CHAPTER II

BASIC THEORIES

In this chapter, we present basic theories used throughout the *skripsi*. First, we present definitions and basic concepts of multiple sequence alignment (MSA) to give a brief view of an MSA problem. The aim of an MSA is to determine the best alignments. Criteria of a best alignment depend on the purpose of the alignment itself. We present scoring scheme as a tool to score every possible alignments. With the help of scoring scheme, we can determine the best alignments based on the purpose of the alignment. However, finding the best alignments is not straightforward. The MSA problem is solved with integer linear programming. The integer linear programming (ILP) model is derived from the graph representation of the MSA problem. Hence, we also present definitions and basic concepts in graph theory, graph representation of an MSA problem, and definitions and basic concepts in integer linear programming. Solving an ILP model requires a different approach than solving a linear programming model since the solution of an ILP model needs to be integer. We devise a program to generate and solve the ILP model of an MSA problem using MATLAB R2008a. MATLAB R2008a has a function to solve a binary ILP problem using branch-and-bound method. Hence, we present the branch-and-bound method at the end of chapter II.

2.1. DEFINITIONS AND BASIC CONCEPTS OF MSA

Sequence alignment is a process of adjusting sequences so that they are in proper relative position. In general, sequence alignment has the following properties (see Althaus E. *et al*. [1]; Kececioglu J.D. *et al*. [4]):

- The length of each of the aligned sequence must be the same after an alignment.
- Each of the aligned sequence before and after an alignment is identical if gap(s) is (or are) ignored.

In bioinformatics, sequence alignment is a way of arranging the primary sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. **Multiple sequence alignment** (**MSA**) is a sequence alignment of more than two sequences (Sequence alignment of two sequences is called **pairwise sequence alignment**). MSA is often used to identify conserved regions across a group of sequences hypothesized to be evolutionarily related. We mainly discuss MSA of DNA sequences in the *skripsi*. DNA (deoxyribonucleic acid) sequences are associated with four alphabet letter: A (adenine), C (cytosine), G (guanine), and T (thymine) each of the alphabets stands for the nucleic acid of DNA. During the course of evolution, residues of the sequences that evolve from a common ancestor may have undergone **substitutions** (when residues are replaced by other residues), **insertions** (when new residues appear in a sequence in addition to the existing ones), or **deletions** (when some residue disappear).

$$\begin{array}{lllllll}
\bar{s}^1: & \bar{s}_1^1 & \bar{s}_2^1 & \cdots & \bar{s}_j^1 & \cdots & \bar{s}_l^1 \\
\bar{s}^2: & \bar{s}_1^2 & \bar{s}_2^2 & \cdots & \bar{s}_j^2 & \cdots & \bar{s}_l^2 \\
\vdots & \vdots & \vdots & & \vdots & & \vdots \\
\bar{s}^i: & \bar{s}_1^i & \bar{s}_2^i & \cdots & \bar{s}_j^i & \cdots & \bar{s}_l^i \\
\vdots & \vdots & \vdots & & \vdots & \ddots & \vdots \\
\bar{s}^k: & \bar{s}_1^k & \bar{s}_2^k & \cdots & \bar{s}_j^k & \cdots & \bar{s}_l^k
\end{array}$$

Fig. 2.1. Alignment set $\bar{S}$.

Let $S = \{s^1, s^2, \ldots, s^k\}$ be a set of k strings over a DNA alphabet A = {A, C, G, T} and let $\bar{A} = A \cup \{-\}$ where '−' (dash) is a symbol that represents gap. An alignment of S is a set $\bar{S} = \{\bar{s}^1, \bar{s}^2, \ldots, \bar{s}^k\}$ of strings over alphabet $\bar{A}$ where all the strings in $\bar{S}$ have the same length and if the dash is ignored, string $\bar{s}^i$ is identical to string s^i. The length of string s is the number of characters that are in string s, denoted by $|s|$. String s consists of characters s_j, $j = 1, 2, \ldots, |s|$. All strings $\bar{s}^i \in \bar{S}$ in an alignment can be represented as an array of k rows and l columns where row i corresponds to string $\bar{s}^i$. Characters of distinct strings in S are said to be aligned under $\bar{S}$ if they are placed on the same column of the alignment array. Figure 2.1 illustrates a general form of an alignment set $\bar{S}$ of k sequences. Alignment set $\bar{S}$ consists of strings $\bar{s}^i$, $i = 1, 2, \ldots, k$. Every string $\bar{s}^i$ consists of characters $\bar{s}_j^i$, j = 1, 2, …, l. Character $\bar{s}_j^m$ and $\bar{s}_j^n$, m, n = 1, 2, …, k; $m \neq n$, is aligned under alignment $\overline{S}$ on column j. A character from s^i and a character from s^j are a **match** if both characters are the same and they are put (aligned) under the same column j, i.e. $\bar{s}_j^m = \bar{s}_j^n$. A character from s^i and a character from s^j are a **mismatch** if both characters are not the same and they are put (aligned) under the same column j, i.e. $\bar{s}_j^m \neq \bar{s}_j^n$. To give a better understanding about array representation of an alignment, consider an MSA problem consisting of three very short DNA sequences given in Example 2.1:

Example 2.1.

Given sequences s^1, s^2, and s^3.

```
s1:  A  G  C
s2:  T  G  C  C  A
s3:  A  T  C  A
```

Some possible alignments of Example 2.1 are given in Figure 2.2. From Figure 2.2 (a), character $\bar{s}_1^1$ (alphabet A from s^1) and character $\bar{s}_1^3$ (alphabet A from s^3) is a match and character $\bar{s}_1^1$ (alphabet A from s^1) and character $\bar{s}_1^2$ (alphabet T from s^2) is a mismatch. Character $\bar{s}_4^2$ (alphabet C from s^2) and character $\bar{s}_4^3$ (dash symbol) is also a mismatch where $\bar{s}_4^3$ is a gap inserted in s^3. A glance of Figure 2.2, the number of inserted dash symbols increases from alignment Figure 2.2 (a) to alignment Figure 2.2 (d).

$$\begin{array}{lccccc}
\bar{s}^1: & A & G & - & C & - \\
\bar{s}^2: & T & G & C & C & A \\
\bar{s}^3: & A & T & C & - & A
\end{array}$$

(a)

$$\begin{array}{lcccccc}
\bar{s}^1: & A & - & G & C & - & - \\
\bar{s}^2: & - & T & G & C & C & A \\
\bar{s}^3: & A & T & - & - & C & A
\end{array}$$

(c)

$$\begin{array}{lcccccc}
\bar{s}^1: & A & - & G & - & C & - \\
\bar{s}^2: & - & T & G & C & C & A \\
\bar{s}^3: & A & T & - & C & - & A
\end{array}$$

(b)

$$\begin{array}{lcccccccc}
\bar{s}^1: & - & - & - & - & - & A & G & C \\
\bar{s}^2: & T & G & C & C & A & - & - & - \\
\bar{s}^3: & - & - & - & - & A & T & C & A
\end{array}$$

(d)

Fig. 2.2. Possible alignments of Example 2.1.

It is obvious that there are many alignments of an MSA. The problem then becomes how to determine the best alignment. In biology, justification of good or bad alignment criteria is obtained through qualitative approach. However, if we want to compute an alignment, such criteria cannot be used. A set of scores obtained through quantitative approach called scoring scheme is used as a criteria of a good or a bad alignment. The alignments that have the best score by definition are the optimal alignments (It is possible that there are more than one such alignment). We will introduce scoring scheme of an alignment on the following subchapter.

2.2. SCORING SCHEME OF AN ALIGNMENT

Scoring scheme is used to score any alignment of the MSA problem. We can distinguish among different alignments by examining their scores. We can make our own scoring scheme according to our own preferences (the preferences must not violate any alignment criteria) or obtain it through biological or statistical observation. A simple scoring scheme assumes independence among columns in an alignment and set the total score of the alignment to be equal to the sum of the scores of each column (see Isaev A. [3]). Such scheme only needs to specify the score of a match and a mismatch. A mismatch includes mismatch between the DNA alphabets and mismatch between the DNA alphabet and the inserted gap. The former is referred as a mismatch score and the latter is referred as a **gap penalty**.

We can make a simple scoring scheme; say +2 for a match, −1 for a mismatch, and 0 for a gap penalty. First, we apply the scoring scheme to alignment (a) in Figure 2.2. From the first column (j = 1), $\bar{s}_1^1 \neq \bar{s}_1^2$ (A ≠ T, a mismatch), $\bar{s}_1^1 = \bar{s}_1^3$ (A = A, a match), and $\bar{s}_1^2 \neq \bar{s}_1^3$ (T ≠ A, a mismatch). The sum of the scores of the first column is −1 + 2 − 1 = 0. From the second column (j = 2), $\bar{s}_2^1 = \bar{s}_2^2$ (G = G, a match), $\bar{s}_2^1 \neq \bar{s}_2^3$ (G ≠ T, a mismatch), and $\bar{s}_2^2 \neq \bar{s}_2^3$ (G ≠ T, a mismatch). The sum of the scores of the second column is 2 – 1 – 1 = 0. From the third column (j = 3), $\bar{s}_3^1 \neq \bar{s}_3^2$ ('−' ≠ C, a gap penalty), $\bar{s}_3^1 \neq \bar{s}_3^3$ ('−' ≠ C, a gap penalty), and $\bar{s}_3^2 = \bar{s}_3^3$ (C = C, a match). The sum of the scores of the third column is 0 + 0 + 2 = 2. From the fourth column (j = 4), $\bar{s}_4^1 = \bar{s}_4^2$ (C = C, a match), $\bar{s}_4^1 \neq \bar{s}_4^3$ (C ≠ '−', a gap penalty), and $\bar{s}_4^2 \neq \bar{s}_4^3$ (C ≠ '−', a gap penalty). The sum of the scores of the fourth column is 2 + 0 + 0 = 2. From the fifth column (j = 5), $\bar{s}_5^1 \neq \bar{s}_5^2$ ('−' ≠ A, a gap penalty), $\bar{s}_5^1 \neq \bar{s}_5^3$ ('−' ≠

A, a gap penalty), and $\bar{s}_5^2$ = $\bar{s}_5^3$ (A = A, a match). The sum of the scores of the fifth column is 0 + 0 + 2 = 2. The total score of alignment (a) in Figure 2.2 is the sum of the scores of each column (*j* = 1, 2, …, 5): 0 + 0 + 2 + 2 + 2 = 6. If we apply the scoring scheme to the rest of the alignments in Figure 2.2, alignment (b), (c), and (d) will have the total score of 12, 12, and −1 respectively. Notice that alignment (b) and (c) have the same total score. However, according to Althaus E. *et al*. [2], "a single longer gap is more likely to arise in reality since it might be caused by a single mutational event". Additionally, the length of the gap should have a smaller impact to the score than the number of gaps. We can achieve this by using convex gap cost functions such as *a* + *b* log *l* where *a* and *b* are arbitrary numbers and *l* is the length of the gap. We can also use an affine gap costs of the form *a* + *bl*.

In alignment (b) Figure 2.2, $\bar{s}^1$ has three gaps: $\bar{s}_2^1$, $\bar{s}_4^1$, and $\bar{s}_6^1$, $\bar{s}^2$ has one gap: $\bar{s}_1^2$, and $\bar{s}^3$ has two gaps: $\bar{s}_3^3$ and $\bar{s}_5^3$. In alignment (c) Figure 2.2, $\bar{s}^1$ has two gaps: $\bar{s}_2^1$ and {$\bar{s}_5^1$, $\bar{s}_6^1$} (both dash symbol $\bar{s}_5^1$ and $\bar{s}_6^1$ are considered as a single but long gap), $\bar{s}^2$ has one gap: $\bar{s}_1^2$, and $\bar{s}^3$ has one gap: {$\bar{s}_3^3$, $\bar{s}_4^3$}. Alignment (b) and (c) have 6 and 4 gaps in total respectively. Hence, alignment (c) should be scored better than alignment (b).

We can solve the MSA problem by using various methods such as dynamic programming (e.g. Needleman-Wunsch algortihm, see Isaev A. [3]) and heuristic (e.g. FASTA, see Isaev A. [3]). A method has been proposed by

Althaus E. *et al*. [1] to solve the MSA problem which is based on ILP. The ILP model is derived from a graph representation of the MSA problem. Before we discuss about the graph representation of the MSA problem further, we will introduce some definitions and basic concept in graph theory on the following subchapter.

2.3. DEFINITIONS AND BASIC CONCEPTS IN GRAPH THEORY

A graph $G(V, E, A)$ consists of V, a nonempty set of **vertices**, E, a set of unordered pairs of distinct elements of V called **edges**, and A, a set of directed edges called **arcs** (see Rosen K.H. [7]). Two vertices u and v in a graph G are called **adjacent** (or **neighbors**) in G if $e = \{u, v\}$ is an edge of graph G. Edge e is called **incident** with vertices u and v. Vertices u and v are called **endpoints** of edge e. Two edges e_1 and e_2 are called **multiple** (or **parallel**) **edges** if both edges are incident to the same endpoints. Vertex u is said to be **adjacent to** vertex v (or vertex v is said to be **adjacent from** vertex u) if $a = (u, v)$ is an arc of graph G. In this case, vertex u is called **initial vertex** of arc a and vertex v is called **terminal** (or **end**) **vertex** of arc a. An arc is called **loop** if the arc has the same initial and end vertex. Graph $H = (W, F, B)$ is a **subgraph** graph $G = (V, E, A)$ if $W \subseteq V$, $F \subseteq E$, and $B \subseteq A$. Graph H is a **spanning subgraph** graph G if $W = V$.

A graph G is called a **simple graph** if it contains no arcs, multiple edges, and loops. G is called a **directed graph** if it contains arcs and (or) loops but no multiple edges. **Walk** of length k is a sequence v_1, u_1, v_2, u_2, v_2, …, v_k, u_k, v_{k+1} of vertices and edge(s) or arc(s) such that all u_i are only edges $\{v_i, v_{i+1}\}$ or arcs (v_i, v_{i+1}), $i = 1, 2, \ldots, k$. **Mixed walk** of length k is a sequence v_1, u_1, v_2, u_2, v_2, …, v_k, u_k, v_{k+1} of vertices, edge(s), and arc(s) such that the sequence contains both arc and edge, u_i can either be an edge $\{v_i, v_{i+1}\}$ or an arc (v_i, v_{i+1}). **Path** is a walk that does not traverse the same vertex more than once, i.e. $v_i = v_j$ if and only if $i = j$, $i, j = 1, 2, \ldots, k + 1$. **Mixed path** is a mixed walk that does not traverse the same vertex more than once. **Cycle** is a path that begins and ends at the same vertex. **Mixed cycle** is a mixed path that begins and ends at the same vertex.

A simple graph G is called **bipartite** if its vertex set V can be partitioned into two disjoint sets V_1 and V_2 such that no edge in graph G connects either two vertices in V_1 or two vertices in V_2. **Complete bipartite graph** $K_{m,n}$ is a graph that its vertex set partitioned into two subsets V_1 of m vertices and V_2 of n vertices such that there is an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset. **Complete *k*-partite graph** $K_{(n_1, n_2, \ldots, n_k)}$ is a graph that has its vertex set partitioned into subsets V_1 of n_1 vertices, V_2 of n_2 vertices, …, and V_k of n_k vertices such that there is an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset.

Figure 2.3 illustrates some graph examples. Graph *G* on Figure 2.3 (a) is a directed graph with 5 vertices, 6 edges, and 4 arcs. Graph H_2 and H_3 on Figure 2.3 (c) and (d) are subgraphs of graph *G*. Graph H_1 on Figure 2.3 (b) is a spanning subgraph of graph *G*. Graph H_2 is a cycle and also a bipartite graph with disjoint vertex sets of $\{v_1, v_2\}$ and $\{v_4, v_5\}$. Graph H_3 is a mixed cycle.

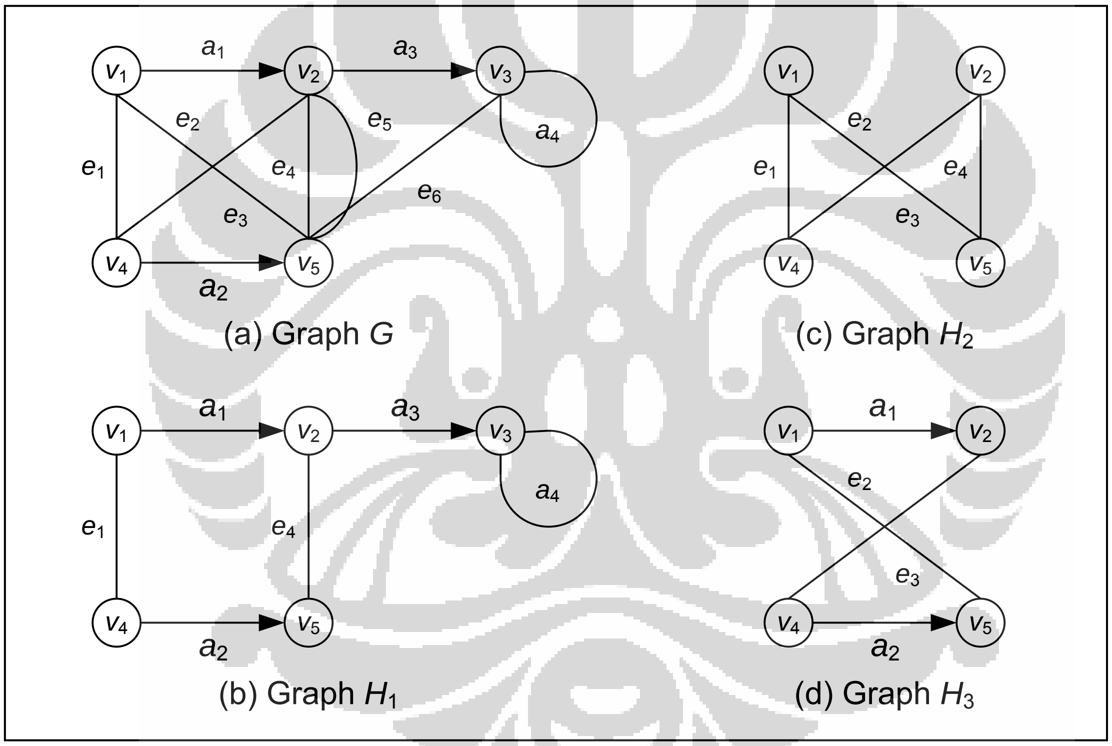


Fig. 2.3. Graph examples.

Now that we have discussed about graph theory, we will present the steps of constructing a graph representation of the MSA problem.

2.4. GRAPH REPRESENTATION OF THE MSA PROBLEM

MSA problem is represented as a graph, each character s^i_j, i = 1, 2, …, k; j = 1, 2, …, $\left|s^i\right|$,is represented as node v^i_j. V^i denotes the set of all nodes corresponding to characters in s^i.

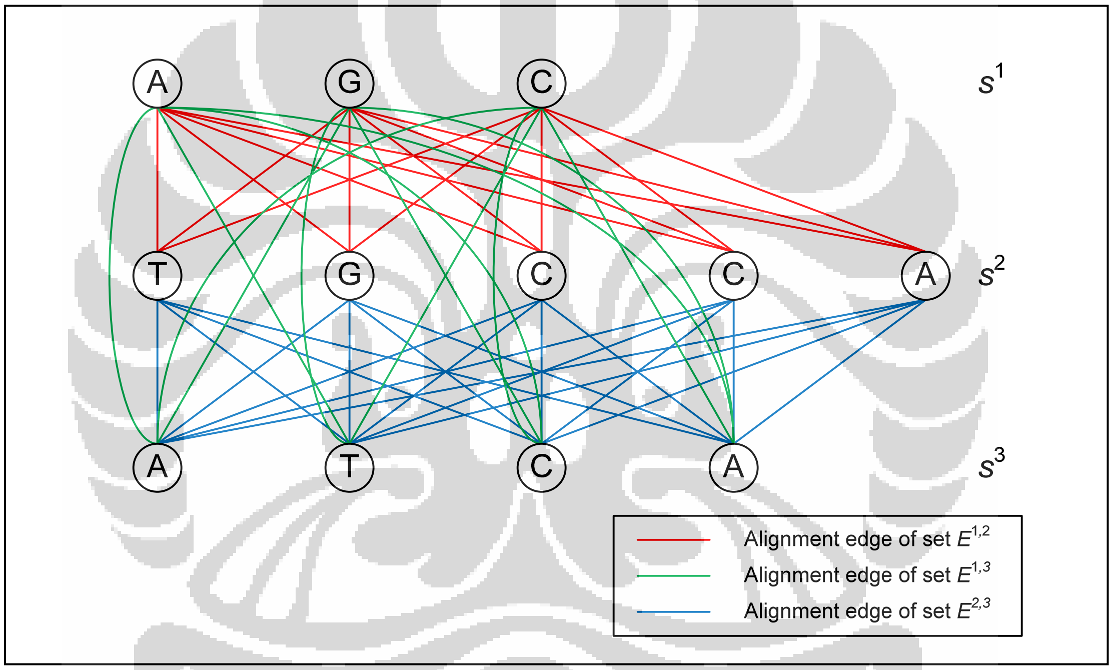

A
G
C
s^1
T
G
C
C
A
s^2
A
T
C
A
s^3
Alignment edge of set $E^{1,2}$
Alignment edge of set $E^{1,3}$
Alignment edge of set $E^{2,3}$

Fig. 2.4. Alignment graph of Example 2.1.

We start by introducing a graph that represents all the possible alignment of the characters s^i_j called an **alignment graph**. Alignment of character $s^{i_1}_{j_1}$ from s^{i_1} and character $s^{i_2}_{j_2}$ from s^{i_2} is represented as edge e = $\left\{v^{i_1}_{j_1}, v^{i_2}_{j_2}\right\}$, i_1, i_2 = 1, 2, …, k; $i_1 \neq i_2$; j_1 = 1, 2, …, $\left|s^{i_1}\right|$; j_2 = 1, 2, …, $\left|s^{i_2}\right|$. Such an edge is called **alignment edge**. E^{i_1,i_2} denotes the set of all alignment edges e = $\left\{v^{i_1}_{j_1}, v^{i_2}_{j_2}\right\}$ of the alignment graph. Graph G = (V, E) is an alignment graph where $V = \{V^1, V^2, \ldots, V^k\}$ and E is the set of all the alignment edge. The alignment graph of Example 2.1 is given in Figure 2.4. We can see in Figure 2.4 that alignment graph of k -sequence MSA problem is a complete k-partite graph.

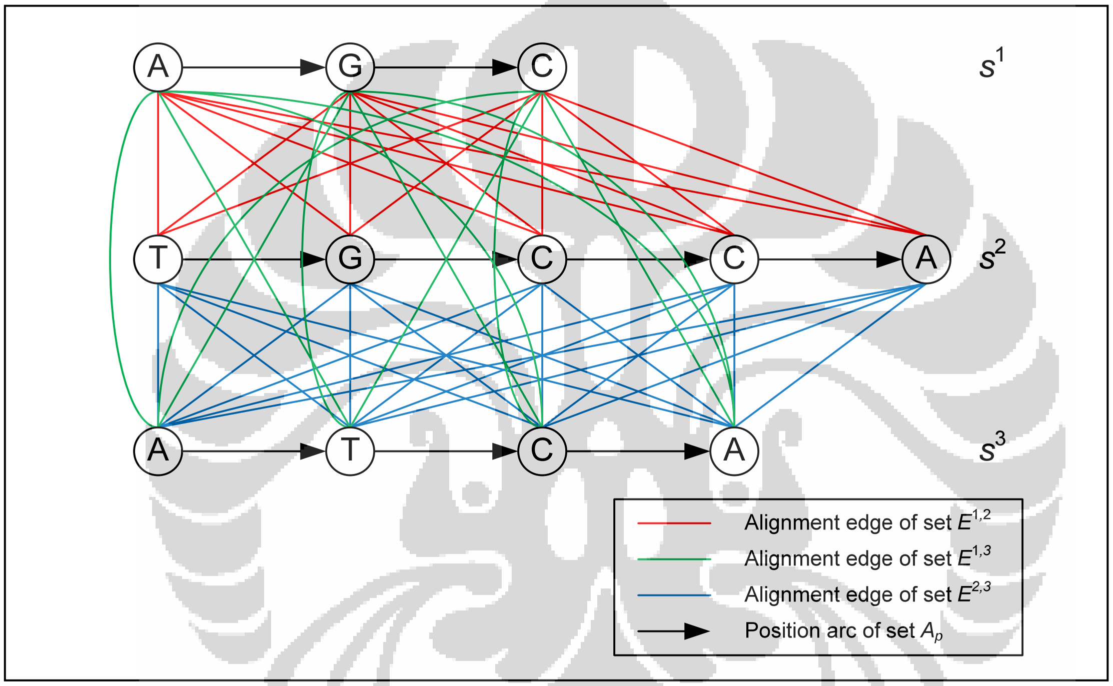

A
G
C
s^1
T
G
C
C
A
s^2
A
T
C
A
s^3
Alignment edge of set $E^{1,2}$
Alignment edge of set $E^{1,3}$
Alignment edge of set $E^{2,3}$
Position arc of set A_p

Fig 2.5. Extended alignment graph of Example 2.1.

However, alignment graph does not represent order of the characters. According to the property of an alignment, the order of the characters of each sequence must be the same before and after the alignment. Hence, we introduce **extended alignment graph**, an alignment graph that also contains the ordering information of the characters of each sequence. Ordering of characters s_j^i within the same string s^i is represented as arc $a_p = \left(v_j^i, v_{j+1}^i\right)$,

$i = 1, 2, \ldots, k$; $j = 1, 2, \ldots, \left|s^i\right| - 1$, which means that the characters are ordered from the left to the right. Such arc is called **position arc**. A_p denotes the set of all position arcs $a_p = \left(v^i_j, v^i_{j+1}\right)$. Graph $G = (V, E, A_p)$ is an extended alignment graph where $V = \{V^1, V^2, \ldots, V^k\}$, E is the set of all the alignment edge and A_p is the set of all the position arc. The extended alignment graph of Example 2.1 is given in Figure 2.5. We can see in Figure 2.5 that the order of each sequence is still preserved by the position arc in the extended alignment graph.

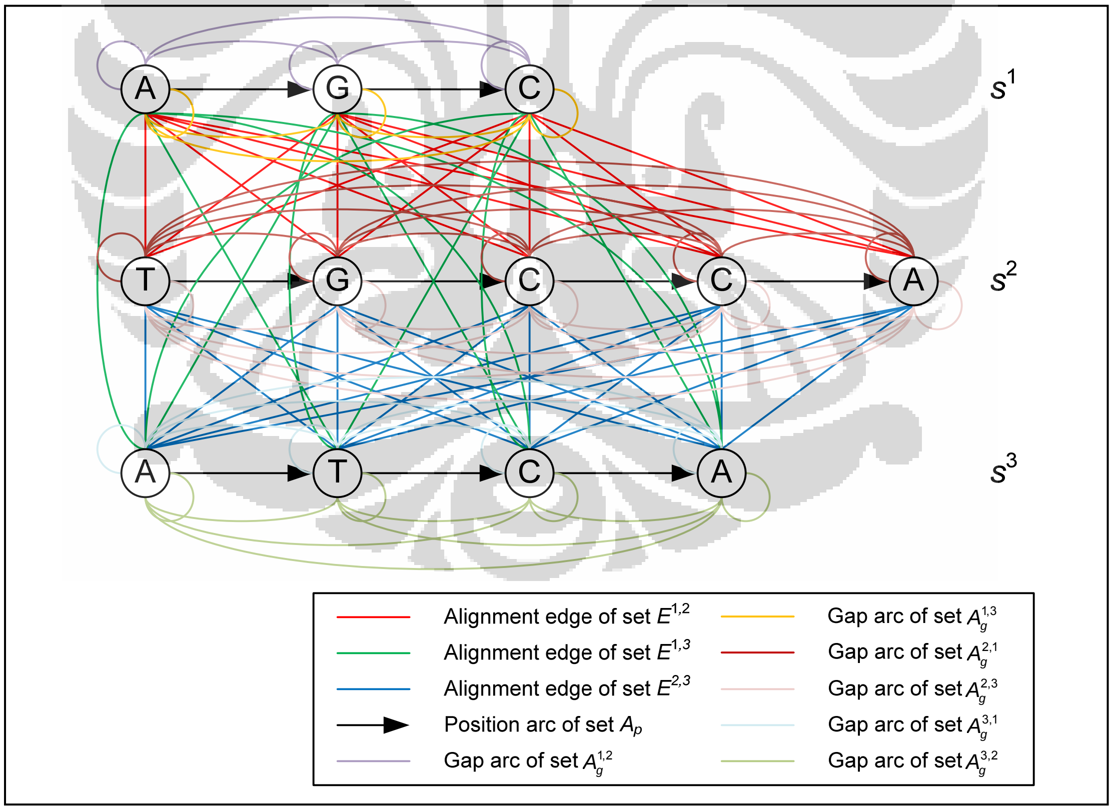

A
G
C
s^1
T
G
C
C
A
s^2
A
T
C
A
s^3
Alignment edge of set $E^{1,2}$
Alignment edge of set $E^{1,3}$
Alignment edge of set $E^{2,3}$
Position arc of set A_p
Gap arc of set $A_g^{1,2}$
Gap arc of set $A_g^{1,3}$
Gap arc of set $A_g^{2,1}$
Gap arc of set $A_g^{2,3}$
Gap arc of set $A_g^{3,1}$
Gap arc of set $A_g^{3,2}$

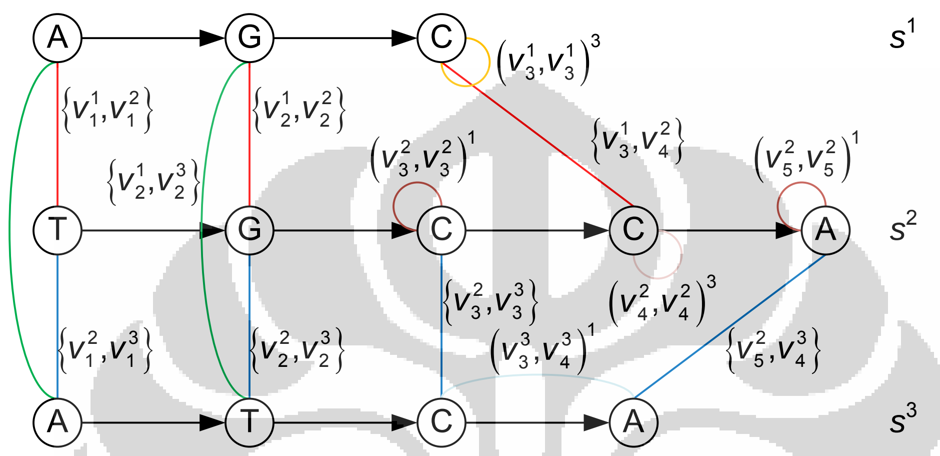
Fig. 2.6. Gapped extended alignment graph of Example 2.1.

Extended alignment graph preserves the ordering information of the sequences but it does not represent the possibility of aligning character(s) to

gap. Hence, we introduce the **gapped extended alignment graph**, an extended alignment graph that also represents the possibility of aligning character(s) to gap. Each character $s_j^{i_1}$ from s^{i_1} that is not aligned to a character from s^{i_2} is aligned to a gap inserted in s^{i_2}, i_1, i_2 = 1, 2, …, k; $i_1 \neq i_2$; j = 1, 2, …, $\left|s^{i_1}\right|$. It is also possible that a substring say $s_{j_1}^{i_1}$ to $s_{j_2}^{i_1}$, j_1, j_2 = 1, 2, …, $\left|s^{i_1}\right|$, of s^{i1} is aligned to a (single longer) gap inserted in s^{i_2}. Hence, for each substring $s_{j_1}^{i_1}$ to $s_{j_2}^{i_1}$ of s^{i_1} that are aligned to a gap inserted in s^{i_2}, there is an arc a_g = $\left(v_{j_1}^{i_1}, v_{j_2}^{i_1}\right)^{i_2}$. Such an arc is called **gap arc**. Substring $s_{j_1}^{i_1}$ to $s_{j_2}^{i_1}$ of s^{i_1} are said to be spanned by gap arc $\left(v_{j_1}^{i_1}, v_{j_2}^{i_1}\right)^{i_2}$. $A_g^{i_1,i_2}$ denotes the set of all gap arcs a_g = $\left(v_{j_1}^{i_1}, v_{j_2}^{i_1}\right)^{i_2}$, j_1, j_2 = 1, 2, …, $\left|s^{i_1}\right|$. There is also another definition for gap arcs that is $A_g^{i_1,i_2}$ ($j_1 \leftrightarrow j_2$). $A_g^{i_1,i_2}$ ($j_1 \leftrightarrow j_2$) :=
$\left\{\left(v_p^{i_1}, v_q^{i_1}\right)^{i_2} : p \leq j_1, q \geq j_2\right\}$ denote the subset of arcs in $A_g^{i_1,i_2}$ that spans character $s_{j_1}^{i_1}$, …, $s_{j_2}^{i_1}$, i_1, i_2 = 1, 2, …, k; $i_1 \neq i_2$; j_1, j_2 = 1, 2, …, $\left|s^{i_1}\right|$. Graph G = (V, E, A) is a gapped extended alignment graph where
V = {V^1 , V^2 , …, V^k}, E is the set of all the alignment edge, A_g is the set of all the gap arc, and A = {$A_p \cup A_g$}. The gapped extended alignment graph of Example 2.1 is given in Figure 2.6. Notice that the gap arc is not represented as an arc but as an edge in the gapped extended alignment graph. It is

represented as an edge because gap arc $\left(v_{j_1}^{i_1}, v_{j_2}^{i_1}\right)^{i_2}$ is the same as gap arc $\left(v_{j_2}^{i_1}, v_{j_1}^{i_1}\right)^{i_2}$, i_1, i_2 = 1, 2, …, k; $i_1 \neq i_2$; j_1, j_2 = 1, 2, …, $\left|s^{i1}\right|$. Rather than representing gap arc $\left(v_l^i, v_m^i\right)^j$ as two different arcs, it is represented as a single edge (an edge is in fact a two way arc). In Figure 2.6, $A_g^{2,1}(2 \leftrightarrow 3)$ denotes the set of gap arcs {$\left(v_1^2, v_3^2\right)^1$, $\left(v_1^2, v_4^2\right)^1$, $\left(v_1^2, v_5^2\right)^1$, $\left(v_2^2, v_3^2\right)^1$, $\left(v_2^2, v_4^2\right)^1$, $\left(v_2^2, v_5^2\right)^1$}.

Gapped extended alignment graph is the graph representation of the MSA problem. Now, we have to find a subgraph from the gapped extended alignment graph that fulfills the alignment properties. We call this subgraph as a gapped trace. The MSA problem becomes a graph problem (or gapped trace problem) which is to find **gapped trace**, a spanning subgraph of the gapped extended alignment graph that corresponds to an alignment of the MSA problem. A gapped trace consists of the set of all vertices and position arcs in the gapped extended alignment graph and sets of realized alignment edges and gap arcs. An alignment edge e = $\left\{v_{j_1}^{i_1}, v_{j_2}^{i_2}\right\}$ is said to be **realized** by an alignment if characters $s_{j_1}^{i_1}$ and $s_{j_2}^{i_2}$ are put in the same column in alignment array. A gap arc a_g = $\left(v_{j_1}^{i_1}, v_{j_2}^{i_1}\right)^{i_2}$ is **realized** by an alignment if the substring $s^{i_1}_{j_1}$ to $s^{i_2}_{j_2}$ of s^{i_1} is not aligned to any character in s^{i_2}, whereas both $s^{i_1}_{j_1-1}$ (if $j_1 > 1$) and $s^{i_1}_{j_2+1}$(if $j_2 < \left|s^{i_1}\right|$) are aligned with some characters in s^{i_2}.


A
G
C
$\left(v^1_3, v^1_3\right)^3$
s^1
$\left\{v^1_1, v^2_1\right\}$
$\left\{v^1_2, v^2_2\right\}$
$\left\{v^1_2, v^3_2\right\}$
$\left(v^2_3, v^2_3\right)^1$
$\left\{v^1_3, v^2_4\right\}$
$\left(v^2_5, v^2_5\right)^1$
T
G
C
C
A
s^2
$\left\{v^2_3, v^3_3\right\}$
$\left(v^2_4, v^2_4\right)^3$
$\left\{v^2_1, v^3_1\right\}$
$\left\{v^2_2, v^3_2\right\}$
$\left(v^3_3, v^3_4\right)^1$
$\left\{v^2_5, v^3_4\right\}$
A
T
C
A
s^3

(a)

$\bar{s}^1$:	A	G	–	C	–
$\bar{s}^2$:	T	G	C	C	A
$\bar{s}^3$:	A	T	C	–	A

(b)

Fig 2.7. Gapped trace of Figure 2.7 that corresponds to alignment (a) of Figure 2.2.

By studying the properties of an alignment presented in subchapter 2.1, the following are gapped trace properties (see Althaus E. *et al*. [1]; Reinert K. [5]):

1. For each pair of strings, each node is either incident to exactly one alignment edge or spanned by exactly one gap arc.
2. There must not be a critical mixed cycle in the gapped trace.
3. There cannot be a pair of conflicting gap arcs for any pair of strings.

4. Whenever two alignment edges incident to the same node are realized, say $\left\{v_{l_1}^{i_1}, v_{l_2}^{i_2}\right\}$ and $\left\{v_{l_2}^{i_2}, v_{l_3}^{i_3}\right\}$, by transitivity, alignment edge $\left\{v_{l_1}^{i_1}, v_{l_3}^{i_3}\right\}$ must be realized too.

We give an example of a gapped trace in Figure 2.7. Gapped trace in Figure 2.7 (a) realizes nine alignment edges and five gap arcs. Figure 2.7 (a) is the gapped trace that represents alignment Figure 2.7 (b). For a better understanding, we give a brief explanation upon the gapped trace properties.

First, for each pair of strings, each node is either incident to exactly one alignment edge or spanned by exactly one gap arc. I.e. each node is either incident to exactly one alignment edge or spanned by exactly one gap arc for each pair of string since for each pair of string $\bar{s}^m$ and $\bar{s}^n$, character $\bar{s}_j^m$ can only be aligned to character $\bar{s}_j^n$ under alignment $\bar{S}$ on column *j*, *m*, *n* = 1, 2, …, *k*; *m* ≠ *n*, *j* = 1, 2, …, *l*.

Second, there must not be a critical mixed cycle in the subgraph. A mixed cycle in a gapped extended alignment graph represents a contradictory ordering in the alignment called **crossing**. A mixed cycle *C* is called **critical** if for all *i*, 1 ≤ *i* ≤ *k*, all vertices in $C \cap S^i$ occur consecutively in *C*. Informally this means that a critical mixed cycle visits (enters and leaves) each sequences at most once. The difference between a mixed and a critical mixed cycle can be seen in Figure 2.8. Figure 2.8 (a) and (b) are subsets of gapped extended alignment graph Figure 2.6 that have a mixed cycle. Figure

2.8 (c) and (d) are subsets of gapped trace Figure 2.8 (a) and (b) that forms the mixed cycle. Mixed cycle (c) is not critical since s^2 is visited more than once in mixed cycle (b). Mixed cycle (d) is critical because each sequence is visited exactly once in mixed cycle (d). Both mixed cycle (c) and (d) represents a contradictory ordering in the alignment. Figure 2.9 illustrates contradictory ordering. Figure 2.9 (a), (b), (c), (d) illustrates the steps of representing mixed cycle Figure 2.8 (c) into the array representation. Figure 2.9 (a) shows that characters s_1^1, s_2^1, and s_3^1 are placed consecutively in different columns in the same row. Figure 2.9 (b) shows that character s_3^1 is aligned to character s_1^2. Figure 2.9 (c) shows that character s_1^2 is aligned to character s_1^3. Figure 2.9 (d) shows that character s_2^3 is aligned to character s_3^2, characters s_1^2, s_2^2, and s_3^2 are placed consecutively in different columns in the same row, and character s_2^3 is moved to the column where character s_3^2 is placed. Figure 2.9 (d) clearly shows the contradictory ordering where character s_3^2 is supposed to be aligned with character. Figure 2.9 (e), (f), (g), (h) illustrates the steps of representing mixed cycle Figure 2.8 (d) into the array representation. Figure 2.9 (e) shows that characters s_1^1, s_2^1, and s_3^1 are placed consecutively in different columns in the same row. Figure 2.9 (f) shows that character s_3^1 is aligned to character s_1^2. Figure 2.9 (g) shows that characters s_1^2, s_2^2, and s_3^2 are placed consecutively in different columns in the same row. Figure 2.9 (h) shows that character s_3^2 is aligned to character s_1^3. Figure 2.9 (h) clearly shows the contradictory ordering where character s_1^3 is supposed to be aligned with character s_1^1. Critical mixed cycle is indeed a mixed cycle and it has been guaranteed that a gapped trace contains a mixed cycle if and only if it contains a critical mixed cycle (see Reinert K. [6]).

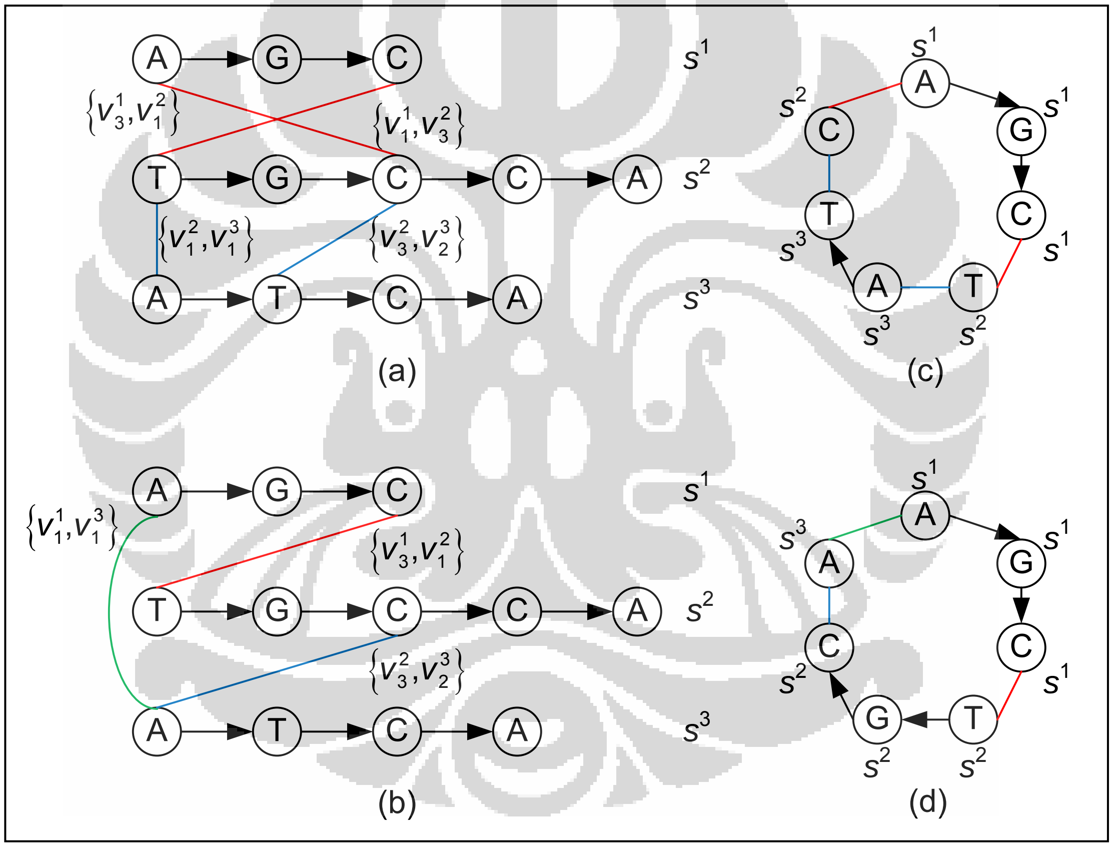


Fig. 2.8. Examples of a mixed cycle and a critical mixed cycle.

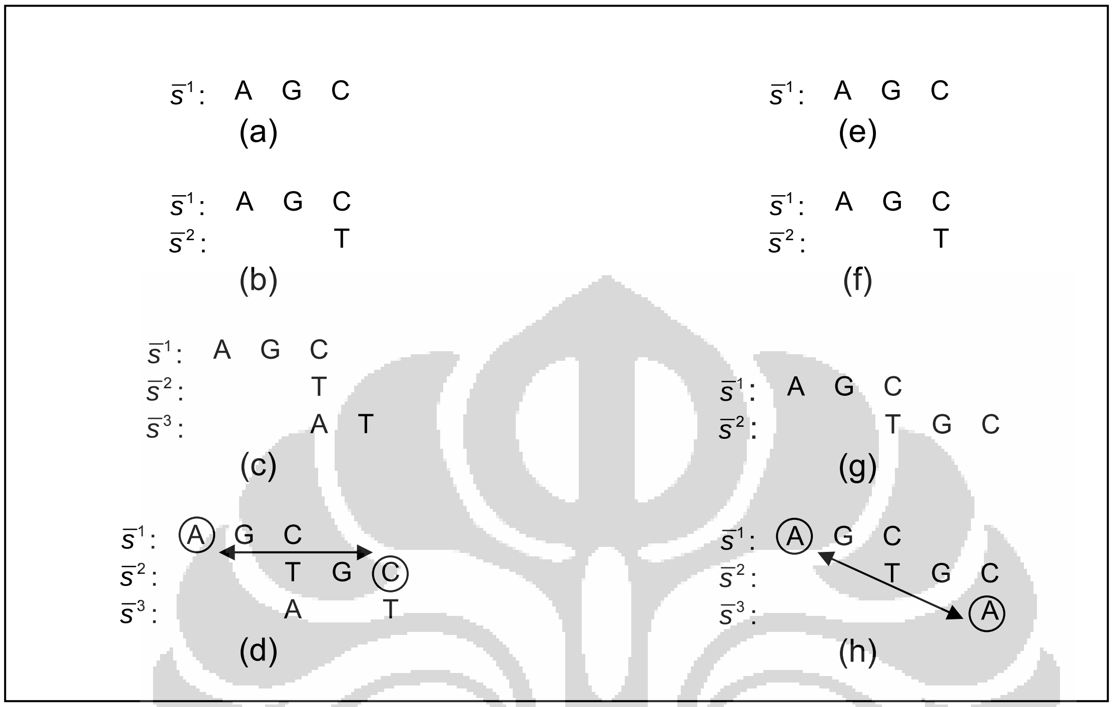


Fig. 2.9. Contradictory ordering of mixed cycle Figure 2.8 (c) and (d).

Third, there cannot be a pair of conflicting gap arcs for a given pair of strings. Gap arcs are said to be **conflict** by definition if the substrings spanned by the gap arcs **overlap** or even **touch** each other. We give an illustration of overlapping and touching gap arcs in Figure 2.10. Figure 2.10 (a) and (b) illustrate examples of overlapping gap arcs. In Figure 2.10 (a), gap arc $\left(v_1^3,v_2^3\right)^1$ spans from characters s_1^3 and s_2^3, gap arc $\left(v_2^3,v_3^3\right)^1$ spans from characters s_2^3 and s_3^3 (notice that gap arcs $\left(v_1^3,v_2^3\right)^1$ and $\left(v_2^3,v_3^3\right)^1$ overlapped each other in spanning character s_2^3), and gap arc $\left(v_3^3,v_4^3\right)^1$ spans from characters s_3^3 and s_4^3, i.e. substring s_1^3 to s_4^3 is aligned to gaps inserted in s^1. By definition of gap arc, gap arcs $\left(v_1^3,v_2^3\right)^1$, $\left(v_2^3,v_3^3\right)^1$, and $\left(v_3^3,v_4^3\right)^1$ must be replaced by gap arc $\left(v_1^3, v_4^3\right)^1$. Figure 2.10 (c) and (d) illustrate examples of touching gap arcs. In Figure 2.10 (c), gap arc $\left(v_1^3, v_1^3\right)^1$ spans character s_1^3, gap arc $\left(v_2^3, v_2^3\right)^1$ spans character s_2^3 (notice that gap arcs $\left(v_1^3, v_1^3\right)^1$ and $\left(v_2^3, v_2^3\right)^1$ touched each other), gap arc $\left(v_3^3, v_3^3\right)^1$ spans character s_3^3, and gap arc $\left(v_4^3, v_4^3\right)^1$ spans characters s_4^3, i.e. substring s_1^3 to s_4^3 is aligned to gaps inserted in s^1. By definition of gap arc, gap arcs $\left(v_1^3, v_1^3\right)^1$, $\left(v_2^3, v_2^3\right)^1$, $\left(v_3^3, v_3^3\right)^1$, and $\left(v_4^3, v_4^3\right)^1$ must be replaced by gap arc $\left(v_1^3, v_4^3\right)^1$. Conflict shown in Figure 2.10 (a), (b), (c) and (d) should be represented by Figure 2.10 (e).

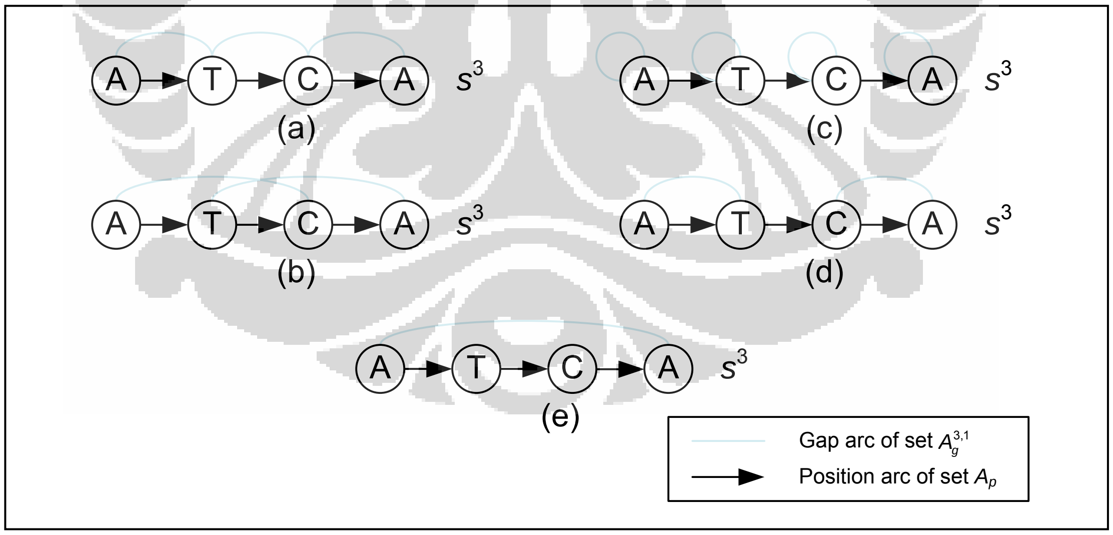


Fig. 2.10. Example of overlapping gap arcs of set $A_g^{3,1}$.

Last, whenever two alignment edges incident to the same node are realized, say $\left\{v_{l_1}^{i_1}, v_{l_2}^{i_2}\right\}$ and $\left\{v_{l_2}^{i_2}, v_{l_3}^{i_3}\right\}$, by transitivity, alignment edge $\left\{v_{l_1}^{i_1}, v_{l_3}^{i_3}\right\}$ must be realized too. From the alignment properties, all characters $\bar{s}_j^i$,

i = 1, 2, …, k, are aligned on the same column j under alignment $\overline{S}$. It means that if a character $\bar{s}_{j_1}^{i_1}$ is aligned to character $\bar{s}_{j_2}^{i_2}$ and character $\bar{s}_{j_2}^{i_2}$ is aligned to character $\bar{s}_{j_3}^{i_3}$, $1 \le i_r \le k$; $1 \le j_r \le \left|s^{i_r}\right|$; r = 1, 2, 3, then character $\bar{s}_{j_1}^{i_1}$ is also aligned to character $\bar{s}_{j_3}^{i_3}$, i.e. we can say that transitivity holds in the array representation of alignment. For an example, say we realize alignment edges $\{v_3^1, v_5^2\}$ and $\{v_5^2, v_1^3\}$. Then by transitivity, alignment edge $\{v_3^1, v_1^3\}$ must be realize too. The illustration is given in Figure 2.11. Notice that the alignment edges $\{v_3^1, v_5^2\}$, $\{v_5^2, v_1^3\}$, and $\{v_3^1, v_1^3\}$ in Figure 2.11 form a cycle. For more detail and further explanation about gapped trace see Reinert K. [6].

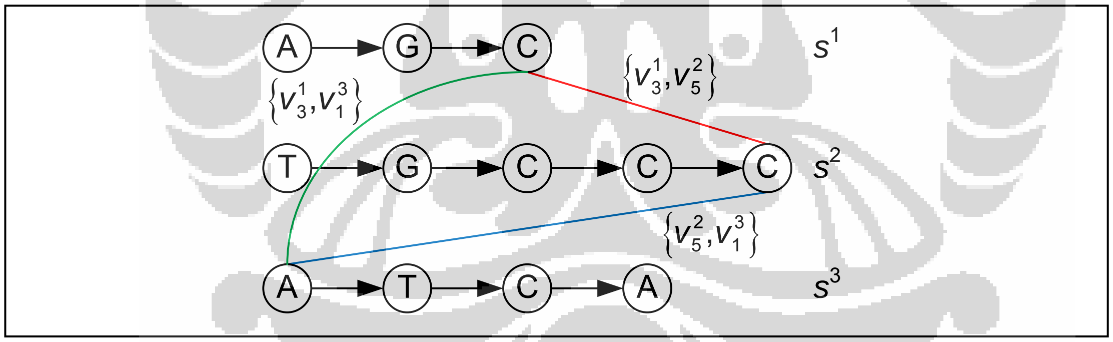


Fig. 2.11. Example of transitivity.

As discussed in subchapter 2.2, scoring scheme is used to score any possible alignment of the MSA problem. Since the MSA problem becomes a gapped trace problem, scoring alignments becomes scoring gapped traces. For scoring gapped traces, each alignment edge and gap arc is assigned a weight that corresponds to the benefit (or cost depending on the purpose of the alignment) of realizing the edge or arc. Let w_e and w_a denote the weight of alignment edge $e \in E$ and gap arc $a \in A_g$. With the weight and the gapped trace properties altogether, we can formulize the general ILP model of the MSA problem. The ILP model along with the detail explanation will be presented in the next chapter. But before we advance to the next chapter, we present definitions and basic concepts in integer linear programming first and then the branch-and-bound method.

2.5. DEFINITIONS AND BASIC CONCEPTS IN INTEGER LINEAR PROGRAMMING

Linear programming is a mathematical technique that selects the best course of action from a set of feasible alternatives (see Wu N. and Coppins R. [8]). It is linear because the relationships among the variables involved are linear. Typical linear programming problem is to optimize an objective function subject to a series of linear restriction (constraints). An optimal solution of a linear program includes set(s) of values for the variables (the solution does not need to be unique) that optimize(s) the corresponding value of the objective function. We can form the standard linear program in symbols as follow:

Optimize $f = c_1x_1 + c_2x_2 + \ldots + c_nx_n = \sum_{j=1}^{n} c_jx_j$

subject to

$$\begin{gathered} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \le b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \le b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \le b_m \end{gathered}$$

Or

$$\sum_{j=1}^{n} a_{ij}x_j \le b_i, \; i = 1, 2, \ldots, m; \; j = 1, 2, \ldots, n$$

$$x_1, x_2, \ldots, x_n \ge 0$$

where $x_1, x_2, \ldots, x_n$ are the decision variables, f is the objective function, $c_1, c_2, \ldots, c_n$ are the coefficients of decision variables in the objective function, $a_{i1}, a_{i2}, \ldots, a_{in}$ are the coefficients of decision variables in the i-th constraint, and b_i are the constant of the i-th constraint. We can also write the standard linear program form with vector notation as follow:

Optimize $f = \boldsymbol{c}'\boldsymbol{x}$

subject to

$$\begin{aligned} \boldsymbol{Ax} &\le \boldsymbol{b} \\ \boldsymbol{x} &\ge \boldsymbol{0} \end{aligned}$$

where $\boldsymbol{c}$, $\boldsymbol{x}$, $\boldsymbol{0}$ are $n \times 1$ vectors, $\boldsymbol{A}$ is an $m \times n$ matrix, and $\boldsymbol{b}$ is an $m \times 1$ vector.

There are many situations which require the decision variables to have integer values. Problems that arise from such situations are referred as **integer programming** problems. One special integer programming problem is **integer linear programming** problem, a linear programming problem with an additional integer value constraint. Among integer programming problems, there is a special class in which the values of the variables are restricted to values 1 or 0 (binary). It is referred as **binary integer linear programming** problems.

In this *skripsi*, we also construct a program that can generate and solve the ILP model of any given MSA problem using MATLAB R2008a. We utilize a function in MATLAB R2008a to solve a binary ILP problem which is based on the branch-and-bound method. Hence, we introduce the branch-and-bound method.

2.6. THE BRANCH-AND-BOUND METHOD

Basically, **branch-and-bound method** refers to a search procedure, i.e. a sequential division of the set of possible solutions to an integer programming problem into subsets. Bounds on the value of the objective function and feasibility criteria are used to limit the search for each subset. The most important property of branch-and-bound is its ability to enumerate the majority of the possible solutions of the integer programming problem implicitly. The number of solutions is finite when the values of the variables are bounded. The more solutions that can be enumerated implicitly, the quicker the optimal solution is identified. This is important for integer programming problems which exhibit explosive growth in the number or possible solutions as the number of variables increases. As an example, a binary integer programming problem of three variables has $2^3 = 8$ possible solutions of which not all may be feasible. A binary integer programming problem of ten and thirty variables has $2^{10} = 1,024$ and $2^{30} = 1,073,741,824$ possible solutions respectively.

The model derived from the graph representation of an MSA problem is a binary ILP model. We know that a binary ILP problem with *n* variables would have 2^n possible solutions. To solve the problem, a modified general branch-and-bound method is used. The method is referred as **implicit enumeration** (Wu N. and Coppins R. [8]). The term implicit enumeration implies that (hopefully) many of the possible 2^n solutions will be discarded by various feasibility tests and bound without requiring explicit enumeration.

We introduce some terminology for the implicit enumeration. Suppose we have assigned values to some of but not all the variables. The solution obtained from the assigned values is called a **partial solution**. The variables that are assigned values in a partial solution are said to be **fixed**, while the remaining variables are said to be **free**. A **completion** is made by assigning a specific set of values to the free variables.

Suppose we have a partial solution at some stage of the solving process. We can generate an upper bound on the optimal value if the objective function for all possible completions by letting each free variable be 1. The upper bound is

$$f_u = \sum_{\text{fixed variables}} c_j x_j + \sum_{\text{free variables}} c_j \ .$$

The feasibility is checked by using a similar approach that is by rewriting each constraint as

$$\sum_{\text{fixed variables}} a_{ij}x_j \le b_i - \sum_{\text{free variables}} a_{ij}x_j \text{ for } i = 1, 2, \ldots, m.$$

A constraint can be satisfied only if

$$\sum_{\text{fixed variables}} \min\left(a_{ij}, 0\right) \le b_i - \sum_{\text{free variables}} a_{ij}x_j$$

i.e. only if the free variables have sufficient negative coefficients. Thus we can eliminate the completion whenever

$$\sum_{\text{fixed variables}} \min\left(a_{ij}, 0\right) > b_i - \sum_{\text{free variables}} a_{ij}x_j .$$

The steps of the implicit enumeration are summarized as follow (Wu N. and Coppins R. [8]):

Step 1 Generate a lower bound f_L^1 by using any feasible solution. If none is obvious, set $f_L^1 = -\infty$. All variables are free.

Step 2 Select a free variable (say x_k) and use it to generate the separation $x_k = 0$ and $x_k = 1$. Move x_k to the set of fixed variables on each branch of the separation.

Step 3 For each new partial solution generated by the separation, compute the upper bound f_u of the objective function over all completions.

Step 4 Select the most recently created partial solution. It is eliminated when (1) $f_u < f_L$, or (2) there are no feasible completions, or (3) there are no free variables, or (4) the upper bound calculation generates a feasible completion. In this case if $f_u > f_L$, replace f_L with the value f_u and store the values of the variables as the new incumbent solution. When a solution has been eliminated, go to step 4, unless there are no remaining partial solutions.

Step 5 If there are no remaining partial solutions, stop. The current incumbent solution is optimal. Otherwise, go to step 2.

By applying the steps of the implicit enumeration to a binary ILP problem, we will obtain a tree that represents the branch-and-bound process of finding solution of the problem.

Consider the following binary ILP maximizing problem of four variables:

Example 2.2

$$\begin{array}{ll} \text{Max} & f = x_1 + 6x_2 + 3x_3 + 2x_4 \\ \text{subject to} & x_1 - 3x_2 - 2x_3 + 5x_4 \le -1 \\ & -2x_1 + x_2 + 2x_3 - 2x_4 \le 0 \\ & x_2 - 2x_3 + x_4 \le 1 \\ & x_1, x_2, x_3, x_4 = 0 \text{ or } 1 \end{array}$$

By applying the steps of the implicit enumeration, we obtain the tree representation of the branch-and-bound process of finding solution for Example 2.2 which is given in Figure 2.12.

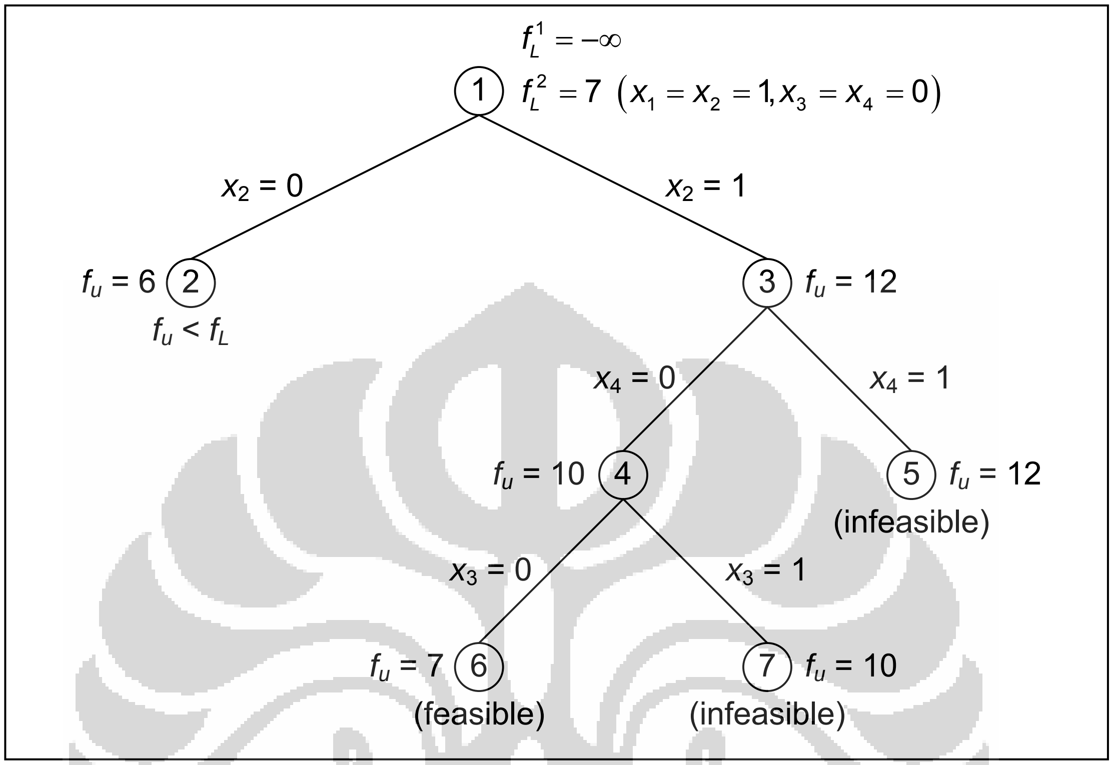


Fig. 2.12. Branch-and-bound tree for Example 2.2

First, we set $f_L^1 = -\infty$ at node 1 since there is no obvious feasible solution. All variables are free. Next, we use the x_2 to generate the separation $x_2 = 0$ (node 2) and $x_2 = 1$ (node 3). At each node we fix x_2. At node 3, $f_u = 12 > f_L^1$. By checking the feasibility upon the constraints of Example 2.2 ($x_2 = 1$), the completion remains feasible as long as $x_4 = 0$. Next, we use x_4 to generate the separation $x_4 = 0$ (node 4) and $x_4 = 1$ (node 5). At each node we fix x_4. By checking the feasibility upon the constraints of Example 2.2 ($x_2 = 1$ and $x_4 = 0$), the completion remains feasible as long as $x_3 = 0$. Next, we use x_3 to generate the separation $x_3 = 0$ (node 6) and $x_3 = 1$ (node 7). At each node we fix x_3. By checking the feasibility upon the constraints of Example 2.2 ($x_2 = 1$, $x_3 = 0$, and $x_4 = 0$), the feasible completion is $x_1 = 1$ with $f_u = 7$.

Since this is a completion and $f_u > f_L^1$, we set $f_L^2 = f_u = 7$ and store $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, and $x_4 = 0$ as the incumbent. At node 2, $f_u = 6 < f_L^2$. Hence, node 2 is eliminated. Since there are no remaining partial solutions, the current incumbent ($x_1 = 1$, $x_2 = 1$, $x_3 = 0$, and $x_4 = 0$) is optimal, with $f = 7$.