Network Flow Problems

Stefano Gualandi
Università di Pavia, Dipartimento di Matematica

email: stefano.gualandi@unipv.it
twitter: @famo2spaghi
blog: http://stegua.github.com

Network Flow Problems

Given a network $G = (N, A)$, with arc cost c_{ij} and flow node balance b_i, with $\sum_{i \in N} b_i = 0$, the minimum cost flow problem is

$$\min \quad \sum_{ij \in A} c_{ij} x_{ij} \tag{1}$$

$$\text{s.t.} \quad \sum_{(j,i) \in A} x_{ji} - \sum_{(i,j) \in A} x_{ij} = -b_i, \qquad \forall i \in N, \tag{2}$$

$$x_{ij} \geq 0. \tag{3}$$

Network Flow Problems

Given a network $G = (N, A)$, with arc cost c_{ij} and flow node balance b_i, with $\sum_{i \in N} b_i = 0$, the minimum cost flow problem is

$$\min \quad \sum_{ij \in A} c_{ij} x_{ij} \tag{1}$$

$$\text{s.t.} \quad \sum_{(j,i) \in A} x_{ji} - \sum_{(i,j) \in A} x_{ij} = -b_i, \qquad \forall i \in N, \tag{2}$$

$$x_{ij} \geq 0. \tag{3}$$

This LP has $n = |N|$ rows and $m = |A|$ columns. The coefficient matrix is a *node-arc incidence matrix*.

Network Flow Problems

Given a network $G = (N, A)$, with arc cost c_{ij} and flow node balance b_i, with $\sum_{i \in N} b_i = 0$, the minimum cost flow problem is

$$\min \quad \sum_{ij \in A} c_{ij} x_{ij} \tag{1}$$

$$\text{s.t.} \quad \sum_{(j,i) \in A} x_{ji} - \sum_{(i,j) \in A} x_{ij} = -b_i, \qquad \forall i \in N, \tag{2}$$

$$x_{ij} \geq 0. \tag{3}$$

This LP has $n = |N|$ rows and $m = |A|$ columns. The coefficient matrix is a *node-arc incidence matrix*.

Depending on the structure of G and the value of b_i, we can formulate different flow problems: Transshipment, Transportation, Hitchcock, Assignment, Shortest Path, Max Flow, Minimum Cost Flow.

Labeling Algorithms

Let us denote the distance from node i to the root r by the *label* v_i.

Labeling Algorithms

Let us denote the distance from node i to the root r by the *label* v_i.
There exists two main type of algorithms:

1. Label-Correcting: the labels represent *approximate* distance to the root and they are iteratively improved until they reach the optimal distances.
2. Label-Setting: the labels are set in systematically order in such a way that at they they all represent the shortest distance to the root.

Labeling-Setting Algorithm

An example of label-setting algorithm is the Dijkstra's algorithm.

Labeling-Setting Algorithm

An example of label-setting algorithm is the Dijkstra's algorithm.

The algorithm use a data structure $\mathcal{F}$ of *finished nodes*, which at the begging is empty, and another $\mathcal{F}^c$ for the *open nodes*. Later, the algorithm, once at the time, remove a node from $\mathcal{F}^c$ that goes into $\mathcal{F}$, while **setting** the correct label v_i one at the time.

Labeling-Setting Algorithm

An example of label-setting algorithm is the Dijkstra's algorithm.

The algorithm use a data structure $\mathcal{F}$ of *finished nodes*, which at the begging is empty, and another $\mathcal{F}^c$ for the *open nodes*. Later, the algorithm, once at the time, remove a node from $\mathcal{F}^c$ that goes into $\mathcal{F}$, while **setting** the correct label v_i one at the time.

Algorithm 3: Labeling-Setting: (Reversed) Dijkstra's Algorithm

Data: Network $G = (N, A)$, cost vector $\boldsymbol{c}$, root r
Result: v_i optimal labels (function value)

$v_r \leftarrow 0, \quad v_i \leftarrow \infty, \ \forall i \in N \setminus \{r\}$ Initial label values;
$\mathcal{F} \leftarrow \{\}, \quad \mathcal{F}^c \leftarrow N$ Two sets of nodes;
while $\mathcal{F}^c \neq \emptyset$ **do**
    $j \leftarrow \arg\min\{v_k \mid k \in \mathcal{F}^c\}$;
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{j\}, \quad \mathcal{F}^c \leftarrow \mathcal{F}^c \setminus \{j\}$;
    for $(i, j) \in A$ *and* $i \in \mathcal{F}^c$ **do**
        if $v_i > c_{ij} + v_j$ **then**
            $v_i \leftarrow c_{ij} + v_j$;
            $p_j \leftarrow i$;

Labeling-Setting Algorithm

An example of label-setting algorithm is the Dijkstra's algorithm.

The algorithm use a data structure $\mathcal{F}$ of *finished nodes*, which at the begging is empty, and another $\mathcal{F}^c$ for the *open nodes*. Later, the algorithm, once at the time, remove a node from $\mathcal{F}^c$ that goes into $\mathcal{F}$, while **setting** the correct label v_i one at the time.

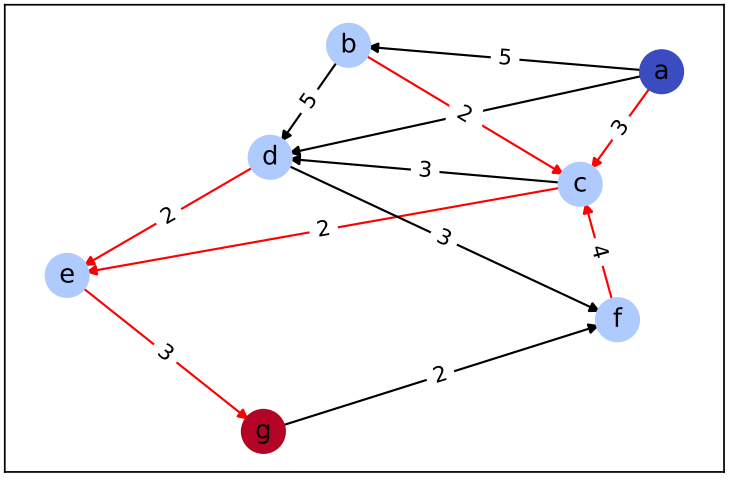
Algorithm 4: Labeling-Setting: (Reversed) Dijkstra's Algorithm

Data: Network $G = (N, A)$, cost vector $\boldsymbol{c}$, root r
Result: v_i optimal labels (function value)

$v_r \leftarrow 0, \quad v_i \leftarrow \infty, \forall i \in N \setminus \{r\}$ Initial label values;
$\mathcal{F} \leftarrow \{\}, \quad \mathcal{F}^c \leftarrow N$ Two sets of nodes;
while $\mathcal{F}^c \neq \emptyset$ **do**
  $j \leftarrow \arg\min\{v_k \mid k \in \mathcal{F}^c\}$;
  $\mathcal{F} \leftarrow \mathcal{F} \cup \{j\}, \quad \mathcal{F}^c \leftarrow \mathcal{F}^c \setminus \{j\}$;
  for $(i, j) \in A$ *and* $i \in \mathcal{F}^c$ **do**
    if $v_i > c_{ij} + v_j$ **then**
      $v_i \leftarrow c_{ij} + v_j$;
      $p_j \leftarrow i$;



Example: Apply Dijkstra's algorithm by Hand



Example: Apply Dijkstra's algorithm by Hand

```
Fc: {'e': 100, 'b': 100, 'g': 0, 'c': 100, 'f': 100, 'd': 100, 'a': 100}
argmin{Fc} = g
update label V[e] = 3

Fc: {'e': 3, 'b': 100, 'c': 100, 'f': 100, 'd': 100, 'a': 100}
argmin{Fc} = e
update label V[c] = 5, update label V[d] = 5

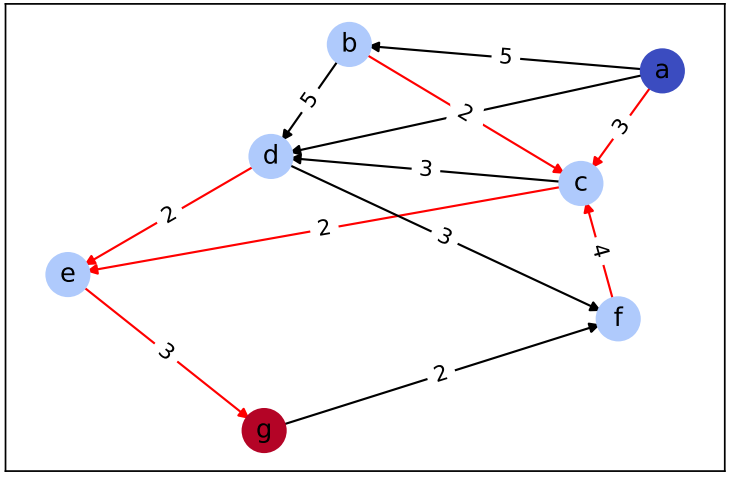
Fc: {'b': 100, 'c': 5, 'f': 100, 'd': 5, 'a': 100}
argmin{Fc} = c
update label V[a] = 8, update label V[b] = 7
update label V[f] = 9

Fc: {'b': 7, 'f': 9, 'd': 5, 'a': 8}
argmin{Fc} = d

Fc: {'b': 7, 'f': 9, 'a': 8}
argmin{Fc} = b

Fc: {'f': 9, 'a': 8}
argmin{Fc} = a

Fc: {'f': 9}
argmin{Fc} = f
```




```
Succes: {'e': 'g', 'b': 'c', 'g': None, 'c': 'e', 'f': 'c', 'd': 'e', 'a': 'c'}
Labels: {'e': 3, 'b': 7, 'g': 0, 'c': 5, 'f': 9, 'd': 5, 'a': 8}
```

Question: Python Implementation?

Question: Python Implementation?

Question: Python Implementation?

First step: Graph Data structure

Question: Python Implementation?

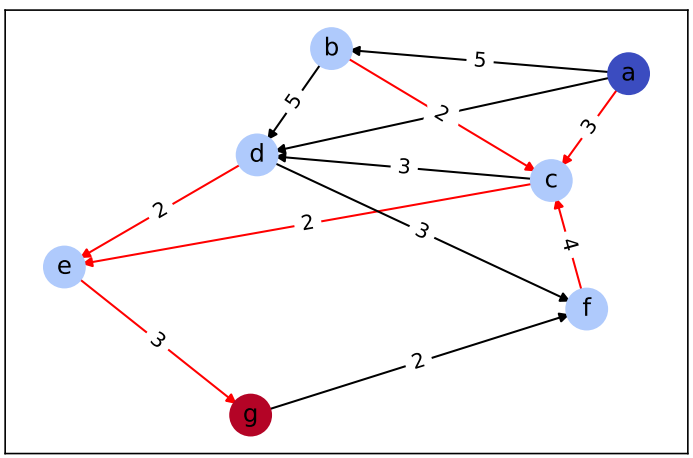
First step: Graph Data structure

```
1   class Digraph(object):
2       def __init__(self):
3           self.N = set()  # Set of nodes (see builtin doc online for SET)
4           self.A = {}     # Double dictionary of arcs: Type: {tail} -> {head} ->c_ij
5
6       def add_arc(self, v, w, c):
7           """ Store Backward star (instead of Forward star) """
8           self.N.add(v)
9           self.N.add(w)
10
11          if w not in self.A:
12              self.A[w] = {}
13          self.A[w][v] = c
14
15      def get_cost(self, v, w):
16          if w not in self.A or v not in self.A[w]:
17              return Inf   # E.g.: Inf = 10**9
18          return self.A[w][v]
19
20      def __str__(self):
21          s = ""
22          for w in self.A:
23              s = s + "{}: ".format(w)
24              for v in self.A[w]:
25                  s = s +"({},{})#{} ".format(v, w, self.A[w][v])
26              s = s + "\n"
27          return s
```

Question: Python Implementation?

Question: Python Implementation?

```
1   # Static description of arcs
2   Ls = [('a', 'b', 5), ('a', 'c', 3), ('a', 'd', 3), ('b', 'c', 2),
3         ('b', 'd', 5), ('c', 'e', 2), ('c', 'd', 3), ('d', 'e', 2),
4         ('d', 'f', 3), ('e', 'g', 3), ('f', 'c', 4), ('g', 'f', 2)]
5
6   # Create digraph
7   graph = Digraph()
8
9   # Add arcs to G
10  for v, w, c in Ls:
11      graph.add_arc(v, w, c)
12
13  print(graph)
14
15  P, V = ReverseDijkstra(graph, 'g')
16
17  print("Succes:", P)
18  print("Labels:", V)
```



Question: Python Implementation?

```
1  def ReverseDijkstra(graph, r):
2      P = dict([(v, None) for v in graph.N])       # Successors
3      V = dict([(v, Inf) for v in graph.N])        # Labels
4      V[r] = 0
5      # Open Nodes
6      Fc = dict([(v, V[v]) for v in V])
7      # Main Loop
8      while len(Fc) > 0:
9          print('\nFc:', Fc)
10         j = ArgMinRemove(Fc)
11         print('argmin{Fc} = ', j)
12         if j in graph.A:
13             for i in graph.A[j]:
14                 if V[i] > graph.A[j][i] + V[j]:
15                     print('update label V[{}] = {}'.format(i, graph.A[j][i]+V[j]))
16                     V[i] = graph.A[j][i] + V[j]
17                     Fc[i] = graph.A[j][i] + V[j]
18                     P[i] = j
19     return P, V
```

Question: Python Implementation?

```python
def ReverseDijkstra(graph, r):
    P = dict([(v, None) for v in graph.N])      # Successors
    V = dict([(v, Inf) for v in graph.N])       # Labels
    V[r] = 0
    # Open Nodes
    Fc = dict([(v, V[v]) for v in V])
    # Main Loop
    while len(Fc) > 0:
        print('\nFc:', Fc)
        j = ArgMinRemove(Fc)
        print('argmin{Fc} =', j)
        if j in graph.A:
            for i in graph.A[j]:
                if V[i] > graph.A[j][i] + V[j]:
                    print('update label V[{}] = {}'.format(i, graph.A[j][i]+V[j]))
                    V[i] = graph.A[j][i] + V[j]
                    Fc[i] = graph.A[j][i] + V[j]
                    P[i] = j
    return P, V
```

```python
def ArgMinRemove(Q):
    keys = sorted(Q)
    j, d = keys[0], Q[keys[0]]
    for v in keys[1:]:
        if Q[v] < d:
            j, d = v, Q[v]
    del Q[j]
    return j
```

Dijkstra's Efficiency: Worst-cast Running Time

In the initialization phase, we perform $O(n)$ elementary operations (one per node).

Dijkstra's algo implementation in the NetworkX python library:
https://networkx.guide/algorithms/shortest-path/dijkstra/

Dijkstra's Efficiency: Worst-cast Running Time

In the initialization phase, we perform $O(n)$ elementary operations (one per node).

The `while` loop is performed at most n times (move i from $\mathcal{F}^c$ to $\mathcal{F}$).

Dijkstra's algo implementation in the NetworkX python library:
https://networkx.guide/algorithms/shortest-path/dijkstra/

Dijkstra's Efficiency: Worst-cast Running Time

In the initialization phase, we perform $O(n)$ elementary operations (one per node).

The `while` loop is performed at most n times (move i from $\mathcal{F}^c$ to $\mathcal{F}$).

At each iteration, we will perform at most n selections from set $\mathcal{F}^c$ and overall at most m controls on arcs, label updates and insertions into $\mathcal{F}^c$ (every arc is considered only once).

Dijkstra's algo implementation in the NetworkX python library:
https://networkx.guide/algorithms/shortest-path/dijkstra/

Dijkstra's Efficiency: Worst-cast Running Time

In the initialization phase, we perform $O(n)$ elementary operations (one per node).

The `while` loop is performed at most n times (move i from $\mathcal{F}^c$ to $\mathcal{F}$).

At each iteration, we will perform at most n selections from set $\mathcal{F}^c$ and overall at most m controls on arcs, label updates and insertions into $\mathcal{F}^c$ (every arc is considered only once).

Therefore, everything depends on the data structures used to implement the set $\mathcal{F}^c$:

1. If an unordered list is used, the selection of the smallest label node implies a scanning of the entire list, so its complexity is $O(n)$, whereas the insertion is made into $O(1)$. Hence, the global complexity is $O(n^2 + m) = O(n^2)$.

Dijkstra's algo implementation in the NetworkX python library:
https://networkx.guide/algorithms/shortest-path/dijkstra/

Dijkstra's Efficiency: Worst-cast Running Time

In the initialization phase, we perform $O(n)$ elementary operations (one per node).

The `while` loop is performed at most n times (move i from $\mathcal{F}^c$ to $\mathcal{F}$).

At each iteration, we will perform at most n selections from set $\mathcal{F}^c$ and overall at most m controls on arcs, label updates and insertions into $\mathcal{F}^c$ (every arc is considered only once).

Therefore, everything depends on the data structures used to implement the set $\mathcal{F}^c$:

1. If an unordered list is used, the selection of the smallest label node implies a scanning of the entire list, so its complexity is $O(n)$, whereas the insertion is made into $O(1)$. Hence, the global complexity is $O(n^2 + m) = O(n^2)$.

2. If we use a priority queue, e.g. a heap di Fibonacci, the selection complexity is $O(1)$, whereas removal, insertion and update are $O(\log n)$. Globally, we will have $O(m + n \log n) = O(n \log n)$.

Dijkstra's Efficiency: Worst-cast Running Time

In the initialization phase, we perform $O(n)$ elementary operations (one per node).

The `while` loop is performed at most n times (move i from $\mathcal{F}^c$ to $\mathcal{F}$).

At each iteration, we will perform at most n selections from set $\mathcal{F}^c$ and overall at most m controls on arcs, label updates and insertions into $\mathcal{F}^c$ (every arc is considered only once).

Therefore, everything depends on the data structures used to implement the set $\mathcal{F}^c$:

1. If an unordered list is used, the selection of the smallest label node implies a scanning of the entire list, so its complexity is $O(n)$, whereas the insertion is made into $O(1)$. Hence, the global complexity is $O(n^2 + m) = O(n^2)$.

2. If we use a priority queue, e.g. a heap di Fibonacci, the selection complexity is $O(1)$, whereas removal, insertion and update are $O(\log n)$. Globally, we will have $O(m + n \log n) = O(n \log n)$.

Dijkstra's algo implementation in the NetworkX python library:
https://networkx.guide/algorithms/shortest-path/dijkstra/

DNA Sequence Alignment

The DNA can be viewed as a string based on the alphabet $\Sigma = \{A, C, G, T\}$. Given two sequences of DNA denoted by $B = b_1 b_2 \dots b_p$ and $D = d_1 d_2 \dots d_q$, we are interested in measuring their similarity.

DNA Sequence Alignment

The DNA can be viewed as a string based on the alphabet $\Sigma = \{A, C, G, T\}$. Given two sequences of DNA denoted by $B = b_1 b_2 \dots b_p$ and $D = d_1 d_2 \dots d_q$, we are interested in measuring their similarity.

A way to measure the (dis-)similarity is to determine the minimum cost required to transform the sequence B into D.

DNA Sequence Alignment

The DNA can be viewed as a string based on the alphabet $\Sigma = \{A, C, G, T\}$. Given two sequences of DNA denoted by $B = b_1 b_2 \dots b_p$ and $D = d_1 d_2 \dots d_q$, we are interested in measuring their similarity.

A way to measure the (dis-)similarity is to determine the minimum cost required to transform the sequence B into D.

To transform B into D we can

1. Insert an element in B at a cost of α units.
2. Delete an element from B at a cost of β units.
3. Mutate an element b_i into an element d_j at a cost $g(b_i, d_j)$.

There exists many ways to transform B into D and it is not trivial to identify the minimum cost transformation. However, we can applied dynamic programming and formulate this problem as a shortest path problem on an appropriately defined network.

Example of DNA Alignment

Consider $B = AGTT$ and $D = CTAGC$. One possible way to transform B into D is to delete one T from B, and adding two elements at the begging, giving the sequence:

$$B = AGTT \quad \Rightarrow \quad B' = \circledast \circledast AGT$$

We then align B' with D and mutate the element T into C, so that the two sequence become identical.

$$\begin{array}{rcc} B' = & \circledast \circledast AGT & CTAGC \\ D = & CTAGC & CTAGC \end{array}$$

The overall cost is

$$\beta + 2\alpha + g(T, C)$$

DNA Alignment: Dynamic Programming

Let $f(i, j)$ denote the minimum cost of transforming the sub-sequence $b_1 b_2 \ldots b_i$ into the sub-sequence $d_1 d_2 \ldots d_j$.

DNA Alignment: Dynamic Programming

Let $f(i,j)$ denote the minimum cost of transforming the sub-sequence $b_1 b_2 \ldots b_i$ into the sub-sequence $d_1 d_2 \ldots d_j$.

The value $f(p,q)$ is the minimum cost of transforming B into D, and it is computed by first computing $f(i,j)$ for all $i = 0, 1, \ldots, p$ and forall $j = 0, 1, \ldots, q$.

DNA Alignment: Dynamic Programming

Let $f(i,j)$ denote the minimum cost of transforming the sub-sequence $b_1 b_2 \ldots b_i$ into the sub-sequence $d_1 d_2 \ldots d_j$.

The value $f(p,q)$ is the minimum cost of transforming B into D, and it is computed by first computing $f(i,j)$ for all $i = 0, 1, \ldots, p$ and forall $j = 0, 1, \ldots, q$.

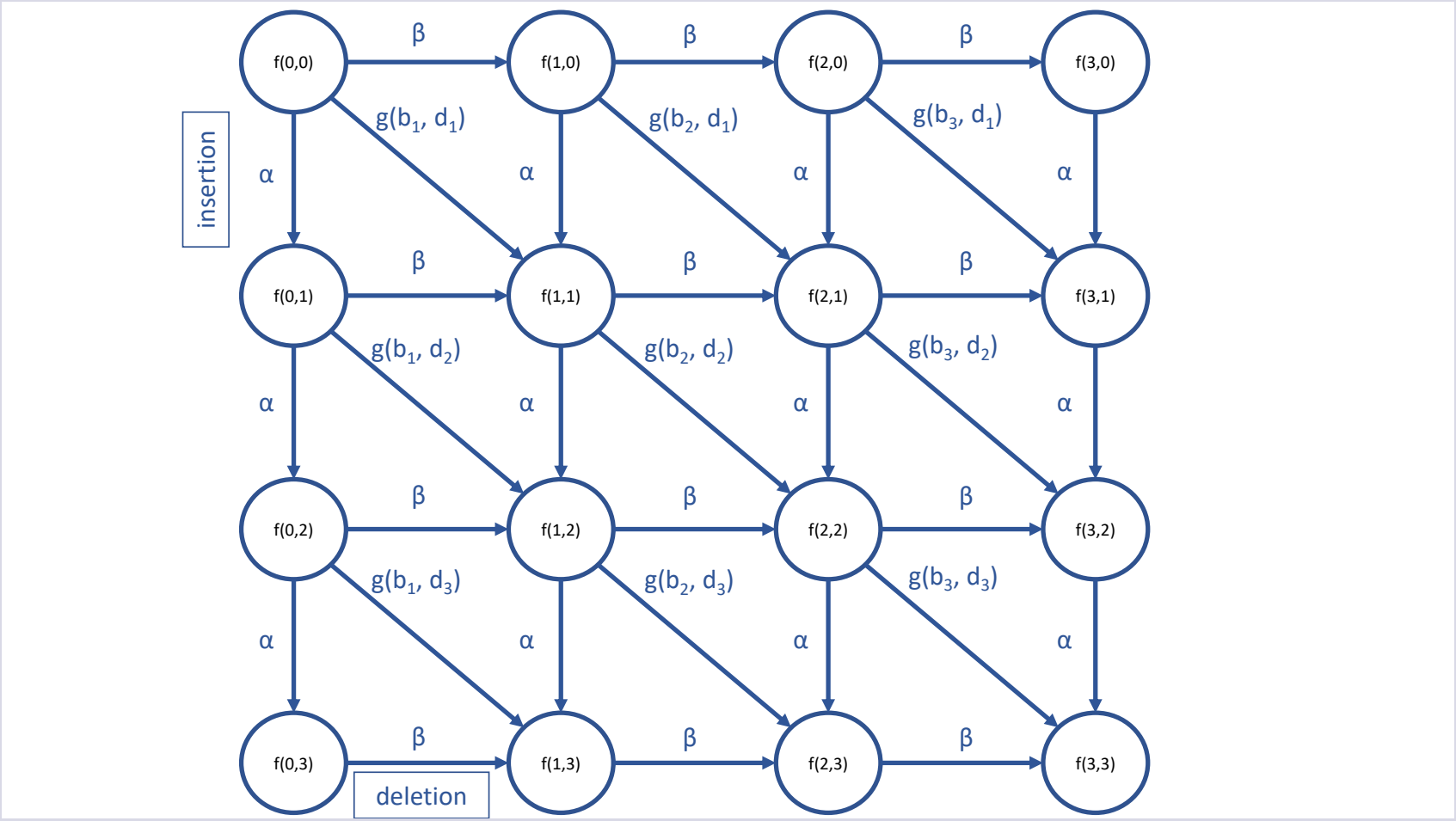
The intermediate $f(i,j)$ are computed as:

- $f(i,0) = \beta i$ for all i
- $f(0,j) = \alpha j$ for all j
- $f(i,j) =$

$$\min \begin{cases} f(i-1,j-1) + g(b_i, d_j) & \text{case: } b_i, d_j \text{ aligned} \\ f(i,j-1) + \alpha & \text{case: } b_i \text{ is not aligned with } d_j \\ f(i-1,j) + \beta & \text{case: } B' \text{ do not contain } b_i. \end{cases}$$

The shortest-path formulation is given in the next slide.

DNA Alignment: Shortest Path



Let $f(i,j)$ denote the minimum cost of transforming the sub-sequence $b_1 b_2 \ldots b_i$ into the sub-sequence $d_1 d_2 \ldots d_j$.

References for Mathematical Modeling

- Chap. 14.1 and 15 in Vanderbei, R.J., 2015. Linear programming. Heidelberg: Springer.
- Chap. 4, 5, 19.4 in Ahuja, R.K., Magnanti, T.L. and Orlin, J.B., 1988. Network flows.