Language
Understanding
$\lambda(\mu)$ Laboratory

oppaWord: DAG-BiMM-LM based Myanmar Word Segmenter

(8th Seminar for Intern-3)

Ye Kyaw Thu
Lab Leader, LU Lab., Myanmar

27 July 2025 (SUN)

Contents

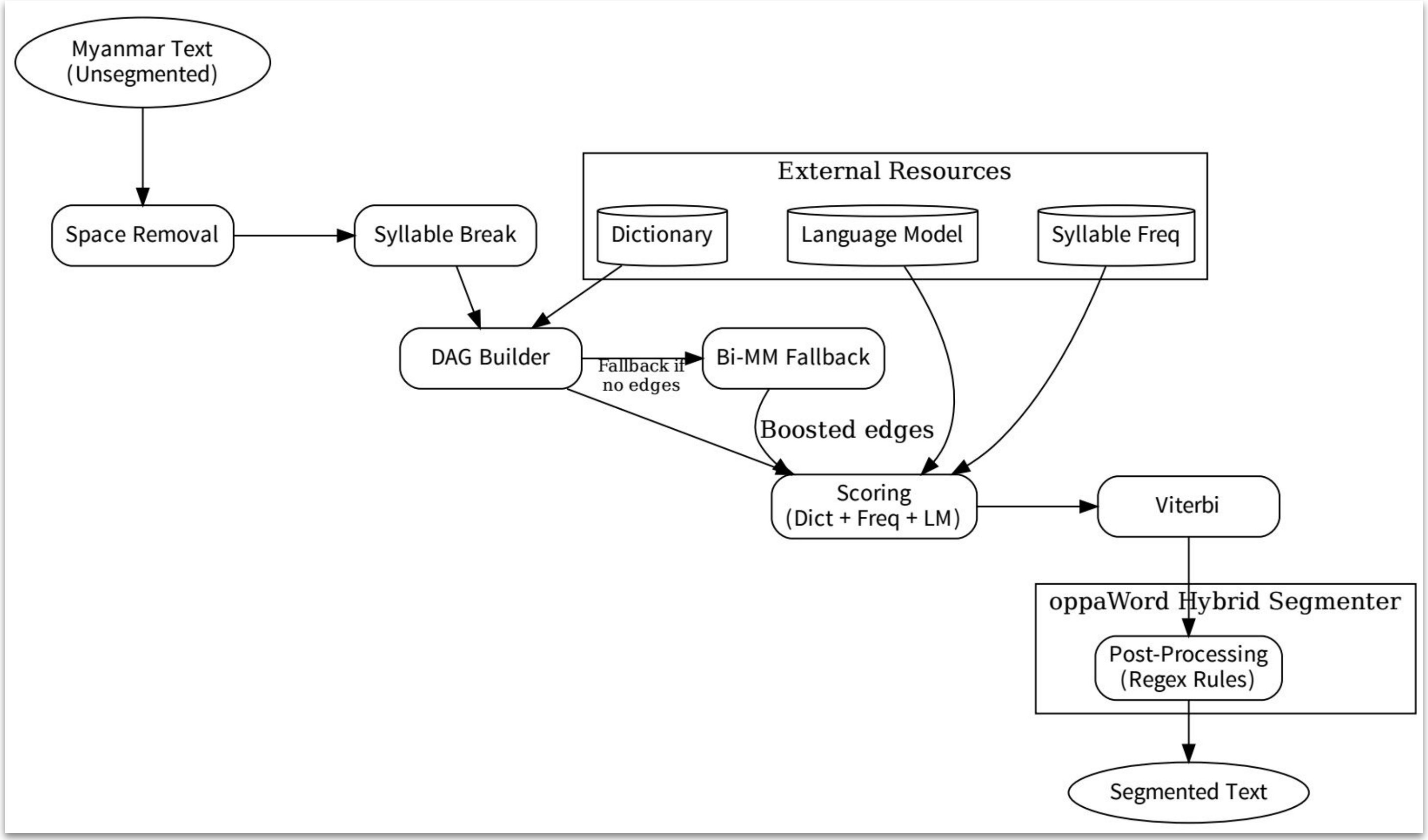
Myanmar Word Segmentation

- Challenge: No explicit word boundaries in Myanmar script (e.g., "ကျောင်းသား" = "ကျောင်း" + "သား").
- Agglutinative morphology: Words formed by combining syllables.
- Applications: NLP tasks like machine translation (MT), search engines, text-to-speech (TTS), automatic speech recognition (ASR), and Summarization.

Core Algorithms

- Hybrid Approach
- DAG Construction: Represent all possible segmentations.
- Bi-MM Fallback: Bidirectional Maximum Matching for robustness.
- N-gramLanguage Model
- Syllable-level processing.
- Multi-feature scoring (dictionary, frequency, LM).

Overview of oppaWord



Smart Space Remover

- Word Segmenter တွေကို input လုပ်တဲ့အခါမှာ space ဖြုတ်ပေးကြရတယ်
- သို့သော် လက်တွေ့မှာ space အကုန်ဖြုတ်လိုက်ရင် နံပါတ် ကိစ္စတွေ၊ မြန်မာစာနဲ့အတူ ရောပါနေတဲ့ အင်္ဂလိပ်စာလုံးတွေ၊ တရုတ်စာလုံးတွေ၊ ဂျပန်စာလုံးတွေ၊ ထိုင်းစာလုံးတွေ ကိုရော ဘယ်လို လုပ်ကြမလဲ။
- မော်ဒ်တွေထဲကို input လုပ်ရတဲ့အခါမှာ preprocessing တွေ လုပ်ပေးကြရတယ်
- အဲဒီ ကိစ္စကို ဒီတခါမှာတော့ smart_space_remover.py နဲ့ ရှင်းခဲ့တယ်

Smart Space Remover

- Chonburi ကိုသွားမယ့်နောက်လေယာဉ်မှာလက်မှတ်တစ်စောင်ပေးပါ
- domain name တိုင်း Top level domain TLD name နဲ့အဆုံးသတ်ရပါတယ်
- ဝမ် ၈၁၀၀ မှာမှတ်ပုံတင်ကြေးဝမ် ၈၀၀ ပေါင်းရင်စုစုပေါင်းဝမ် ၈၉၀၀ ကျပါတယ်
- ဂျပန်တို့ကစင်ကာပူကိုရှိုးနန်တိုး 昭南島 Shōnantō ဟုအမည်ပေးခဲ့ပြီး 昭和の時代に得た南の島 သို့ရှိုဝါခေတ်တွင်ရရှိခဲ့သောတောင်ဘက်ကျွန်းဟုအဓိပ္ပါယ်ရသည်
- အမျိုးသားဦးရေ ၂၉ ဒသမ ၇၂ သန်းနှင့်အမျိုးသမီးဦးရေ ၃၀ ဒသမ ၀၆ သန်းရှိသည်
- ဒီကြီးတဲ့ပစ္စည်းအတွက်ဆိုရင်တစ်နေ့ကို ၁.၅၀ ကျတယ်

Syllable Breaking (edited-sylbreak)

- Original sylbreak.py ရဲ့ Other words ဆိုတာကို ဖြုတ်လိုက်တယ်

```python
# From oppaWord.py
consonants = r"က-အ"
punctuation = r"၊|။"
subscript = r"္"  # Virama (kinzi)
a_that = r"်"  # Asat

break_pattern = re.compile(
    rf"((?<!{subscript})([{consonants}]|{punctuation})(?![{a_that}{subscript}]))"
)
```

DAG (Directed Acyclic Graph)

Definition:

- A Directed Acyclic Graph (DAG) is a graph with directed edges and no cycles.
- Nodes: Represent syllable positions in the input text.
- Edges: Represent candidate words (valid dictionary entries or single syllables).

Key Property:

- No cycles → Ensures linear progression from start to end of the sentence.

DAG Consturction

- In mathematics, particularly graph theory, and computer science, a directed acyclic graph (DAG) is a directed graph with no directed cycles.

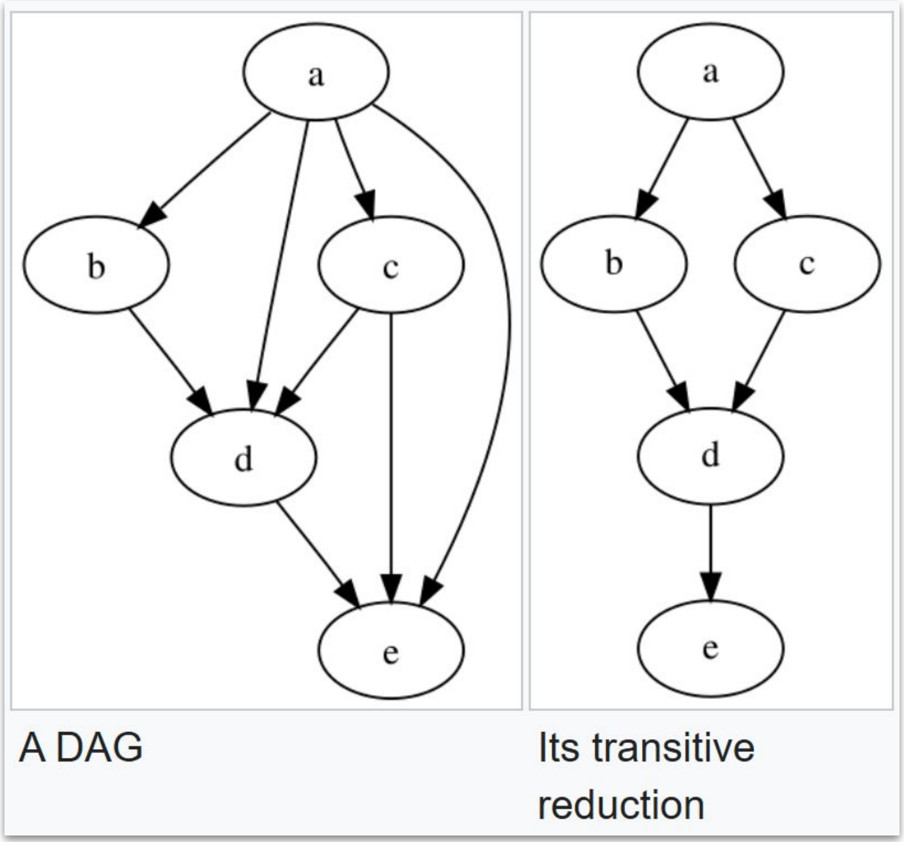


Fig. Transitive reduction (from wiki)

DAG Construction

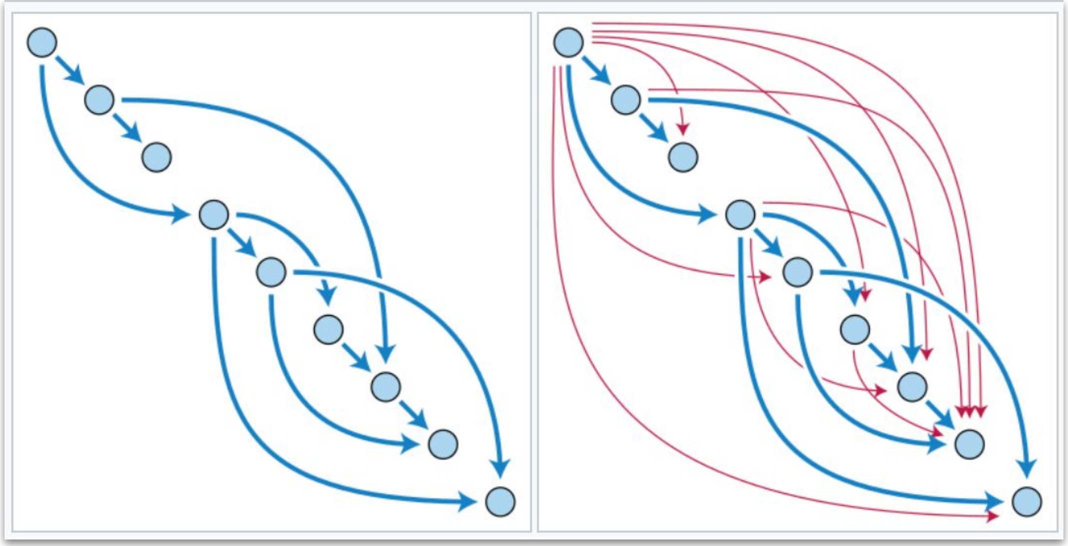


Fig. (from wiki)

- Adding the red edges to the blue directed acyclic graph produces another DAG, the transitive closure of the blue graph. For each red or blue edge u → v, v is reachable from u: there exists a blue path starting at u and ending at v.

DAG Construction (Why DAGs for Word Segmentation?)

Problem:

- Myanmar text has no explicit word boundaries (e.g., "ကျေးဇူးပြု၍" → "ကျေးဇူး" + "ပြု" + "၍").
- Exponentially many possible segmentations.

Solution:

- DAG compactly encodes all valid segmentations in a single structure.
- Enables efficient search for the best path (e.g., using Viterbi).

DAG Construction (Step by Step)

- **Input:** Syllable list (e.g., ["ကျွန်တော်", "မ", "သိ", "ပါ"]).
- **Sliding Window:** For each syllable position i, check substrings of length 1 to max_word_len (e.g., 6).
- **Validation:** Keep substrings that exist in the dictionary or are single syllables.

```python
for i in range(len(syllables)):
    for j in range(i+1, min(i+max_word_len+1, len(syllables)+1)):
        word = syllables[i:j]
        if word in dictionary or j-i == 1:
            dag[i].append((j, word))  # Edge to node j
```

DAG Construction (Example)

- **Input Sentence:** "ကျေးဇူးပြု၍" (syllables: ["ကျေးဇူး", "ပြု", "၍"]).
- **DAG Edges:**
  - **0 → 1:** ကျေးဇူး
  - **0 → 2:** ကျေးဇူးပြု
  - **1 → 2:** ပြု
  - **2 → 3:** ၍

DAG Construction (vs Other Approaches)

Approach	Pros	Cons
DAG	Captures all possibilities	Requires scoring to pick best path
Bi-MM	Fast, simple	Greedy; may miss optimal splits
Pure LM	Context-aware	Slow for long sentences

DAG Construction (Scoring in oppaWord)

Edge Weights: Combine:

1. Dictionary membership (dict_weight).
2. Syllable frequency (log probability).
3. Language model score (n-gram probability).

Example:

- Edge 0 → 1 ("ကျေးဇူး"):
  score = 10.0 (dict) + -1.2 (freq) + -0.5 (LM) = 8.3

DAG Construction (Path Finding with Viterbi)

- **Input Sentence:** Find the highest-scoring path from start (node 0) to end (node N).
- **Dynamic Programming:**
  - scores[j] = max(scores[i] + edge_score) for all edges i → j.
- **Implementation:**

```
for i in range(n):
    for (j, word, is_bimm) in dag[i]:
        scores[j] = max(scores[j], scores[i] + get_score(word))
```

DAG Construction (Visualization)

- Tool: Graphviz (dot).

Example:

```
digraph DAG {
  0 -> 1 [label="ကျေးဇူး (8.3)"];
  0 -> 2 [label="ကျေးဇူးပြု (5.1)"];
  1 -> 2 [label="ပြု (7.0)"];
  2 -> 3 [label="၍ (1.0)"];
}
```

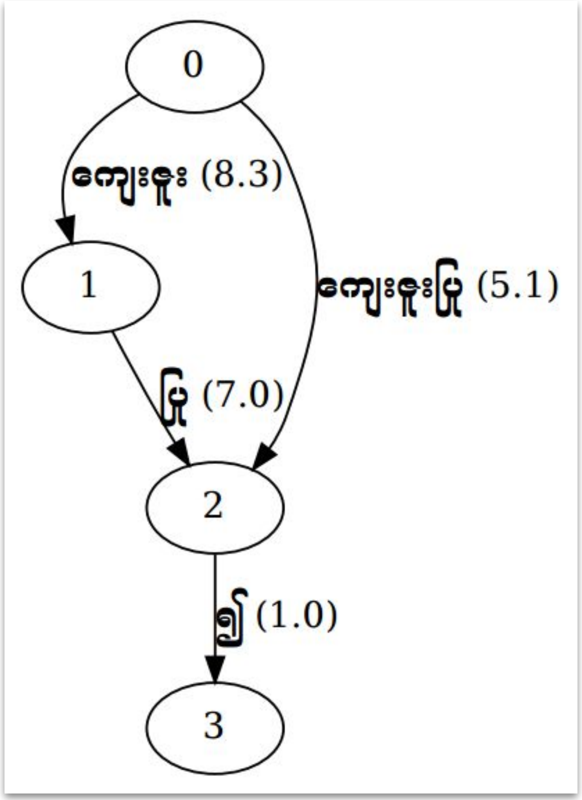


Fig. Visualization

DAG Construction (Limitation)

- **Memory:** Can grow large for long sentences (O(N × max_word_len) edges).
- **Ambiguity:** Multiple valid paths may exist
- **Mitigation:**
  - Limit max_word_len (default: 6).
  - Fall back to Bi-MM for low-confidence paths.

Bi-MM (Bidirectional Maximum Matching)

- **MM Definition:** A greedy algorithm that segments text by matching the longest possible word from a dictionary.
- **Directions:**
  - **Forward Maximum Matching (FMM):** Left-to-right.
  - **Backward Maximum Matching (BMM):** Right-to-left.
- **Why Bidirectional?**
- Unidirectional MM can be direction-sensitive.
  - Example: "အပြင်သွား" → FMM: "အပြင်" + "သွား",
  - BMM: "အ" + "ပြင်" + "သွား".

Bi-MM (Bidirectional Maximum Matching)

- **Solution:**
  - Run both FMM and BMM, then pick the better result (fewer segments or higher score).
- **Advantage:**
  - More robust to dictionary gaps and ambiguities.

Bi-MM (Algorithm Steps)

- **Forward Pass (FMM):**
  - Start at the first syllable.
  - Match longest valid word from the dictionary.
  - Move to the next unmatched syllable.
- **Backward Pass (BMM):**
  - Start at the last syllable.
  - Match longest valid word backwards.
- **Conflict Resolution:**
  - Choose the segmentation with fewer words (or apply scoring).

Bi-MM (Implementation)

```python
def _forward_mm(syllables):
    result = []
    i = 0
    while i < len(syllables):
        for j in range(min(max_word_len, len(syllables)-i), 0, -1):
            word = ''.join(syllables[i:i+j])
            if word in word_dict:
                result.append((i, i+j, word))  # (start, end, word)
                i += j
                break
        else:  # No match → single syllable
            result.append((i, i+1, syllables[i]))
            i += 1
    return result
```

Bi-MM (Integration with DAG in oppaWord)

- **Role of Bi-MM:**
  - **Fallback Path:** When DAG edges are missing (low-confidence/no dictionary match).
  - **Score Boosting:** Bi-MM paths get a bonus (bimm_boost parameter).

```
if use_bimm_fallback:
    bimmpath = self._get_bimm_segmentation(syllables)
    for start, end, word in bimmpath:
        dag[start].append((end, word, True))  # Mark as Bi-MM edge
```

Bi-MM (vs. DAG Scoring)

Feature	Bi-MM	DAG
Approach	Greedy, rule-based	Exhaustive, probabilistic
Speed	Faster	Slower (scales with sentence length)
Accuracy	Lower (dictionary-dependent)	Higher (LM + frequency-aware)
Use Case	Fallback for DAG gaps	Primary segmentation

Bi-MM (Configuration in oppaWord)

- **Parameters:**
  - **--use-bimm-fallback:** Enable/disable Bi-MM.
  - **--bimm-boost 0.5:** Boost Bi-MM path scores (default: 0.0).
- **Trade-offs:**
  - Higher boost → Prefer Bi-MM over DAG paths (useful for noisy text).

Bi-MM (Limitations)

- **Dictionary Bias:**
  - Fails if dictionary is incomplete (e.g., missing compound words).
- **Directional Bias:**
  - BMM and FMM may disagree (requires heuristic resolution).
- **No Context Awareness:**
  - Ignores LM/frequency scores (fixed in DAG).

Bi-MM (Practical Tips)

- **Dictionary Quality:**
  - Ensure coverage of common words and compounds.
- **Tuning bimm_boost:**
  - Increase if DAG often produces unnatural splits.
  - Decrease to rely more on LM.
- **Visualization:**
  - Use **--visualize-dag** to see Bi-MM fallback edges (marked with *).

N-gram LM

- **Definition:**
  - Predicts the probability of a word given its preceding **n-1** words.
- **Example (Trigram: n=3):**
  - P("သွား" | "ကျောင်း" + "ကို") = probability of "သွား" after "ကျောင်းကို".
- **Types:**
  - **Unigram:** Single word frequency (P("ကျေးဇူး")).
  - **Bigram:** Conditional probability (P("ပြု" | "ကျေးဇူး")).
  - **Higher-order:** Trigram, 4-gram, etc.

N-gram LM (for Segmentation?)

- **Problem:**
  - Dictionary-based methods (e.g., Bi-MM) lack context awareness.
  - Example: "ဘုန်းကြီးကျောင်းသား" vs. "ဘုန်းကြီး" + "ကျောင်းသား"
  - LM picks the more probable sequence.
- **Solution:**
  - LMs score candidate segmentations using real-world word co-occurrence statistics.

N-gram LM (Implementation in oppaWord)

- **Supported Formats:**
  - **ARPA:** Human-readable n-gram counts and probabilities.
  - **KenLM Binary:** Efficient memory-mapped format.

```
def _get_lm_score(self, history, word):
    if isinstance(self.lm, dict):  # ARPA
        ngram = ' '.join(history[-n:] + [word])  # Construct n-gram
        return self.lm.get(ngram, self.unk_logprob)
    else:  # KenLM
        return self.lm.score(' '.join(history + [word]))
```

N-gram LM (Scoring)

- **Input Syllables:** ["နေ", "ကောင်း", "လား"]
- **Candidate Paths:**
  a. ["နေ", "ကောင်း"] + ["လား"]
     - Bigram score: P("ကောင်း" | "နေ") + P("လား" | "ကောင်း")
  b. ["နေကောင်း"] + ["လား"]
     - Unigram score: P("နေကောင်း") + P("လား")
- Result: Higher-probability path wins.

N-gram LM (Integration with DAG)

- **Step-by-Step:**
- **DAG Construction:** All possible segmentations (edges).
- **Viterbi Decoding:** Scores paths using:

```
total_score = dict_weight + syl_freq + lm_score + (bimm_boost if fallback)
```

- **Visualization:**
  - DAG edges show LM scores (e.g., "နေကောင်း (LM:-1.2)").

N-gram LM (Configuring LM in oppaWord)

- **Parameters:**
- **Candidate Paths:**
  - **--arpa lm.arpa:** Path to ARPA/KenLM file.
  - **--max-order 5:** Max n-gram order (default: 5).
- **Trade-offs:** \
  - Higher max-order → More context but slower.

N-gram LM (Limitations & Mitigations)

- **Sparsity:** Rare n-grams → fallback to lower-order models.
- **oppaWord's Solution:** Uses unk_logprob for unknown words.
- **Memory:**
  - Large LMs need KenLM binaries.
- **Summary:**
  - **n-gram LMs** add context awareness to segmentation.
  - **oppaWord Hybrid:** Combines DAG, Bi-MM, and LM for optimal accuracy.

Viterbi Algorithm

- **Definition:**
  - A dynamic programming algorithm to find the most likely sequence of states (here, word segments) in a probabilistic model (DAG + LM).
- **Key Properties:**
  - **Optimal:** Guarantees the highest-scoring path.
- **Efficient:** O(N×M) time (N = syllables, M = max word length).

Viterbi Algorithm (for Word Segmentation?!)

- **Challenge:**
  - Exponentially many possible segmentations
- **Solution:**
  - Viterbi efficiently explores all paths in the DAG using memoization.

Viterbi Algorithm (Core Component in oppaWord)

- **DAG Representation:**
  - Nodes = syllable positions.
  - Edges = candidate words (e.g., 0 → 2: "ကျေးရူး").
- **Scoring:**
  - Each edge has a score:

```
score = dict_weight + syl_freq + lm_score + (bimm_boost if fallback)
```

Viterbi Algorithm (Step by Step)

- **Initialization:**
  - **scores[0] = 0** (start node), others = -∞.
  - **paths[j]** stores the best path to node **j**.
- **Recursion:**
  - For each node **i**, update all outgoing edges *i → j*:

```
if scores[j] < scores[i] + edge_score:
    scores[j] = scores[i] + edge_score  # Update best score
    paths[j] = (i, word)                # Track best path
```

- **Termination:**
  - Backtrack from the end node (e.g., node N) using paths.

Viterbi Algorithm (Example Walkthrough)

- **Input Syllables:** ["နေ", "ကောင်း", "လား"]
- **DAG Edges:**
  - 0 → 1: "နေ" (score=8), 0 → 2: "နေကောင်း" (score=12)
  - 1 → 2: "ကောင်း" (score=7), 2 → 3: "လား" (score=5)
- **Viterbi Steps:**
  1. Scores: [0, 8, 12, -∞]
  2. Update node 2: max(8+7=15, 12+5=17) → scores=[0,8,17]
  3. Best path: 0 → 2 → 3 → "နေကောင်း လား"

Viterbi Algorithm (Implementation)

```python
# Initialize
scores = [-float('inf')] * (n+1)
scores[0] = 0
paths = [None] * (n+1)

# Recursion
for i in range(n):
    for (j, word, is_bimm) in dag[i]:
        edge_score = dict_score + syl_score + lm_score
        if scores[j] < scores[i] + edge_score:
            scores[j] = scores[i] + edge_score
            paths[j] = (i, word)

# Backtracking
segmented = []
idx = n
while idx > 0:
    i, word = paths[idx]
    segmented.append(word)
    idx = i
```

Viterbi Algorithm (Summary)

- **Viterbi** finds the globally optimal segmentation in O(N×M) time.
- **oppaWord Hybrid:** Combines DAG, LM scores, and Viterbi for accuracy.

References

1. DAG: https://en.wikipedia.org/wiki/Directed_acyclic_graph
2. Viterbi Algorithm: https://en.wikipedia.org/wiki/Viterbi_algorithm
3. Viterbi AJ (April 1967). "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm". IEEE Transactions on Information Theory. 13 (2): 260–269.
4. Gai, R. L., Gao, F., Duan, L. M., Sun, X. H., & Li, H. Z. (2014). Bidirectional Maximal Matching Word Segmentation Algorithm with Rules. Advanced Materials Research, 926–930, 3368–3372.
5. N-gram: https://en.wikipedia.org/wiki/N-gram
6. Word n-gram Language Model: https://en.wikipedia.org/wiki/Word_n-gram_language_model
7. KenLM: https://github.com/kpu/kenlm
8. KenLM website: https://kheafield.com/code/kenlm/

References

9. Sylbreak: https://github.com/ye-kyaw-thu/sylbreak
10. oppaWord: https://github.com/ye-kyaw-thu/oppaWord
11. Dynaic Programming: https://en.wikipedia.org/wiki/Dynamic_programming
12. myWord Segmentation Tool: https://github.com/ye-kyaw-thu/myWord
13. Ngram Format: http://www.speech.sri.com/projects/srilm/manpages/ngram-format.5.html
14. ARPA Language Models: https://cmusphinx.github.io/wiki/arpaformat/