



인공지능 실습 Chapter 3

Search

포항공과대학교 컴퓨터공학과

3.1. Search 개요

3.2. Back-tracking Search

3.3. Uniform Cost Search

3.4. Text Reconstruction

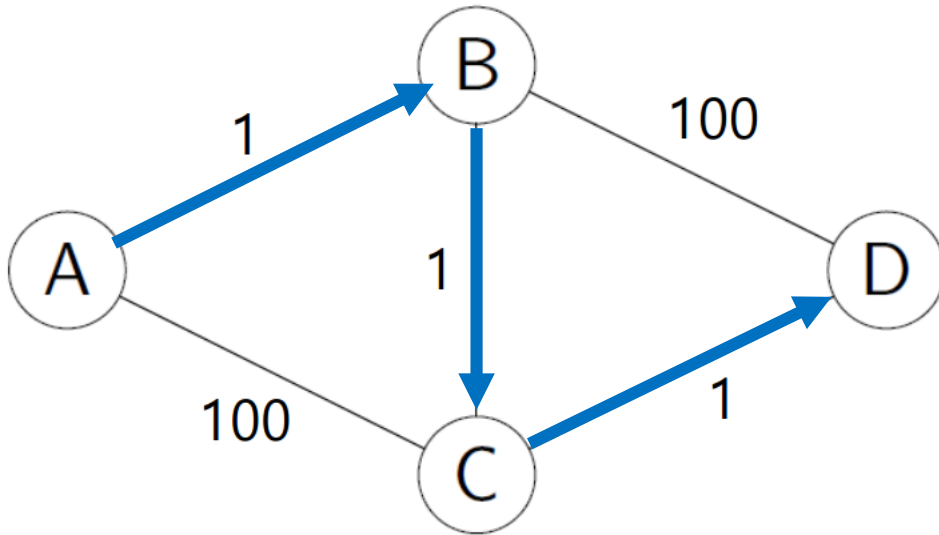
3.5. A* Search

3.1. Search 개요

Greedy Algorithm

4

- 각 단계마다 지역적인(local) 최적의 선택을 함으로써 전역적인(global) 최적의 해답을 찾고자 하는 알고리즘
- 좋은 예. 그래프에서의 최단 경로 탐색



A -> B -> C -> D : Cost = 3

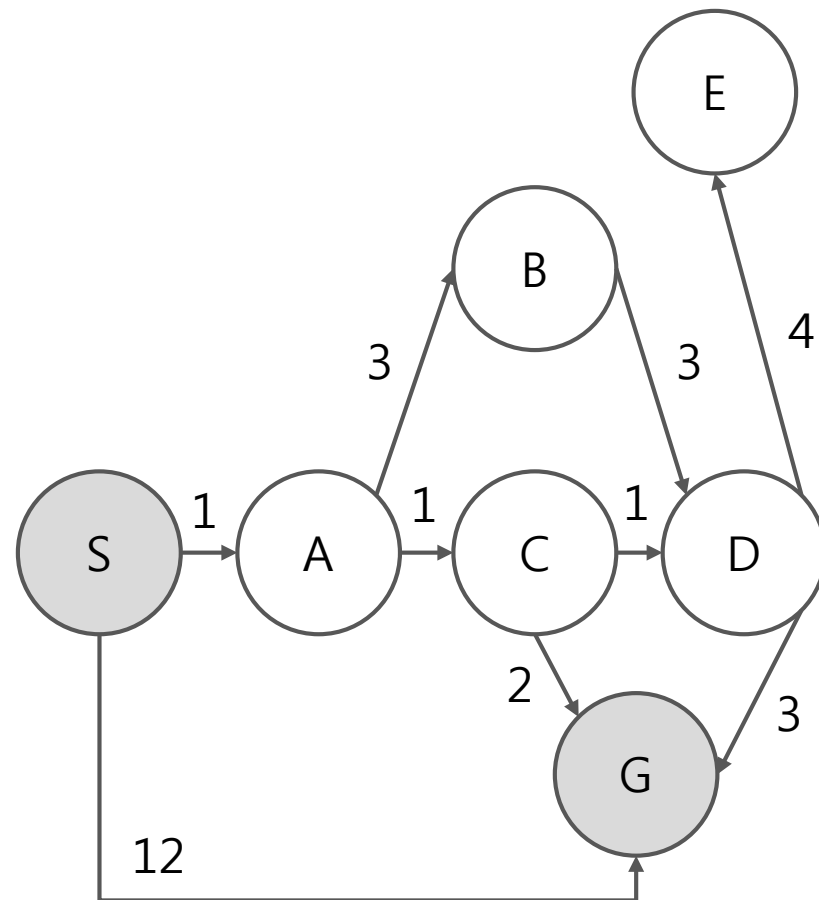
'A→B'라는 action과 'B'라는 상태를 구분하는 것이 바람직

즉, 최적의 솔루션은 다음 세 action의 연속
[A->B, B->C, C->D]

풀고자 하는 문제 1. Graph

5

- S에서 G까지의 최단경로 탐색



- Q. 최단경로와 그 비용은?

- Q. Greedy policy로 선택된 경로와 그 비용은?

풀고자 하는 문제 2. Transportation

6

- 1에서 7까지 최단경로 탐색

- Street with blocks numbered 1 to n .
- **Walking** from s to $s+1$ takes 1 minute.
- Taking a magic **tram** from s to $2s$ takes 2 minutes
- How to travel from 1 to n in the least time?

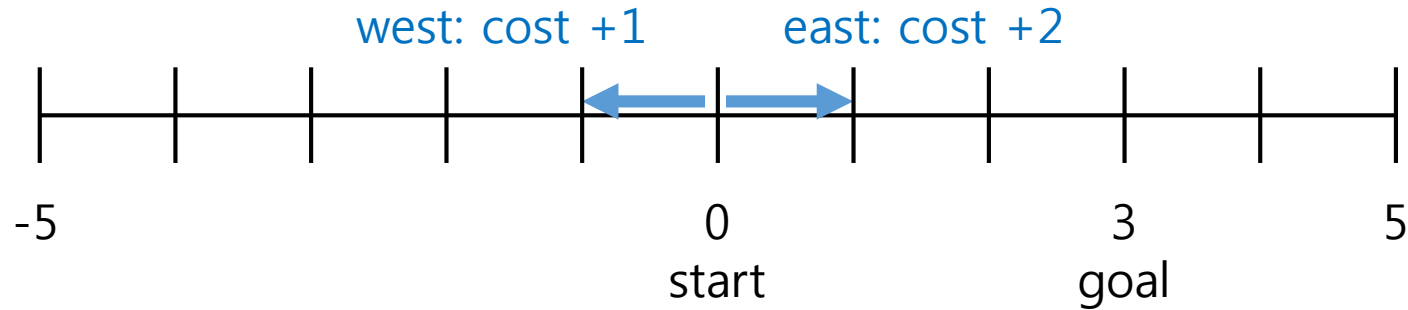


- Q. 최단경로와 그 비용은?
- Q. Greedy policy로 선택된 경로와 그 비용은?

풀고자 하는 문제 3. Number Line

7

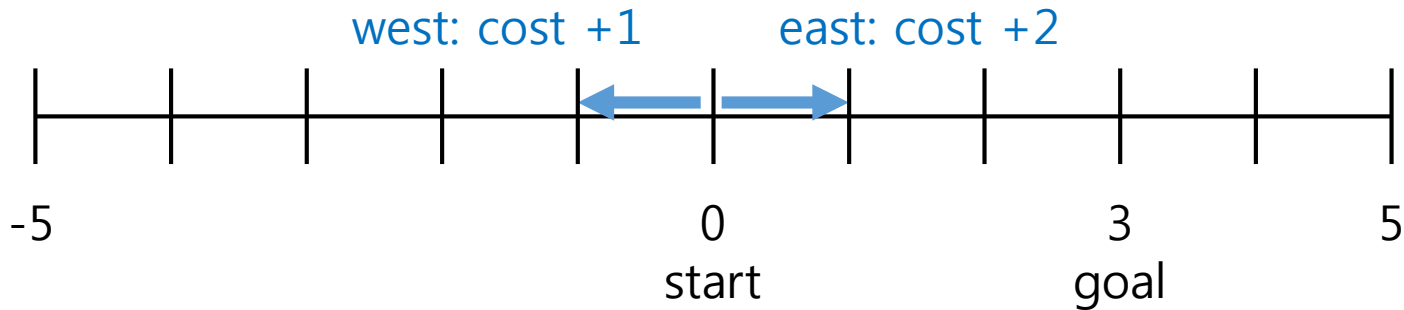
- 0에서 3까지 최단경로 탐색



- Q. 최단경로와 그 비용은?
- Q. Greedy policy로 선택된 경로와 그 비용은?

- 상태 표현 (state representation)
 - "A state is a summary of all the past actions sufficient to choose future actions optimally"
- Search problem 구성 요소
 - s_{start} : 시작 상태
 - $\text{Actions}(s)$: s 에서 취할 수 있는 모든 action
 - $\text{Cost}(s, a)$: s 에서 a 를 취하는 cost(비용)
 - $\text{Succ}(s, a)$: s 에서 a 를 취했을 때 다음 상태
 - $\text{IsEnd}(s)$: s 의 종료 조건 만족 여부

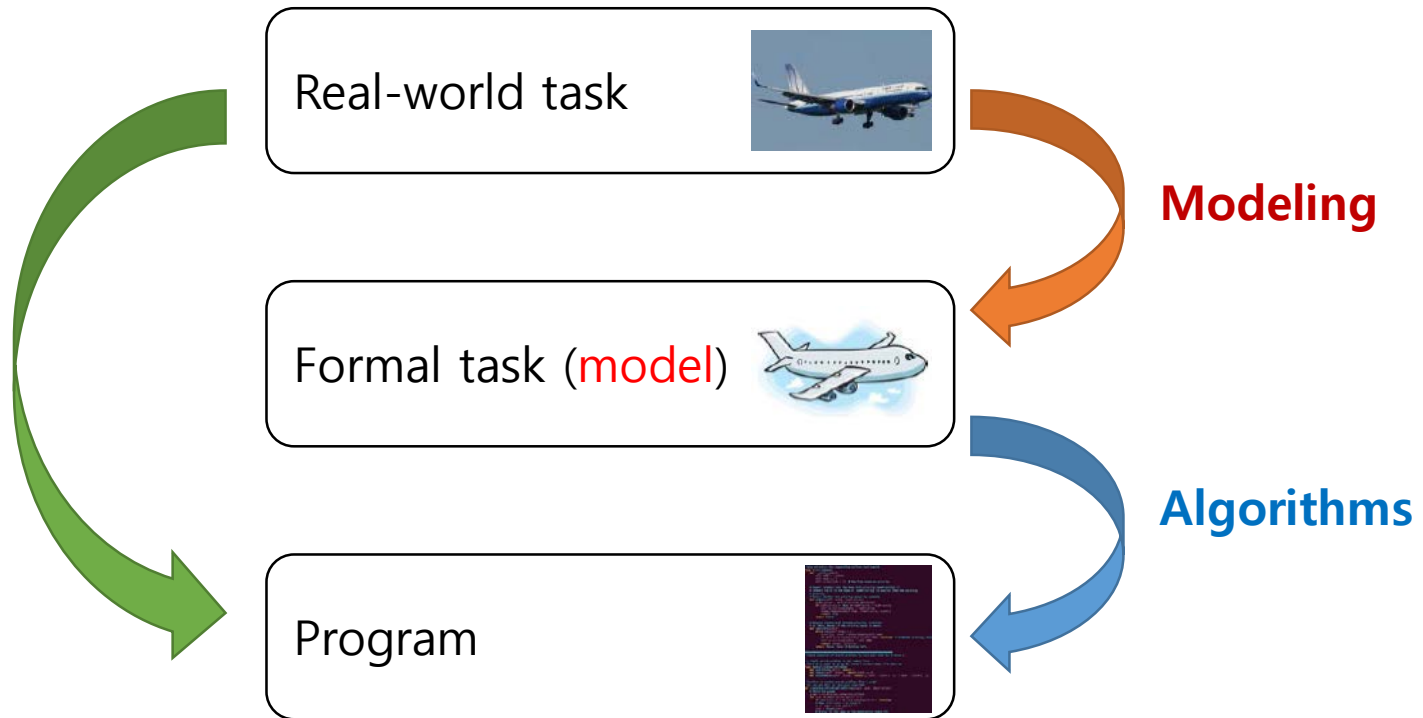
- 0에서 3까지 최단경로 탐색



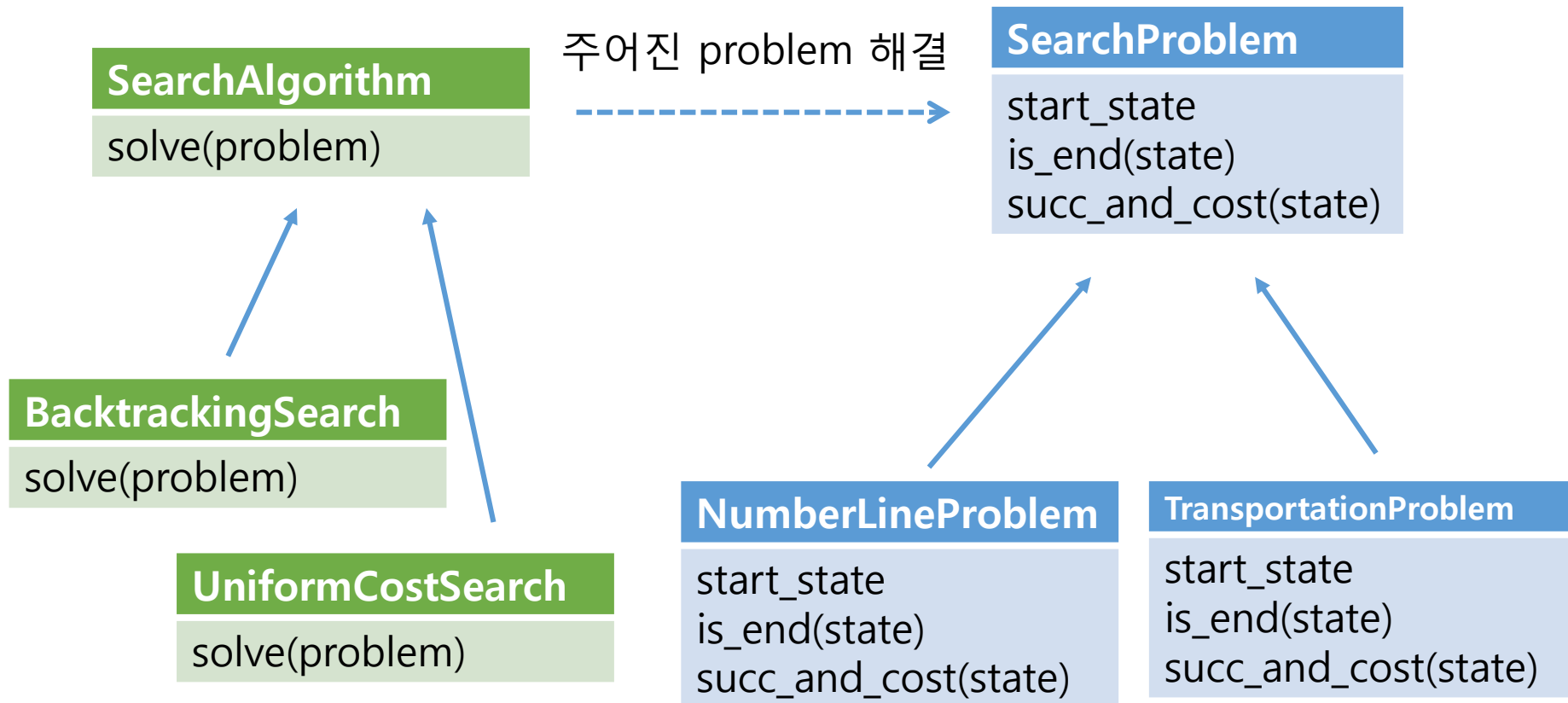
- Search problem 구성 요소

- $s_{\text{start}} = 0$
- $\text{Actions}(s) = \{\text{west}, \text{east}\}$
- $\text{Cost}(s, a) = \begin{cases} 1, & \text{if } a = \text{west} \\ 2, & \text{if } a = \text{east} \end{cases}$
- $\text{Succ}(s, a) = \begin{cases} s - 1, & \text{if } a = \text{west and } s > -4 \\ s + 1, & \text{if } a = \text{east and } s < 4 \end{cases}$
- $\text{IsEnd}(s) = \begin{cases} \text{True}, & \text{if } s = 3 \\ \text{False}, & \text{otherwise} \end{cases}$

How to Tackle these challenging AI tasks? 10



- Separate **what** to compute (**modeling**) from **how** to compute it
- Advantage: division of labor



util.SearchProblem

- 탐색 문제에 대한 추상 클래스
- 새로운 클래스로 상속해 다음 함수 구현
 - start_state: 시작 상태
 - is_end(state): 종료 조건 만족 여부 (True or False) 반환
 - succ_and_cost(state): 현재 state에서 가능한 (action, new_state, cost)의 리스트 반환
 - cost는 해당 action을 취했을 때의 cost (누적값 X)

util.SearchAlgorithm

- 탐색 알고리즘에 대한 추상 클래스
- 새로운 클래스로 상속해 solve 함수 구현

- Search 문제
 - Graph [graph_problem.py](#)
 - Transportation [transportation_problem.py](#)
 - Number line [numberline_problem.py](#) (제공)
- Search 알고리즘
 - Back-tracking search [backtracking_search.py](#)
 - Uniform cost search [uniform_cost_search.py](#) (제공)



테스트

```
class NumberLineSearchProblem(util.SearchProblem):
    def __init__(self, size, end_state):
        ...
    def start_state(self):
        ...
    def is_end(self, state):
        ...
    def succ_and_cost(self, state):
        ...

problem = NumberLineSearchProblem(10, 5)

import uniform_cost_search
ucs = uniform_cost_search.UniformCostSearch(verbose=3)
print ucs.solve(problem)
```

Uniform cost search 결과

- 최단경로: east, east, east (비용 6)

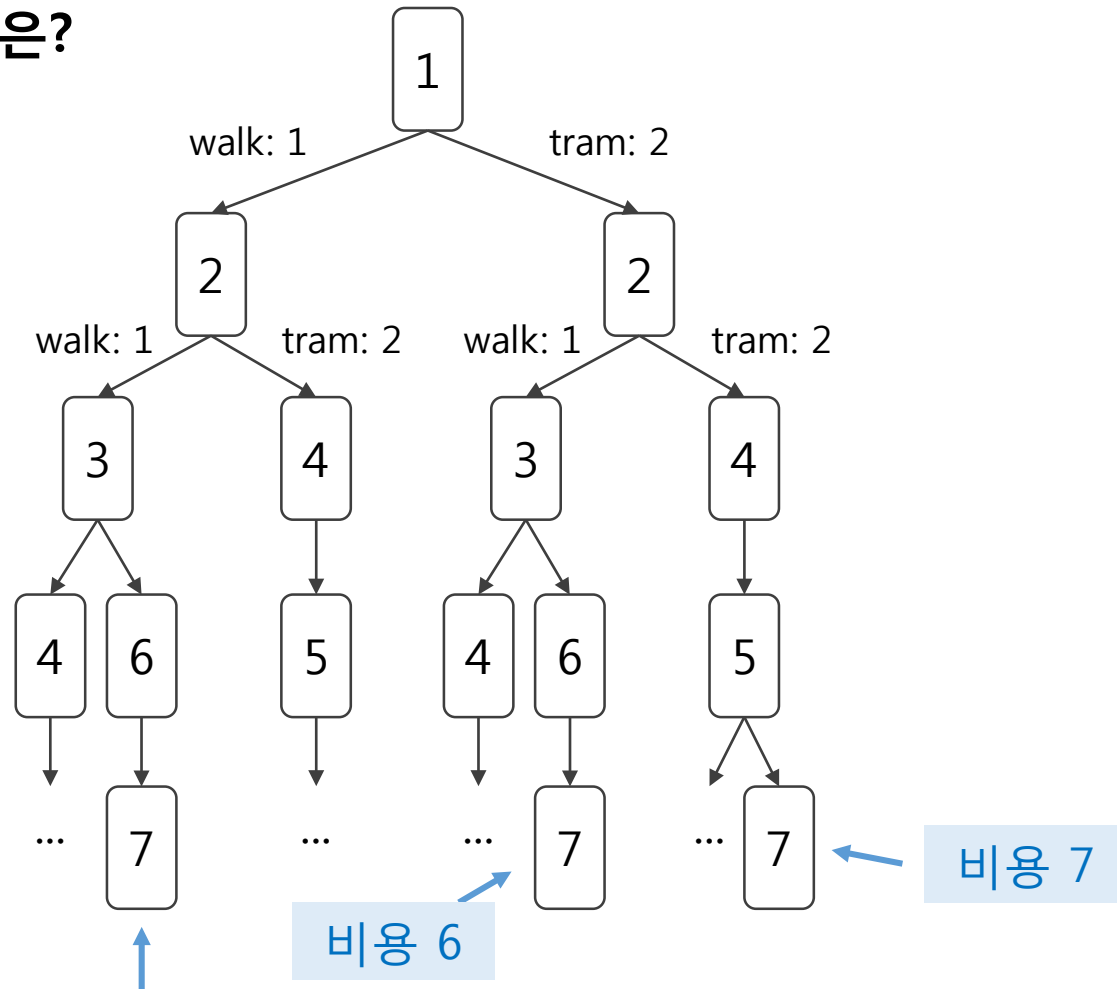
※ python numberline_problem.py 로 실행

- **Graph 및 Transportation 문제에 대한 Search Problem class를 구현하고 UniformCostSearch 적용**
 - Graph `graph_problem.py`
 - Transportation `transportation_problem.py`
- **Uniform cost search 결과와 실제 최단경로 비교**
 - Graph 최단경로: S->A, A->C, C->G (비용 6)
 - Transportation 최단경로: Walk, Walk, Walk, Tram, Walk (비용 5)

※ verbose

3.2. Back-tracking Search

- Q. 이 방법의 단점은?



비용 5 솔루션 (walk, walk, tram, walk)으로 목표 도달 가능

BacktrackingSearch 알고리즘

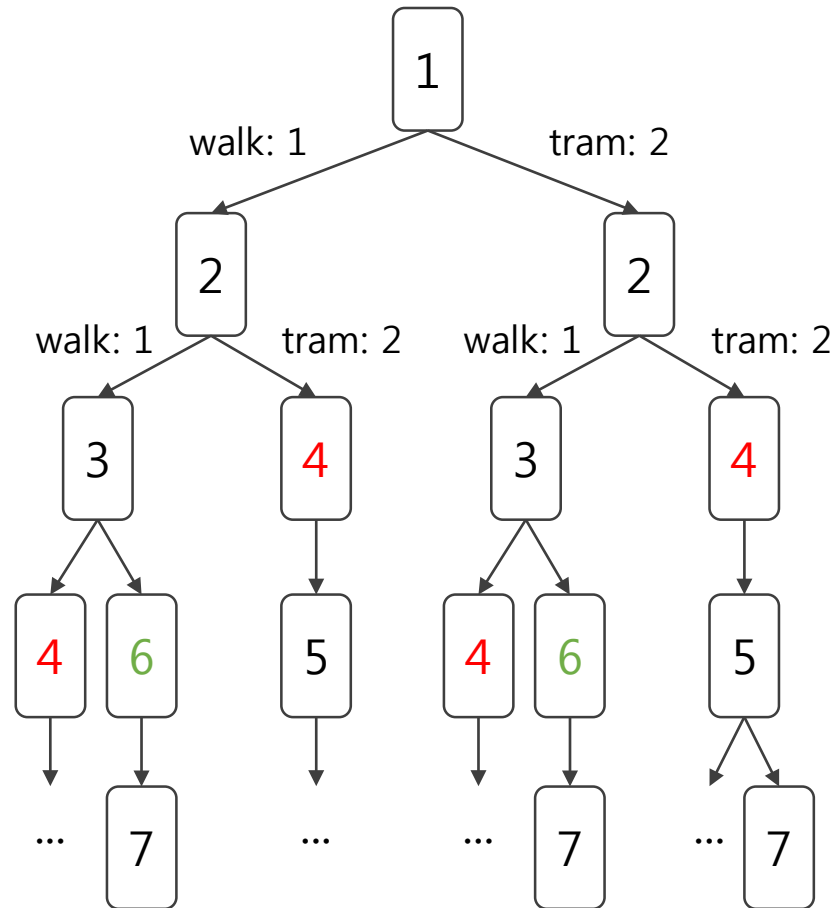
```
problem ← 문제 정의 (예. TransportationProblem)
best_path, best_path_cost 초기화

def backtrackingSearch(state, path, path_cost):
    If problem.is_end(state):
        best_path, best_path_cost 업데이트
    For each (action, next_state, action_cost) in problem.succ_and_cost(state):
        extended_path ← path를 확장
        extended_path_cost ← extended_path의 비용
        backtrackingSearch(next_state, extended_path, extended_path_cost)

backtrackingSearch(problem.start_state, (), 0)
best_path, best_path_cost 리턴
```

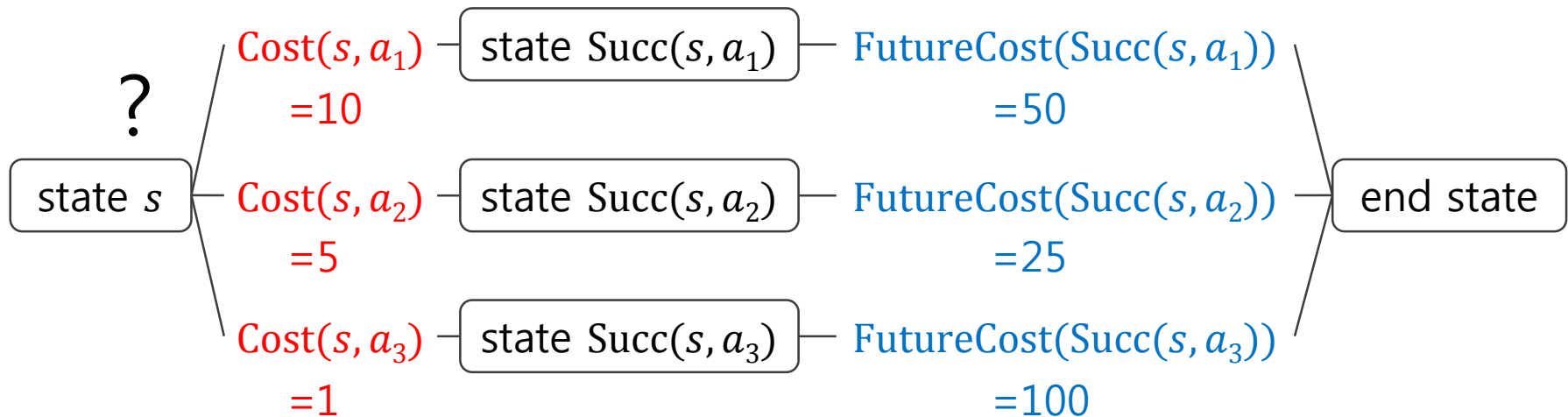
실습을 통해 이론을 더 잘 이해 할 수 있는 기회!

1. [backtracking_search.py](#) 구현
2. Transportation 및 Graph 문제에 적용
3. 실제 최단경로와 일치여부 확인



- 동일한 연산을 반복

- state s 에서 end state로 가는 비용을 최소화하는 action 결정

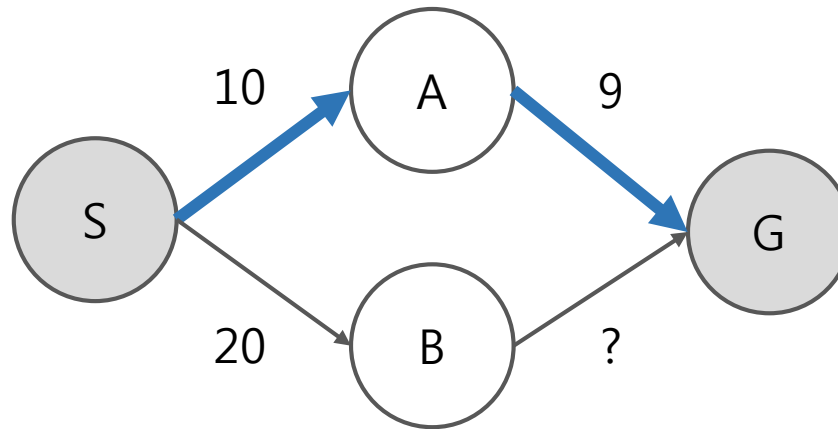


- FutureCost를 재귀적으로 정의

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{\{a \in \text{Actions}(s)\}} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

3.3. Uniform Cost Search

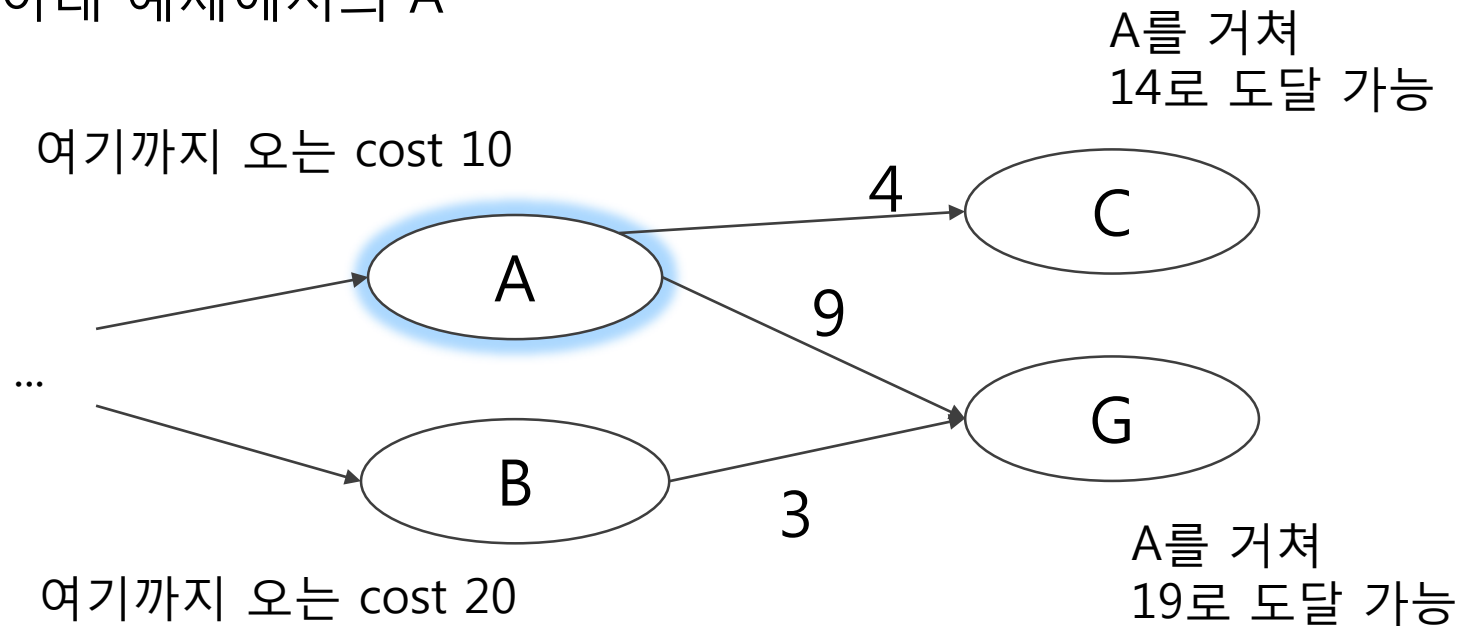
- 1. Acyclic 그래프에만 적용 가능
 - 예. Line number search에 back-tracking 적용 시 비정상
- 2. 현재 $S \rightarrow A \rightarrow G$ 라는 cost 7 경로를 찾은 상황



Q. 현재 상황에서 $S \rightarrow B$ action에 대해서도 고려해 봐야 할까?

- Cost가 낮은 것부터 고려해보자

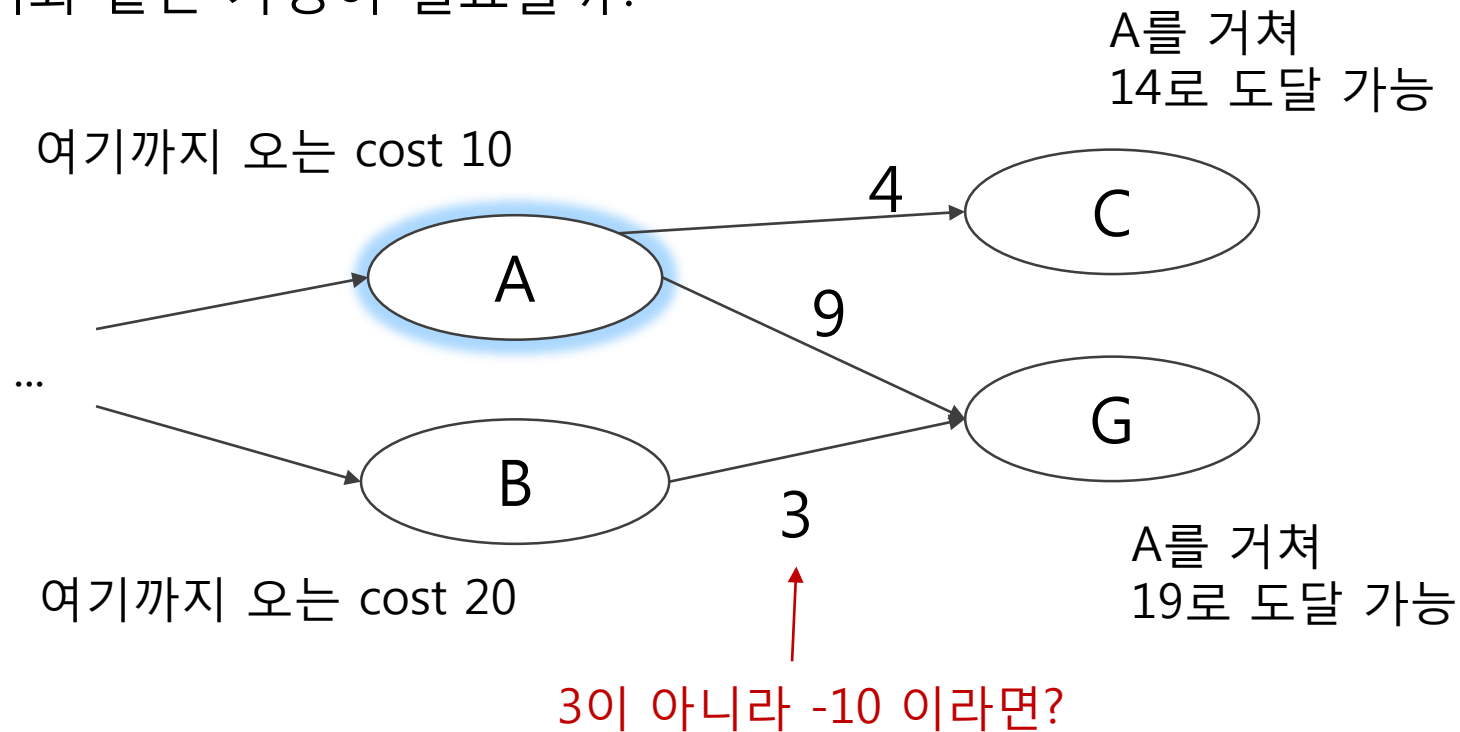
- 아래 예제에서의 A



B의 cost(20)가 G의 cost(19)보다 높으므로 고려 할 필요 없음!

- 이를 통해 탐색 시간을 줄일 수 있음

- 모든 action의 비용이 **음수**가 아님: $\text{cost}(s, a) \geq 0$
 - 즉 현재까지 찾은 solution에서 action을 취했을 때 solution의 비용이 감소 할 수 없음
- 왜 위와 같은 가정이 필요할까?



시작 단계

Frontier: [(S, 0)]

Explored: []

Backpointer: {}

※ *Frontier*: 현재 탐색 중인 상태들
(cost를 기준으로 오름차순 정렬)

※ *Explored*: 이미 최적의 경로를 찾은 상태들

1단계

Frontier: [

(A, 10),

(B, 20)]

Explored: [S]

Backpointer: {

A:S,

B:S}

2단계

Frontier: [

(C, 14),

(G, 19),

(B, 20)]

Explored: [S, A]

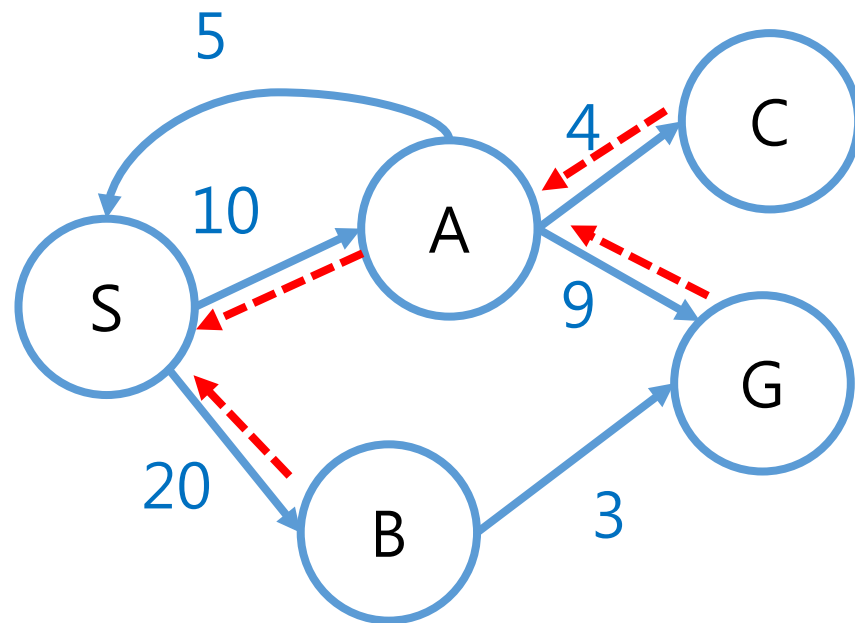
Backpointer: {

A:S,

B:S,

C:A,

G:A}



3단계:

Frontier: [

(G, 19),

(B, 20)]

Explored: [S, A, C]

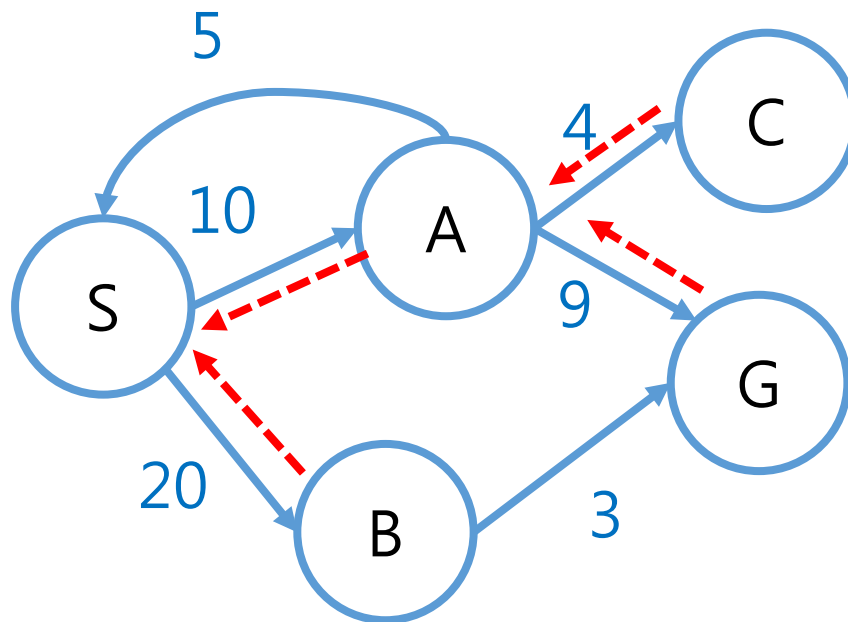
Backpointer:{

A:S,

B:S,

C:A,

G:A}



4단계:

- 목표 G에 도착 (비용 19)

- 최적경로 선택

1. Backpointer[G] = A

2. Backpointer[A] = S

최적경로 = ([S->A->G], 19)



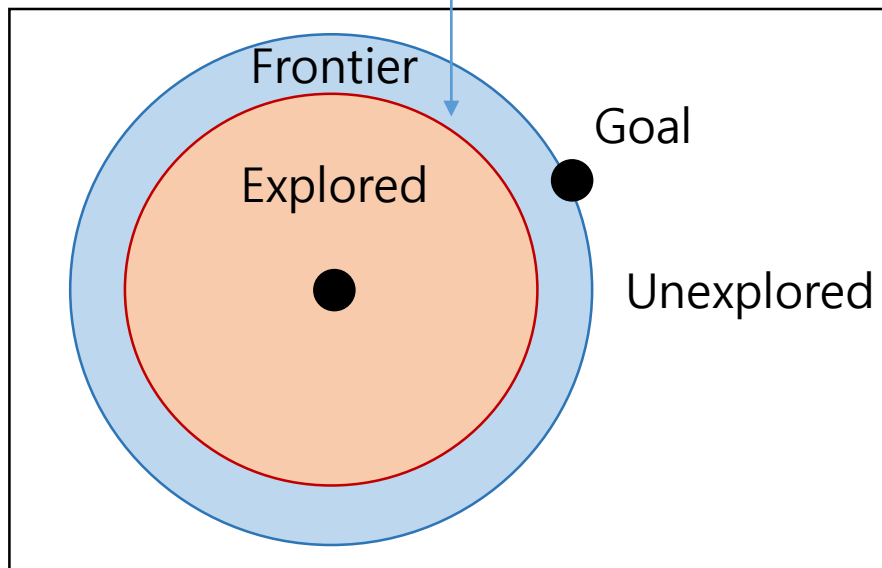


<https://www.youtube.com/watch?v=A138vmAAKrA>

Unexplored, Frontier, Explored

29

이 중 최소 cost 상태를 먼저 explore



왜 이름이 Uniform Cost Search 인가?

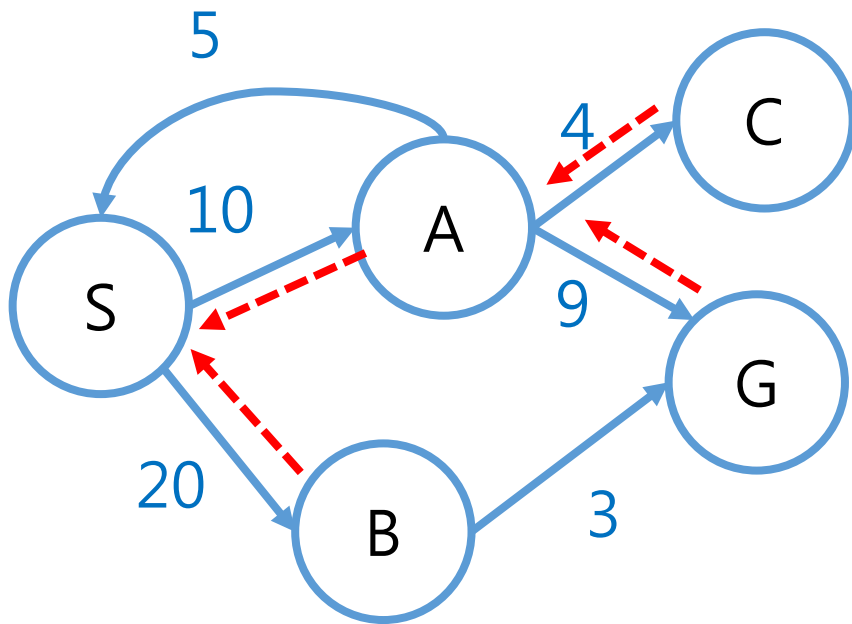
- 초기화: 시작 상태를 *frontier*에 추가
 - *Frontier*에는 탐색된 솔루션들이 비용(cost)의 오름차순으로 정렬되어 있음
- 반복 (*frontier*가 텅 빌 때까지)
 - *Frontier*에서 최소 비용 p 를 갖는 솔루션 s 를 제거
 - s 를 *explored*에 추가
 - s 가 목표에 도달했으면:
 - 솔루션 반환하고 알고리즘 종료
 - s 에서 취할 수 있는 모든 가능한 action a 에 대해:
 - s 에서 a 를 취했을 때의 새로운 상태 s' 계산
 - s' 가 *explored*에 없으면:
 - $Frontier.update(solution=s', cost=p + cost(s, a))$
 - s' 에 대한 최적경로가 바뀌면:
 - $Backpointer[s'] = s$

- Greedy Search : 최적(optimal) 솔루션 보장 X
 - 무조건 방금 전에 본 상태에서 탐색을 이어감
- Back-tracking Search: 최적 솔루션 보장 O
 - 모든 솔루션을 탐색
- Dynamic Programming: 최적 솔루션 보장 O
 - 모든 솔루션을 탐색하지만 동일한 연산을 반복하는 문제 해결
- Uniform Cost Search: 최적 솔루션 보장 O
 - 어떤 상태를 먼저 볼 지 잘 결정 (상태의 cost를 기준으로)
 - 결과적으로 할 필요가 없는 탐색은 하지 않음
- A* Search : ?

Dijkstra Algorithm v.s. UCS

32

- 시작 노드부터 모든 노드에 대한 최단거리를 계산



Frontier: []

Explored: [S, A, B, C]

Backpointer: {

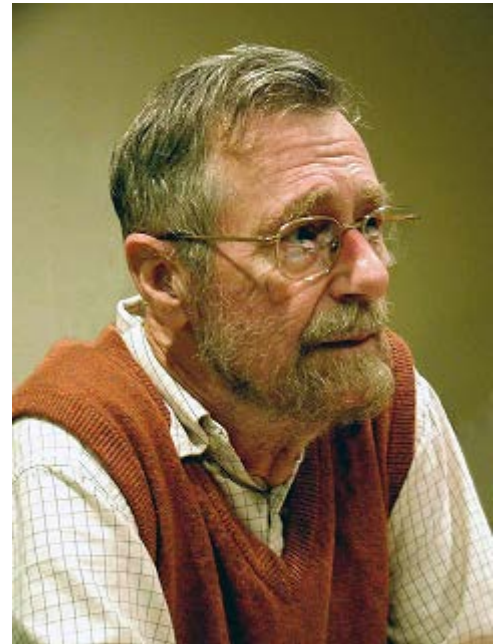
A:S,

B:S,

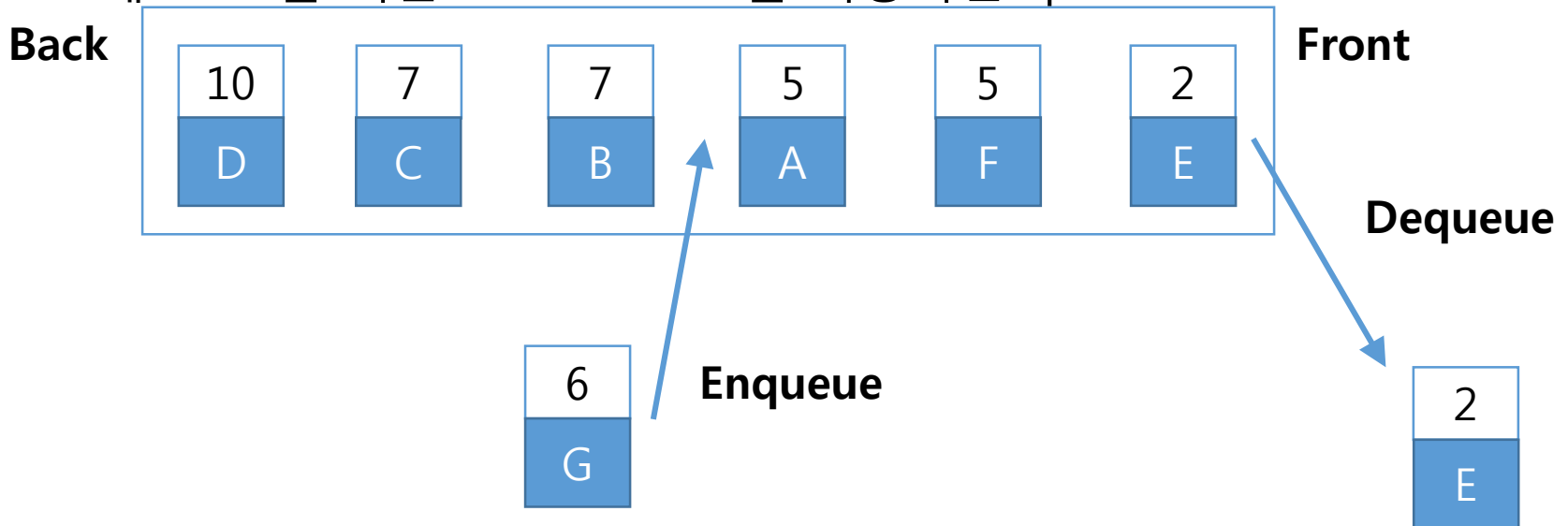
C:A,

G:A}

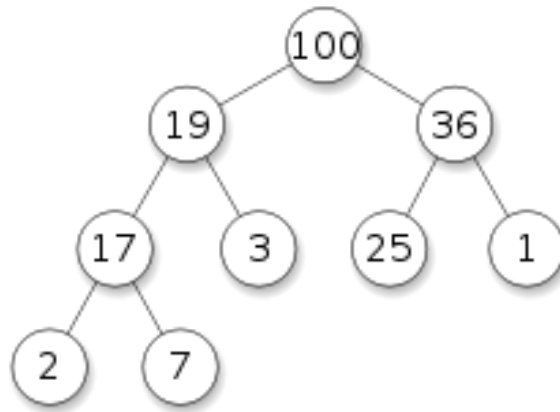
Edsger W. Dijkstra
Turing Award Winner



- 우선순위 큐
 - 내부적인 기준에 따라 오름차순 혹은 내림차순으로 정렬된 아이템들의 큐
 - Enqueue: 우선순위에 따라 큐에 데이터 삽입
 - Dequeue: 큐 가장 앞의 데이터 제거 및 조회
 - 예. Cost를 기준으로 solution을 저장하는 queue

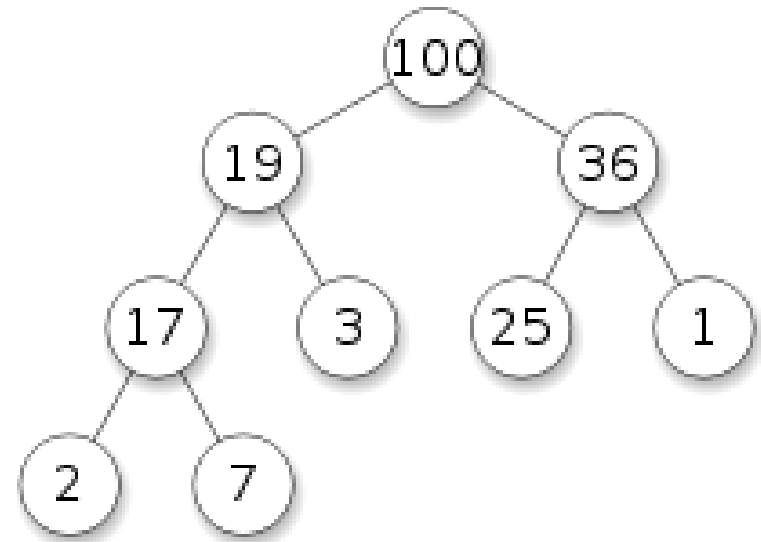


- Priority queue는 일반적으로 구현에 heap을 사용
- Heap
 - 최대값 및 최소값 검색 연산을 빠르게 하기 위한 자료구조
 - **heap property**를 만족하는 특수한 tree로 구현
 - “A가 B의 부모노드이면, A의 키값과 B의 키값 사이에는 대소관계가 성립한다”

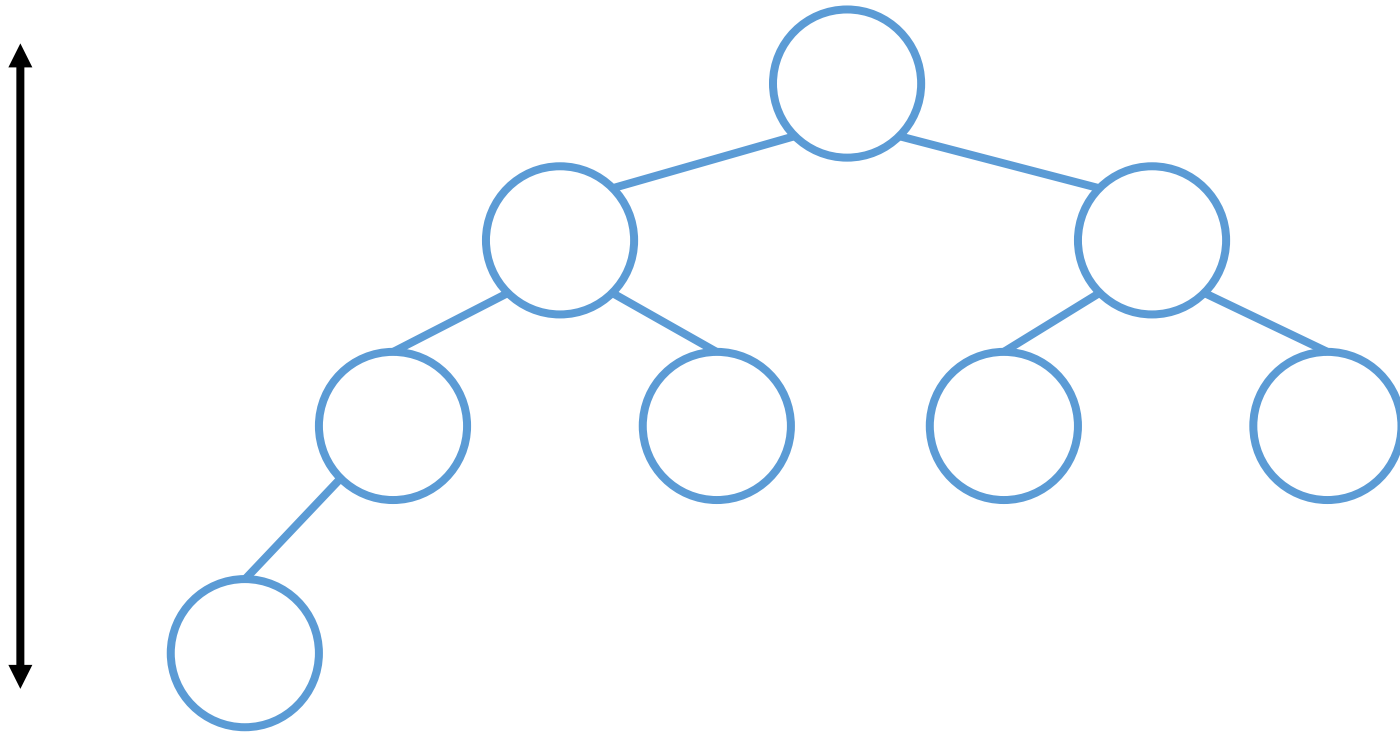


Max heap: 최대값이 먼저 나오는 heap (v.s. Min heap)

- 삽입 연산 (예. "23")
 - 새로운 노드를 마지막 노드 다음에 위치시킴
 - 해당 노드를 부모 노드와 비교해 힙의 성질(대소관계)이 만족되지 않으면 자리를 바꿈 (반복)
- 삭제 연산 (최상단)
 - 루트 노드를 제거하고 마지막 노드를 루트 노드에 위치시킨다
 - 해당 노드를 자식 노드와 비교해 힙의 성질(대소관계)이 만족되지 않으면 자리를 바꾼다 (반복)



- n 개의 아이템이 있을 때 $\log_2 n$
 - 예. $\log_2 8 = \log_2 2^3 = 3$
- Heap 삽입/삭제 연산의 시간은 높이에 비례



- 특징
 - Search에서는 비용이 낮은 것을 우선시하기 때문에 min heap 사용
 - 같은 item 중복 저장을 허용하지 않음
 - Heap 자체는 (item, priority) 튜플의 리스트로 저장하며, heapq package 사용해 이 리스트에 대해 연산
- 함수
 - `update(item, new_priority)`: 우선순위 큐에 item 삽입
 - 기존에 동일한 item이 있는 경우, priority 값이 높은 item을 제거
 - 변경사항이 있는 경우 True 반환
 - `remove_min()`
 - `is_empty()`

test_pq.py

- 1. 우선순위 큐 생성
- 2. 우선순위 큐에 다음 아이템 순차 삽입
 - item='A', priority=10
 - item='B', priority=20
 - item='C', priority=30
 - item='A', priority=5
- 3. 우선순위 큐의 첫 번째 아이템 출력
- 4. 우선순위 큐의 아이템 remove

- Uniform cost search

- Graph 문제에 대한 state 및 그 비용 저장

start state에서 해당 state로 가는 비용

```
frontiers = util.PriorityQueue()

# 주변 state를 frontier에 추가
...
frontiers.update(C, 8)
...
frontiers.update(A, 5)
...
frontiers.update(C, 15)
...

#최소비용 state 먼저 탐색
...
state, priority = frontiers.remove_min()
...
state, priority = frontiers.remove_min()
```

- [uniform_cost_search.py](#) 참조
 - SearchProblem 객체를 이용해 문제에 접근
 - PriorityQueue를 이용해 frontier 관리

3.4. Text Reconstruction

- 언어 모델 (language model)
 - 자연어 문장의 fluency (or likelihood)를 측정
 - 입력: 문장, 출력: 확률

$$\text{예. } P(I, \text{love}, \text{you}) > P(I, \text{you}, \text{love})$$

- 문장의 확률을 각 단어의 확률의 곱으로 표현

$$\text{조건부 확률: } P(A, B) = P(A)P(B|A)$$

$$\begin{aligned}\text{예. } P(I, \text{love}, \text{you}) &= P(I, \text{love})P(\text{you}|I, \text{love}) \\ &= P(I)P(\text{love}|I)P(\text{you}|I, \text{love})\end{aligned}$$

- LM 응용: Statistical machine translation (SMT)

- 예. 불어 문장 f 를 영어 문장 e 로 번역

$$\tilde{e} = \arg \max_{e \in e^*} p(e|f) = \arg \max_{e \in e^*} \underbrace{p(f|e)}_{\text{Translation Model}} \underbrace{p(e)}_{\text{LM}}$$

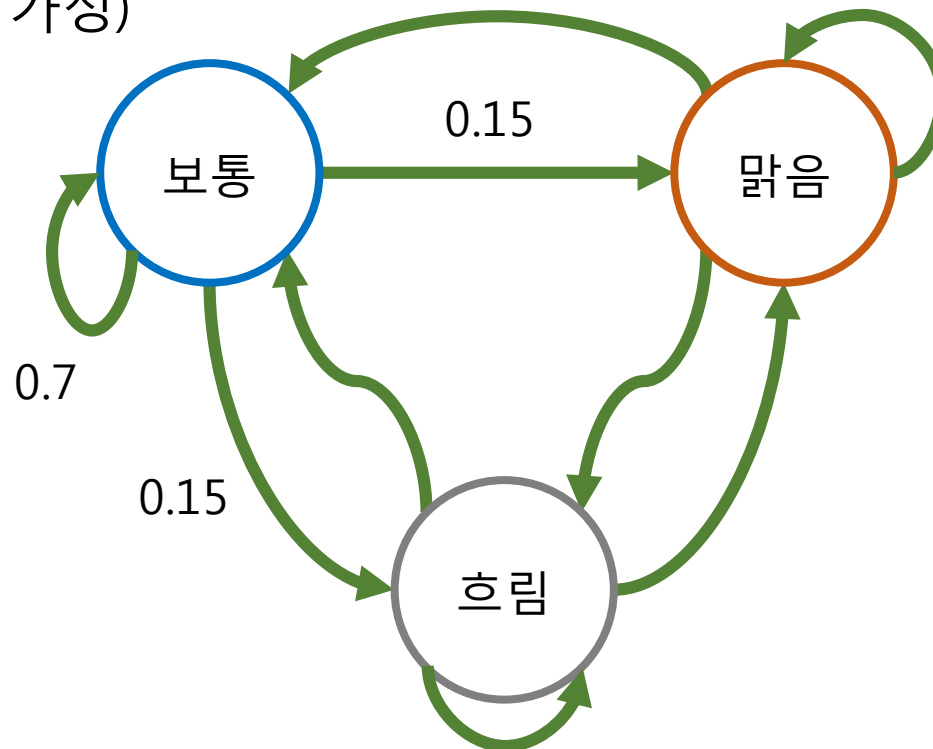
- $P(\text{high winds tonight}) > P(\text{large winds tonight})$

- LM 응용: Spell correction
 - 입력: "The office is about fifteen **mineuts** from my house"
 - $P(\text{about fifteen } \mathbf{minutes} \text{ from})$
 - > $P(\text{about fifteen } \mathbf{mineuts} \text{ from})$
- LM 응용: Speech recognition
 - 입력: speech signal
 - $P(\text{I saw a van}) \gg P(\text{eyes awe of an})$

- LM 응용: Word segmentation (오늘 실습 내용)
 - 입력: "iloveyou"
 - $P(i \text{ love you}) > P(il \text{ ove you})$
- LM 응용: Vowel insertion (오늘 실습 내용)
 - 입력: "cll m bck"
 - $P(call \text{ me back}) > P(cell \text{ my back})$

- **Markov property**

- 과거가 아닌 현재 상태만이 미래 상태에 대한 확률에 영향을 주는 성질
- 예. 내일 날씨가 현재 날씨에만 영향을 받는다 (비현실적인 가정)



- n-gram 언어 모델

- 모든 케이스를 기억하려면 큰 용량이 필요하고 sparse한 문제가 있음
- 앞의 n-1개 단어를 보고 단어를 예측
- 예. 1-gram (unigram) 모델

$$P(I, love, you) = P(I)P(love)P(you)$$

- 예. 2-gram (bigram) 모델

$$P(I, love, you) = P(I | - BEGIN -)P(love|I)P(you|love)$$

- **LM 학습** (based on simple counting)
 - 2-gram 사용

$$P(w|w') = \frac{\text{count}(w', w)}{\text{count}(w')}$$

Example:

-BEGIN- I am Sam
-BEGIN- Sam I am
-BEGIN- I do not like green

$$P(I|-\text{BEGIN}-) = \frac{2}{3} = 0.666\dots$$

$$\text{Q1. } P(\text{Sam}|-\text{BEGIN}-) = ?$$

$$\text{Q2. } P(\text{do}|I) = ?$$

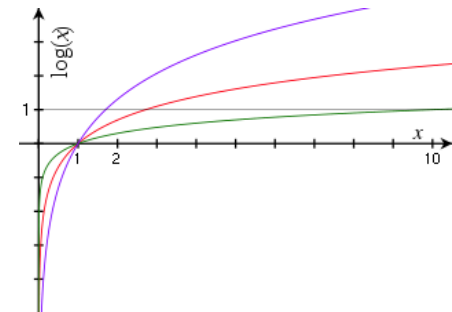
$$\text{Q3. } P(\text{green}|\text{like}) = ?$$

- **Underflow**

- 언어 모델에서 다루는 확률 p ($0 \leq p \leq 1$)는 굉장히 작음
- 0에 가까운 확률을 여러 번 곱하면 underflow 발생
- 즉 긴 문장의 확률은 무조건 0이 될 수 있음

- **Log-likelihood**

- 확률의 로그(log)값을 취함
- 문장의 확률은 각 로그확률의 합으로 계산
 - 예. $\log P(I) + \log P(\text{love}) + \log P(\text{you})$
- 본래 목적은 언어 모델 값의 비교이므로 로그값을 그대로 사용해도 무방
- UCS는 non-negative cost를 정의해야 함



- `lm_unigram.py` 및 `lm_bigram.py` 구현
- 문장의 리스트가 corpus로 주어짐
- 각 코드가 unigram(혹은 bigram) 확률의 $-\log$ 값을 출력하도록 구현
- Q. n-gram 언어 모델을 바탕으로 감정 분석 할 수 있는 방법?

- 공백(띄어쓰기)이 없는(space-free) 문자열에 공백을 삽입해 **unigram 언어 모델(LM)** 상에서 자연스러운 문장 생성
 - 테스트 케이스. "thisisnotmybeautifulhouse"
=> "this is not my beautiful house"
- **1) Search 형태의 문제 정의**
 - state, start state, end state, actions, cost
- **2) [word_segmentation.py](#)의 SegmentationProblem 클래스 멤버함수 구현**
- **3) UCS 및 Back-tracking search 적용**

※ 후자의 경우 제 시간에 안 끝나므로 "thisishouse" 에 대해 테스트

- e.g. "this is not my beautiful house"
 - `unigramCost('this') + unigramCost('is') + unigramCost('not') + unigramCost('my') + unigramCost('beautiful') + unigramCost('house')`
- wordsegUtil의 makeLanguageModels 사용

```
unigramCost, bigramCost = wordsegUtil.makeLanguageModels('leo-will.txt')  
  
unigramCost('love')  
  
bigramCost('beautiful', 'house')
```

- 모음이 없는(vowel-free) 문자열에 모음을 삽입해 **bigram** 언어 모델(LM) 상에서 자연스러운 문장 생성
 - Vowel (모음): A, E, I, O, and U; never Y
 - 테스트 케이스. "thts m n th crnr"
=> "thats me in the corner"
- 1) Search 형태의 문제 정의
 - state, start state, end state, actions, cost
- 2) [vowel_insertion.py](#)의 VowelInsertionProblem 클래스 멤버함수 구현
- 3) UCS 및 Back-tracking search 적용

- possibleFills 함수 사용
 - vowel이 없는 단어를 입력으로 받아 모든 가능한 vowel insertion 후보 반환
 - 예. possibleFills('fg') => set(['fugue', 'fog'])
 - 만약 후보가 없다면 vowel이 없는 단어 자체가 후보가 되도록 구현
- 문장의 시작은 SENTENCE_BEGIN으로 나타냄
 - SENTENCE_BEGIN = "-BEGIN-"

- 띄어쓰기가 없고(space-free) 모음이 없는(vowel-free) 문자열에 띄어쓰기와 모음을 삽입해 1-gram 및 2-gram 언어 모델(LM)의 **smooth cost** 상에서 자연스러운 문장 생성
- 테스트 케이스. "mgnllthppl"
=> "imagine all the people"
- 1) Search 형태의 문제 정의
 - state, start state, end state, actions, cost
- 2) [joint_task.py](#) 의 JointSegmentationInsertionProblem 클래스 멤버 함수 구현
- 3) UCS 및 Back-tracking search 적용

- 문제 Vowel Insertion과 달리 단어엔 무조건 모음이 포함되어 있어야 함
- 생성되는 단어는 반드시 하나 이상의 자음을 포함해야 하며, 모음만을 포함해서는 안 됨
 - 예. "a" or "I"는 X

3.5. A* Search

UCS



A* Search



- 두 방법론 모두 최적(optimal) 솔루션 보장 (cost가 동일)
- 그러나 A* Search가 훨씬 더 빠르게 답을 찾을 수 있다
 - HOW? 휴리스틱(heuristic)

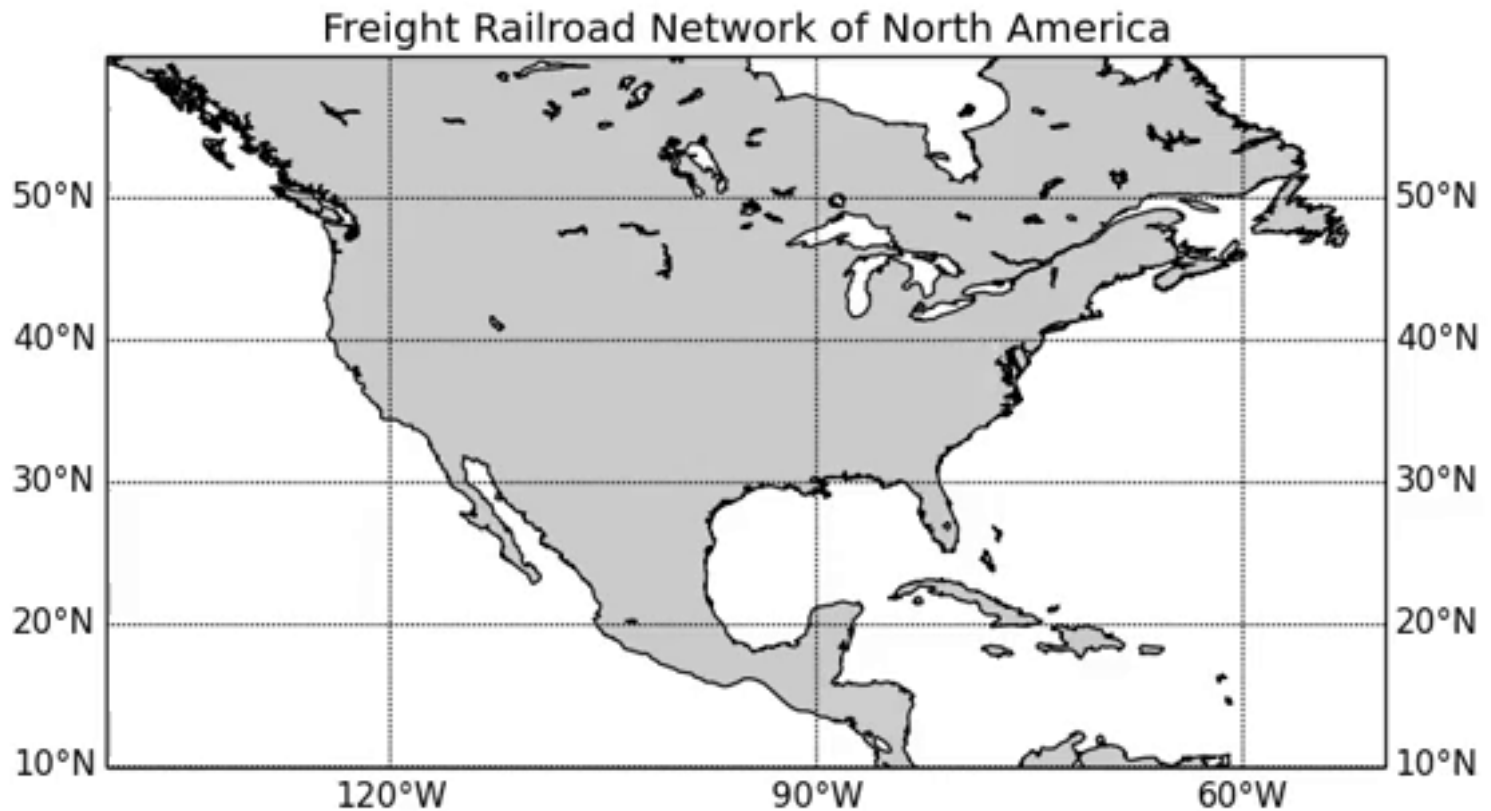
A* Search 적용 예

59



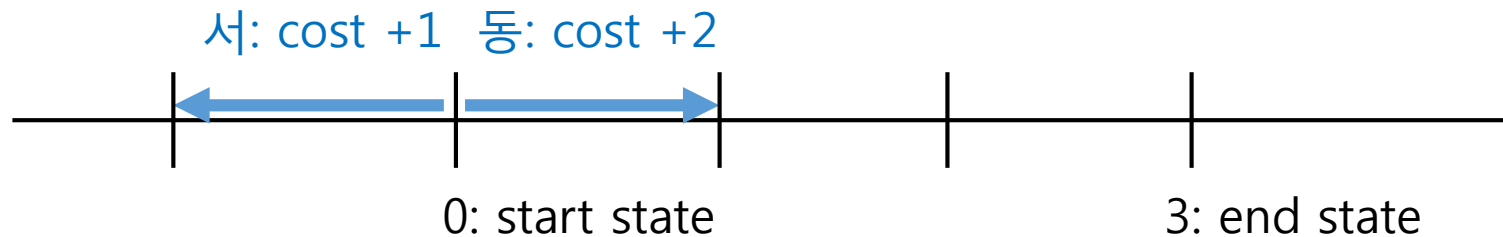
A*는 UCS 보다 더 빠르게 최적의 솔루션을 찾음

- 워싱턴 D. C.에서 LA까지 최단 철도 경로 찾기







- 목표까지 도달하는데 필요한 비용에 대한 추정(estimation)값
 - 즉, FutureCost(s)에 대한 추정값
- <https://ko.wikipedia.org/wiki/발견법>
 - 인간과 기계에서 어떤 문제를 해결하거나 제어하기 위해 필요한 정보를 위해 느슨하게 적용시키는 접근을 시도하는 전략
 - 문제를 풀기 위한 정보가 완전히 주어지지 않을 때 발견법이 사용 가능하다
- UCS는 모든 휴리스틱 값이 0인 A* Search의 특수한 경우로 생각 할 수 있다

- Q. Number Line 문제에서의 휴리스틱은?



- Q. 아래 문제에서의 휴리스틱은?
- Q. 장애물이 있다면?

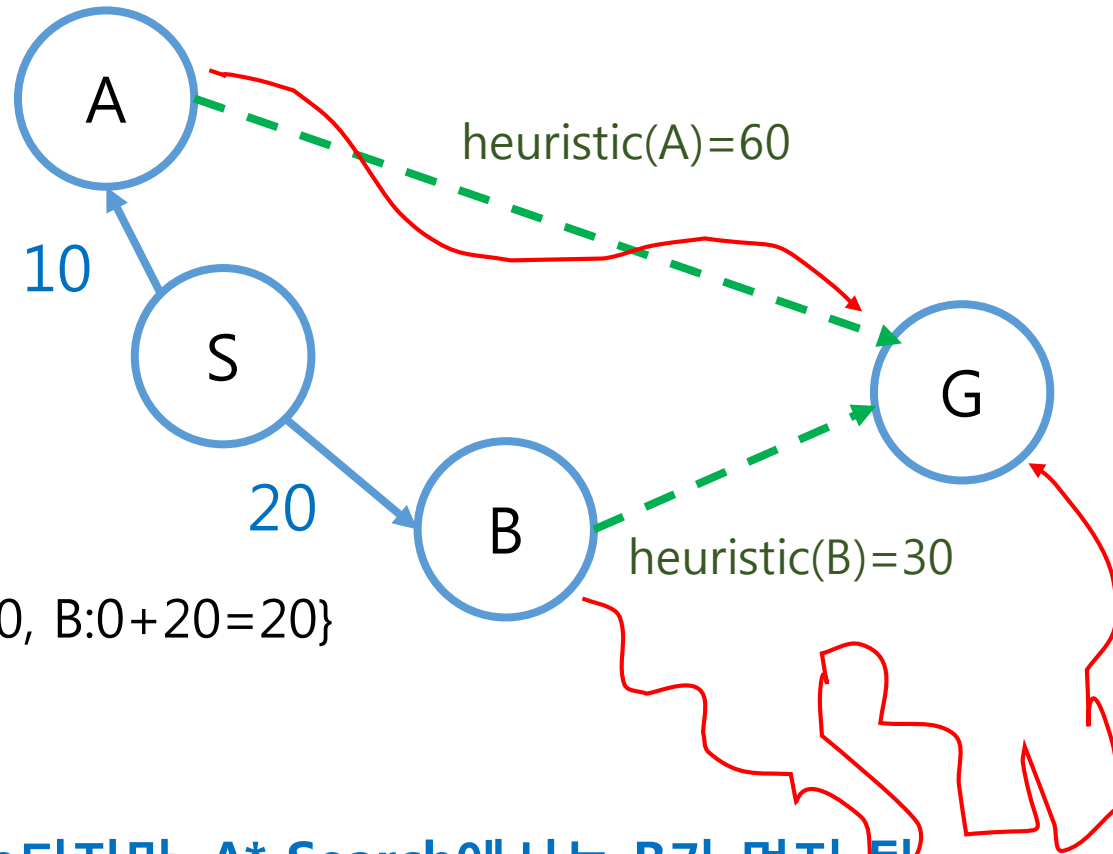
end state				
		 상: cost +2		
	 좌: cost +2	start state	 우: cost +1	
		 하: cost +1		

시작 단계

Frontier: [
 ([A], 0)]
cost_so_far={S:0}
Explored: []

1단계

Frontier: [
 ([S->B], $0+20+30=50$),
 ([S->A], $0+10+60=70$)]
cost_so_far={S:0, A: $0+10=10$, B: $0+20=20$ }
Explored: [S]



UCS에서는 A가 먼저 explore되지만, A* Search에서는 B가 먼저 됨

실제로 어떨지는 모르겠지만 B가 더 가까운 것 같으니 B 먼저 살펴보자!

- 초기화
 - 시작 상태를 *frontier*와 *cost_so_far*에 추가
- 반복 (*frontier*가 텅 빌 때까지)
 - *Frontier*에서 최소 비용 p 를 갖는 솔루션 s 를 제거
 - s 가 목표에 도달했으면:
 - 솔루션 반환하고 알고리즘 종료
 - s 를 *explored*에 추가
 - s 에서 취할 수 있는 모든 가능한 action a 에 대해:
 - s 에서 a 를 취했을 때의 새로운 상태 s' 계산
 - s' 가 *Explored*에 없으면:
 - *Frontier*에 $\text{cost_so_far}[s] + \text{cost}(s, a) + \text{heuristic}(s')$ 비용을 갖는 s' 추가
 - *Cost_so_far*에 s' 의 값을 $\text{cost_so_far}[s] + \text{cost}(s, a)$ 로 업데이트 (최소 비용인 경우)

이전엔 p 값을 그대로 사용

이로써 *heuristic*
값을 고려해
solution을 search
가능

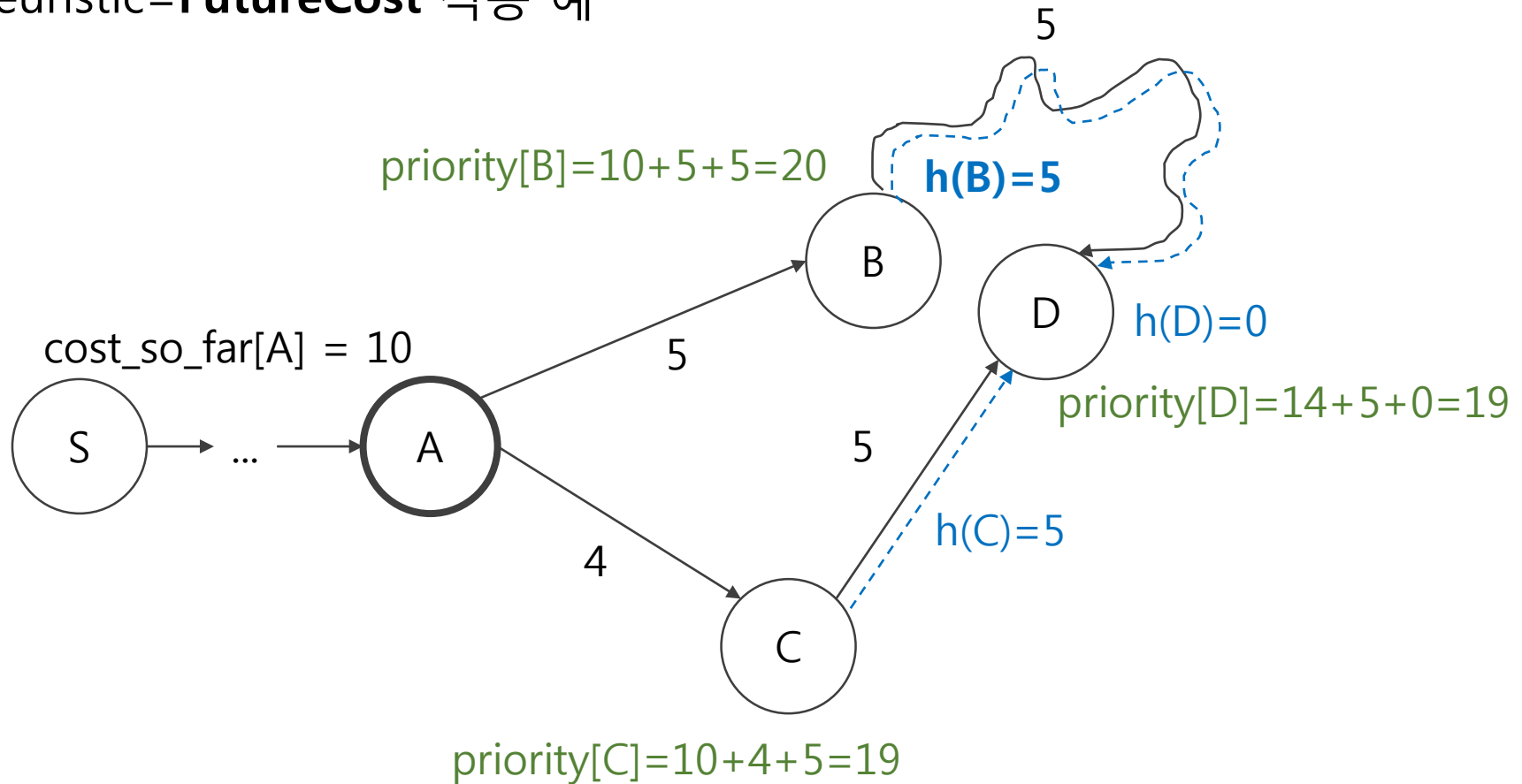
- 풀고자 하는 문제의 SearchProblem class에 **heuristic(state)** 함수 추가
- **Uniform cost search**의 **solve** 함수를 수정해 **A* search** 구현

- A heuristic h is **admissible** if

$$h(s) \leq \text{FutureCost}(s)$$

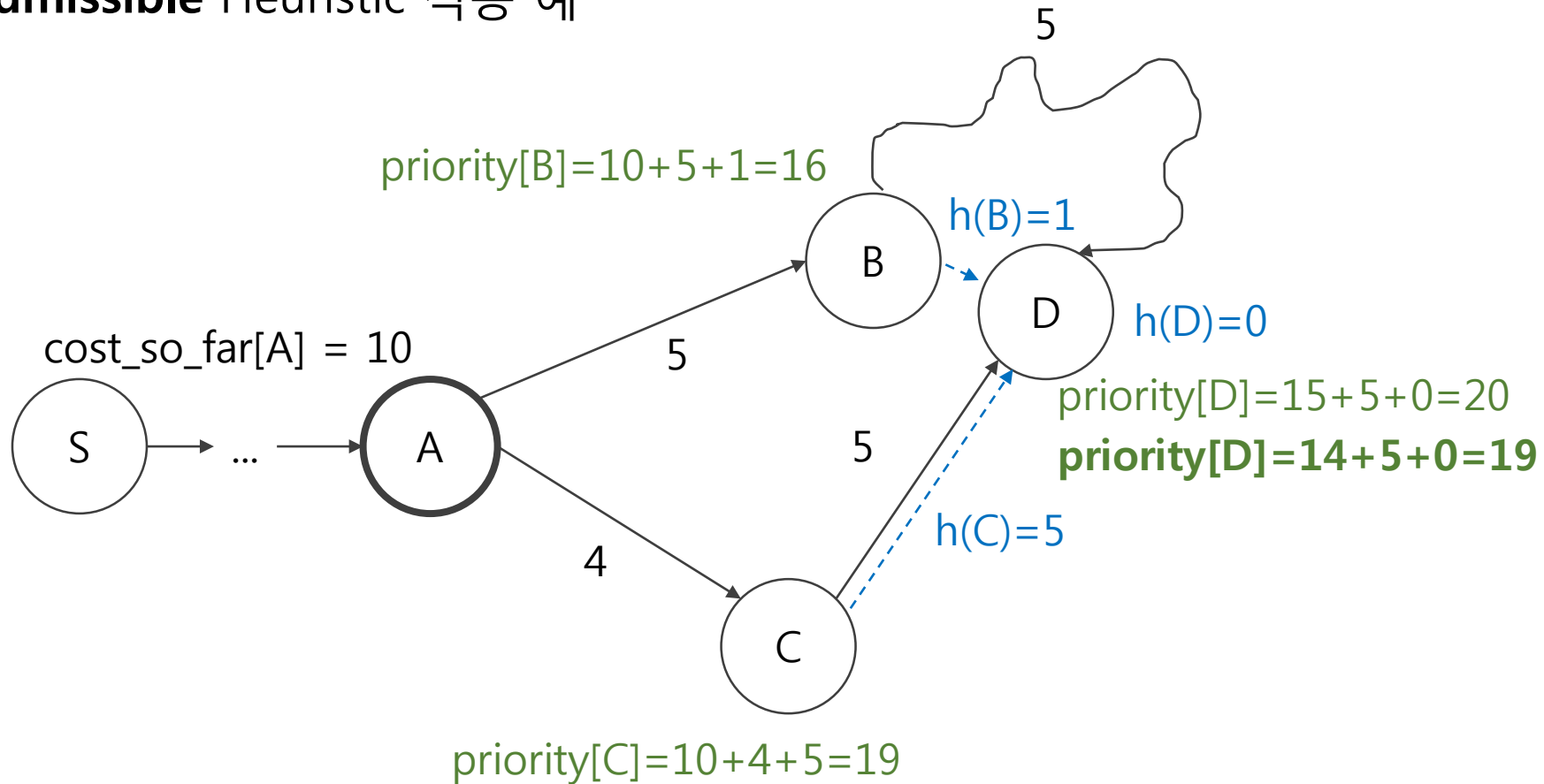
- Admissible heuristic는 future cost를 과소평가(underestimate)
- Admissible heuristic을 사용하는 A*는 최적의 답을 찾는 것이 보장

- Heuristic=**FutureCost** 적용 예



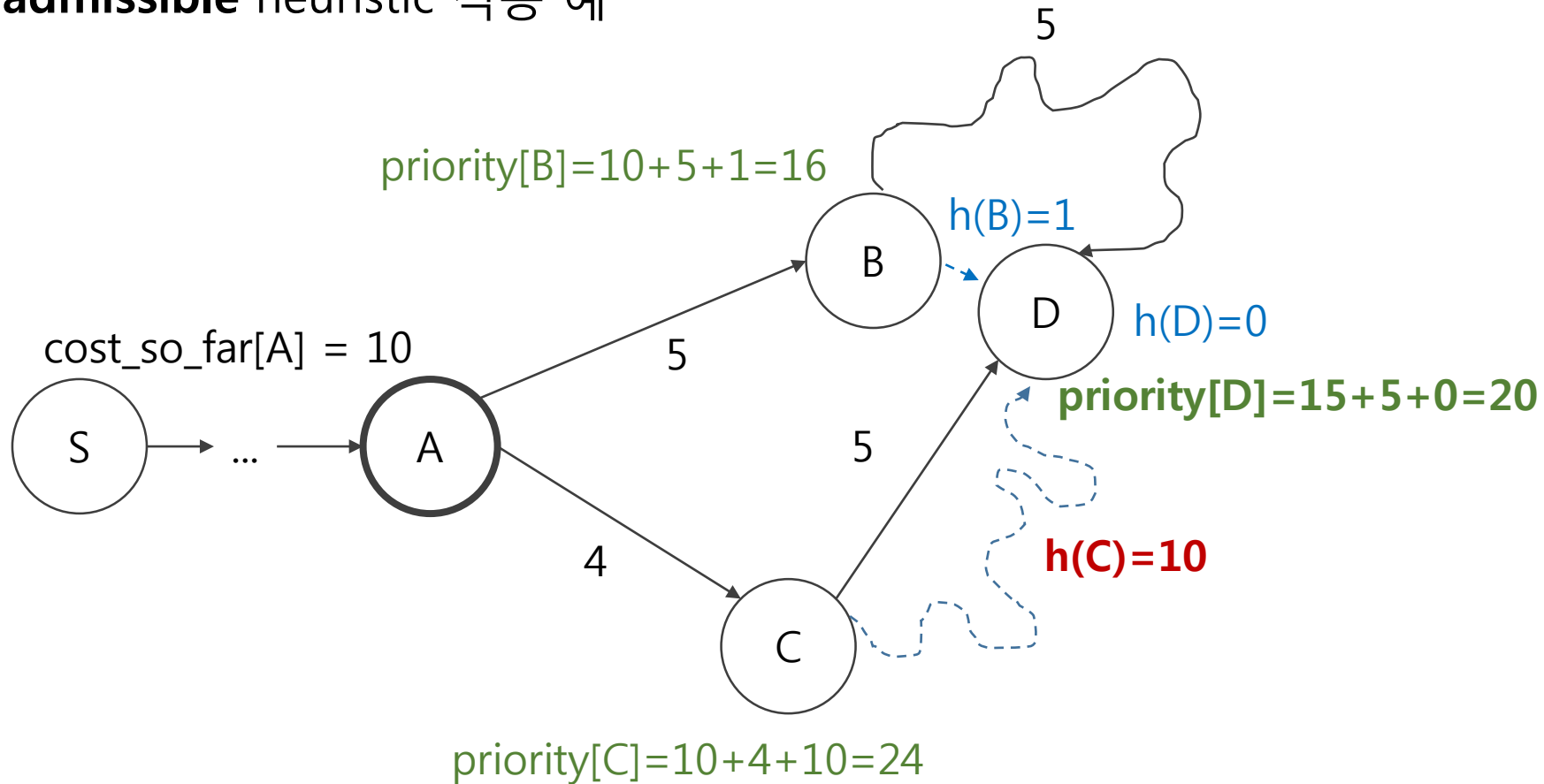
최적의 경로 경로 $S \rightarrow \dots \rightarrow A \rightarrow C \rightarrow D$ 를 바로 선택

- **Admissible** Heuristic 적용 예



최종적으로는 최적의 경로 경로 $S \rightarrow \dots \rightarrow A \rightarrow C \rightarrow D$ 를 선택

- **Inadmissible** heuristic 적용 예



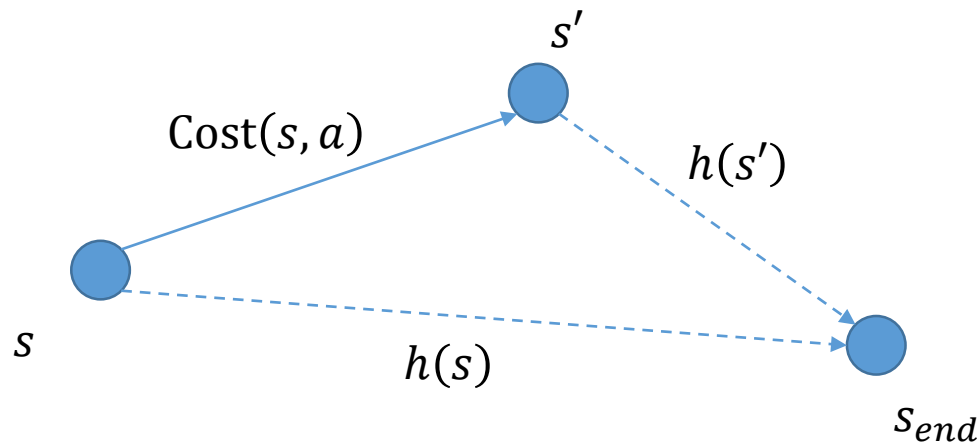
Optimal 하지 않은 경로 $S \rightarrow \dots \rightarrow A \rightarrow B \rightarrow D$ 선택

- A heuristic h is **consistent** if

$$h(s_{end}) = 0$$

and

$$\text{Cost}'(s, a) = \text{Cost}(s, a) + h(s') - h(s) \geq 0, \text{ where } s' = \text{Succ}(s, a)$$



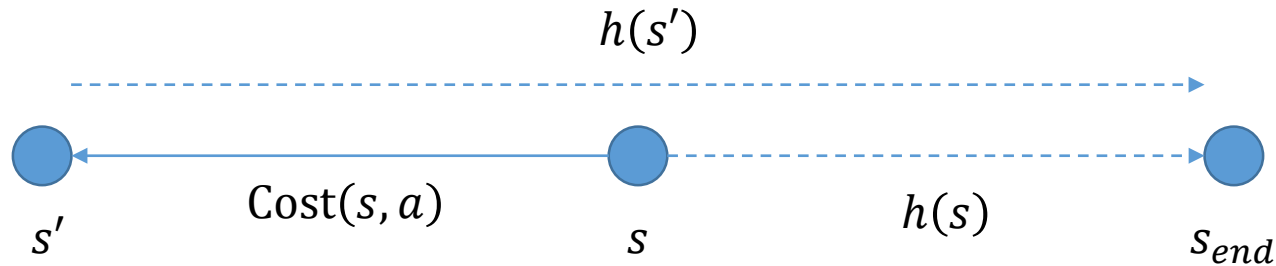
Triangle Inequality

$$\text{Cost}(s, a) + h(s') \geq h(s)$$

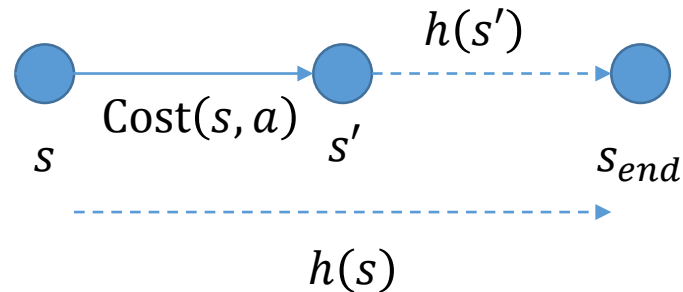
Triangle Inequality

$$\text{Cost}(s, a) + h(s') \geq h(s)$$

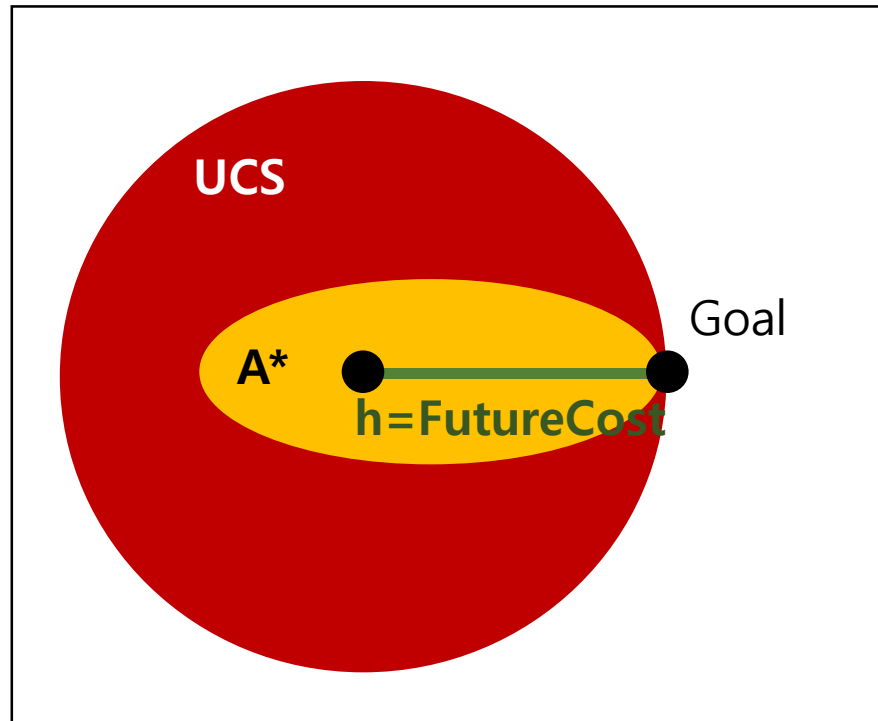
Case 1.



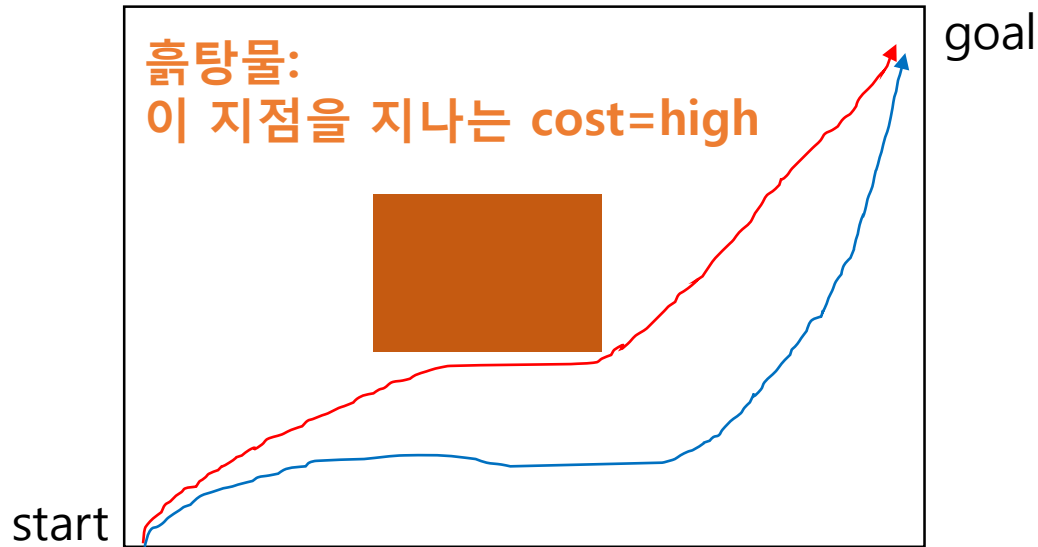
Case 2.



- Consistency implies admissibility



- if $h(s) = 0$, then A*는 UCS와 동일
- if $h(s) = \text{FutureCost}(s)$, then A*는 최단경로만을 탐색
- 실제로는 그 중간 정도



- MDP
 - Uncertainty in the real world

