Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação

Trabalho Prático 1

Algoritmos de Busca - Arbitragem com Moedas

Disciplina: SCC0530 – Inteligência Artificial
Profª. Dra. Solange Oliveira Rezende

Aluno	**NUSP**
Danilo Zecchin Nery	8602430
Felipe Scrochio Custódio	9442688
Caroline Jesuíno Nunes da Silva	9293925
Henrique Martins Loschiavo	8936972
Henrique F.M. Freitas	8937225
Isadora Maria Mendes de Souza	8479318

Índice

Introdução

Inteligência Artificial (IA) é uma subdivisão da área de Ciências de Computação que busca fazer com que máquinas realizem tarefas mimetizando a abordagem de solução que seria usada pela inteligência humana.

No escopo deste curso foram apresentadas aplicações da IA e alguns métodos e algoritmos comumente usados para solução de problemas na área, focando-se, nesta primeira parte, em buscas.

Nossa tarefa neste trabalho foi, então, aplicar um dos algoritmos discutidos em sala de aula e algum outro não apresentado (porém baseado em um dos tradicionais) para resolver um problema a nossa escolha, com espaço de busca grande. Escolhemos resolver o problema de Arbitragem de Moedas através de três algoritmos: Floyd-Warshall, Bellman-Ford e Shortest Paths Faster Algorithm (SPFA).

Como pudemos escolher qualquer linguagem de programação, escolhemos a linguagem R e, para a visualização dos resultados, Javascript.

Por fim, tivemos de comparar os algoritmos entre si, baseando-nos em seus tempos de execução.

Definição do Problema

Arbitragem é a estratégia de comprar algo em um mercado e simultaneamente vender em outro a um preço maior, obtendo lucro a partir dessa diferença temporária de preços. É possível encontrar oportunidades lucrativas de arbitragem em comércio de moedas, convertendo seu dinheiro original a partir de uma sequência de câmbio que resulta em lucro na moeda original.

Essa sequência pode ser encontrada a partir de um laço em um grafo direcionado, onde os nós são as moedas e as arestas são a taxa de conversão entre elas.

Exemplo de exploração:

	USD	**EUR**	**JPY**	**BTC**
USD	-	0.7779	102.4590	0.0083
EUR	1.2851	-	131.7110	0.01125
JPY	0.0098	0.0075	-	0.0000811
BTC	115.65	88.8499	12325.44	-

Valor original: **$100.00000**

- Converter $100 para €77.79
- Converter €77.79 para .8751375 BTC
- Converter .8751375 BTC para $101.20965.

Valor após ciclo: **$101.20965**

A seguir explicaremos a caracterização como um problema de busca, as estratégias de solução e pré-processamento de dados usadas e os algoritmos aplicados.

Implementação

Caracterização do problema de busca

O espaço de busca é um grafo complexo onde nós representam modas ou ativos financeiros e arestas representam taxas de conversão entre cada par de ativos financeiros. Dado um par financeiro A/B, por exemplo, com cada ativo sendo representado por um nó (A e B), a aresta E(A, B), ou seja, a conexão direcionada A->B, tem peso igual ao *bid* ou valor de venda do par (o máximo de B pode ser obtido pela venda de 1 unidade de A). A aresta E(B, A), por sua vez, terá peso igual ao *bid* do par B/A (o máximo de A que pode ser obtido pela venda de 1 unidade de B). Observe que:

$$Bid\ A/B = \frac{1}{(Ask\ B/A)}$$
$$Ask\ A/B = \frac{1}{(Bid\ B/A)}$$

Em nossa aplicação, procuramos nesse grafo um ciclo do tipo A->B->C->A, com qualquer número factível de passos, tal que o produto dos pesos ao se percorrer esse ciclo seja maior que 1. Em outras palavras, um ciclo tal que, começando com 1 unidade de A, façamos as transações descritas pelas arestas e terminemos com mais de 1 A.

Seja P(A,B) o peso da aresta entre os nós A e B, ou seja, o valor de venda (*bid*) do par de ativos A/B, e tomando como exemplo o ciclo infinitamente otimizável de exemplo do parágrafo anterior; se buscamos um ciclo tal que:

$$1 * P(A,B) * P(B,C) * P(C,A) = P(A,B) * P(B,C) * P(C,A) > 1$$

Temos, então, que a seguinte manipulação é válida:

$$P(A,B) * P(B,C) * P(C,A) > 1$$
$$log(P(A,B) * P(B,C) * P(C,A)) > log(1)$$
$$log(P(A,B)) + log(P(B,C)) + log(P(C,A)) > 0$$
$$-(log(P(A,B)) + log(P(B,C)) + log(P(C,A))) < 0$$
$$-log(P(A,B)) - log(P(B,C)) - log(P(C,A)) < 0$$

É portanto, um movimento legal reescrevermos os pesos da seguinte forma:

$$R(A,B) = -\log(P(A,B)),\ \textit{para qualquer par de nós } (A,B)$$

Assim estabelecemos um problema válido de **busca por ciclos negativos.** Utilizamos algoritmos de busca de todos para todos outros nós, e também de um para todos, com otimizações ao longo do caminho. Especificamente, foram aplicados os algoritmos de *Floyd-Warshall, Bellman-Ford* e *Shortest Paths Faster Algorithm,* que é por sua vez uma boa melhoria sobre o caso médio de performance do *Bellman-Ford.*

Algoritmos e estratégias

Floyd-Warshall

O algoritmo de Floyd-Warshall faz buscas de todos os nós no grafo para todos os outros e calcula uma **matriz de distâncias mínimas.** Seu funcionamento generaliza e estende o do algoritmo de Dijkstra, que é um procedimento de busca bem explorado no contexto da disciplina de Inteligência artificial.

Seja V o conjunto de nós do nosso grafo reconstruído, e $weight(i,j)$ o peso da aresta entre os nós **i** e **j,** o algoritmo é o seguinte:

```
1  Create a |V| x |V| matrix, M, that will describe the distances between vertices
2  For each cell (i, j) in M:
3     if i == j:
4        M[i][j] = 0
5     if (i, j) is an edge in E:
6        M[i][j] = weight(i, j)
7     else:
8        M[i][j] = infinity
9  for k from 1 to |V|:
10    for i from 1 to |V|:
11       for j from 1 to |V|:
12          if M[i][j] > M[i][k] + M[k][j]:
13             M[i][j] = M[i][k] + M[k][j]
```

Ao observarmos a diagonal da matriz M, dado o funcionamento do algoritmo, se tivermos um valor negativo, ou seja, uma distância ótima negativa de um nó para ele mesmo, fica claro que tal nó participa de um ciclo negativo e portanto infinitamente otimizável. Podemos então reconstruir esse ciclo e calcular o lucro teórico possível, caminhando pelos nós do ciclo no grafo original (com pesos em valor monetário).

A complexidade de tempo média do algoritmo de Floyd-Warshall é de $O(|V|^3))$, com V sendo, de novo, o conjunto de nós (ou seja, a cardinalidade do conjunto elevado ao cubo), por conta dos 3 laços aninhados. Não há nenhum fator de redução gradativa do espaço de busca do algoritmo e, portanto, é uma aplicação consistentemente lenta.

Bellman-Ford

O algoritmo de Bellman-Ford propõe uma busca de um nó no grafo para todos os outros, resultando em um **vetor de distâncias mínimas.** Na presença de pesos negativos nas arestas, ele é particularmente útil na detecção de ciclos negativos.

Dado que o nó de origem é único, precisamos adaptar o grafo antes: adicionamos um nó falso $v0$ no grafo, e posicionamos arestas direcionadas partindo dele para todos os outros nós, todas com peso 0. Executando o processo a partir de $v0$, temos a garantia de poder alcançar ciclos negativos originados em qualquer outro nó e, ao mesmo tempo, não introduzindo nenhum novo ciclo (ou seja, a topologia original é mantida). Temos o pseudocódigo:

```
1  for v in V:
2      v.distance = infinity
3      v.p = None
4  source.distance = 0
5  for i from 1 to |V| - 1:
6      for (u, v) in E:
7          relax(u, v)
8  for (u, v) in E:
9      if v.distance > u.distance + weight(u, v):
10         print "A negative weight cycle exists"
```

Onde a função de relaxamento das arestas é dada por:

```
1  relax(u, v):
2     if v.distance > u.distance + weight(u, v):
3        v.distance = u.distance + weight(u, v)
4        v.p = u
```

O uso do Bellman-Ford em comparação com Floyd-Warshall é importante pois o anterior tem complexidade de tempo $O(|V| * |E|)$, onde V e E são os conjuntos de nós e arestas, respectivamente. A melhoria em relação ao caso anterior é grande.

Otimização: fila de nós candidatos (SPFA)

A proposta do Bellman-Ford clássico ainda oferece boas oportunidades de otimização. Em particular, o uso de uma **fila de nós candidatos à exploração** no passo seguinte, já reduz severamente a performance do caso geral do algoritmo.

A estratégia é, a cada passo, enfileirar os nós vizinhos cujas arestas incidentes foram relaxadas no passo atual, e continuar a exploração até que esta fila esteja vazia OU até que o número de iterações máximas (ou seja, o número de vértices, no caso tradicional do Bellman-Ford). O pseudocódigo segue:

```
procedure Shortest-Path-Faster-Algorithm(G, s)
 1  for each vertex v ≠ s in V(G)
 2     d(v) := ∞
 3  d(s) := 0
 4  offer s into Q
 5  while Q is not empty
 6     u := poll Q
 7     for each edge (u, v) in E(G)
 8        if d(u) + w(u, v) < d(v) then
 9           d(v) := d(u) + w(u, v)
10           if v is not in Q then
11              offer v into Q
```

Vale notar que o limite de iterações não é checado no código acima, isso porque a princípio assume-se que não existem arestas com peso negativo e, portanto, nenhum caminho infinitamente otimizável. Fizemos o ajuste devido em nossa implementação.

Uma otimização adicional em cima do já proposto é, a cada passo, explorar os nós que estão mais próximos da origem da busca (imitando uma expansão radial a partir do ponto inicial). Isso implica priorizarmos os elementos que são retirados de Q por suas distâncias d. Essa otimização é chamada de *Small Labels First* (SLF) e, para implementá-la, logo após a inserção v em Q incluímos uma chamada ao seguinte procedimento:

```
procedure Small-Label-First(G, Q)
    if d(back(Q)) < d(front(Q)) then
        u := pop back of Q
        push u into front of Q
```

É importante ressaltar que a complexidade de pior caso do SPFA é a **mesma da do algoritmo de Bellman-Ford,** no entanto a performance do caso médio sobe drasticamente, particularmente nos casos onde existem pesos negativos, que se alinha com nosso interesse.

Análise de tempos de Execução

Para cada algoritmo, a média e o desvio padrão dos tempos de 15 execuções foram medidos. Obtivemos a seguinte tabela:

	Tempo médio de execução	**Desvio padrão**
Floyd-Warshall	3.54773842	0.68704999
Bellman-Ford	1.40677368	0.02638175
SPFA	0.01104989	0.01035597

Observe que o tempo de execução dos algoritmos supracitados distam consideravelmente entre si.

O algoritmo Floyd-Warshall é claramente mais lento, por causa de sua complexidade cúbica $(O(|V|^3),\ tal\ que\ V\ é\ o\ número\ de\ nós)$; o algoritmo Bellman-Ford, em comparação, já é significativamente mais rápido, tendo complexidade $O(|V| * |E|)$. O *Shortest Paths Faster Algorithm*, por sua vez, é muito mais rápido, mesmo sem estabelecer uma complexidade assintótica menor (no pior caso é igual ao

Bellman-Ford). Esse comportamento é, de fato, esperado pois o algoritmo limita consideravelmente o espaço de busca possível a cada passo.

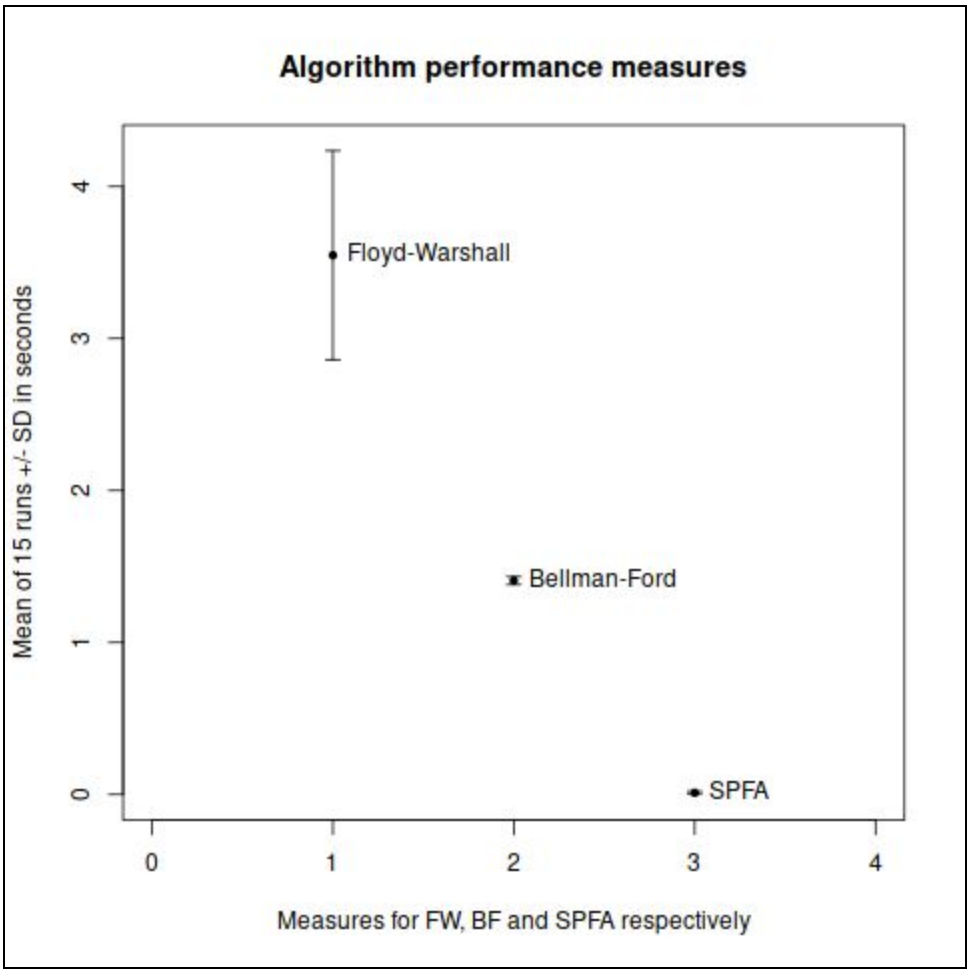

Gráfico 1: A média de tempo de execução dos algoritmos é representada em segundos, no eixo y, pelos pontos, enquanto as linhas medem o desvio padrão (para cima ou para baixo). O eixo x apenas serve para indicar qual é o algoritmo representado (1: Floyd-Warshall, 2: Bellman-Ford, 3: SPFA).

Vídeo de apresentação

O vídeo de apresentação dá uma introdução ao problema de forma bem visual e simples, mostrando também a execução dos três algoritmos.

https://youtu.be/0HxWQiEfiss

Execução do programa

Configuração do ambiente R

Para implementação, a linguagem de escolha foi R. Caso já não tenha instalado no sistema, o CRAN da instituição Case Western é uma escolha confiável e usamos diversas vezes no desenvolvimento do projeto: https://cran.case.edu/ .

Adicionalmente, usamos alguns pacotes externos na implementação, também é necessário instalá-los para execução do script. Isso pode ser feito (sem necessidade de direitos administrativos) via execução do script **`install-packages.r`**:

```
Rscript --vanilla install-packages.r
```

Exemplos de execução e saídas

Executando o script principal com a `flag --help` ou simplesmente sem passar argumentos forçará o programa a exibir instruções sobre os argumentos passados:

```
Rscript main.r --help
```

Execução com escolha do algoritmo de busca

A aplicação do algoritmo Bellman-Ford sobre o dataset de entrada `summary_btx_30-04.txt` gerando um padrão de saída out é dada pelo comando:

```
Rscript --vanilla main.r -i datasets/summary_btx_30-04.txt -o out -a bf
```

Arquivos gerados

Arquivo out.nodes: todos os nós exclusivos grafo
Arquivo out.cycle: as arestas que compõem o ciclo negativo, se algum existir
Arquivo out.summary: um sumário da execução do algoritmo, em formato legível por humanos

Execução do benchmark de todos os algoritmos

A aplicação de todos os algoritmos e as médias e desvios padrões de 15 execuções de cada um, sobre o dataset de entrada `summary_btx_30-04.txt` é dada pelo comando:

```
Rscript --vanilla main.r -i datasets/summary_btx_30-04.txt --benchmark
```

Arquivos gerados

Arquivo out.plot: um gráfico das médias e desvios padrões dos tempos de execução de todos os algoritmos, como apresentado na seção de análise
Arquivo out.summary: um sumário da execução do algoritmo, em formato legível por humanos

Casos de exemplo adicionais

A aplicação de detecção de ciclos sobre o dataset de entrada `summary_btx_30-04.txt` usando o algoritmo de Floyd-Warshall gerando padrão de saída out é dado por:

```
Rscript --vanilla main.r -i datasets/summary_btx_30-04.txt -o out -a fw
```

A aplicação de detecção de ciclos sobre o dataset de entrada `summary_btx_30-04.txt` dataset usando o SPFA gerando padrão de saída out é dado por:

```
Rscript --vanilla main.r -i datasets/summary_btx_30-04.txt -o out -a spfa
```

Visualização

Implementamos uma visualização para deixar o problema mais fácil de entender, permitindo ver geograficamente quais moedas foram visitadas durante as conversões. Porém, a *API* para obter dados de *Foreign Exchange (FOREX)* possuía um limite para a versão gratuita, e nossas *requests* necessárias ultrapassaram o limite. Deixamos a visualização no código com um ciclo *dummy* para mostrar como funcionaria. Para acessá-la, apenas abrir o arquivo **index.html** na pasta **visualization.**

Cada moeda foi representada por seu país de origem no mapa, as arestas entre eles são criadas quando o algoritmo percorre esse caminho. A animação permite visualizar todo o caminho que o algoritmo percorreu até encontrar o ciclo, deixando a aresta atual com a cor mais forte a cada vez que passa por ela. Assim, o ciclo fica bem visível no final.

Nossa implementação dos algoritmos de busca gera um arquivo com as moedas visitadas. No pré-processamento da visualização, geramos o caminho comparando a saída do programa com um dataset de países e suas moedas.

Implementação feita utilizando Javascript com a biblioteca P5.js, com datasets de latitude e longitude dos países e suas moedas, com o mapa gerado utilizando Mapbox.

Figura 1: screenshot da visualização rodando no browser.

Fontes

“Bellman-Ford Algorithm.” *Brilliant Math & Science Wiki*, brilliant.org/wiki/bellman-ford-algorithm/.

“Bitcoin Arbitrage.” *Priceonomics*, priceonomics.com/jobs/puzzle/.

“Floyd-Warshall Algorithm.” *Brilliant Math & Science Wiki*, brilliant.org/wiki/floyd-warshall-algorithm/.

“Shortest Path Faster Algorithm.” *Wikipedia*, Wikimedia Foundation, 30 Apr. 2018, en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm.

Staff, Investopedia. “What Is Arbitrage?” *Investopedia*, Investopedia, 1 Mar. 2018, www.investopedia.com/ask/answers/04/041504.asp.