맹목적(무정보) 탐색

목표에 대한 힌트 없이 해를 찾는 방법

개요

I. 탐색 알고리즘 종류

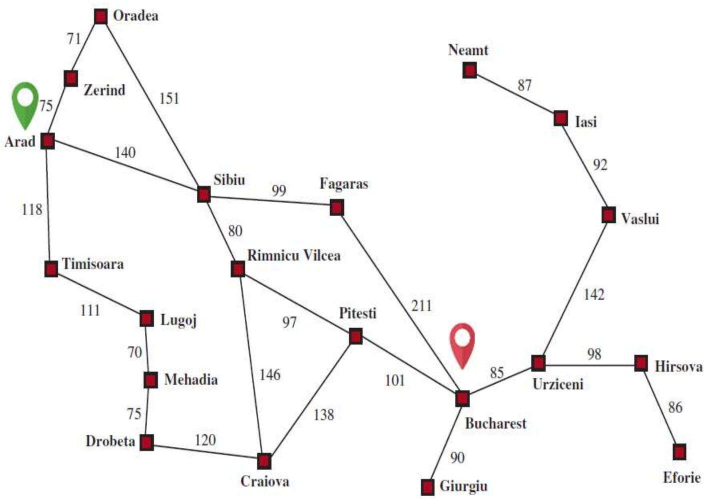
II. 너비 우선 탐색

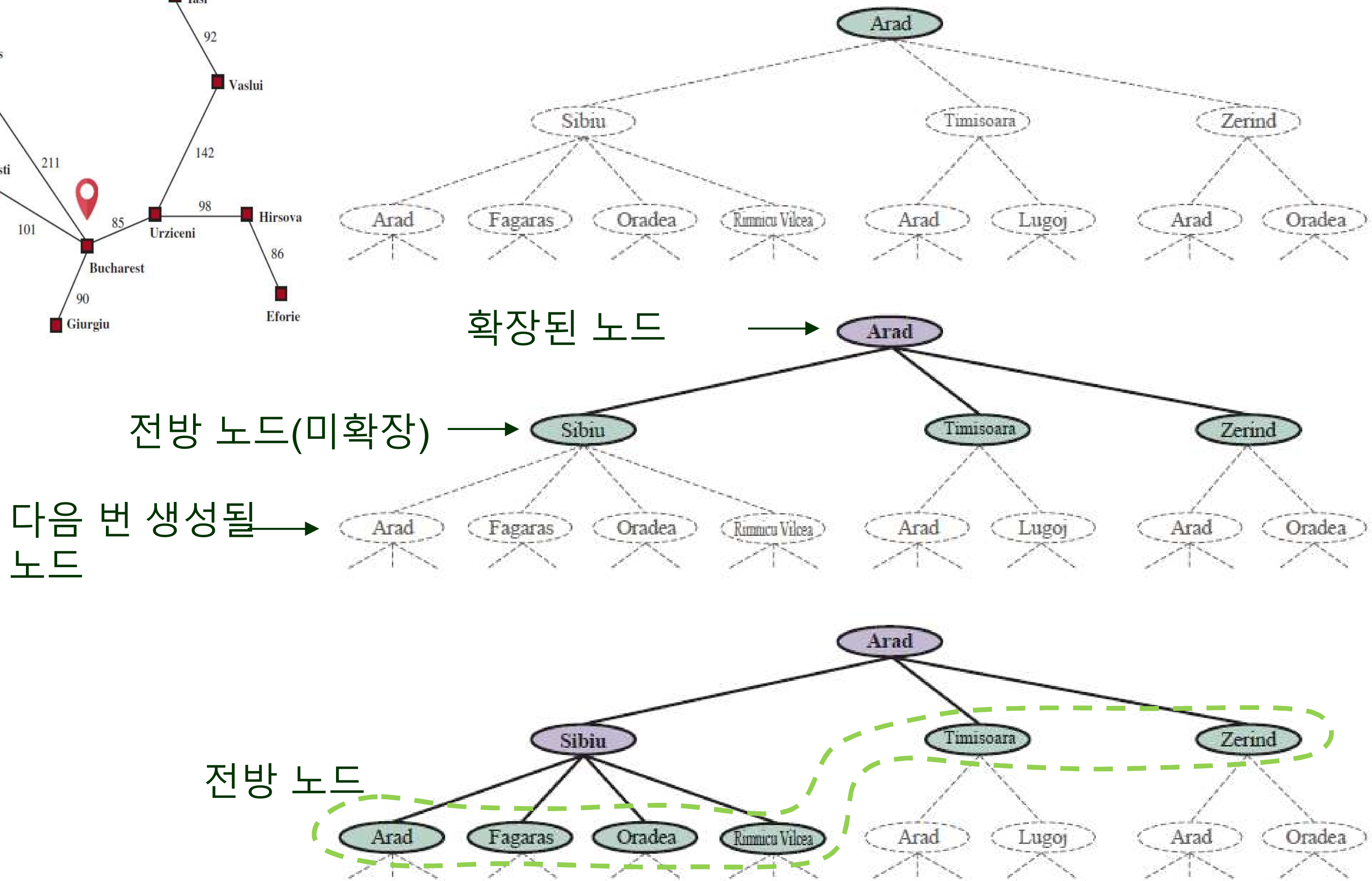
III. 깊이 우선 탐색

IV. 반복적 깊이 증가

V. 양방향 탐색

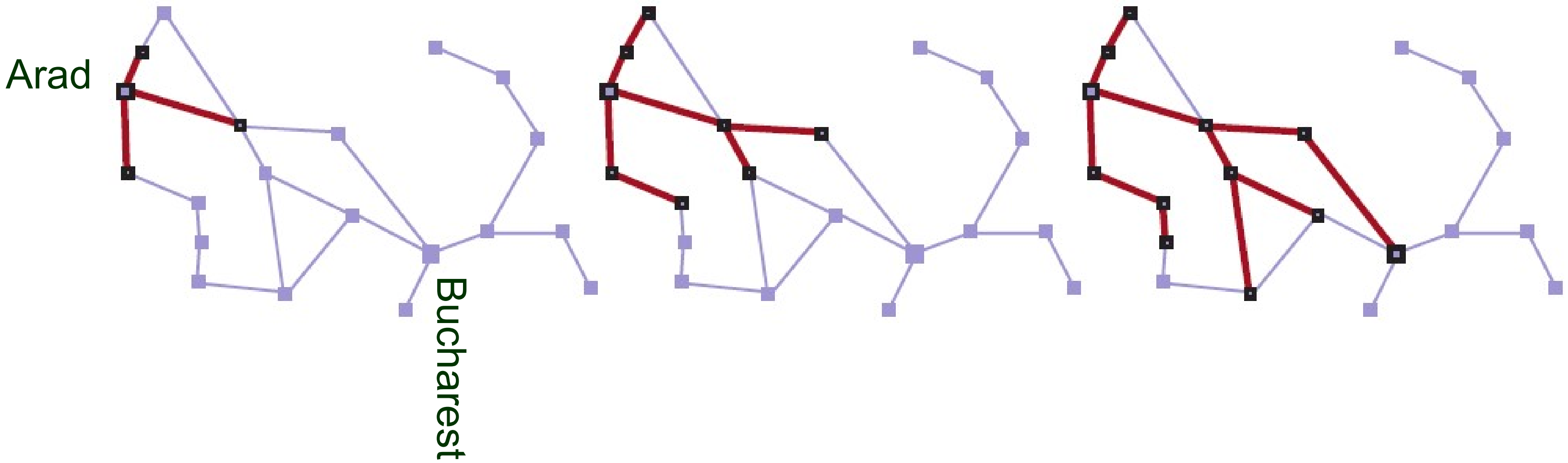
I. 트리 탐색


Oradea
Zerind
Arad
Sibiu
Fagaras
Timisoara
Rimnicu Vilcea
Lugoj
Pitesti
Mehadia
Drobeta
Craiova
Bucharest
Giurgiu
Neamt
Iasi
Vaslui
Urziceni
Hirsova
Eforie
71
75
151
140
118
99
80
211
111
97
70
146
101
75
138
120
90
85
98
86
142
92
87


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea
확장된 노드
전방 노드(미확장)
다음 번 생성될 노드
전방 노드

생성된 탐색 트리

상태 공간 그래프 위에 탐색 트리를 겹쳐 놓고 초기 상태에서 목표 상태까지의 경로를 탐색



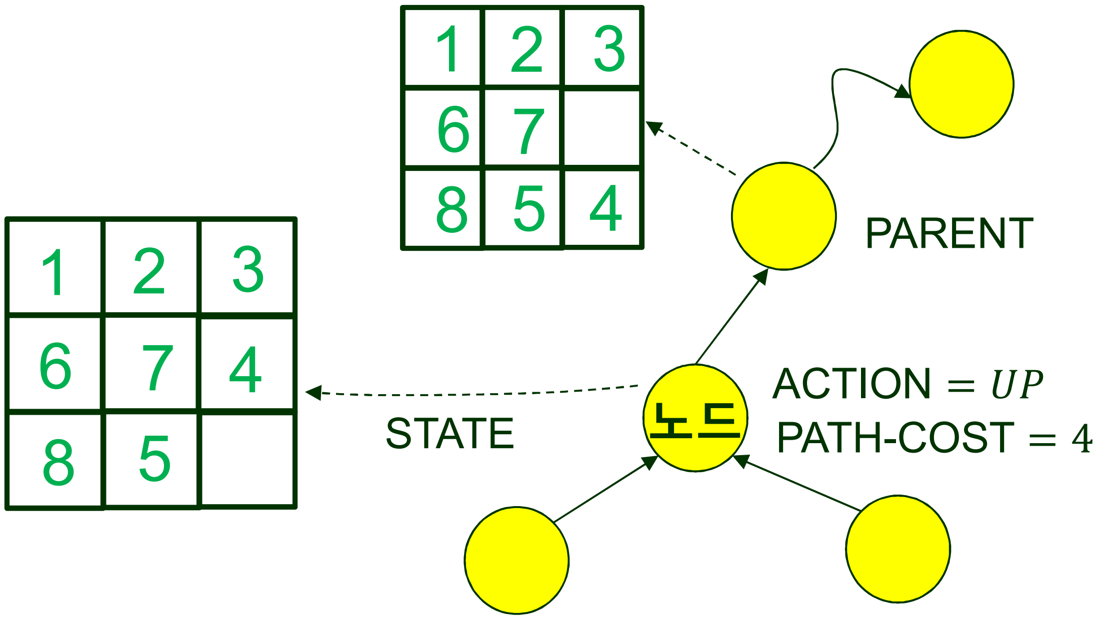
다음 번 순서로 확장되어야 할 노드는?

최선 우선 탐색

특정 평가 함수 $f(n)$이 최소가 되도록 하는 노드 n을 선택

- 목표 상태가 되면 리턴
- 아니면, 자식 노드 생성 및 전방 배치

노드 구조:

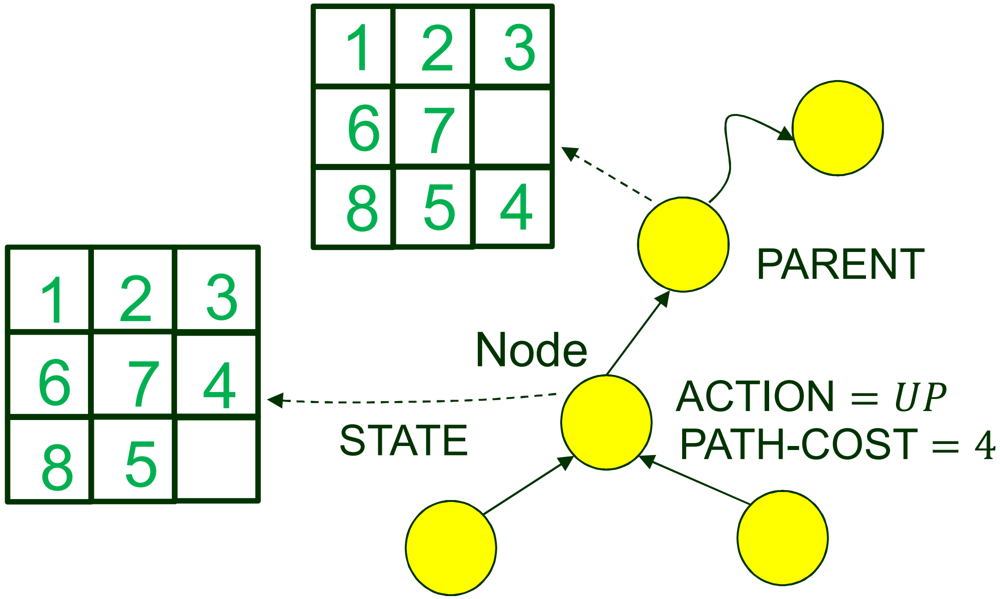


- *node*.STATE
- *node*.PARENT
- *node*.ACTION: 부모 노드에서 이 노드를 생성할 때 적용한 액션
- *node*.PATH-COST: 시작 노드부터 현재 노드까지의 총비용 $g(n)$

자료 구조

전방:

- IS-EMPTY(*frontier*)
- POP(*frontier*)
- TOP(*frontier*)
- ADD(*node*, *frontier*)



세가지 큐

- 우선 순위 큐: 최소 비용을 가진 노드를 팝(pop)
- FIFO 큐: 먼저 추가된 노드를 팝(너비 우선 탐색에서 사용)
- LIFO 큐: 나중에 추가된 노드를 팝(깊이 우선 탐색에서 사용)

최선 우선 탐색 알고리즘

큐 종류 바꾸면 BFS나
DFS로 변경 가능

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

우선 순위 큐에 노드 추가
(전방 배치)

방문한 적이 있는 노드인지 확인

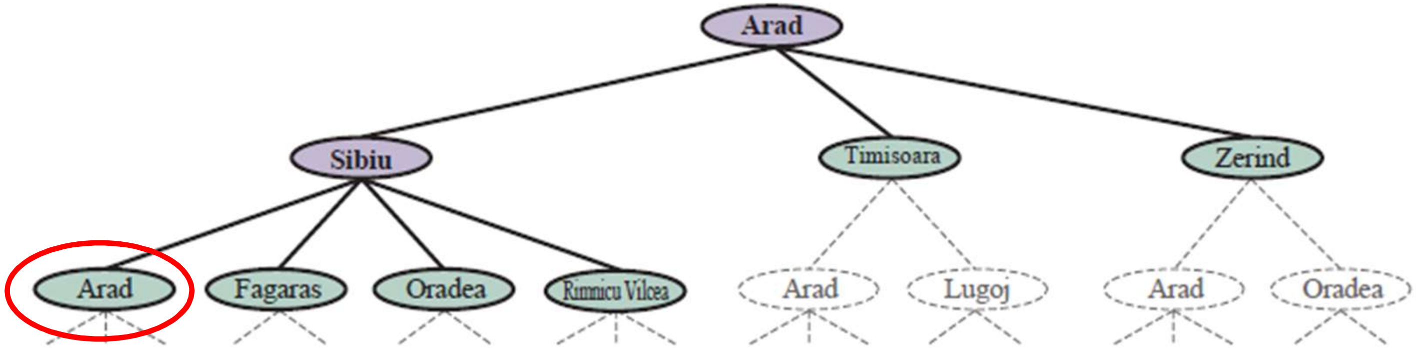
자식 노드에 방문한 적이 없거나, 있더라도 이전 비용 보다 적을 경우

```
function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

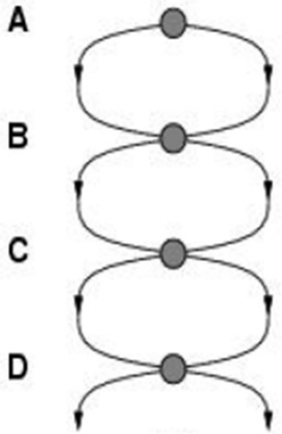
최선 우선 탐색 알고리즘 (해설)

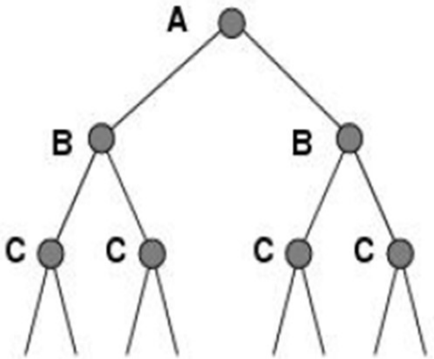
1. 시작 노드 초기화
2. 우선순위 큐에 추가
3. 방문했었는지 확인하기 위한 테이블을 만들고, 초기 상태에 대한 항목 추가
4. 우선순위 큐가 비어 있지 않은 동안...
   1. 우선순위 큐에서 노드 꺼내기(POP)
   2. 해당 노드가 목표 상태인지 확인. 목표 상태라면 해당 노드 반환
   3. 해당 노드 확장하여 자식 노드들 생성
   4. 각 자식 노드에 대해...
      1. 자식 노드 상태 확인 및 이전에 방문했던 상태인지 확인
      2. 새로운 자식 노드가 이전에 도달한 상태보다 더 낮은 비용을 도달할 수 있다면, 테이블 갱신하고 우선순위 큐에 해당 자식 노드를 추가 (frontie로)
5. 끝까지 목표 상태를 찾지 못했다면 '실패'를 반환

이미 방문했던 상태 재방문



- 반복된 상태를 감지하지 못하면 해결 가능한 문제가 해결 불가능한 문제로 변할 수 있음





- 각 상태로 오는 최상의 경로만 유지

성과 측정 기준

- 완전성: 대상 알고리즘은 해가 존재하기만 하면 무조건 찾아 낼 수 있는가?

상태 공간은 무한대 크기일 수 있다!

- 비용 최적성: 모든 해 중에 최소 경로 비용을 갖는 해를 찾을 수 있는가?

- 시간 복잡도: 물리적인 시간, 또는 상태나 액션의 개수

- 공간 복잡도: 탐색에 필요한 메모리 크기

$$|V| + |E|?$$

깊이와 분기 요소

- 그래프가 명시적으로 주어진 경우에는 $|V|+|E|$를 기준으로 삼는 것이 적절함

- 현실에서는 대부분의 그래프 표현이 암묵적임 (노드와 에지 불분명)
  - 초기 상태
  - 액션
  - 전이 모델

  주어진 정보로부터 점진적으로 추정해야 함

- 따라서 복잡도는 아래 요소들을 기반으로 측정됨
  - d: 깊이 (최적해에 포함된 액션 개수)
  - m: 임의의 경로를 구성하는 액션의 최대 개수
  - b: 분기 요소(현재 노드의 후임 노드 개수)

무한대 상태 공간

크누스의 추정: 4보다 큰 모든 정수는 4에 대해 제곱근, 내림(floor), 팩토리얼 연산을 적용하여 도달 가능

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

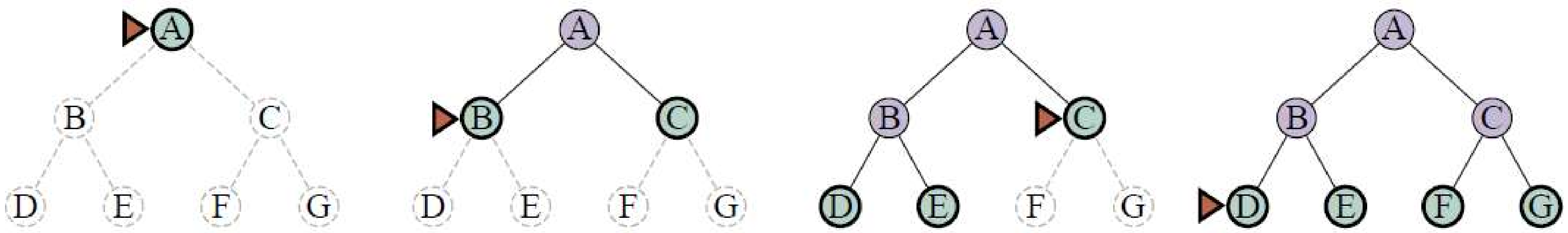
$$4 \;-\; 4! \;-\; (4!)! \;-\; \left((4!)!\right)! \;-\; \vdots$$

알고리즘 팩토리얼 연산 반복 적용

너비 우선 탐색

맹목적(무정보) 탐색: 현재 상태가 얼마나 목표 상태에 가까워졌는지 모름

루트 노드를 맨 먼저 확장(액션 적용하여 자식노드 생성)한 뒤, 후임 노드 생성 ⟹ 이 과정을 반복 적용



- 체계적인 탐색 방식
- 상태 공간이 무한할 경우에도 완료 가능 (탐색 깊이를 제한하거나, 목표 범위 지정 등)
- 항상 최소한의 행동으로 해를 찾음

BFS (너비우선) 알고리즘

- BEST-FIRST-SEARCH 알고리즘의 평가 함수를 $f(n)$ = node depth 로 변경함으로써 BFS 구현가능(예. 깊이 1인 노드 먼저 다 처리하고 깊이 2로...)
- 추가로 FIFO 큐 사용(우선순위 큐는 효율이 떨어짐), 빠른(이른) 목표 테스팅(노드 생성 시) 적용

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

시간 공간 복잡성

분기 요소

각 노드가 b개의 후임을 갖는 무정보 트리에 적용된 BFS 알고리즘

- 루트 노드로부터 생성된(depth=1) 노드는 b개
- Depth=1인 각 노드로부터 다시 b개의 노드 생성 ==> b^2 개의 노드 (depth=2)
- 계속 반복....

해가 depth d에 존재 $\Longrightarrow$ #nodes $= 1 + b + \cdots + b^d = O(b^d)$

$$= \frac{b^{d+1}-1}{b-1} \text{ (단, } b > 1)$$

시간 공간 복잡도

(모든 노드는 메모리에 존재한다고 가정)

- 지수적 시간 복잡성으로 인해 소규모 문제만 해결 가능
- 메모리가 시간보다 더 큰 이슈임

BFS의 시간 공간 요구사항

트리 탐색: $O(b^d)$

Depth	Nodes	Time	Memory
2	110	0.11 milliseconds	107 KB
4	11,110	11 milliseconds	10.6 MB
6	10^6	1.1 seconds	1 GB
8	10^8	2 minutes	103 GB
10	10^{10}	3 hours	10 TB
12	10^{12}	13 days	1 PB
14	10^{14}	3.5 years	99 PB
16	10^{16}	350 years	10 EB

(단, $b = 10$ 가정)

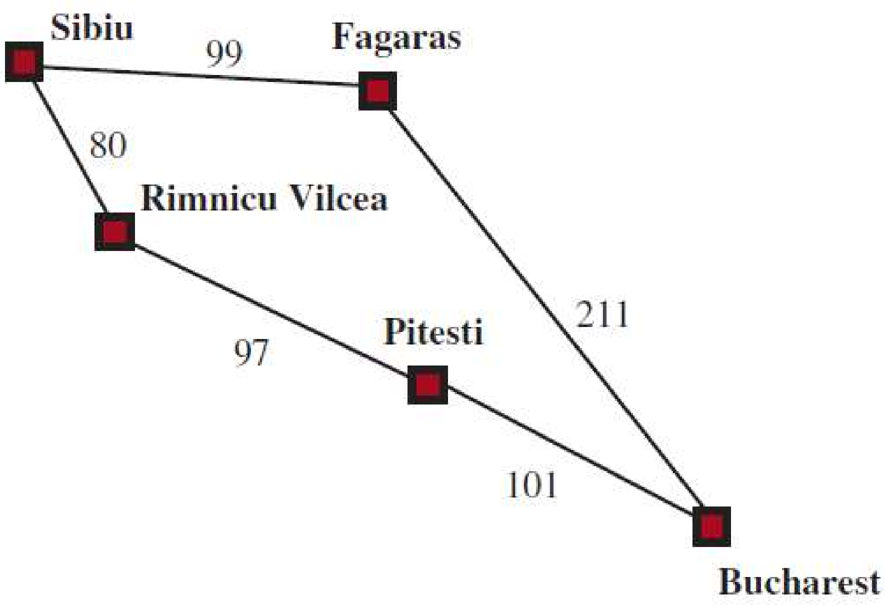
그래프 탐색 알고리즘이 선호됨: 그래프 탐색 알고리즘은 상태 공간의 크기에 비례하여 필요한 시간과 공간의 크기가 결정됨 (대부분$O(b^d)$보다 작다).

균일 비용 탐색(다익스트라 알고리즘)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

최선 우선 탐색과 동일.
단, 경로 비용을 적용

- 단위가 균일한 경로 비용이 웨이브를 형성하면서 전파



Rimnicu Vilcea (80)
Fagaras (99)

⇩

Fagaras (99)
Pitesti (80 + 97 = 177)

⇩

Pitesti (177)
Bucharest (99 + 211 = 310)

⇩

Bucharest (177 + 101 = 278 < 310)

균일 비용 탐색: 완전성과 복잡성

♦ 완전성: 모든 경로를 탐험할 수 있도록 체계적일 것 - 특정 경로에 갇히지 않음

♦ 최적성: 다익스트라 알고리즘의 최적성을 추종

♦ 복잡성: 깊이가 아니라 비용을 고려함

$$O(b^{1+\lfloor C^*/\epsilon \rfloor})$$

C^*: 최적 해의 비용 (최적 비용)

ϵ: 액션에 부과된 비용의 최소 하한선

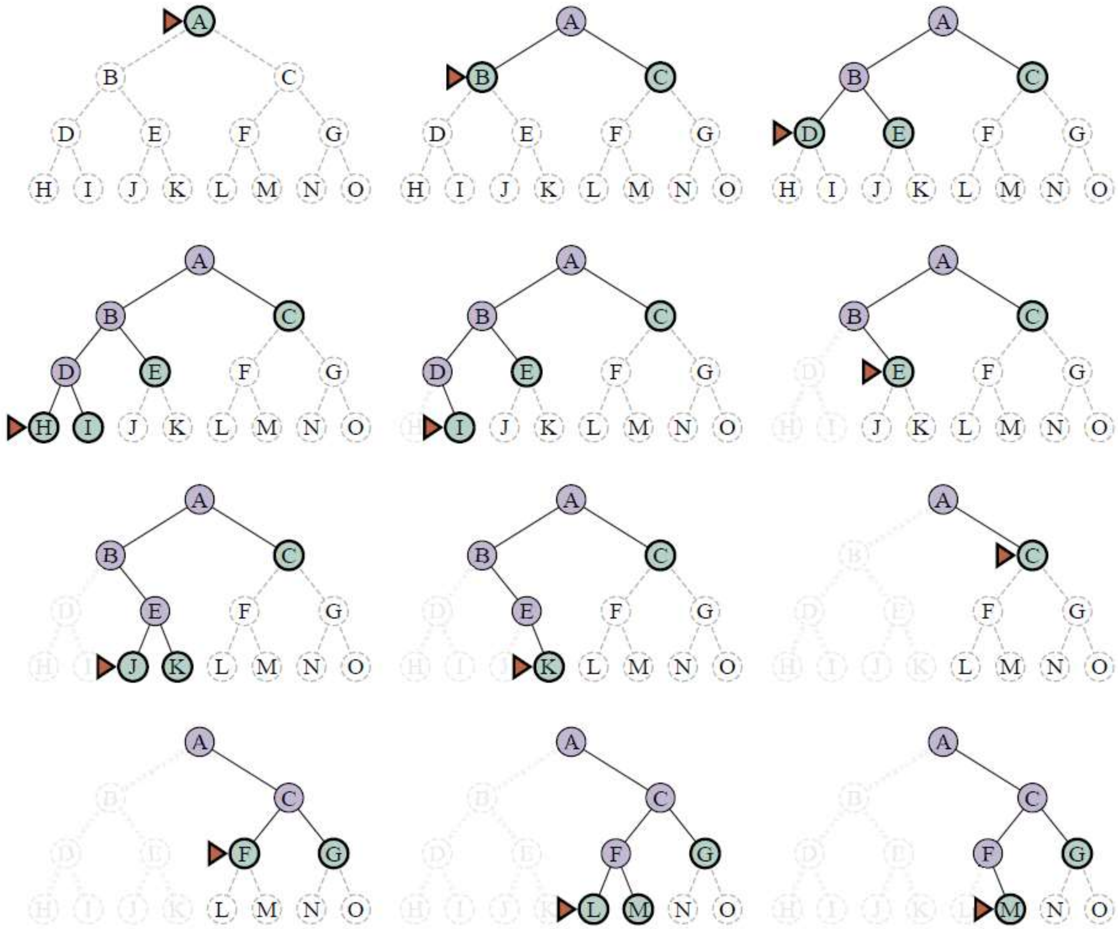
$\gg$ b^d , 즉 b^d 보다 훨씬 커질 수 있음
(예를 들어 짧은 단계로 구성된 사소한 경로를 먼저 탐험하는 경우)

$=$ b^{d+1} (모든 액션에 동일한 비용이 부과된 경우)

깊이 우선 탐색

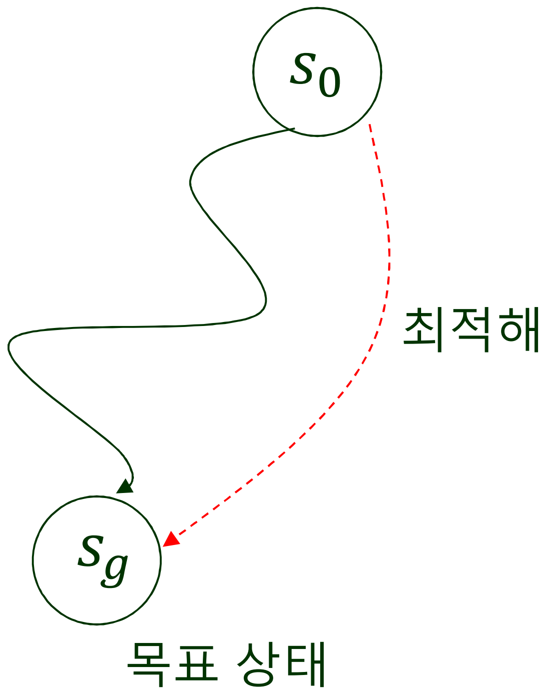
가장 깊은 곳에 있는 노드를 확장하여 전방에 우선 배치

- 최선 우선 탐색 알고리즘의 f(n)에 음의 경로를 적용하여 구현



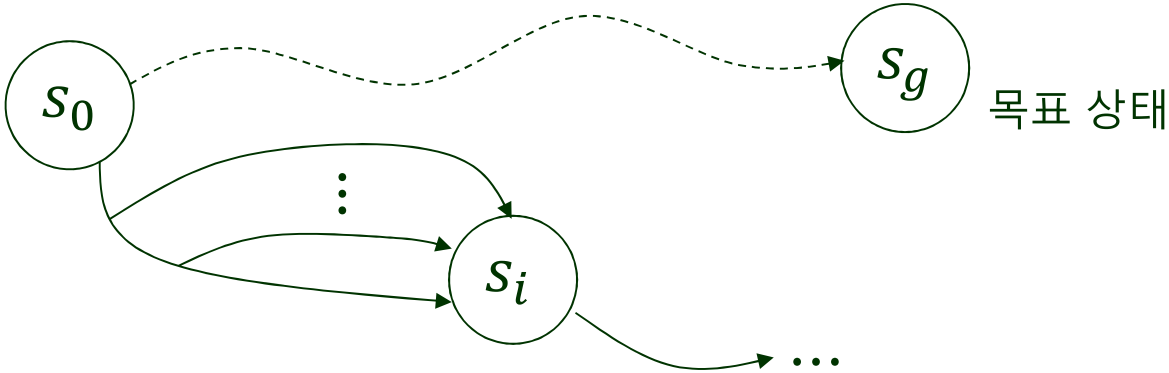
DFS의 비최적성

최저 비용의 해가 아닐지라도 조건에 맞는 첫 번째로 발견된 해를 반환

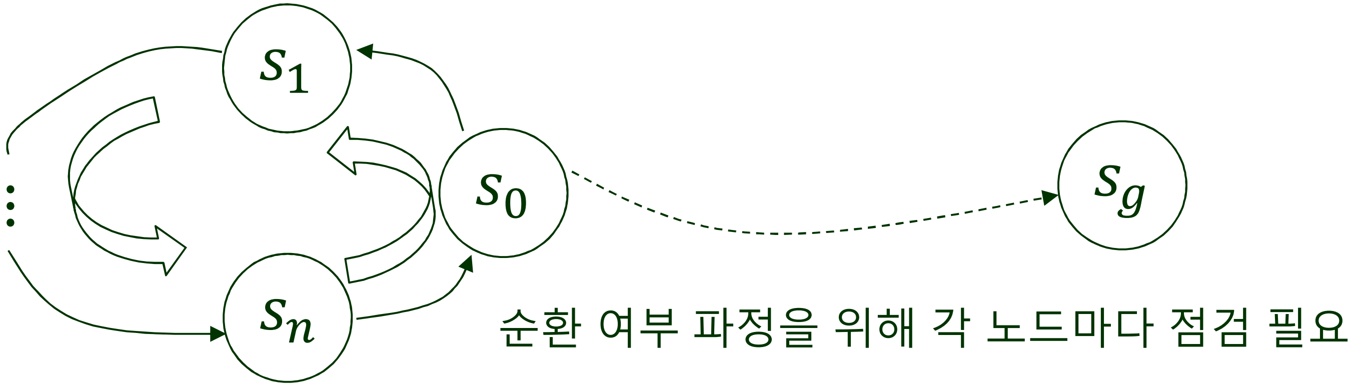


DFS의 비효율성

여러 경로를 통해 특정 상태에 도달할 수 있는 경로가 여러개 존재할 수 있으므로 해당 상태가 여러번 중복해서 확장될 수 있음. 하지만 비순환(acyclic)이기만 하다면 결국에는 전체 공간을 체계적으로 탐색하게됨

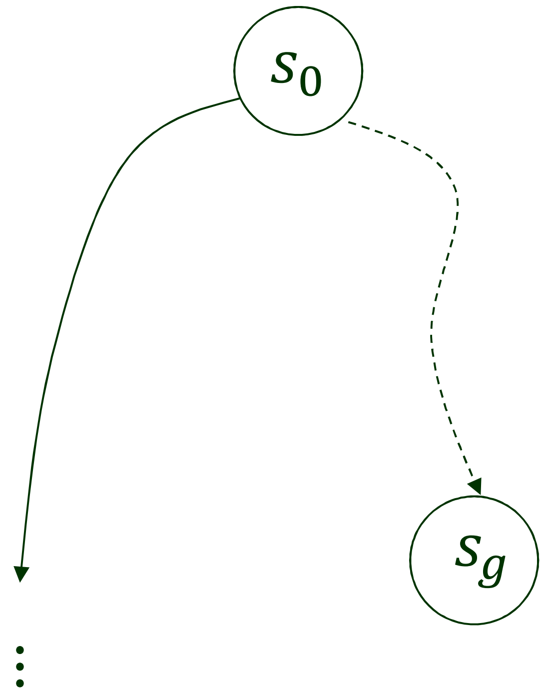


순환 상태 공간인 경우에는 무한 루프에 빠질 수 있음



DFS의 불완전성

순환이 존재하지 않는데도 무한 길이의 경로로 내려갈 수 있음

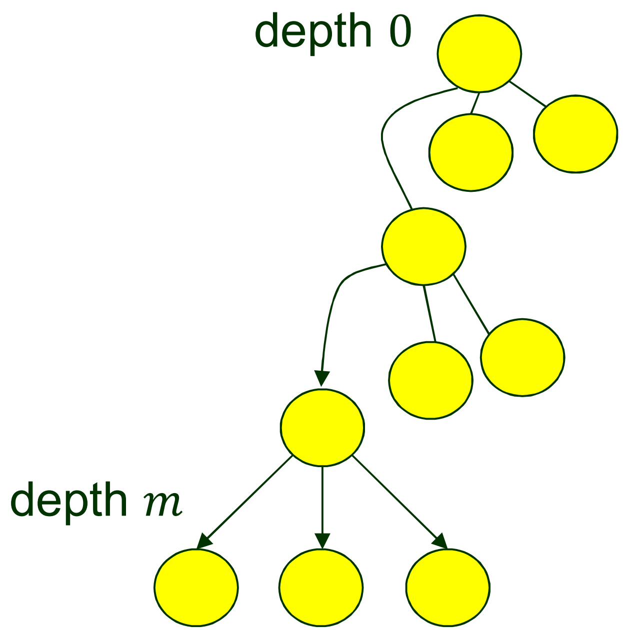


그래도 DFS를 쓰는 이유

트리 형태의 탐색이 가능한 문제이기만 하다면 **작은 메모리**로 구현 가능

기존에 방문 했었는지 여부를 기록하기
위한 테이블이 불필요

전방 노드들의 개수가 적다



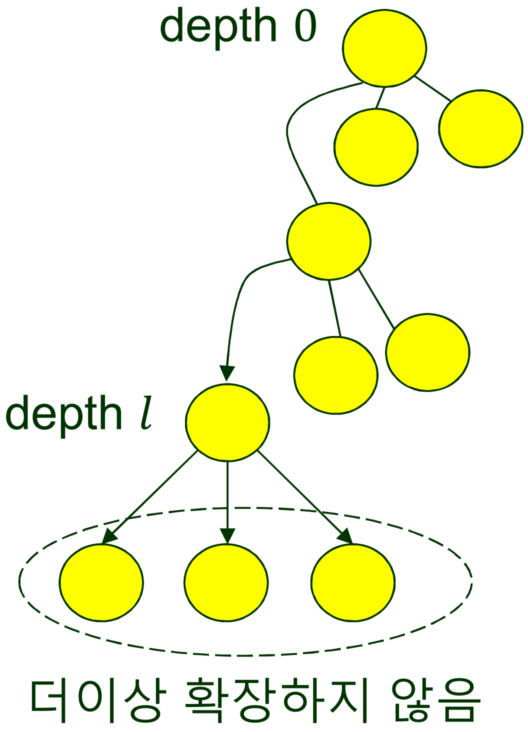
- 검색 시간: O(#nodes).
- 메모리 소비량: $O(bm)$.

분기 요소
(후임 노드 수)

최대 깊이

- 제약 충족, 논리 프로그래밍 등의 실용적 도구

깊이 제한 탐색



- To 무한대 경로로의 탐험을 피하기 위해 깊이 제한값 l 을 추가로 고려
- 깊이 에 위치한 모든 노드는 후임이 있어도 끝단으로 간주됨

시간: $O(b^l)$

메모리: $O(bl)$

- l을 어떻게 설정하느냐에 따라 성능이 크게 좌우됨

l값을 잘 선택하기 위해 어떤 전략을 쓸 수 있을까?

반복 심화 탐색

깊이 제한 l 값을 0, 1, 2, ... 로 차례로 늘려 가면서 반복 탐색

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution node or *failure*
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ *cutoff* **then return** *result*

DEPTH-LIMITED-SEARCH 알고리즘이 반환하는 값

- 목표 상태: 발견 된다면
- "실패": 모든 노드를 탐색했고 해가 없다고 확인 될 때
- 컷오프: 현재 제한 값 l 의 한도 내에서는 해를 찾을 수 없을 때 (depth > l 인 경우 해가 존재할 수도 있음)

- DFS와 BFS의 장점만 결합한 형태
- BFS의 완정성과 최정성을 보유

반복 심화 탐색의 시간과 공간

Case 1 목표 상태가 depth *d* 에서 발견된 경우

맨 밑의 노드들은 한 번만 생성됨
맨 밑에서 바로 위에 있는 노드들은 두번씩 생성

⋮

시간 ~ 노드수 $\leq b^d + 2b^{d-1} + \cdots + (d-1)b^2 + db$

$= O(b^d)$ BFS와 동일

메모리: $O(bd)$

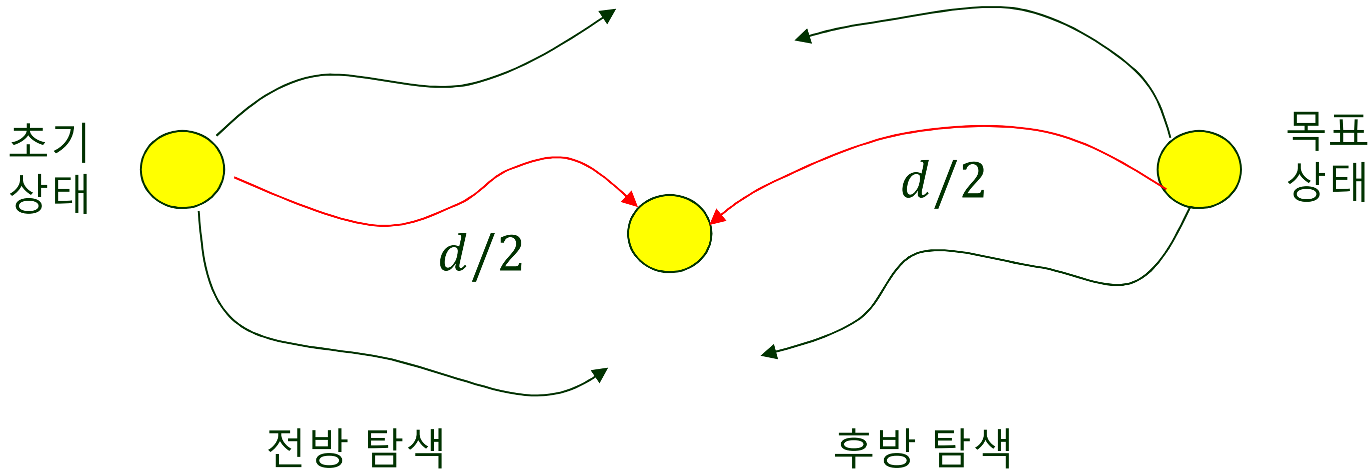
Case 2 목표 상태가 유한한 상태 공간에서 발견되지 않는 경우

시간 $O(b^m)$

메모리 $O(bm)$ m: 최장 경로의 액션 수

- DFS와 유사하게 적정한 메모리가 소요됨

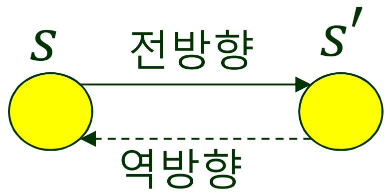
양방향 탐색



동기:

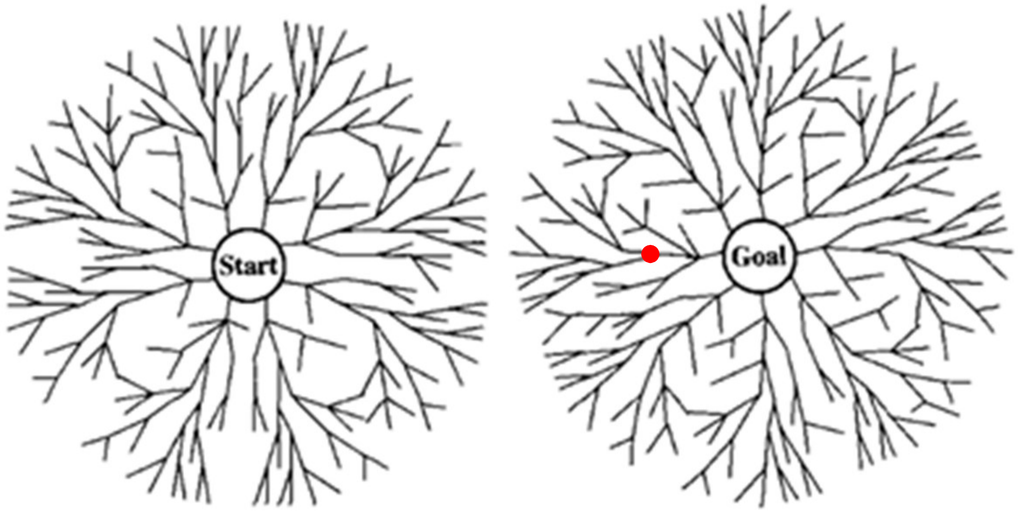
$$b^{d/2} + b^{d/2} \ll b^d$$

역방향으로의 추론이 필요함



역방향으로 보면 s 는 s'의 후임이 됨 (전방향에서는 선임이지만)

후향 추론



- 모든 액션이 뒤집을 수 있는 경우 쉽게 구현 가능

8-퍼즐

루마니아 도시 여행 경로

- 목표가 추상적으로 설정된 경우 구현이 어려움

8-퀸

양방향 최선 우선 탐색

문제를 두가지 버전으로 준비(전방향, 후방향)

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                         // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                         // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution
```

전방 배치된 노드를 확장하고 자식 노드가 다른 전방
라인의 노드에 의해 도달했던 상태인지 확인

무정보 탐색 알고리즘 비교

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

b: 분기 요소

d: 해가 존재하는 최소 깊이

l: 깊이 제한값

m: 탐색 트리 깊이 최대값

C^*: 최적해의 비용

ϵ: 최소 액션 비용

1. b가 유한하고 상태 공간에 해가 존재하며 상태 공간도 유한하다면 yes

2. 모든 비용이 $\geq \varepsilon > 0$ 이면 yes

3. 모든 비용이 동일하면 yes

4. 양방향 모두 너비 우선이거나 균일비용이면 yes