Assignment5.cpp

Directed Graph

```
// node for the directed graph
class Node
{
public:
    int id; // vector number
    vector<pair<Node*, int>> neighbors; // list of neighbors

    // add a neighbor
    void addNeighbor(Node* neighbor, int cost)
    {
        neighbors.push_back(make_pair(neighbor, cost));
    }

};

// directed graph class
class DirectedGraph
{
public:
    // store all nodes in the graph
    vector<Node*> nodes;

    // add a node to the graph
    void addNode(Node* node)
    {
        nodes.push_back(node);
    }

    // display the directed graph
    void printGraph()
    {
        for (Node* node : nodes)
        {
            cout << "Node " << node->id << " is connected to: ";
            for (pair<Node*, int> neighbor : node->neighbors)
            {
                //             id of neighbor,                cost of neighbor
                cout << "(" << neighbor.first->id << ", " << neighbor.second << ") ";
            }
            cout << endl; // next line
        }
    }
};
```

The simple node class is reworked to hold the id (vertex number) as well a list of its neighbors associated with the cost to reach that neighbor. The directed graph class has all the functionality - including adding nodes and printing the graph's paths.

Fractional Knapsack Problem

```
class Item
{
public:
    string name;
    double totalPrice;
    int qty;

    Item(string n, double p, int q)
    {
        name = n;
        totalPrice = p;
        qty = q;
    }
};

// function to compute best case in fractional knapsack problem
double fractionalKnapsack(int capacity, Item* items[]) {

    double totalValue = 0.0;

    for (int i = 0; i < 4; i++)
    {
        // if the whole item fits in the bag
        if (capacity >= items[i]->qty) {
            totalValue += items[i]->totalPrice;
            capacity -= items[i]->qty;
            // add it to the bag and decrease bag quantity
        }
        // take all that fits in the bag
        else {
            double fraction = (double)capacity / items[i]->qty;
            totalValue += fraction * items[i]->totalPrice;
            break;
        }
    }

    return totalValue;
}
```

The knapsack function iterates through the list of items (sorted from most valueble to least based on unit price) and adds as much of the most valuable item possible. If there is enough space, the maximum amount of the next most valuable item is added. This continues until the knapsack is full and the total value of the contents is returned.

Loading Items From File

```
void loadGraphs()
{
    // initialize file
    fstream itemFile;
    itemFile.open("graphs1.txt", ios_base::in);

    if (itemFile.is_open())
    {
        // counter
        int i = 0;

        // initialize objects
        Matrix m;
        AdjacencyList adj;
        int n = 0; // size

        // while file still has items to read
        while (itemFile.good())
        {
            string line; // initialize item string
            getline(itemFile, line); // get line

            if (line.find("--") != std::string::npos)
            {
                // ignore this line
                //std::cout << "ignore\n";
            }

            else if (line.find("graph") != std::string::npos)
            {
                m.print();      // print old matrix
                adj.print();    // print old adjacency list
                n = 0;          // reset size
            }

            else if (line.find("vertex") != std::string::npos)
            {
                // adjust sizing for each new vertex
                n++;
                m.size = n;
                adj.size = n;
                //std::cout << "vertex" << n << "\n";
            }

            else if (line.find("edge") != std::string::npos)
            {
                // find v1 and v2
                int v1 = 0;
                int v2 = 0;
                m.addEdge(v1, v2);
                adj.addEdge(v1, v2);
                //std::cout << "edge\n";
            }
```

```
            i++; // increment counter
        }
        itemFile.close();
    }
}
```

Main Function

```
int main()
{

    // fractional backpack problem
    Item* red       = new Item("red", 4.0, 4);      // unit value: 1
    Item* green     = new Item("green", 12.0, 6);   // unit value: 2
    Item* blue      = new Item("blue", 40.0, 8);    // unit value: 5
    Item* orange    = new Item("orange", 18.0, 2);  // unit value: 9

    Item* items[] = {orange, blue, green, red};

    cout << "Knapsack with capacity 1 has a value of " << fractionalKnapsack(1, items) <<
         endl;
    cout << "Knapsack with capacity 6 has a value of " << fractionalKnapsack(6, items) <<
         endl;
    cout << "Knapsack with capacity 10 has a value of " << fractionalKnapsack(10, items)
        << endl;
    cout << "Knapsack with capacity 20 has a value of " << fractionalKnapsack(20, items)
        << endl;
    cout << "Knapsack with capacity 21 has a value of " << fractionalKnapsack(21, items)
        << endl;
}
```

The main function creates the Items and initializes the data associated with them, then loads them into a list with the highest unit value first. The items are then used to determine the value of knapsacks of different capacities.

Sample Output

```
Knapsack with capacity 1 has a value of 9
Knapsack with capacity 6 has a value of 38
Knapsack with capacity 10 has a value of 58
Knapsack with capacity 20 has a value of 74
Knapsack with capacity 21 has a value of 74
```

This is a sample of the output after the program is run to display the value of the contents of knapsacks solved with the fractional knapsack function.