CMPT 435 - Fall 2024 - Dr. Labouseur
Assignment Four – Algorithms

Stefano Farro
stefano.farro1@marist.edu

December 6, 2024

1 Weighted & Directed Graphs

1.1 Graph Structure & Operation

The Graph class implements a weighted, directed graph through a linked object representation. Each Vertex object maintains its ID, a set of neighbor relationships including destination vertex and weight, and attributes for tracking distances and predecessors during path-finding operations. The Graph constructor initializes these structures and provides methods for vertex and edge manipulation.

Listing 1: Graph Constructor & Simple Operations

```cpp
Graph::Graph() : title(), vertices() {}

void Graph::setTitle(std::string title) {
    this->title = title;
}

std::string Graph::getTitle() {
    return title;
}

void Graph::addVertex(int id) {
    vertices.emplace(id, Vertex(id));
}

void Graph::addEdge(int from, int to, int weight) {
    vertices[from].neighbors.push_back({to, weight});
}
```

The class structure is designed to efficiently represent weighted edges and support the path-finding operations required by Bellman-Ford. Each vertex stores its outgoing edges as pairs of destination vertices and weights, enabling efficient traversal of edges during the relaxation steps.

Listing 2: Class Structure for Weighted & Directed Graphs

```cpp
class Graph {
private:
    struct Vertex {
        int id;
        std::vector<std::pair<int, int>> neighbors;  // (
            destination_vertex, weight)
        int distance;
        int predecessor;

        Vertex() : id(0), distance(INT_MAX), predecessor(-1)
             {}
        Vertex(int num) : id(num), distance(INT_MAX),
            predecessor(-1) {}
    };

    std::string title;
    std::map<int, Vertex> vertices;  // maps vertex ID to
        unique Vertex object

public:
    Graph();

    void setTitle(std::string title);
    std::string getTitle();

    void addVertex(int id);
    void addEdge(int from, int to, int weight);
    void SSSP(int source);
    void relax(int u, int v, int w);

    static std::vector<Graph*> parseGraphList(std::vector<
        std::string>& list);
};
```

1.2 Single Source Shortest Path

The Bellman-Ford implementation systematically relaxes edges to find shortest paths. The outer loop runs V-1 times, where V is the number of vertices, ensuring that paths of increasing length are considered. The inner loop examines each edge, potentially updating distance values if a shorter path is found. A final pass checks for negative cycles.

The relaxation function iterates graph edge, updating vertex distances when shorter paths are found. For each edge (u,v), if the distance to u plus the edge weight is less than the current distance to v, the distance to v is updated and its predecessor is set to u. This process continues for —V—-1 iterations to find shortest paths, followed by a final pass checking for negative cycles.

Listing 3: Bellman-Ford Relaxation Helper Function

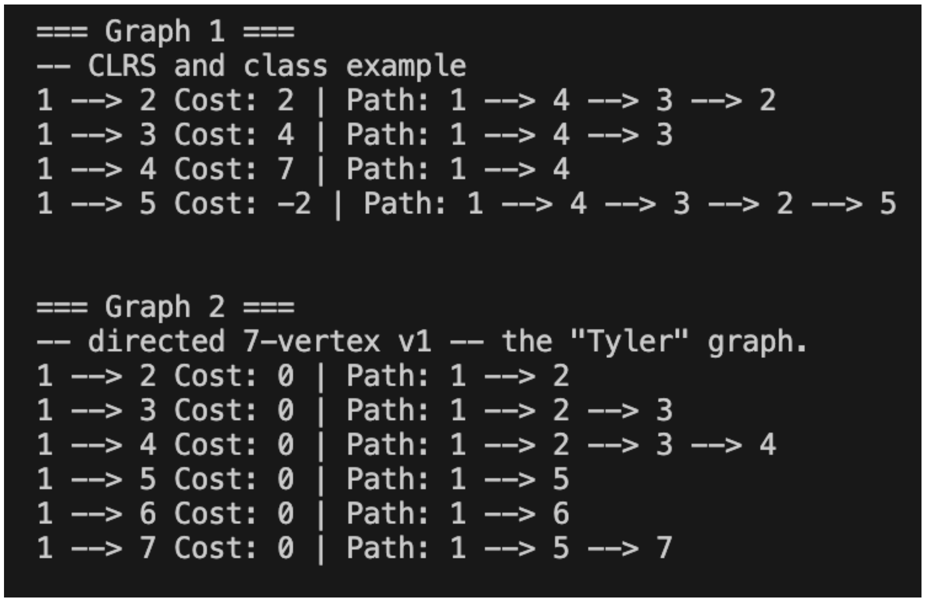
```
void Graph::relax(int u, int v, int w) {
    if (vertices[u].distance != INT_MAX &&
        vertices[u].distance + w < vertices[v].distance) {
        vertices[v].distance = vertices[u].distance + w;
        vertices[v].predecessor = u;
    }
}
```

Listing 4: Bellman-Ford Single Source Shortest Path Algorithm

```
void Graph::SSSP(int source) {
    vertices[source].distance = 0;

    // relax edges |V| - 1 times
    int V = vertices.size();
    for (int i = 1; i <= V - 1; i++) {
        for (const auto& vertex : vertices) {
            int u = vertex.first;
            for (const auto& edge : vertex.second.neighbors)
                {
                int v = edge.first;    // destination vertex
                int w = edge.second;   // edge weight
                relax(u, v, w);
            }
        }
    }

    // negative weight cycle check
    for (const auto& vertex : vertices) {
        int u = vertex.first;
        for (const auto& edge : vertex.second.neighbors) {
            int v = edge.first;
            int w = edge.second;
            if (vertices[u].distance != INT_MAX &&
                vertices[u].distance + w < vertices[v].
                    distance) {
                std::cout << "Graph contains negative weight
                     cycle!" << std::endl;
                return;
            }
        }
    }
```

Listing 5: SSSP Printing implementation (continuation of above function)

```cpp
    // pretty pritning
    for (const auto& vertex : vertices) {
        int v = vertex.first;
        if (v != source) {
            std::cout << source << " --> " << v << " Cost: "
                      << vertices[v].distance << " | Path: ";

            std::vector<int> path;
            int current = v;
            while (current != -1) {
                path.push_back(current);
                current = vertices[current].predecessor;
            }

            for (int i = path.size() - 1; i >= 0; i--) {
                std::cout << path[i];
                if (i > 0) std::cout << " --> ";
            }
            std::cout << std::endl;
        }
    }
}
```



```
=== Graph 1 ===
-- CLRS and class example
1 --> 2 Cost: 2 | Path: 1 --> 4 --> 3 --> 2
1 --> 3 Cost: 4 | Path: 1 --> 4 --> 3
1 --> 4 Cost: 7 | Path: 1 --> 4
1 --> 5 Cost: -2 | Path: 1 --> 4 --> 3 --> 2 --> 5


=== Graph 2 ===
-- directed 7-vertex v1 -- the "Tyler" graph.
1 --> 2 Cost: 0 | Path: 1 --> 2
1 --> 3 Cost: 0 | Path: 1 --> 2 --> 3
1 --> 4 Cost: 0 | Path: 1 --> 2 --> 3 --> 4
1 --> 5 Cost: 0 | Path: 1 --> 5
1 --> 6 Cost: 0 | Path: 1 --> 6
1 --> 7 Cost: 0 | Path: 1 --> 5 --> 7
```

Figure 1: Output of SSSP

2 Fractional Knapsack Problem

The fractional knapsack solution uses a greedy approach to maximize value while respecting capacity constraints. Items are sorted by value per unit weight (value per scoop in this case), then allocated to knapsacks in descending order of value density.

2.1 SpiceType Class

The SpiceType class encapsulates spice properties and supports value comparison. Each spice has a name, per-scoop value, and available quantity. The class implements comparison operators to enable sorting by value density, which is crucial for the greedy allocation strategy.

Listing 6: Spice Object Class

```cpp
class SpiceType {
private:
    std::string name;
    double valuePerScoop;
    double availableQty;

public:
    SpiceType(std::string n, double total_price, double qty)
        ;
    std::string getName() const;
    double getValue() const;
    double getQty() const;
    void reduceQty(double amount);
    bool operator<(const SpiceType& other) const;
};
```

The Constructor initializes spice properties and calculates per-scoop value. Core operations include quantity management (getQty/reduceQty) and comparison operators for value-based sorting.

Listing 7: Spice Object Operations

```cpp
SpiceType::SpiceType(std::string n, double total_price,
   double qty)
    : name(n), valuePerScoop(total_price/qty), availableQty(
        qty) {}

std::string SpiceType::getName() const { return name; }
double SpiceType::getValue() const { return valuePerScoop; }
double SpiceType::getQty() const { return availableQty; }
void SpiceType::reduceQty(double amount) { availableQty -=
   amount; }
bool SpiceType::operator<(const SpiceType& other) const {
    return valuePerScoop > other.valuePerScoop;
}
```

2.2 Knapsack Class

The Knapsack class manages capacity constraints and spice allocation. It tracks remaining capacity, maintains a map of contained spices and their quantities, and calculates total value. The class ensures fractional allocations are handled correctly when splitting spices across multiple knapsacks.

Listing 8: Napsack Object Class

```cpp
class Knapsack {
private:
    double capacity;
    std::map<SpiceType, double> contents;
    double remainingCapacity;
    double totalValue;

public:
    explicit Knapsack(double cap);
    void addSpice(const SpiceType& spice, double scoops);
    double getCapacity() const;
    double getRemainingCapacity() const;
    double getTotalValue() const;
    const std::map<SpiceType, double>& getContents() const;
    void printContents() const;
};
```

The Constructor sets initial capacity. addSpice tracks contents, remaining capacity, and total value. Accessor methods support capacity/value queries. printContents formats output with proper spacing and decimal precision for spice quantities and total value.

Listing 9: Napsack Object Operations

```cpp
Knapsack::Knapsack(double cap)
    : capacity(cap), remainingCapacity(cap), totalValue(0.0)
         {}

void Knapsack::addSpice(const SpiceType& spice, double
    scoops) {
    if (scoops > 0) {
        contents[spice] = scoops;
        remainingCapacity -= scoops;
        totalValue += scoops * spice.getValue();
    }
}
```

2.3 Spice Heist Class

This wrapper class orchestrates the overall solution, coordinating between available spices and knapsacks. It implements the core greedy algorithm: sorting spices by value density, then allocating them to knapsacks to maximize total value while respecting capacity constraints.

Listing 10: Spice Heist Greedy Algorithm Wrapper

```
class SpiceHeist {
private:
    std::vector<SpiceType> availableSpices;
    std::vector<Knapsack> knapsacks;

public:
    void addSpiceType(const std::string& name, double
        total_price, double qty);
    void addKnapsack(double capacity);
    void optimizeKnapsacks();
    void printResults() const;

    static std::vector<SpiceHeist*> parseSpiceHeist(std::
        vector<std::string>& list);
};
```

The addSpiceType/addKnapsack functions populate available resources. optimizeKnapsacks implements the greedy algorithm - sorting spices by value and allocating highest-value spices first. parseSpiceHeist processes input files, handling comment lines and command parsing to construct the heist scenario.

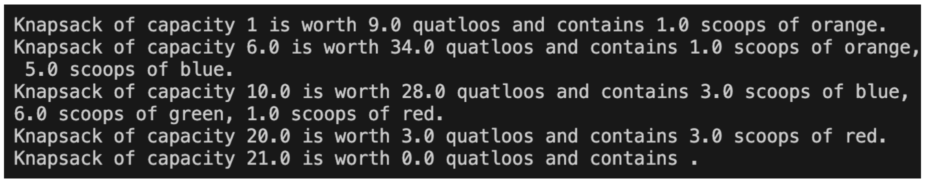
Listing 11: Spice Heist Operations

```cpp
void SpiceHeist::addSpiceType(const std::string& name,
    double total_price, double qty) {
    availableSpices.emplace_back(name, total_price, qty);
}

void SpiceHeist::addKnapsack(double capacity) {
    knapsacks.emplace_back(capacity);
}

void SpiceHeist::optimizeKnapsacks() {
    // sort spices by value per scoop (descending)
    std::sort(availableSpices.begin(), availableSpices.end()
        );

    // fill each knapsack independently
    for (auto& knapsack : knapsacks) {
        double spaceLeft = knapsack.getCapacity();

        // each knapsack gets to try using the full amount
            of each spice
        for (const auto& spice : availableSpices) {
            if (spaceLeft <= 0) break;

            // take as much as we can of this spice
            double amountToTake = std::min(spaceLeft, spice.
                getQty());
            if (amountToTake > 0) {
                knapsack.addSpice(spice, amountToTake);
                spaceLeft -= amountToTake;
            }
        }
    }
}

void SpiceHeist::printResults() const {
    for (const auto& knapsack : knapsacks) {
        knapsack.printContents();
    }
}
```

2.3.1 Algorithm Analysis

The optimizeKnapsacks method implements the following steps:

- Sorts available spices by value per scoop in descending order
- For each knapsack:
  - Tracks remaining capacity
  - Takes maximum possible amount of each spice in value order
  - Updates remaining space after each allocation
- Allows fractional allocations when remaining capacity is less than available spice quantity



```
Knapsack of capacity 1 is worth 9.0 quatloos and contains 1.0 scoops of orange.
Knapsack of capacity 6.0 is worth 34.0 quatloos and contains 1.0 scoops of orange,
 5.0 scoops of blue.
Knapsack of capacity 10.0 is worth 28.0 quatloos and contains 3.0 scoops of blue,
6.0 scoops of green, 1.0 scoops of red.
Knapsack of capacity 20.0 is worth 3.0 quatloos and contains 3.0 scoops of red.
Knapsack of capacity 21.0 is worth 0.0 quatloos and contains .
```

Figure 2: Output of the Spice Heist

The output demonstrates & validates the greedy allocation strategy:

- Orange spice (9.0 quatloos/scoop) is allocated first
- Blue spice (5.0 quatloos/scoop) fills remaining space
- Green (2.0 quatloos/scoop) and red (1.0 quatloos/scoop) spices are used last
- Larger knapsacks receive more valuable spices by accessing full quantities

3 Asymptotic Analysis

3.1 Bellman-Ford SSSP Analysis

The Bellman-Ford algorithm has the following time complexity components:

- Outer loop (vertices - 1 times): O(V) - Required for path length consideration
- Inner loop (all edges): O(E) - Edge relaxation attempts
- Overall complexity: O(VE) - Product of nested loop iterations

3.2 Fractional Knapsack Analysis

The fractional knapsack algorithm has the following time complexity components:

- Sorting spices by value: O(n log n) - Required for greedy choice property
- Processing each knapsack: O(n) - One pass through sorted spices
- Overall complexity: O(n log n) - Dominated by initial sorting