UCSD CSE 101 WI'18 : : PA2

Professor Andrew B. Kahng
Discussion by Nathan Ng and Joseph Chen

Housekeeping

```
template <class T>
class Edge {
public:
    T src;
    T dest;

    Edge(T s, T d){
        this->src = s;
        this->dest = d;
    }

    bool operator<(const Edge<T>& l) const{
        return (l.src < this->src) ||
               ((l.src == this->src) && (l.dest < this->dest));
    }
};
```

Edge<T> class:

- Generic
- Compatible with std::map<T> and other ordered STL data structures
- Can be created, modified, and accessed via helper functions in Graph<T>
    - Values must be accessed via Graph<T>'s get_weight()!
- In general, don't worry about the Edge class :)

Housekeeping

Using Edge<T>:

```
std::map<Edge<T>, float> weights;

g.vertices[ /* T id of src */ ]->edges.insert( /* T id of dest */ );
g.set_weight( /* T id of src */ , /* T id of dest */ , /* Weight of edge */ );
g.get_weight( /* T id of src */ , /* T id of dest */ );

for(auto it = g.weights.begin(); it != g.weights.end(); it++){
     // it->first.src
     // it->first.dest
     // it->second
```

EDGE WEIGHT ACCESSORS ARE PROVIDED VIA THE GRAPH CLASS

Housekeeping

Data structures you may want to use:

```
std::priority_queue<T>
```

```
std::vector<T>
```

Note that while std::vector is typically not used in the class of algorithms covered, the underlying implementation of C++'s priority queue uses a vector for storage and will require using an std::vector declaration when overriding properties of the priority queue:

```
std::priority_queue</* custom class */, std::vector</* custom class */>, /* custom comparison "struct" function */> pq;
```

Housekeeping

Using Alarm<T>

- Concept of “alarm” from Dijkstra’s
- Can use as a general way to alert algorithm of which edge to next explore
  - May be useful for Prim’s and PrimDijk…
- Priority queue to sort based on alarm time
  - std::priority_queue<Alarm<T>, std::vector<Alarm<T>>> pq;

```
template <class T>
class Alarm {
  public:
   T src;
   T dest;
   float time;

   Alarm(T src, T dest, float time) {
     this->src = src;
     this->dest = dest;
     this->time = time;
   }

   bool operator<(const Alarm<T>& l) const{
     return this->time > l.time;
   }
};
```

Q1 :: Dijkstra’s Shortest Paths Tree

Pseudocode for Dijkstra’s algorithm can be found in lecture slides

With Dijkstra’s algorithm, we can find all shortest paths from a root vertex *u* to every other vertex in the graph.

You can use a priority queue to keep track of which vertices to visit next. NOTE: be careful when using the priority queue, as the elements should represent “snapshots” of the state of the path thus far.

Get your implementation working **PERFECTLY** before continuing to the next parts.

Q2 :: Prim’s Minimum Spanning Tree

Prim’s algorithm finds the minimum spanning tree such that the distance between any two vertices *u* and *v* is minimized.

Your implementation will read the same graph input as in Q1 and start Prim’s algorithm at a given vertex.

Strategy: list the differences between Dijkstra’s and Prim’s and make the appropriate changes in your code.

Q3 :: Prim’s/Dijkstra’s Hybrid

Greedy decisions:

Prim’s:

- Vertex added optimal based exclusively on the weight of the edge to reach it

Dijkstra’s:

- Vertex added optimal based on the sum of edge weight and weight of path from the source vertex

Q3 :: Prim’s/Dijkstra’s Hybrid

Hybrid:

- “Prim-Dijkstra”: attach a weighting “factor” onto the cost of the path leading to the source vertex ‘c’
- Weight ‘c’ will determine the closeness of the match between the hybrid algorithm and Prim’s or Dijkstra’s.

More: [Lecture 6, Slide 29]

Q3 :: Prim's/Dijkstra's Hybrid

```
template <class T>
float prim(Graph<T>& g, T src) {
    float cost = 0.0;
```

```
template <class T>
float dijkstra(Graph<T>& g, T src) {
    float cost = 0.0;
```

```
template <class T>
float primdijk(Graph<T>& g, T src, float c) {
    float cost = 0.0;
```

Q3 :: Prim’s/Dijkstra’s Hybrid

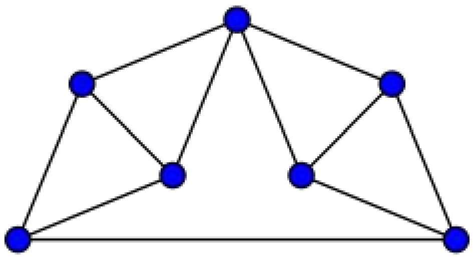
You will be asked to experiment with varying values of ‘c’ to observe how paths are picked in different graphs.

Graph generator:

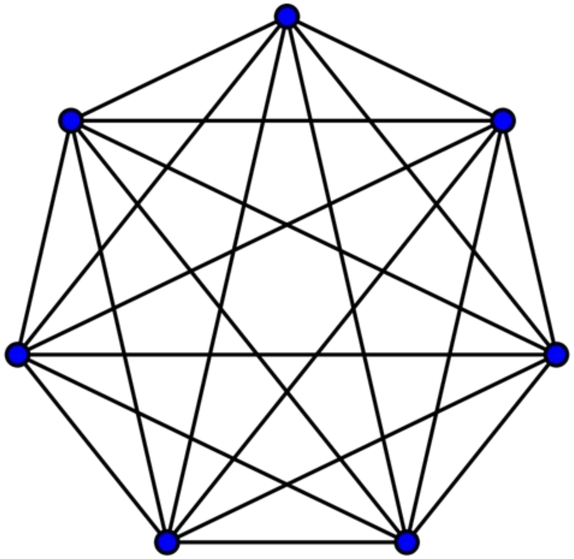
- We’ve included a graph generator written in Python to help you generate test cases, which can be directly fed into the PrimDijk tester.
    - **p**: probability that an edge exists between two vertices
    - **n**: number of vertices
    - **w**: maximum weight of vertices
- Generated graphs can be sparse or denser

Q4 :: Prim's/Dijkstra's Hybrid

Sparse graph: less edges -> fewer possible paths between vertices



Dense graph: more edges -> more possible paths between vertices



Q3 :: Analysis

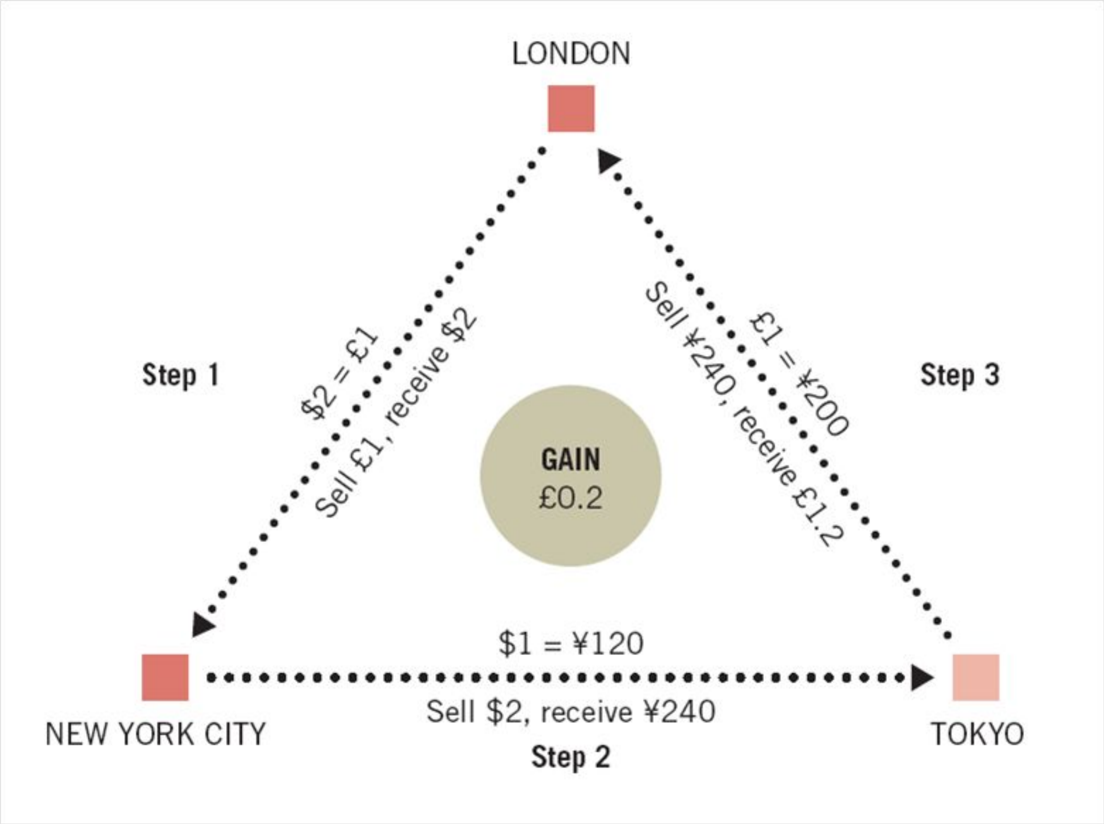
Observations:

- Granularity of path length variation
- Deducing graph structure from variating ‘c’
- Effects of p, n, and w

Produce a write-up describing the above, along with the requested tabular data.

Keep in mind that the graphs will take longer to generate and run with larger p and n values, and be wary of your disk quota!

Q4 :: Bitvest



LONDON
Step 1
$2 = £1
Sell £1, receive $2
Step 3
£1 = ¥200
Sell ¥240, receive £1.2
GAIN
£0.2
$1 = ¥120
Sell $2, receive ¥240
NEW YORK CITY
TOKYO
Step 2

Q4 :: Bitvest

In real life, exploiting such situations induces a surge in demand that stores equilibrium to prices.

In our problem we are interested in a snapshot of the cryptocurrency markets at a certain point in time.

- Prices will not return to equilibrium in a snapshot.

Q4 :: Bitvest

In real life, exploiting such situations induces a surge in demand that stores equilibrium to prices.

In our problem we are interested in a snapshot of the forex market at a certain point in time.

- Prices will not return to equilibrium in a snapshot.
- **Infinite** profit???

Q4 :: Bitvest

In real life, exploiting such situations induces a surge in demand that stores equilibrium to prices.

In our problem we are interested in a snapshot of the forex market at a certain point in time.

- Prices will not return to equilibrium in a snapshot.
- Infinite profit???
- This sounds familiar.

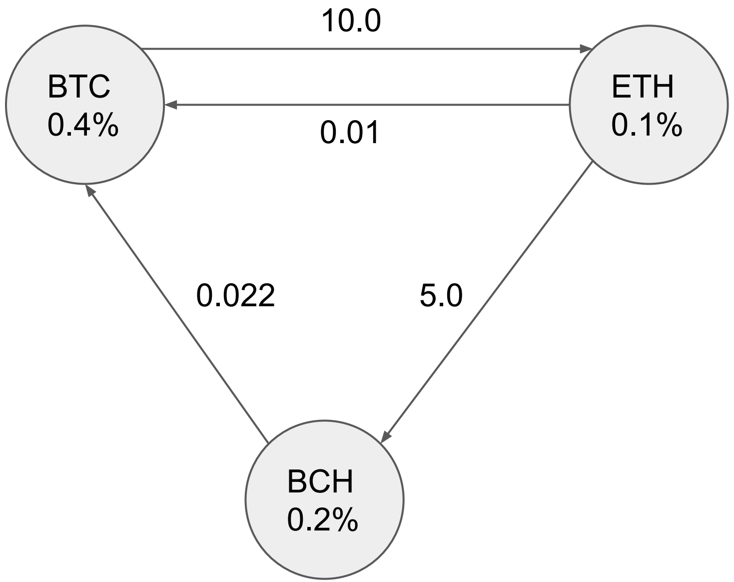
Example Input (profitable):

```
BTC 0.004
ETH 0.001
BCH 0.002

BTC ETH 10.0
ETH BTC 0.01
ETH BCH 5.0
BCH BTC 0.022
```

Example Output (profitable):

```
Trade status: 1 // Profit!
```



Example Input (profitable):

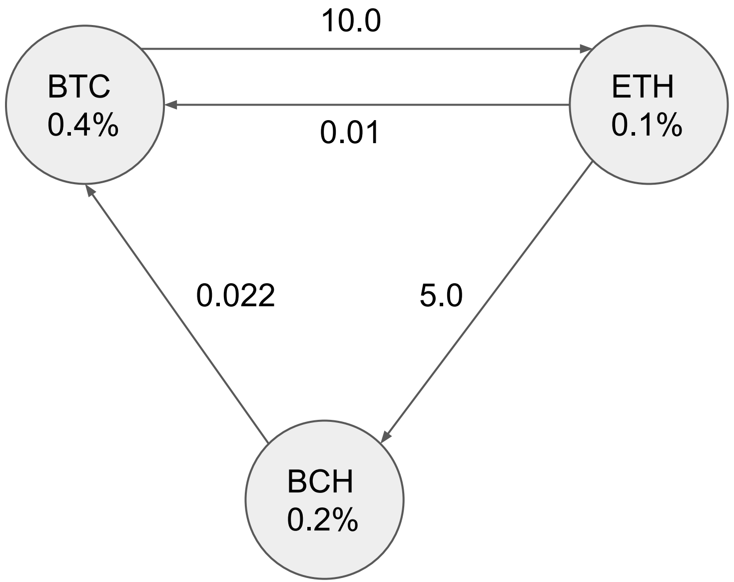
```
BTC 0.004
ETH 0.001
BCH 0.002

BTC ETH 10.0
ETH BTC 0.01
ETH BCH 5.0
BCH BTC 0.022
```

Example Output (profitable):

```
Trade status: 1 // Profit!
```

```
x = 100.0
```



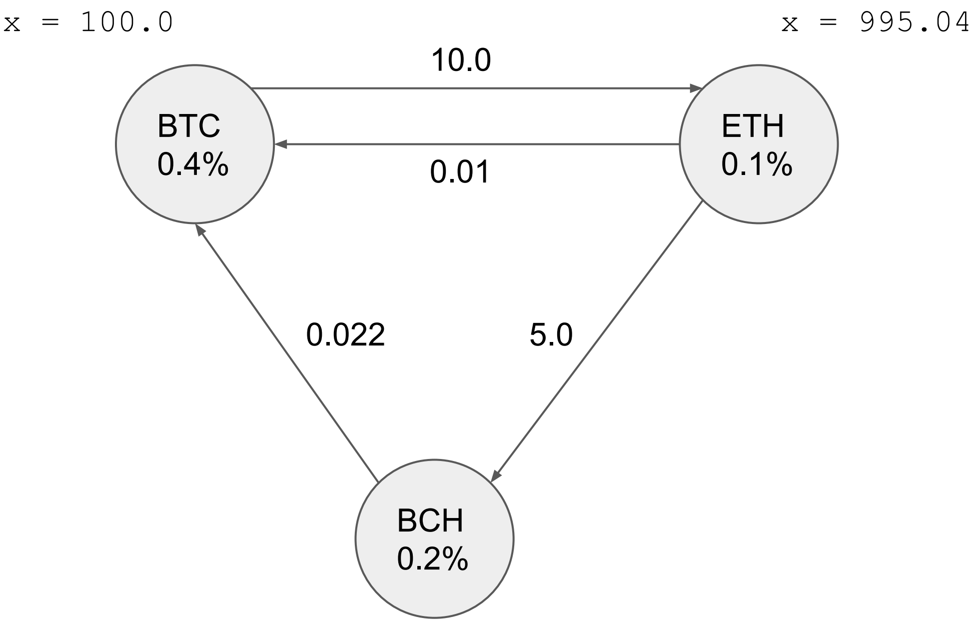
Example Input (profitable):

```
BTC 0.004
ETH 0.001
BCH 0.002

BTC ETH 10.0
ETH BTC 0.01
ETH BCH 5.0
BCH BTC 0.022
```

Example Output (profitable):

```
Trade status: 1 // Profit!
```



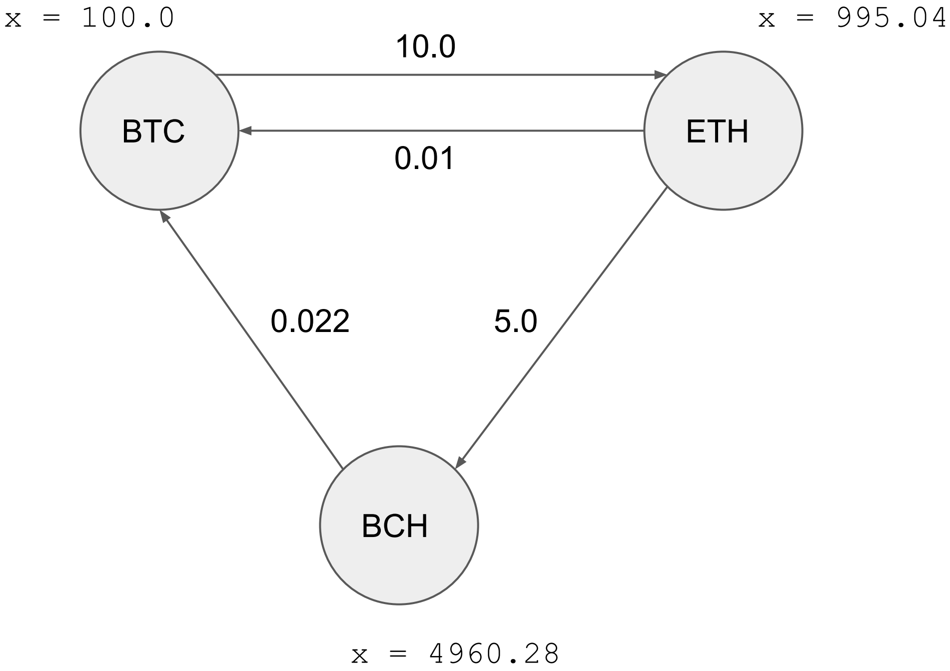
Example Input (profitable):

```
BTC 0.004
ETH 0.001
BCH 0.002

BTC ETH 10.0
ETH BTC 0.01
ETH BCH 5.0
BCH BTC 0.022
```

Example Output (profitable):

```
Trade status: 1 // Profit!
```



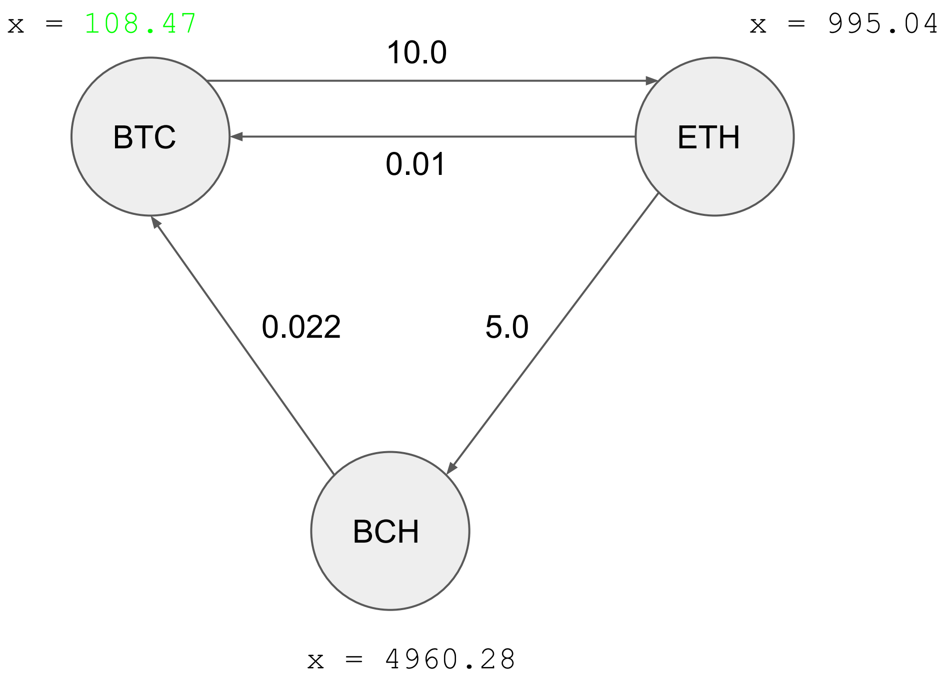
Example Input (profitable):

```
BTC 0.004
ETH 0.001
BCH 0.002

BTC ETH 10.0
ETH BTC 0.01
ETH BCH 5.0
BCH BTC 0.022
```

Example Output (profitable):

```
Trade status: 1 // Profit!
```



Q2 :: Bitvest

Build-your-own graph

```
std::list<Exchange> exchanges

struct Exchange {
   Exchanges(std::string in, std::string out, float rate){
       this->in = in;
       this->out = out;
       this->rate = rate;
   }
   std::string in;
   std::string out;
   float rate;
};
```

Ticker is a simple struct that simply contains information about an available trade.

- “in” specifies the currency the trader seeks
- “out” specifies the currency the trader has
- “rate” specifies the exchange rate of the currency

Graph = (V, E)

V? E?

```
g.vertices[/* id of vertex */] = new Vertex<std::string>(/* id of vertex */, /* weight of vertex */);
```

Q2 :: Bitvest

Build-your-own graph

```
std::list<Exchange> exchanges

struct Exchange {
   Exchanges(std::string in, std::string out, float rate){
        this->in = in;
        this->out = out;
        this->rate = rate;
   }
   std::string in;
   std::string out;
   float rate;
};

g.vertices[/* id of vertex */] = new Vertex<std::string>(/* id of vertex */, /* weight of vertex */);
```

Making a helper function to build your graph will simplify your code!

Q2 :: Bitvest

Fees

```
std::map<std::string, float> fees
```

- Map from currency name to flat fraction fee (not percentage)

Questions?