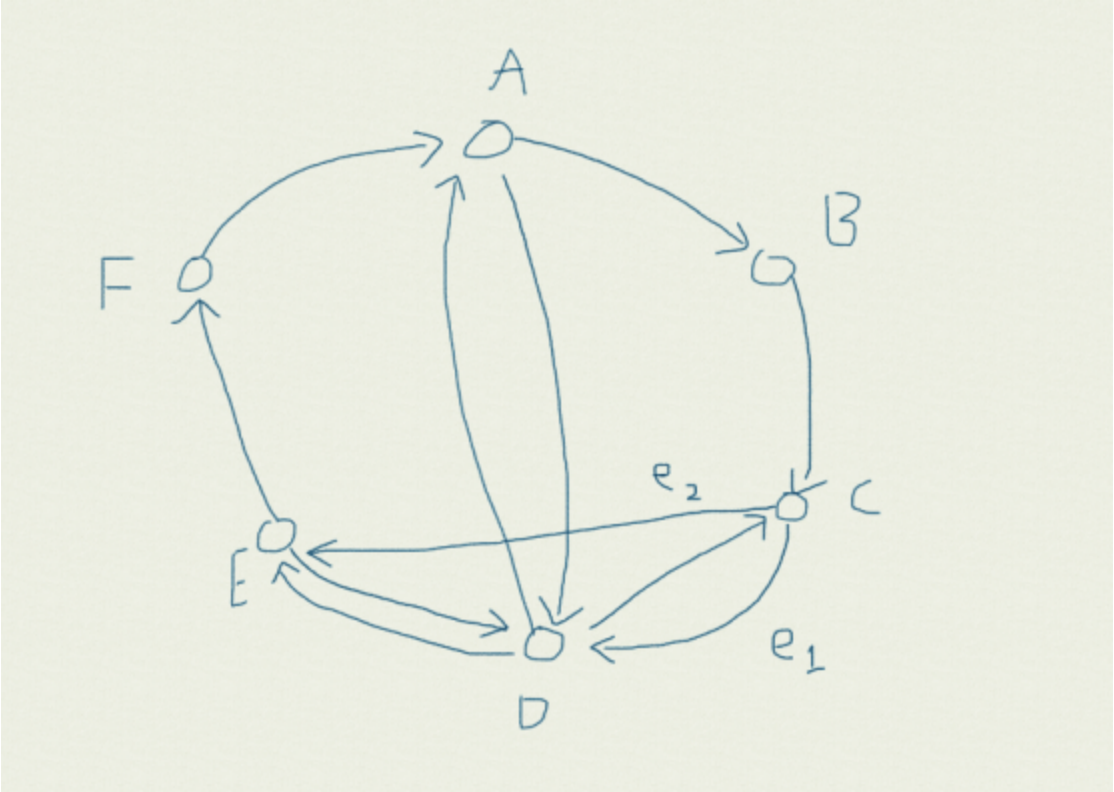
25100198 – A3

Problem 1

a) For there to be a path between two arbitrary sites, these two sites must be connected. Since we can select S_t arbitrarily, a path must exist from any distinct S_t and S_u. For this to hold in general, the graph must be connected. Furthermore, since the roads we use **must be unique** (no backtracking), we need the degree of all vertices to be even. For directed graphs, this means that the evenness must result from the sum of the out-degree and in-degree, where each node must have an in/out-degree of >= 1. For undirected graphs, it just refers to the vertex's degree.
   This will ensure we can enter and leave using different roads. If the two conditions above are granted, we are left with a sub-graph that is connected by a unique set of roads, and a path exists from one node to the other that does not re-use roads (edges). This is called an Eulerian circuit.

b) If we assume that the conditions above will hold, then the sub-graph which will contain a valid Eulerian circuit. All we need to do now is use a depth-first search to compute all the paths. For this algorithm, we must maintain a set of visited nodes, a set of used edges, and a variable that contains our starting node. The algorithm looks like the following:
   i) Check if all nodes have degrees that are even.
   ii) Check if the graph is connected by performing some traversal. Preferably use a breadth-first search.
   iii) If steps 1 and 2 yield True, perform a depth-first-search, saving the paths and their associated costs. The termination condition should be such that our set of visited nodes contains all the nodes, and that our current node is our starting node. Furthermore, if a path contains an edge already in our visited edges list, then we shall not use that path.
   iv) If need be, select the lowest cost path.

c) Refer to the picture below, which describes a scenario where our algorithm above will fail to find a global minimum if e2 << e1. The starting node is A.



Problem 2

For the purposes of this algorithm, I will maintain three data structures. Firstly, it'll be an array of tuples, where the first index of the tuple contains the price/kg of a gemstone, and the second index contains the total weight of the gemstone. I'll call this data structure the gemstone_arr. Secondly, I'll maintain an array called selected_gemstones, which will empty initially. Finally, I'll maintain a third variable called the total_value. As a pre-processing step, we can sort the gemstone_arr based on decreasing price/kg.

We define a recursive function Max_Value that takes the gemstone_arr, capacity, selected_gemstones and total_value as follows:

1. Check if capacity is zero.
    a. Return total_value, selected_gemstones
2. Check if the capacity is less than the weight of the current gemstone.
    a. If yes, then add capacity * value_of_gemstone to total_value.
    b. Add current gemstone to the selected_gemstones list, modifying the second index of the tuple to capacity.
    c. return total_value, selected_gemstones
3. Else
    a. Compute both branches:

i. Max_value(gemstone_arr[1:], capacity, selected_gemstones, total_value)
ii. Max_value(gemstone_arr[1:], capacity - gemstone_arr[0][0], selected_gemstones ++ gemstone_arr[0], total_value + (gemstone_arr[0][1] * gemstone_arr[0][1]))

b. Compare the value of total_value for both branches and set selected_gemstones and total_value equal to the returned values for whichever branch had the greater total_value!
c. Return total_value, selected_gemstones

Problem 3

a) For this problem, I will maintain an array of tuples called friend_wishes. Here, the index will represent the n_i, where n_i is the ith friend in the set of friends. The first element in the tuple will be the friend's first wish, and the second element the second. The m toppings will be assigned a number from 1 to m to identify them uniquely. If the algorithm returns an empty set, no such combination is possible. The algorithm will work as follows:
   i) Make a 2D array called *topping_matrix*, with the number of friends as the rows, and the number of toppings as the columns. For each friend n_i's wish about a topping m_j, set *topping_matrix[i][j]* equal to 1 for topping requested, -1 for topping rejected, and 0 for no request. With such a scheme, contentious toppings will have both a +1 and a -1 in their columns.
   ii) Furthermore, make a *wish_granted* array, which will be of length n, with an index i corresponding to friend n_i. We will also make a *final_toppings* array, which will be empty initially.
   iii) Now, we can grant all the non-contentious toppings. To do so, do the following:
      1) Iterate through the topping_matrix. If a column does not contain both +1s and -1s, we can safely grant this wish for all friends who have sent a request for this topping.
      2) To do so, we increment the respective entries in the *wish_granted* array.
      3) We will also add the topping to the *final_toppings* array if the column contains all add requests, and do nothing if the column contains all subtract requests.
   iv) Once all non-contentious toppings are granted, we can move on to granting contentious toppings, categorized by having both a +1 and a -1 in their column. To do so:

1) Iterate through the topping_matrix. Count the number of subtract and add requests, while keeping track of the friends who made each request.
2) If the length of the subtract and add request is both zero, then this is an uncontended topping. We can simply skip this topping.
3) If the length of the subtract and add requests is equal, then grant the request (add or subtract) for whichever collection of friends has the lower amount of wishes.
    (a) We can figure this out by iterating over the add and subtract request arrays which contain the indices of the friends that have made the request.
    (b) We can use those indices to query the *wish_granted* array, and figure out the total number of wishes.
    (c) Again, add the topping to *final_toppings* if the add requests win, and do nothing if the subtract requests win.
    (d) If both wish counts are equal, we pretend as if the add request won.
4) If the length of the add requests is greater, grant the wishes of the friends who made an add request for this topping and add the topping to *final_toppings.*
5) If the length of the subtract requests is greater, grant the wishes of the friends who made a subtract request for this topping.

v) Once we are done granting the contentious toppings, we will have a final *wish_granted* array, which will contain the number of wishes granted for each friend. If any of the elements in this array is a zero, then **no combination of toppings exist.** Otherwise, our *final_toppings* array will contain the largest subset of toppings possible!

b) When designing an answer to this question, I accidentally answered (b) within (a). As such, simply refer to the algorithm above. Specifically, it guarantees that this is the largest subset of toppings possible as it penalizes the number of wishes granted (we break ties by comparing wishes, and prefer to add when wishes are tied). However, one modification must be made to ensure the largest possible subset:

i) Instead of skipping uncontended (or unrequested) toppings, we will add them! In other words, if a topping's column contains all 0s, we will add this topping by default.