# Beam Search

## Winter Vacation Capstone Study

TEAM Kai.Lib

발표자 : 원철황

2019.12.27 (FRI)

# Beam Search

## 탐색 (探索, search)

- 문제의 해(solution)가 될 수 있는 것들의 집합을 공간(space)으로 간주하고,문제에 대한 최적의 해를 찾기 위해 공간을 체계적으로 찾아보는 것

## 탐색문제의 예

- 선교사 –식인종 강건너기 문제

- 8–퍼즐 문제

| | | |
|---|---|---|
| 8 | | 6 |
| 5 | 4 | 7 |
| 2 | 3 | 1 |

- 8–퀸(queen) 문제

- 틱–택–토(tic–tac–toe)

- 루빅스큐브 (Rubik's cube)

- 순회 판매자 문제 (traveling salesperson problem, TSP)

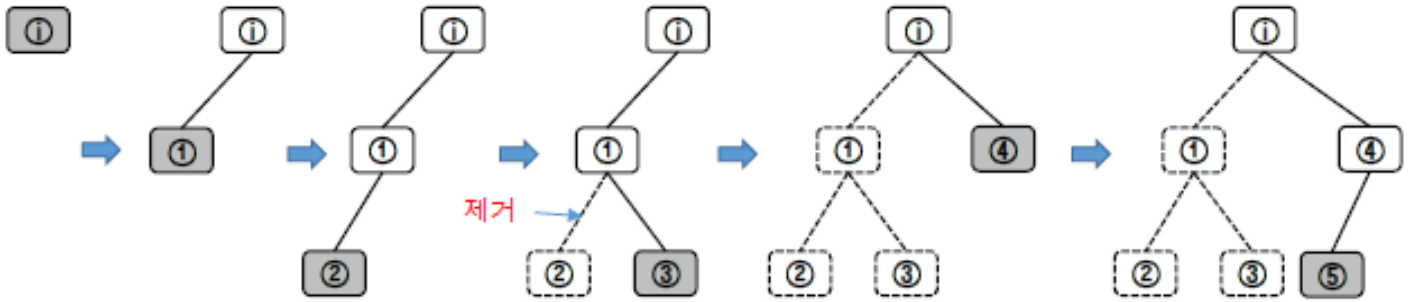

- 해(解, solution)
  일련의 동작으로 구성되거나 하나의 상태로 구성

# Beam Search

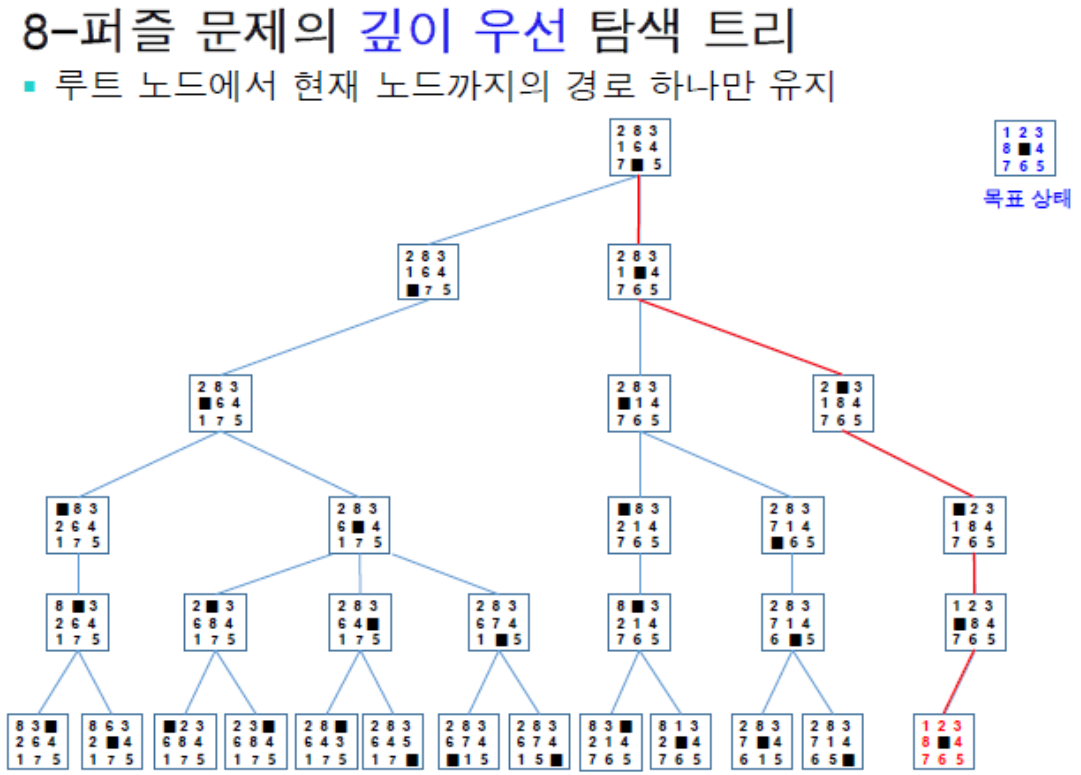
## 맹목적 탐색(blind search)

- 정해진 순서에 따라 상태 공간 그래프를 점차 생성해 가면서 해를 탐색하는 방법

- 깊이 우선 탐색(depth-first search, DFS)
  - 초기 노드에서 시작하여 깊이 방향으로 탐색
  - 목표 노드에 도달하면 종료
  - 더 이상 진행할 수 없으면, 백트랙킹(backtracking, 되짚어 가기)
  - 방문한 노드는 재방문하지 않음

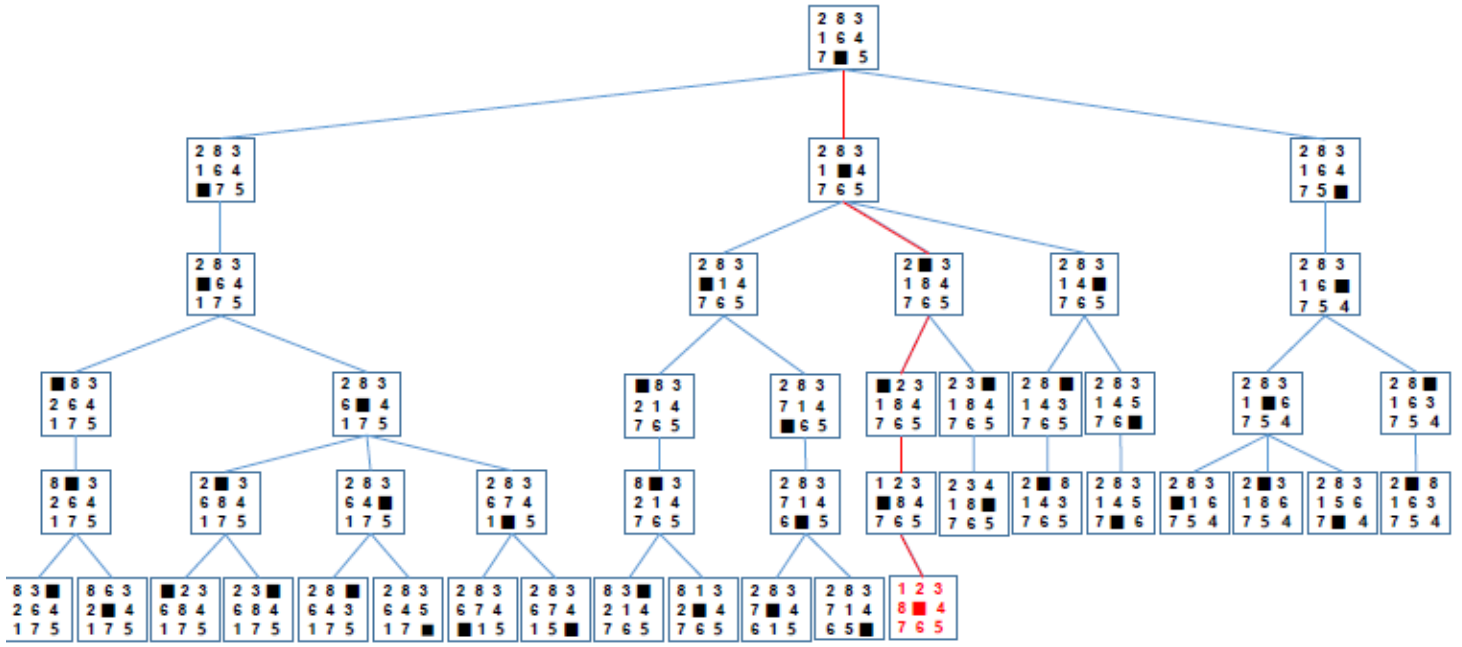


# Beam Search



# Beam Search

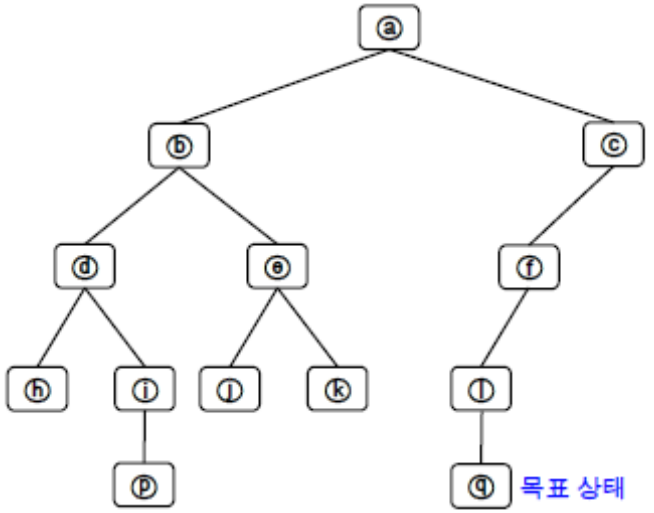
- 8-퍼즐 문제의 너비 우선 탐색 트리
  - 전체 트리를 메모리에서 관리



# Beam Search

## 반복적 깊이심화 탐색(iterative-deepening search)

- 깊이 한계가 있는 깊이 우선 탐색을 반복적으로 적용




깊이 0: ⓐ
깊이 1: ⓐ,ⓑ,ⓒ
깊이 2: ⓐ,ⓑ,ⓓ,ⓔ,ⓒ,ⓕ,ⓖ
깊이 3: ⓐ,ⓑ,ⓓ,ⓗ,ⓘ,ⓔ,ⓙ,ⓚ,ⓒ,ⓕ,ⓛ,ⓜ,ⓖ,ⓝ,ⓞ
깊이 4: ⓐ,ⓑ,ⓓ,ⓗ,ⓘ,ⓟ,ⓔ,ⓙ,ⓚ,ⓒ,ⓕ,ⓛ,ⓠ

# Beam Search

## 맹목적 탐색 방법의 비교

- 깊이 우선 탐색
  - 메모리 공간 사용 효율적
  - 최단 경로 해 탐색 보장 불가
- 너비 우선 탐색
  - 최단 경로 해 탐색 보장
  - 메모리 공간 사용 비효율
- 반복적 깊이심화 탐색
  - 최단 경로 해 보장
  - 메모리 공간 사용 효율적
  - 반복적인 깊이 우선 탐색에 따른 비효율성
    - 실제 비용이 크게 늘지 않음
    - 각 노드가 10개의 자식노드를 가질 때,
      너비 우선 탐색 대비 약 11%정도 추가 노드 생성
  - 맹목적 탐색 적용시 우선 고려 대상

- 탐색 효율을 더 발전시키는 방법은 없을까?

- 인간처럼 탐색할 수 없을까?

- 사고과정을 모방하는 탐색 방법을 구현할 수 있을까?

# Beam Search

## 정보이용 탐색(informed search)

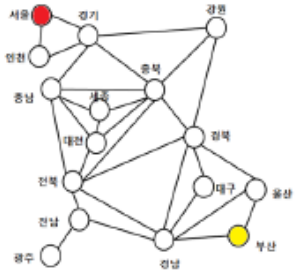
- 휴리스틱 탐색(heuristic search)
- 언덕 오르기 방법(hill climbing), 최상 우선(best-first) 탐색, 빔(beam) 탐색, A* 알고리즘 등

- 휴리스틱(heuristic)
  - 그리스어 Εὑρίσκω (Eurisko, 찾다, 발견하다)
  - 시간이나 정보가 불충분하여 합리적인 판단을 할 수 없거나, 굳이 체계적이고 합리적인 판단을 할 필요가 없는 상황에서 신속하게 어림짐작하는 것
  - 예.
    - 최단 경로 문제에서 목적지까지 남은 거리
      - 현재 위치에서 목적지(목표 상태)까지 지도상의 직선 거리

# Beam Search

## 휴리스틱 비용 추정의 예

- 최단경로 문제
  - ➔ 현재 위치에서 목적지까지 직선 거리




- 8-퍼즐 문제
  - ➔ 제자리에 있지 않는 타일의 개수

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

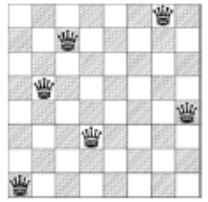
현재 상태

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

목표 상태

추정비용 : 4

- 8-퀸 문제
  - ➔ 충돌하는 회수



# Beam Search

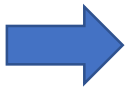
## 빔 탐색(beam search)

- 휴리스틱에 의한 평가값이 우수한 일정 개수의 확장 가능한 노드만을 메모리에 관리하면서 최상 우선 탐색을 적용

# Beam Search

$$P(w_1, w_2) = P(w_1)P(w_2 \mid w_1),$$

$$\text{because } P(w_2 \mid w_1) = \frac{P(w_1, w_2)}{P(w_1)}$$



$$P(w_1, w_2, \cdots, w_n) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1, w_2)\cdots P(w_n \mid w_1, w_2, \cdots, w_{n-1})$$

**※단어들 사이의 확률적 연쇄법칙**

$$\begin{aligned} P(A,B,C,D) &= P(D \mid A,B,C)P(A,B,C) \\ &= P(D \mid A,B,C)P(C \mid A,B)P(A,B) \\ &= P(D \mid A,B,C)P(C \mid A,B)P(B \mid A)P(A) \end{aligned}$$

이로부터 단어와 단어 사이의 확률을 비교할 수 있게 되었다.

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_{<i})$$

따라서, 고정적이지 않은 list의 index 값을 input으로 하여 서로 다른 길이의 단어 확률을 계산할 수 있다.

※ 공식의 일반화 => Flexible Coding 가능

# Beam Search

이때, 문장이 길어지면 (Character가 길어지면) 확률에 대한 곱셈이 거듭되면서 확률이 매우 낮아져 계산 또는 표현이 어려워진다.

$$\log P(w_1, w_2, \cdots, w_n) = \sum_{i=1}^{n} \log P(w_i \mid w_{<i})$$

따라서 양 변에 로그를 취해 덧셈으로 바꾼 형태로 연산한다.
※ 계산상으로는 차이가 거의 없으나 Computing 연산 면에서 이득.

$$C = \{s_1, s_2, \cdots, s_n\}$$

$$s_i = \{\text{BOS}, \text{나는}, \text{학교에}, \text{갑니다}, \text{EOS}\}$$

$$P(\text{BOS}, \text{나는}, \text{학교에}, \text{갑니다}, \text{EOS}) = P(\text{BOS})P(\text{나는} \mid \text{BOS})P(\text{학교에} \mid \text{BOS}, \text{나는})P(\text{갑니다} \mid \text{BOS}, \text{나는}, \text{학교에})\, P(\text{EOS} \mid \text{BOS}, \cdots, \text{갑니다})$$

이제 확률을 계산하는 방법으로 사후 확률 (Posterior Probability)는 단어 조합의 출현 빈도를 세어 추정할 수 있다.
따라서 수집한 말뭉치에 따라, 데이터를 어떻게 구성함에 따라 다음 확률 값이 달라지며 정확도가 결정된다.

$$P(\text{갑니다} \mid \text{BOS}, \text{나는}, \text{학교에}) \approx \frac{\text{Count}(\text{BOS}, \text{나는}, \text{학교에}, \text{갑니다})}{\text{Count}(\text{BOS}, \text{나는}, \text{학교에})}$$

이렇게 되면 희소성 문제에 봉착하게 된다. 수많은 문장을 온라인에서 수집해도, 애초에 출현 가능한 단어의 조합의 경우의 수는 무한히 많다고 가정할 수 있으며, 이는 곧 단어의 조합이 조금만 길어져도 분자가 0이 되면서 (곱이 많아지면서) 확률 값이 0으로 근사하게 된다.

# Beam Search

그래서. 그렇기에 마르코프 가정을 사용하는 것이었다. 특정 시점의 상태 확률을 이전 모든 상태를 고려하는 것이 아니라 직전 상태에만 의존하여 독립적으로 계산한다는 논리이다.
※ 물론 " 언어" 라는 System 상에서 각 단어의 "독립성 " 은 절대적 모순이지만 그 "상관성 " 역시 "언어 " 라는 시스템 안에서 매우 미미하기 때문에 이 마르코프 가정을 활용하는데 큰 무리는 없다고 생각합니다. (본인 생각)

$$P(x_i \mid x_1, x_2, \cdots, x_{i-1}) \approx P(x_i \mid x_{i-k}, \cdots, x_{i-1})$$

여기서 k가 앞의 단어의 수를 나타낸다. K = 2 라는 것은 앞 단어 2개를 참조하여 다음 단어 (x_i) 확률을 근사하여 나타낸다는 뜻이다.

$$P(x_i \mid x_{i-2}, x_{i-1})$$

여기에 연쇄법칙을 적용하면 문장에 대한 확률도 다음과 같이 표현할 수 있다.

$$P(x_1, x_2, \cdots, x_n) \approx \prod_{i=1}^{n} P(x_i \mid x_{i-k}, \cdots, x_{i-1})$$

이를 마지막으로 로그로 표현하면 다음과 같다.

$$\log P(x_1, x_2, \cdots, x_n) \approx \sum_{i=1}^{n} \log P(x_i \mid x_{i-k}, \cdots, x_{i-1})$$ ——— 여기서 k 가 무한으로 늘어난 것이 이전까지 진행했던 확률 계산 방법입니다.

이 방법을 n-gram이라고 한다. N이 커질수록 앞서 수식에 따르면 corpus에 출현하지 않은 단어 시퀀스가 들어간 사후 확률 (직접 관측하여 얻는 확률) 값은 0이 되지만, 실제 세상에 없지는 않으므로 모델과 오차가 생기게 된다.

$$P(x_i \mid x_{i-2}, x_{i-1}) = \frac{\mathrm{Count}(x_{i-2}, x_{i-1}, x_i)}{\mathrm{Count}(x_{i-2}, x_{i-1})}$$ ——— n 이 3일 때 확률 추론

# Beam Search

X는 Character 집합을 뜻한다.

Given $X = x_1, x_2, \cdots, x_n$,

we need to find $b^*$

b-star 는 가장 최적의 Path

where $\mathcal{B}_t = \{b_t^1, b_t^2, \cdots, b_t^K\}$ and $b_t^i = \{\hat{y}_1^i, \cdots, \hat{y}_t^i\}$

앞서 확인했던 이전 t-1 step까지 path들의 확률 값을 고려한 t번째 가장 높은 확률 값을 찾는 공식

$$\hat{y}_t^k = \underset{y \in \mathcal{Y}}{\text{argmax}}^k \{\log P_\theta(y_t \mid X, b_{t-1}^1) + \log P_\theta(b_{t-1}^K \mid X), \cdots, \log P_\theta(y_t \mid X, b_{t-1}^1) + \log P_\theta(b_{t-1}^K \mid X)\}$$

$$\log \tilde{P}(\hat{Y} \mid X) = \log P(\hat{Y} \mid X) \times \text{penalty}$$

$$\text{penalty} = \frac{(1+\text{length})^\alpha}{(1+\beta)^\alpha}$$

where $\beta$ is hyper-parameter of minimum length.

이때. 문장 길이가 길어질 수록 확률 값이 작아지므로 위와 같은 패널티를 두어 길이간 확률 간 오차를 줄인다. 알파는 1.2. 베타는 5 값을 주는 것이 일반적이다.

$$b_t^k = \{\hat{y}_1^k, \cdots, \hat{y}_{t-1}^k\} + \{\hat{y}_t^k\} = \{\hat{y}_1^k, \cdots, \hat{y}_t^k\}$$
$$= \underset{b \in \mathcal{B}_t}{\text{argmax}}^k \{\log P_\theta(y_t \mid X, b_{t-1}^1) + \log P_\theta(b_{t-1}^K \mid X), \cdots, \log P_\theta(y_t \mid X, b_{t-1}^1) + \log P_\theta(b_{t-1}^K \mid X)\} + \{\hat{y}_t^k\}$$

b_t 값은 t-1 까지 step의 확률 값을 고려한 다음 그 다음 t 값을 더해 고려한다.

where $\log P_\theta(b_t^i \mid X) = \log P_\theta(y_t = \hat{y}_t^i \mid X) + \log P_\theta(b_t^i \mid X)$

$$= \sum_{i=1}^{t} \log P_\theta(y_t = \hat{y}_t^i \mid X, \hat{y}_{<t}^i)$$

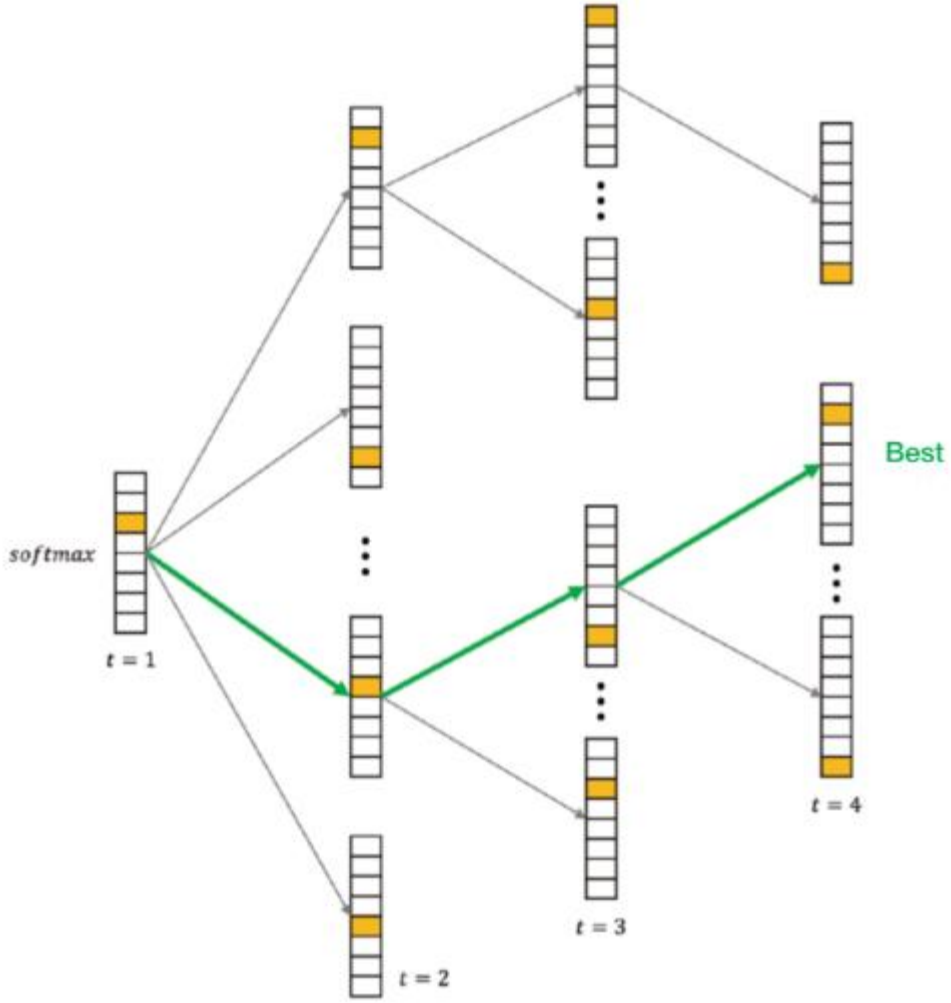
앞서 확인했던 확률의 연쇄법칙을 beam이라는 path의 확률과 y_t 라는 time step t 번째에서 target word로 나타낸 것

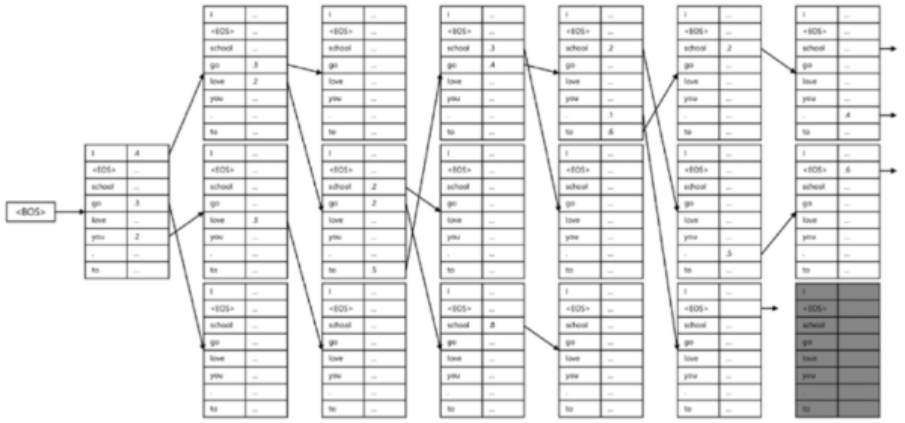
b[-1] 가 문장의 끝일 때, 연산을 멈추고 그 값을 판단한다.

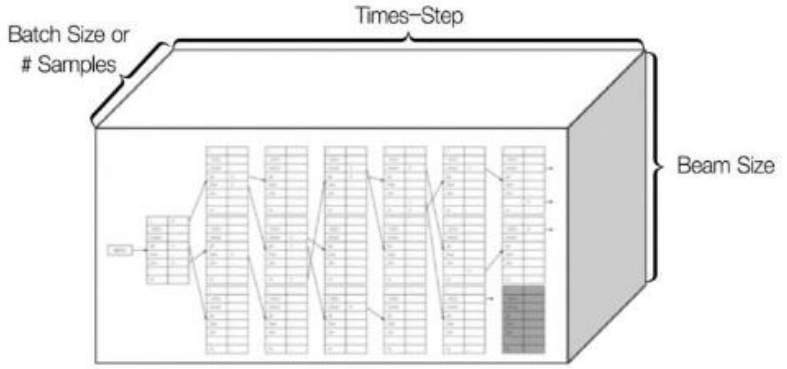
Pick best beam path $b$,

where $\hat{b} = \underset{b}{\text{argmax}} \{b_1^1, \cdots, b_1^k, b_2^1, \cdots, b_t^k : \text{where } b_j^i[-1] = \text{EOS}\}$

# Beam Search




이렇게 매 step마다 확률 값을 판단하여 path를 3개 씩 유지하며 진행하는 모습을 도식화 한 그림이다.

Beam 의 개수 == EOS 값이 될 때까지 진행되며, 그림처럼 EOS가 하나 발생할 때, 다시 3개의 Search를 시작함을 관찰할 수 있다.

# Beam Search

| 번호 | 빔 크기 k=1 | 빔 크기 k=5 |
|---|---|---|
| 1 | 너 스스로 말하고, 나를 위해 말하지마. | 당신 자신을 위해 말하세요, 당신은 나를 위해 말하지 않아요. |
| 2 | 걱정마. 난 그것에 익숙해. | 걱정마. 난 그것에 익숙해져 있어. |
| 3 | 너는 내 상사에게 말해야 한다. | 너는 나의 상사에게 말해야 한다. |
| 4 | 그녀는 그와 결혼하지 않기로 결심한 것을 느꼈다. | 그녀는 그와 결혼하지 않기로 결심한 그녀의 결심을 느꼈다. |
| 5 | 배달 채널을 삭제할 수 없습니다. | 배달 채널이 삭제되지 않았습니다. |
| 6 | 하지만 여러모로 나는 행복한 사람이 나를 화나게 하지 않는다. | 하지만 많은 면에서 나는 행복한 사람이 나를 화나게 하지 않는다. |
| 7 | 그 점원의 불의의 정중함은 나를 기분 좋게 만들었다. | 그 점원의 예상치 못한 공손함이 나를 기분 좋게 만들었다. |
| 8 | 내가 너의 조종사를 보고 있는 것을 신경쓰지 않기를 바란다. | 내가 너의 조종사를 감시하는 것을 신경쓰지 않았으면 좋겠어. |
| 9 | 저 소녀는 너와 이야기하고 싶어해. | 저 소녀는 너에게 말하고 싶어해. |

Beam Search를 적용하면 대체적으로 표현이 풍부해 짐을 알 수 있다. K=1인 경우는 Greedy Algorithm이라 생각해도 무방하다.

# Beam Search

## 선언부

```
import collections
from operator import itemgetter

import torch
import torch.nn as nn

import data_loader

LENGTH_PENALTY = .2
MIN_LENGTH = 5
```

# Beam Search

class SingleBeamSearchSpace():

```
def __init__(self,
             device,
             prev_status=None, # list of tuple, (status_name, status, batch_dim)
             beam_size=5,
             max_length=255,
             ):
    self.beam_size = beam_size
    self.max_length = max_length

    super(SingleBeamSearchSpace, self).__init__()

    # To put data to same device.
    self.device = device
    # Inferred word index for each time-step. For now, initialized with initial time-step.
    self.word_indice = [torch.LongTensor(beam_size).zero_().to(self.device) + data_loader.BOS]
    # Index origin of current beam.
    self.prev_beam_indice = [torch.LongTensor(beam_size).zero_().to(self.device) - 1]
    # Cumulative log-probability for each beam.
    self.cumulative_probs = [torch.FloatTensor([.0] + [-float('inf')] * (beam_size - 1)).to(self.device)]
    # 1 if it is done else 0
    self.masks = [torch.ByteTensor(beam_size).zero_().to(self.device)]

    # We don't need to remember every time-step of hidden states:
    #       prev_hidden, prev_cell, prev_h_t_tilde
    # What we need is remember just last one.
```

각 time step 값에서 추론 값들을 저장할 공간
시작은 BOS로 초기화

t-1 번째 값까지 0으로 초기화

t-1 이전까지 축적된 확률 값

# Beam Search

```
self.prev_status = collections.OrderedDict()
self.batch_dims = collections.OrderedDict()
for status_name, status, batch_dim in prev_status:
    if status is not None:
        self.prev_status[status_name] = torch.cat([status] * beam_size, dim=batch_dim)
    else:
        self.prev_status[status_name] = None
    self.batch_dims[status_name] = batch_dim

self.current_time_step = 0
self.done_cnt = 0
```

# Beam Search

Penalty 값을 계산하는 부분

```
def get_length_penalty(self,
                       length,
                       alpha=LENGTH_PENALTY,
                       min_length=MIN_LENGTH
                       ):
    # Calculate length-penalty,
    # because shorter sentence usually have bigger probability.
    # Thus, we need to put penalty for shorter one.
    p = ((min_length + length) / (min_length + 1)) ** alpha

    return p
```

# Beam Search

Search의 End 확인

```
def is_done(self):
    # Return 1, if we had EOS more than 'beam_size'-times.
    if self.done_cnt >= self.beam_size:
        return 1
    return 0
```

# Beam Search

```
def collect_result(self, y_hat, prev_status):
    # |y_hat| = (beam_size, 1, output_size)
    # prev_status is a dict of followings:
    # if model != transformer:
    #     |hidden| = |cell| = (n_layers, beam_size, hidden_size)
    #     |h_t_tilde| = (beam_size, 1, hidden_size)
    # else:
    #     |prev_state_i| = (beam_size, length, hidden_size)
    output_size = y_hat.size(-1)

    self.current_time_step += 1

    # Calculate cumulative log-probability.
    # First, fill -inf value to last cumulative probability, if the beam is already finished.
    # Second, expand -inf filled cumulative probability to fit to 'y_hat'.
    # (beam_size) --> (beam_size, 1, 1) --> (beam_size, 1, output_size)
    # Third, add expanded cumulative probability to 'y_hat'
    cumulative_prob = self.cumulative_probs[-1].masked_fill_(self.masks[-1], -float('inf'))
    cumulative_prob = y_hat + cumulative_prob.view(-1, 1, 1).expand(self.beam_size, 1, output_size)
    # Now, we have new top log-probability and its index.
    # We picked top index as many as 'beam_size'.
    # Be aware that we picked top-k from whole batch through 'view(-1)'.
    top_log_prob, top_indice = torch.topk(
        cumulative_prob.view(-1),
        self.beam_size,
        dim=-1,
    )
    # |top_log_prob| = (beam_size)
    # |top_indice| = (beam_size)
```

Model에 따라 다른 적용방식

Time_step +1 증가

# Beam Search

```
# Because we picked from whole batch, original word index should be calculated again.
self.word_indice += [top_indice.fmod(output_size)]
# Also, we can get an index of beam, which has top-k log-probability search result.
self.prev_beam_indice += [top_indice.div(output_size).long()]

# Add results to history boards.
self.cumulative_probs += [top_log_prob]
self.masks += [torch.eq(self.word_indice[-1],
               data_loader.EOS)
               ]  # Set finish mask if we got EOS.
# Calculate a number of finished beams.
self.done_cnt += self.masks[-1].float().sum()

# In beam search procedure, we only need to memorize latest status.
# For seq2seq, it would be lastest hidden and cell state, and h_t_tilde.
# The problem is hidden(or cell) state and h_t_tilde has different dimension order.
# In other words, a dimension for batch index is different.
# Therefore self.batch_dims stores the dimension index for batch index.
# For transformer, lastest status is each layer's decoder output from the biginning.
# Unlike seq2seq, transformer has to memorize every previous output for attention operation.
for k, v in prev_status:
    self.prev_status[k] = torch.index_select(
        v,
        dim=self.batch_dims[k],
        index=self.prev_beam_indice[-1]
    ).contiguous()
```

# Beam Search

```
def get_n_best(self, n=1, length_penalty=.2):
    sentences, probs, founds = [], [], []

    for t in range(len(self.word_indice)):  # for each time-step,
        for b in range(self.beam_size):  # for each beam,
            if self.masks[t][b] == 1:  # if we had EOS on this time-step and beam,
                # Take a record of penaltified log-proability.
                probs += [self.cumulative_probs[t][b] / self.get_length_penalty(t, alpha=length_penalty)]
                founds += [(t, b)]

    # Also, collect log-probability from last time-step, for the case of EOS is not shown.
    for b in range(self.beam_size):
        if self.cumulative_probs[-1][b] != -float('inf'):
            if not (len(self.cumulative_probs) - 1, b) in founds:
                probs += [self.cumulative_probs[-1][b]]
                founds += [(t, b)]

    # Sort and take n-best.
    sorted_founds_with_probs = sorted(zip(founds, probs),
                                      key=itemgetter(1),
                                      reverse=True
                                      )[:n]
    probs = []

    for (end_index, b), prob in sorted_founds_with_probs:
        sentence = []

        # Trace from the end.
        for t in range(end_index, 0, -1):
            sentence = [self.word_indice[t][b]] + sentence
            b = self.prev_beam_indice[t][b]

        sentences += [sentence]
        probs += [prob]

    return sentences, probs
```

Best predicted 출력