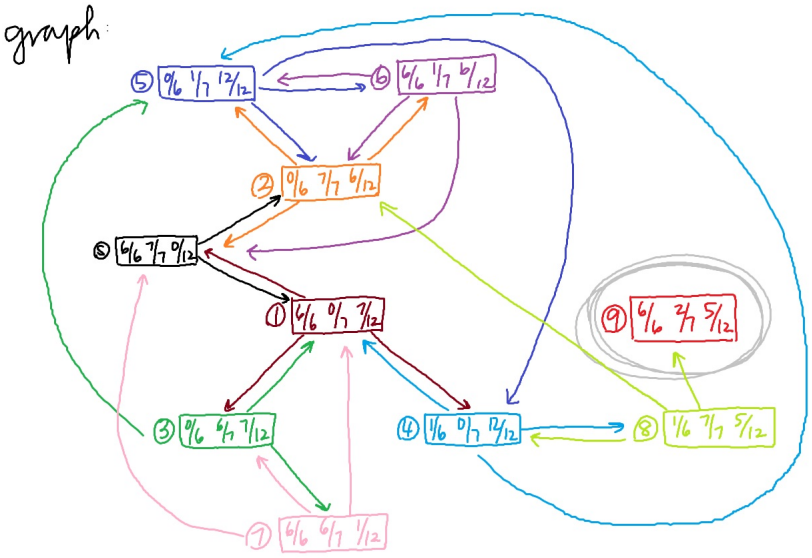
   (a) *Hermione knows that this is actually a graph problem. Give a precise definition of how to model this problem as a graph and state the specific question about this graph that must be answered.*

   This problem can be solved by a directed graph paired with a four-column array. The graph itself has vertices that carry three values: $\frac{x}{6}, \frac{y}{7}, \frac{z}{12}$, where x, y, and z represent the number of morts in the 6-mort, 7-mort, and 12-mort batteries, respectively. At each vertex, we can create a directed edge to a new vertex if the transfer spell can be used on any of the batteries (we continue transferring if none of the batteries have 2 morts in them AND there exists a battery that is not completely full; since there are only 13 morts to start with and we have a capacity of 25 morts, the graph terminates when we find a battery with 2 morts in it or we have exhausted all options and no new unique vertices can be made). Note that every vertex should have an out-degree of three (except for the source vertex, which has an out-degree of two) unless we have already found a way to isolate two morts since there are always three possible transfers that can be made. There should be as many edges as needed to exhaust all possibilities of the use of the transfer spell. Our source vertex is $(\frac{6}{6}, \frac{7}{7}, \frac{0}{12})$.

   The four-column array with coordinates (i, x, y, z) keeps track of whether or not the combination of energy levels in the batteries have already appeared in the graph. Here, i is the index of the vertex with the battery levels of x, y, z. If the combination exists in an existing vertex, the edge created should route to the

existing vertex holding those values. This creates a cycle, since we know all the possibilities that can emerge from that combination of mort levels in the batteries. Each time we want to create a vertex, we search the array for the values of x, y, and z for the vertex we want to create. If the values already exist, the graph creates the edge to the existing node. If not, we then store the values of x, y, and z in the array and index the new vertex with $max(i) + 1$. This keeps track of what combinations we have and have not seen, allow us to know which vertex we need to create a cycle to if necessary, and allows us to ensure we have exhausted all possibilities of the transfer spell.

Therefore, the graph and the array look like this:



(b) *What algorithm should Hermione apply to solve the graph problem?*

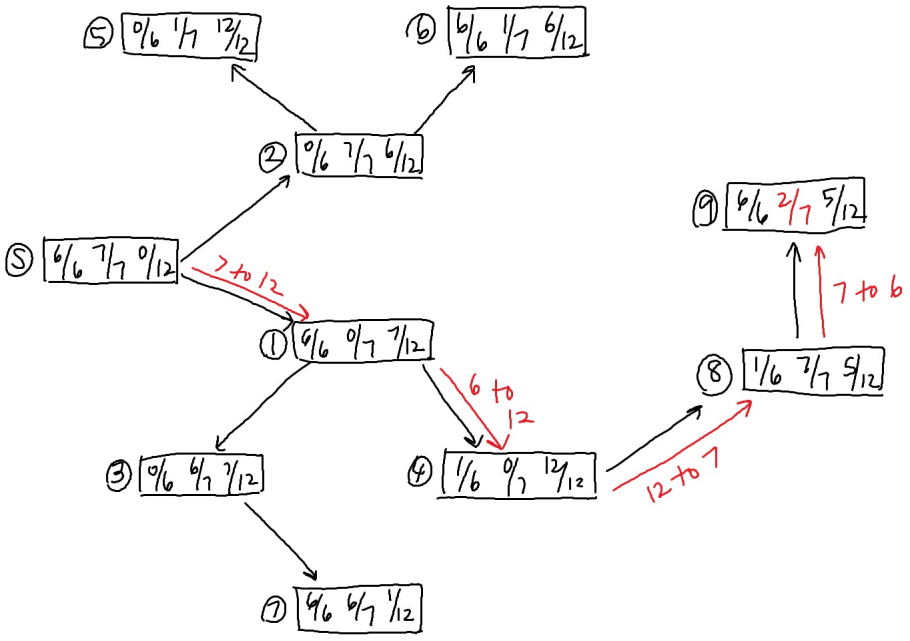
Hermione should use breadth-first search to solve this graph problem. The search should terminate when there are no new vertices in the queue or when one of the vertices has a value of $x = 2, y = 2$, or $z = 2$, whichever comes first.

(c) *Apply that algorithm to Snape's question. Report and justify the answer.*

Applying breadth-first search to the graph above, we discover through the first-in-last-out queue (where the adjacency matrix has vertices in ascending order) the vertices in this order:

$$S, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

where vertex 9 contains the desired level of two morts. The graph discovered by breadth-first search looks like this:



Therefore, there is a sequence of mort transfer spells that leaves exactly 2 morts in the 7-mort battery. The sequence is:
Source: $\frac{6}{6}, \frac{7}{7}, \frac{0}{12}$
Now, transfer from 7-mort battery to 12-mort battery.
1: $\frac{6}{6}, \frac{0}{7}, \frac{7}{12}$
Now, transfer from 6-mort battery to 12-mort battery.
2: $\frac{1}{6}, \frac{0}{7}, \frac{12}{12}$
Now, transfer from 12-mort battery to 7-mort battery.

3: $\frac{1}{6}, \frac{7}{7}, \frac{5}{12}$
Now, transfer from 7-mort battery to 6-mort battery.
4: $\frac{6}{6}, \frac{2}{7}, \frac{5}{12}$
The 7-mort battery now has 2 morts in it and the problem is solved.