Global Note:
In each of the files I included a global variable "DEBUG" set to false by default.
If this is made true, all of my print statements I used to track the process through
testing will be displayed

I also apologize for the formatting, I did them as .txt files so I could edit them in mimir whilst I was writing the code, so that's what the awkward spacing is all about. I fixed it as best I could.

Pathfinder Write up
CPSC 2120
Lake Summers

I implemented a similar BFS to the one that I used when implementing ford fulkerson.
It wasn't too difficult to figure out how it can apply so it was nice to be able to see how writing the pseudocode I did for that BFS helped a LOT here.
I used BFS because although it is not the fastest, it does return the shortest path.
I also thought of the following strategy for the bonus of avoiding bombs:
Treat bombs like walls (mark them visited) on the first run so that the BFS will avoid them and still find the shortest path possible.
If the first run fails, treat bombs like empties (mark them unvisited at start) and find the shortest path possible through the bombs.
Tradeoff of this bomb-avoidance method:
Although it is great at avoiding all bombs, if the BFS fails and it runs treating bombs as empties, it could hit unnecessary bombs (any besides the one bomb that originally
prevented it from reaching the finish) when taking the shortest path
References:
Same video as in supplyChain which spoke about ford fulkerson and a BFS:
https://www.youtube.com/watch?v=GoVjOT30xwo

Inserting at beginning of vector:
https://www.geeksforgeeks.org/vector-insert-function-in-c-stl/

Runtime complexity:
O(n^2) because it runs through the whole array and is called multiple times.
However, the coefficient does not matter.
This may not be exact, but at least big O covers anything less

Space Complexity:
O(n^2) also, because I am storing a bool for each index in a map
This may not be exact, but at least big O covers anything less

Here is the breakdown of how I used the BFS to find the shortest path
(this includes avoiding bombs if possible):
Global variables:
map of index to parentIndex
map<pair<int,int>,pair<int,int>> parent

Functions:
ALL CHECKS USED IN BFS

checkLeft

if index is on left wall (currentIndex.second ==0)
return (-1,-1) (could be any pair of indexes outside of the 2D array)
else
return currentIndex with -1 off column

checkRight

if index is on right wall (currentIndex.second ==cols-1)
return (-1,-1) (could be any pair of indexes outside of the 2D array)
else
return currentIndex with +1 on column

checkDown

if index is on bottom wall (currentIndex.first ==rows-1)
return (-1,-1) (could be any pair of indexes outside of the 2D array)
else
return currentIndex with +1 on row

checkUp

if index is on top wall (currentIndex.first ==0)
return (-1,-1) (could be any pair of indexes outside of the 2D array)
else
return currentIndex with -1 off row

BFS

clear the parents map in case you have to run twice

make pair to store start index
pair<int,int>
make map to store if index has been visited
map<pair<int,int>,bool> visited

Go through whole array
mark walls as visited so they are never added to the queue
mark empties as unvisited
note index of start and mark unvisited
note index of finish and mark unvisited
if not 'w','e','s','f' then it is a bomb or gold
if the passed in "failed" boolean is true
mark bombs and gold as unvisited so they are treated like empties
else
mark bombs and gold as visited so they are treated like walls

make queue to store nextIndex pair
queue<pair<int,int>> searchQueue

push start onto queue
mark start as visited
make parent of start (999,999) or any pair that isn't an index in the map

```
        this will be used to backtrack

    make pair to keep track of currentIndex
    make pair to keep track of nextIndex (when searching in the 4 cardinal
directions)

    while searchQueue isn't empty
        set currentIndex = front of queue
        pop off the front of the queue

        check all 4 cardinal direction using "check___" function for
bounds checking
            set result to nextIndex
                if result != (-1,-1) (this is what is outputted by check
function when
                out of range) &&
                if nextIndex hasn't been visited
                    push nextIndex onto searchQueue
                    mark it as visited
                    update nextIndex's parent to be currentIndex

    if finish was visited
        return true
    else
        return false

pathfinder
    declare boolean to keep track if BFS failed, and set to false because it hasn't failed

    if BFS failed because a path coudn't be found without a bomb
(BFS(maze,failed) is false)
        set failed to true
        run BFS again (the failed boolean will change it to go through the
bomb)
    else
        do nothing

    make moves vector

    go through parents map starting with finish until the parent is
(999,999) as set in BFS
        NOTE: keep in mind, backtracking so everything is reversed
        if previous index was above (parent[index].first == index.first-1), last move was down
            insert 'D' onto front of moves (going backwards so thats why
on front)
        if previous index was below (parent[index].first == index.first+ 1), last move was up
            insert 'U' onto front of moves (going backwards so thats why
on front)
        if previous index was left (parent[index].second == index.second-1), last move was
right
            insert 'L' onto front of moves (going backwards so thats why
```

on front)
if previous index was below (parent[index].second == index.second+1), last move was left
insert 'R' onto front of moves (going backwards so thats why on front)
make sure to update index so not caught in infinite loop

return moves

printArray
strictly used for testing, just prints contents of maze before and after pathfinder in my main

Faux Battle Write Up
CPSC 2120
Lake Summers

I created my own algorithm for this in which I attempt to deal the maximum damage possible.
I do this by sorting yourUnits by power to start, and theirUnits by HP to start,
then targeting all bestAgainst first for maximum initial damage. I also set their targetClass
to be the class they are best against.
I then repeat this process by sorting theirUnits by HP. This is because their HP's should be
lower if any attacks occured. I then repeat the process by using my highest power unit with
moves remaining to target theirUnit with the highest HP remaining.

The algorithm was chosen because how long it takes to determine and return this
plan factors into the outcome, so choosing the first thing in theirUnits that is the
bestAgainst class and attacking until they're dead or you don't have speed left
allows them to attack quickly and effectively. This is opposite of a different strategy
I considered, which was looking for what has lowest HP and targeting that to get the most
kills. Then once the bestAgainst attacks are done, attacking theirUnits in the order of highest HP
remaining with the highest power yourUnit is the way to try and do the most total damage per turn.

References:
Sorting a vector of structs by a certain value of that struct (sorting by HP and power):
https://stackoverflow.com/questions/4892680/sorting-a-vector-of-structs

Runtime complexity:
O(n^3) where n is the number of units, because there is a triple nested for loop.
Obviously this is slow, but I am making that tradeoff for maximum damage
This may not be exact, but at least big O covers anything less

Space Complexity:
O(n) where n is the number of units, because I am storing values for each unit
This may not be exact, but at least big O covers anything less

Algorithm Explanation:
```
//QuickSort (take into account amortized)
Sort yourUnits by power (highest first)
//QuickSort (take into account amortized)
Sort theirUnits by currentHP (highest first)
```

NOTE: I also considered sorting theirUnits by lowest currentHP first because there may be priority in taking out any amount of units first rather than a focus on pure amount of damage dealt. However, this runs the risk of wasting a lot of the power of yourUnits when attacking theirUnits with low currentHP, so I decided to prioritize amount of damage dealt.

Attack all bestAgainst first for maximum initial damage
HOW:
```
For each yourUnit
    For each bestAgainst
        For each theirUnit
            while (conditions 1-3)
```

(1)theirUnit has the yourUnit's corresponding bestAgainst class,
(2)their currentHP is > 0, and
(3)the number of actions has not reached speed
Attack
If number of actions == speed
break the search through theirUnits loop
If number of actions == speed
break the bestAgainst loop and go to next yourUnit

After attacking all bestAgainst first, resort theirUnits by HP and attack highest HP with highest power until everything is dead or all yourUnit's number of actions have reached their speed

HOW:

For all yourUnit

With highest power yourUnit (will be first after sort, so you go in order), attack highest currentHP theirUnit

Attack until their currentHP is < 0 || number of moves has reached speed

If currentHP > 0 or number of actions has reached speed numMoves < speed

move to next highest currentHP (next index)

For each yourUnit

For each theirUnit

while (conditions 1 and 2)

(1)their currentHP is > 0 (not dead)

(2)the number of actions < speed (actions left)

Attack

If number of actions == speed

break the search through theirUnits loop and move to next yourUnit

Supply Chain Write Up
CPSC 2120
Lake Summers

For this I implemented Ford Fulkerson. A lot of my implementation is inspired by the following youtube video explaining ford fulkerson with a breadth first search: https://www.youtube.com/watch?v=GoVjOT30xwo

This video was sent to me by clemson username: tsallur and I forwarded it to the following people because they were having trouble with ford fulkerson as a concept: mgrimsl, jenekaw, colin7

I wanted to let you know that I forwarded them the video in case they follow a similar design pattern so you'd know I wouldn't just copy off of them. The video is just psuedo code and an in depth explanation of how ford fulkerson can be implemented using an adjacency matrix, and I found it quite helpful for getting an idea of how to do a breadth first search as well.

I chose Ford fulkerson because this problem was easily represented with a graph and the maximum throughput was needing to be found each time. I also knew that I would be able to easily complete the bonus objective of multiple starts and ends with one commodity after that.
After coming to your office hours you reminded me of what we did in bipartite matching:
Adding a synthetic source with edges to all the sources with infinite weight
and adding a synthetic sink with edges to all the sources with infinite weight
The only thing is that you just need to make sure these aren't included in the final output because those edges don't truly exist.
So I added that to my code, and obviously it will still work for cases with only 1 start and source as well because it is simply just adding one extra edge at the beginning and end.

Runtime complexity:
O(n^2) where n is the number of nodes. This is because each node must be checked multiple times when looking for paths.
This may not be exact, but at least big O covers anything less

Space Complexity:
O(n) where n is the number of edges because we are storing values for each edge
This may not be exact, but at least big O covers anything less

Ford Fulkerson implementation Explanation:

- Global Variables:
  - set of all cities: allCities
  - map of pair of cities to capacity and ID: citytoCapandID
  - map of each city to a parent (used in BFS): parent
- Functions:
  - readEdges(string)
    - read in the ID, startCity, endCity, capacity, and cost
    - map to respective values in citytoCapandID
  - checkForPathBFS(startCity,endCity)
    - -reset the parent map
    - -make a map of city to bool true if visited false if not visited: cityVisited

-for all cities
    mark visted as false
-make a queue
-add the startCity to the queue
-mark the startCity as visited
-make the parent of the startCity "source" or any string
    this string will be used when backtracking as a stopping point
-while the queue isn't empty
    pop the first thing off the queue and store it as currentCity
    for all cities (referred to as nextCity from here on)
        If nextCity not visited and,
        the edge (currentCity,nextCity) exists and
        the capacity of the edge is > 0
            add the nextCity to the queue
            Mark nextCity as visited
            set the parent of the nextCity to currentCity

if the endCity was visited
    return true
else
    return false

organizeLogistics(vector<pair<string,double>> start, vector<pair<string,double>> end):

****THIS IS WITH THE BONUS, BUT OBVIOUSLY IT STILL WORKS FOR THE BASE CASES***

readEdges (use function readEdges) with worldmap.txt passed in

make an unordered_map of ID's of edges to cargo put on them

set maxCapacity to a large double

Keep track of used edges with a vector

***Code added for bonus**
Add a node pointing to each start city (sources) with essentially infinite edge capacity
This way, they are not the limiting factor when doing multiple start and end locations
Add a node which edges from all end cities point to (sinks) with essentially infinite edge capacity
Set the ID of all these edges to an arbitrary thing so we know not to add them to the final map
Add these cities to allCities

while a path is available (Use BFS to check, which will update the parent map)
    reset the maxCapacity to be a large double
    reset the usedEdges

    backtrack through the parent map to get your list of used edges
        keep track of maxCapacity while doing this

go through list of used edges
for their ID's, update the final unordered_map
**unless it is one of the edges that you added (where arbitrary ID comes in)

update the citytoCapandID capacities so you can run the BFS again

return your final unordered_map

I had also originally implemented organizeLogistics where the double being passed in in the pair was the amount of cargo you can possibly put through (regardless of the edge capacities). Therefore, when the maxCapacity of a path was found, if there is less amount of cargo than that maxCapacity, only add the amount of cargo. You then keep a running total of the amount of cargo, and go until it is 0 or a path isn't found.

I marked all the differences to the original algorithm

This function is named organizeLogisticsWithCargoAmount and included at the bottom of the .h file

organizeLogisticsWithCargoAmount(vector<pair<string,double>> start, vector <pair<string,double>> end):
^^ to clarify, this is the amount of cargo you start with

readEdges (use function readEdges)

set remainingCargo to starting cargo amount

make an unordered_map of ID's of edges to cargo put on them

set maxCapacity to a large double

Keep track of used edges

while a path is available (Use BFS to check, which will update the parent map)
reset the maxCapacity to be a large double
reset the usedEdges

backtrack through the parent map to get your list of used edges
keep track of maxCapacity while doing this

**difference
if maxCapacity > remainingCargo (don't have enough cargo to take advantage of maxCapacity)
set maxCapacity to remainingCargo

go through list of used edges
for their ID's, update the final unordered_map by adding maxCapacity

update the citytoCapandID capacities so you can run the BFS again

```
        **difference
        update remainingCargo
            subtract maxCapacity

        **difference
        if remainingCargo is 0
            break out of the while loop

    return your final unordered_map
```