가장 기초적인 탐색 기법

"Uninformed" Search

특정한 사전 정보 없이 문제해결을 위해 탐색공간을 탐색하는 알고리즘

Note: this material was originated from the slides provided by Prof. Padhraic Smyth

Search Algorithms

- Uninformed Blind search
  - Breadth-first
  - depth-first
  - Iterative deepening depth-first
  - uniform cost
- Informed Heuristic search
  - Greedy search, Heuristics, hill climbing,
- Important concepts:
  - Completeness
  - Time complexity
  - Space complexity
  - Quality of solution

성능 판단을 위해 따져볼 것들

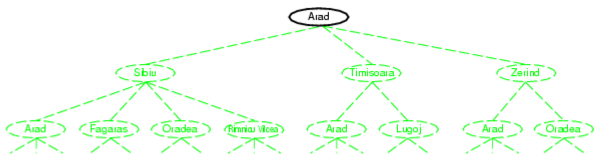
탐색은 보통 tree base로 이루어진다

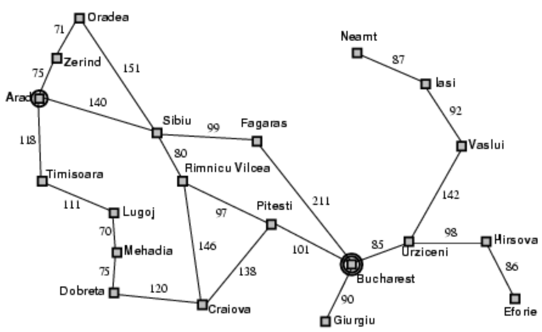
Tree-based Search

- Basic idea: node 탐색
  - (Exploration of state space) by generating successors of already-explored states (a.k.a. expanding states). = child node
    = 갈 수 있는 다음 노드
    탐색이 된 공간으로부터 다음 노드를 탐색한다
  - Every state is evaluated: *is it a goal state*?
    여기 도착하면 완료

= generating successor
= expanding states

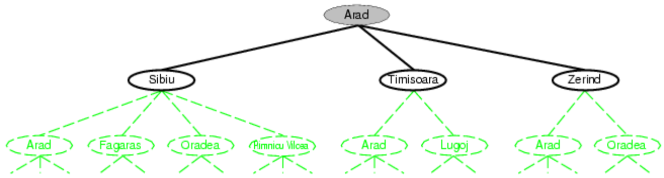
Tree search example





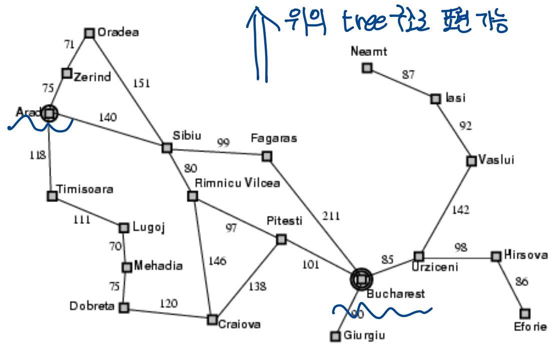
Tree search example

계속 expand 하다
goal state인 B가 나오면
이 경로를 A→B path라고
할 수 있다



node를 expand 한다
(child를 계속 붙여
나가는 행위)

위의 tree 구조 표현 가능



Search Tree for the 8 puzzle problem

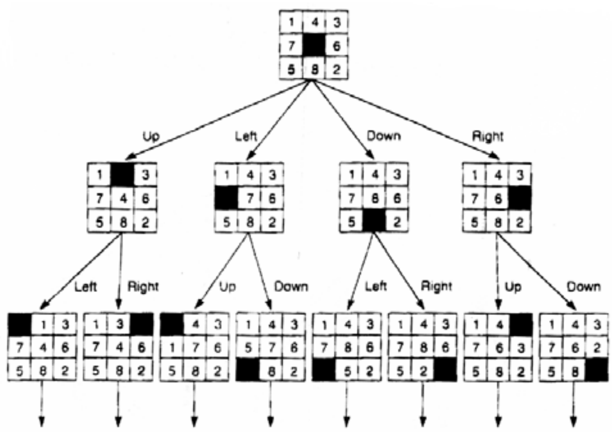


Figure 3.6 State space of the 8-puzzle generated by "move blank" operations.

Search Strategies

- A search strategy is defined by picking the order of node expansion

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists? → 항상 정답을 찾아낼 수 있는가?
  - time complexity: number of nodes generated → 탐색 시간
  - space complexity: maximum number of nodes in memory → 메모리 요구량
  - optimality: does it always find a least-cost solution? → 찾은 Solution이 Optimal한가?

- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree → 확장 가능한 child 개수 ⇒ b개 까지 확장 가능하다
  - *d:* depth of the least-cost solution → Solution의 depth
  - *m*: maximum depth of the state space (may be ∞)

  ↳ maximum depth ↳ 무한대일 수도 있다

Breadth-First Search (BFS)

- Expand shallowest unexpanded node (가까운 순서로 확장)

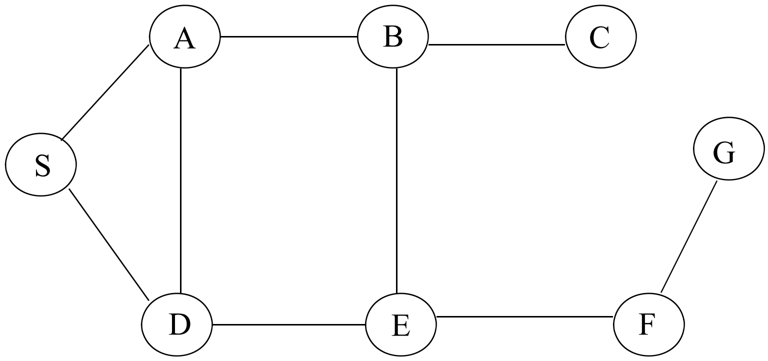
- Fringe: nodes waiting in a queue to be explored, also called OPEN

- Implementation:
  - For BFS, *fringe* is a first-in-first-out (FIFO) queue
  - new successors go at end of the queue

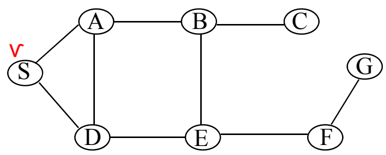
- Repeated states?
  - Simple strategy:
    - ①번 strategy ↱ 이미 갔던 노드는 queue에 추가하지 않는다
    - do not add an already-expanded node to the queue
    - do not expand an already-expanded node
    - ↳ 이미 확장된 node는 확장하지 마라
    - ②번 strategy

Example: Map Navigation

State Space:
S = start, G = goal, other nodes = intermediate states, links = legal transitions



BFS Search Tree



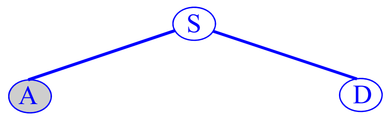
Queue = {S}

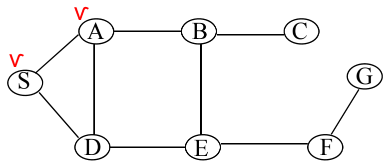
Select S

Goal(S) = true?

If not, Expand(S)

BFS Search Tree





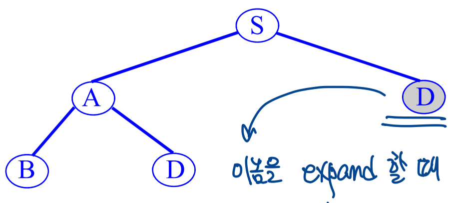
Queue = {A, D}

Select A

Goal(A) = true?

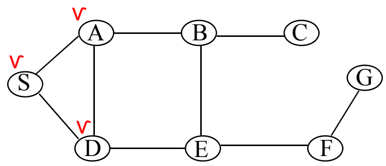
If not, Expand(A)

BFS Search Tree



이놈은 expand 할 때 A는 이미 expand 했던 놈이므로 queue에 넣지 않는다

⇒ 1번 strategy



Queue = {D, B, D}

Select D

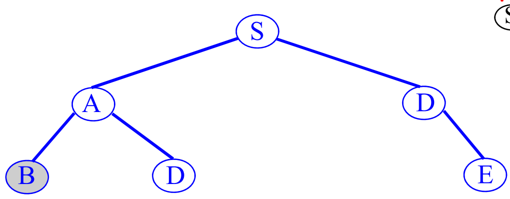
Goal(D) = true?

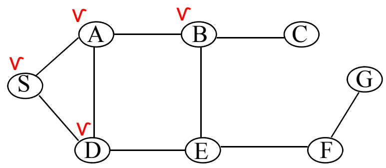
If not, expand(D)

D가 enqueue된 이유

⇒ ①, ②번 strategy에 해당되지 않으므로

BFS Search Tree





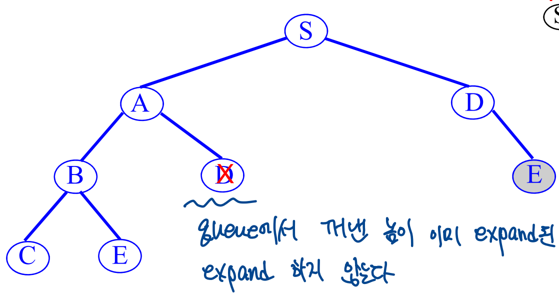
Queue = {B, D, E}

Select B

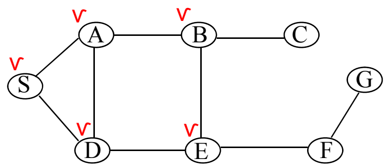
Goal(B) = true?

If not, expand(B)

BFS Search Tree



queue에서 꺼낸 놈이 이미 expand된 놈이면
expand 하지 않는다
⇒ ②번 strategy



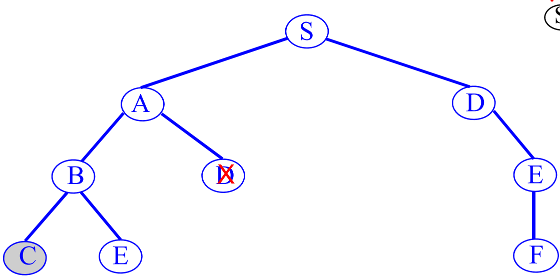
Queue = {~~D~~, E, C, E}

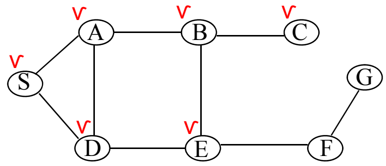
Select E (D already expanded)

Goal(E) = true?

If not, expand(E)

BFS Search Tree





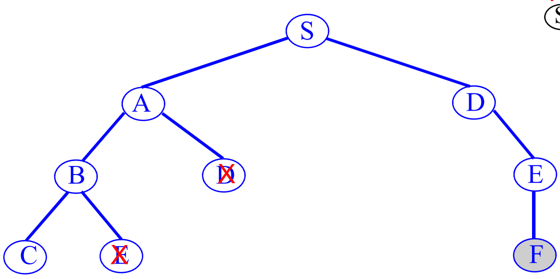
Queue = {C, E, F}

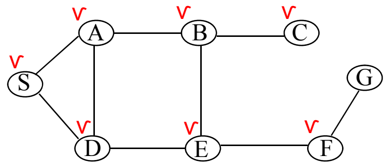
Select C

Goal(C) = true?

If not, expand(C)
(but nothing expanded)

BFS Search Tree





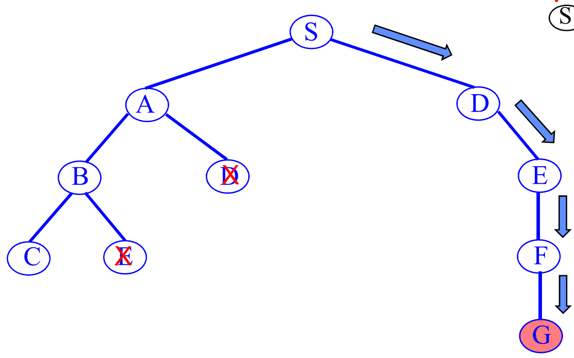
Queue = {~~E~~, F}

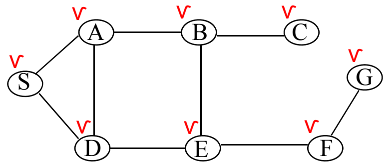
Select F (E already expanded)

Goal(F) = true?

If not, expand(F)
(but nothing expanded)

BFS Search Tree





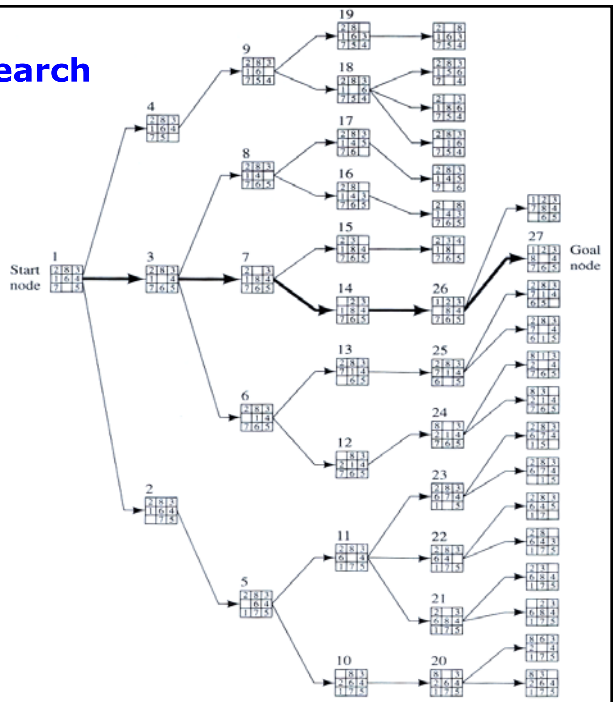
Queue = {G}

Select G

Goal(G) = true?

If yes, finish with goal G

Breadth-First Search



Depth-First Search (DFS)

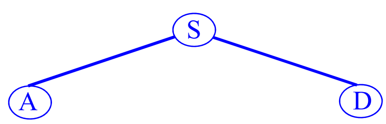
- Expand deepest unexpanded node

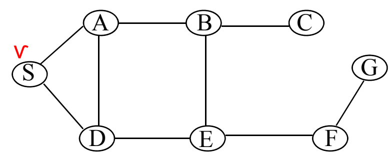
- Implementation:
  - For DFS, *fringe* is a Last-in-first-out (LIFO) stack
  - new successors go at beginning of the stack

- Repeated nodes?
  - Simple strategy: do not add an already-expanded node to the stack
    do not expand an already-expanded node

    BFS와 동일한 strategy

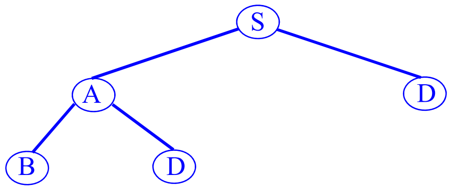
DFS Search Tree

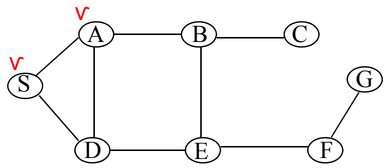




Stack = {A,D}

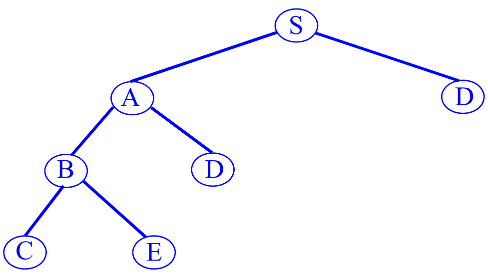
DFS Search Tree

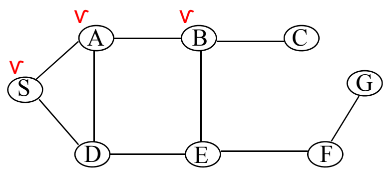




Stack = {B,D,D}

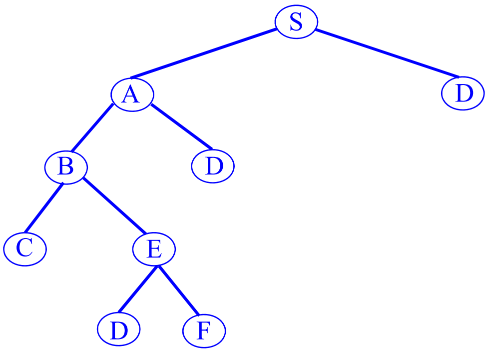
DFS Search Tree

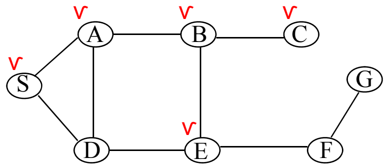




Stack = {C,E,D,D}

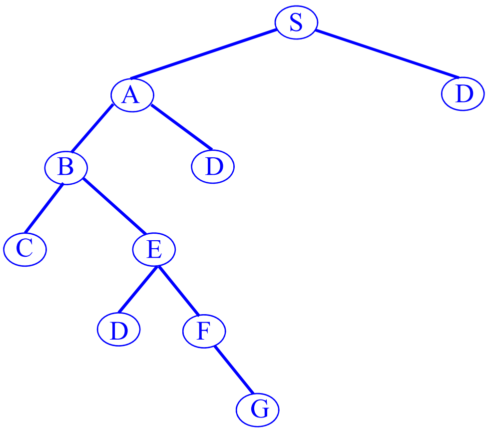
DFS Search Tree

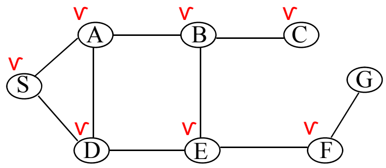




Stack = {D,F,D,D}

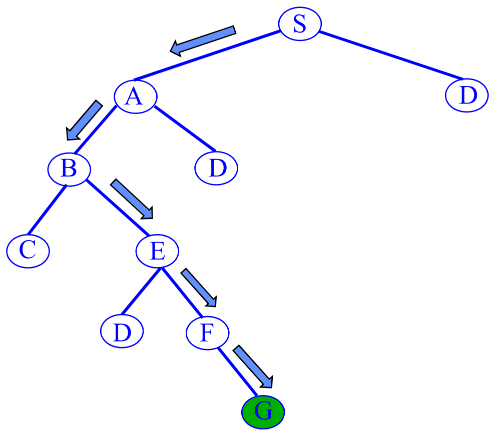
DFS Search Tree

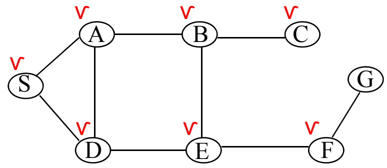




Stack = {G,D,D}

DFS Search Tree





Stack = {G,D,D}

Select G

Goal(G) = true?

If yes, finish with goal G

Evaluation of Search Algorithms

- Completeness
  - does it always find a solution if one exists?
- Optimality
  - does it always find a least-cost (or min depth) solution?
- Time complexity
  - number of nodes generated (worst case)
- Space complexity
  - number of nodes in memory (worst case)

- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)

Breadth-First Search (BFS) Properties

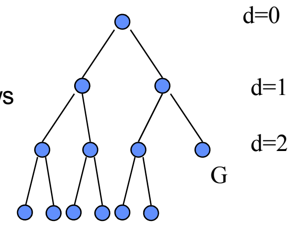
- Complete? Yes → 층별로 모두 탐색하므로
- Optimal? Yes → level order로 가까운 곳부터 찾으므로
- Time complexity $O(b^d)$
- Space complexity $O(b^d)$

- Main practical drawback? exponential space complexity

Complexity of Breadth-First Search

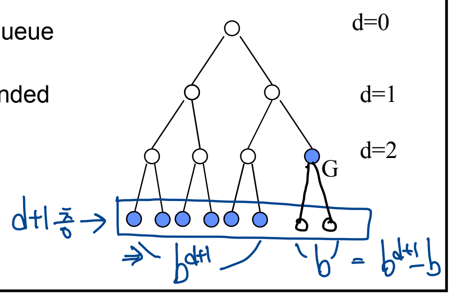
- **Time Complexity**
  - assume (worst case) that there is 1 goal leaf at the RHS at depth d
  - so BFS will generate nodes as follows

$$\boxed{\begin{aligned} &= b + b^2 + \ .....\ + b^d + b^{d+1} - b \\ &= \mathbf{O\ (b^{d+1})} = O\ (b \cdot b^d) = O\ (b^d) \end{aligned}}$$



- **Space Complexity**
  - how many nodes can be in the queue (worst-case)?
  - at depth d there are b^{d+1} unexpanded nodes in the Q as follows

$$\boxed{\begin{aligned} &= b^{d+1} - b \\ &= \mathbf{O\ (b^{d+1})} \end{aligned}}$$



Examples of Time and Memory Requirements for Breadth-First Search

↱ 10이면 간단한 문제

Assuming b=10, speed =10000 nodes/sec, node_size=1kbyte/node

Depth of Solution	Nodes Generated	Time	Memory
2	1100	0.11 seconds	1 MB
4	111,100	11 seconds	106 MB
8	$\approx 10^9$	$\approx$ 31 hours	1 TB
12	$\approx 10^{13}$	$\approx$ 35 years	10 PB

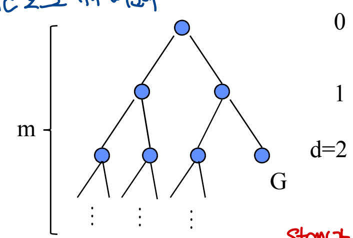
부담　　　더 큰 부담

What is the Complexity of Depth-First Search?

최대 depth까지 모두 들어간 뒤 다른 노드로 가기 때문에

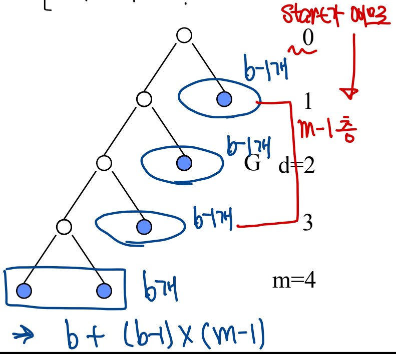
- Time Complexity
  - maximum tree depth = m
  - assume (worst case) that there is 1 goal leaf at the RHS at depth d
  - so DFS will generate $\boxed{O(b^m)}$

    b^m



- Space Complexity
  - how many nodes can be in the queue (worst-case)?
  - at depth m we have b nodes
  - and b-1 nodes at earlier depths
  - total = b + (m-1)*(b-1) = $\boxed{O(bm)}$

  ↳ ✱ Linear 해졌다



Examples of Time and Memory Requirements for Depth-First Search

Assuming b=10, m = 12, speed=10000 nodes/sec, node_size=1kbyte/node

Depth of Solution	Nodes Generated	Time	Memory
2	$\approx 10^{12}$	≈ 3 years	120kb
4	$\approx 10^{12}$	≈ 3 years	120kb
8	$\approx 10^{12}$	≈ 3 years	120kb
12	$\approx 10^{12}$	≈ 3 years	120kb

DFS의 단점 Good point

⇒ But,, BFS나 DFS나 exponential complexity이다

Depth-First Search (DFS) Properties

DFS의 문제점

- Complete? → depth가 ∞이면 노답
  - Not complete if tree has unbounded depth
- Optimal?
  - No ,, ⇒

→ optimal

→ But 이놈 먼저 찾음

- Time complexity?
  - Exponential → BFS와 비슷하거나 좀 더 안좋다
- Space complexity?
  - Linear ⇒ DFS의 장점

Comparing DFS and BFS

- Time complexity: same, $\boxed{\text{but}}$
  - In the worst-case, BFS is generally better than DFS
  - Sometime, on the average DFS is better if:
    - many goals, no loops and no infinite paths

      ↳ 이 경우 DFS가 나을 수 있다
- BFS is much worse memory-wise
  - DFS is $\boxed{\text{linear space}}$
  - BFS may store the order of the whole search space.
- In general ↱ BFS가 더 좋은 경우
  - BFS is better if goal is not deep, if infinite paths, if many loops, if small search space
  - DFS is better if many goals, not many loops, no infinite paths
  - DFS is much better in terms of memory ↳ DFS가 더 좋은 경우

    ↳ But optimality. Completeness 가 보장되지 않는다

↱ Depth Limit을 주는 DFS 방식

DFS with a depth-limit L

- Standard DFS, but tree is not explored below some depth-limit L
- Solves problem of infinitely deep paths with no solutions
  - But will be incomplete if solution is below depth-limit

    ↳ depth-limit 아래 답이 있는 경우만 Solve 가능
- Depth-limit L can be selected based on problem knowledge
  - E.g., diameter of state-space:
    - E.g., max number of steps between 2 cities
  - But typically not known ahead of time in practice

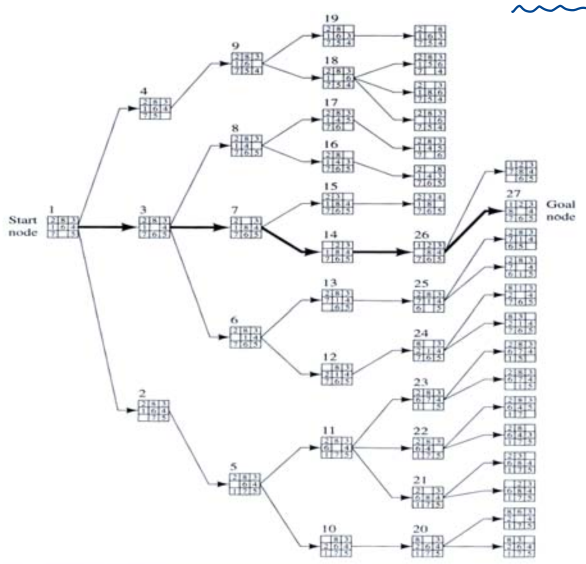
    ↳ DFS의 단점 보완

    But Goal이 몇번째 깊이에 존재하는지 모르면 이 방법을 쓸 수 없다

무한 깊이에 빠지는 것을 막을 수 있다

Limit 사용

Depth-First Search with a depth-limit, L = 5



Iterative Deepening Search (IDS)

- Run multiple DFS searches with increasing depth-limits

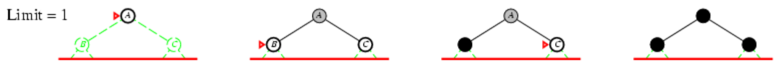
↳ DFS with Depth Limit을 Limit을 늘려가며 반복적으로 수행

Iterative deepening search

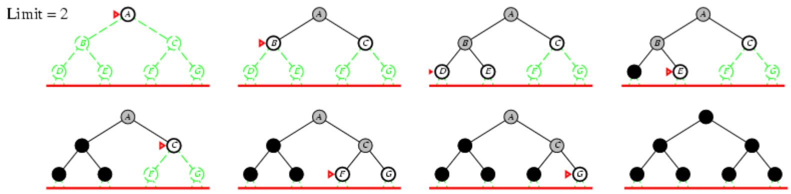
- L = 1
- While no solution, do
  - DFS from initial state S_0 with cutoff L
  - If found goal,
  - stop and return solution,
  - else, increment depth limit L

Iterative deepening search L=0

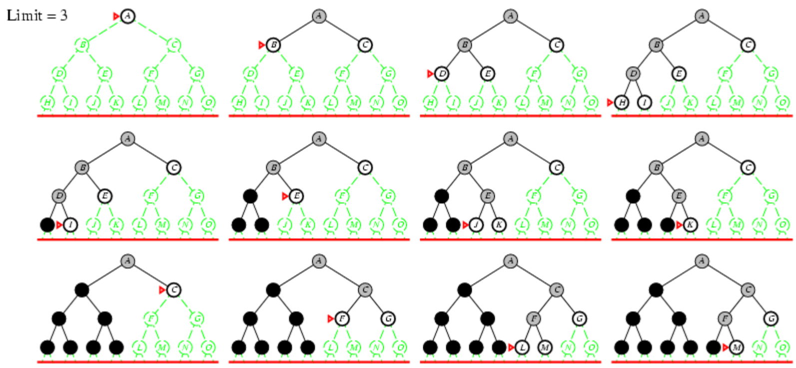
Iterative deepening search L=1



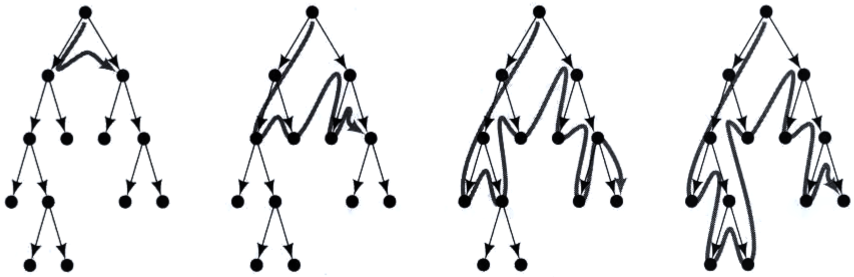
Iterative deepening search L=2



Iterative Deepening Search L=3



Iterative deepening search



Stages in Iterative-Deepening Search

장점 1. Memory Complexity가 linear하다

2. Limit을 늘려가며 DFS를 수행하기 때문에 optimal하다 ⇒ BFS 장점인 Optimality

3. Goal이 존재한다면 반드시 Goal을 찾을 수 있다 ⇒ BFS 장점인 Completeness

Properties of Iterative Deepening Search

DFS에서 O(bm)이었는데 IDS에서는 m=d이므로 O(bd)

- Space complexity = $\boxed{O(bd)}$
  - (since its like depth first search run different times, with maximum depth limit d)

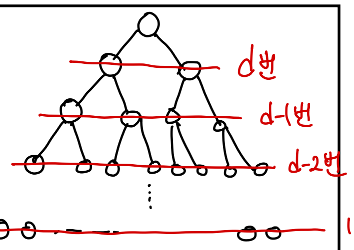
rebundent하게 safe한 node를 반복적으로 접근하는 부분

- Time Complexity L=2
  - b + (b+b^2) + …….($\boxed{b+\ldots}$b^d) = $\boxed{O(b^d)}$ L=d L=1 ⇒ |△| = |△|+|△|+…+|△|
    (i.e., asymptotically the same as BFS or DFS to limited depth d in the worst case)

한번에 다 방문하는 것과 IDS 처럼 조금씩 반복되게 모두 방문하는 것은 O-notation으로 봤을 때 동일하다

- Complete?
  - Yes
- Optimal
  - Yes as long as path cost is a non-decreasing function of depth
- IDS combines the small memory footprint of DFS, and has the completeness guarantee of BFS

IDS in Practice



↱ IDS의 반복되는 탐색의 낭비 정도를 알아보자

- Isn't IDS wasteful?
  - Repeated searches on different iterations

    ↱ IDS와 BFS 비교
  - Compare IDS and BFS: ↱ 0층 1층 2층 기준
    - E.g., b = 10 and d = 5

      ↱ 2층은 d번중 d-1 일때는 방문하지 않으므로 (d-1)
    - N(IDS) ~ $db + (d-1)b^2 + \ldots\ldots b^d = 123{,}450 \approx$ $\boxed{b/b\text{-}1 \text{ times of } N(BFS)}$

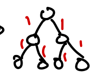
      ↳ depth limit 만큼 중복되게 방문하기 때문에 곱해준다
    - N(BFS) ~ $b + b^2 + \ldots\ldots b^d = 111{,}110$

      L=1의 node 개수 ↳ L=2의 node 개수
    - Difference is only about 11%
      - Most of the time is spent at depth d, which is the same amount of time in both algorithms

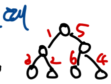
        ↳ 계산량이 생각보다 많이 증가하지는 않는다

- In practice, IDS is the preferred uniform search method with a large search space and unknown solution depth

⇒ 결론 : 계산량이 조금 증가했지만
DFS, BFS의 장점만 가져온 탐색 방법이다



DFS, BFS는 edge의 비용이 모두 동일한 경우 사용했지만
↱ Uniform Cost Search의 경우 edge의 비용이 다를 때
사용할 수 있는 일반화된 BFS 방식이다



Uniform Cost Search

- Optimality: path found = lowest cost 보장한다
  - Algorithms so far are only optimal under restricted circumstances

    ↱ 시작 node
- Let g(n) = cost from start state S to node n → S부터 n까지의 expand 비용

  ↳ n node 함수 g(n)

- Uniform Cost Search:
  - Always expand the node on the fringe with minimum cost g(n)

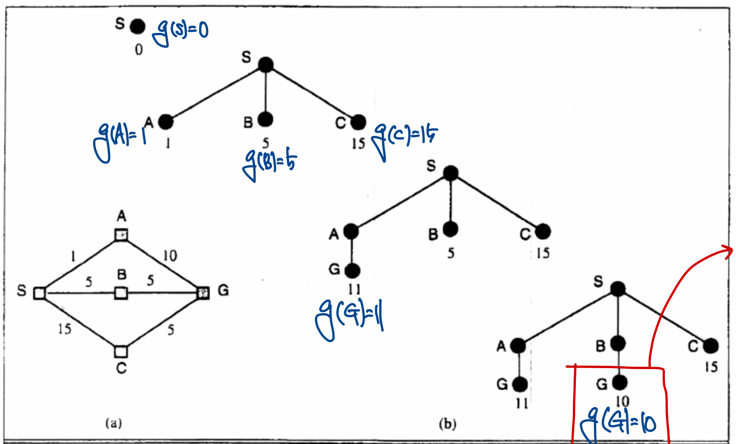
    ↳ g(n)이 최소가 되는 값을 먼저 expand 한다
  - Note that if costs are equal (or almost equal) will behave similarly to BFS

    ↳ Cost가 동일하면 BFS와 동일하게 작동

↳ 문제 성질에따라
최대를 구하면 Max Heap
최소를 구하면 Min Heap 사용

DFS : Stack
BFS : Queue
UCS : Heap (Priority Queue)

Uniform Cost Search



여기서 선택되고 종료

Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

Optimality of Uniform Cost Search?

↱ 모든 Cost가 0보다 큰 경우 보장

- Assume that every step costs at least $\varepsilon > 0$

- Proof of Completeness:

  Given that every step will cost more than 0, and assuming a finite branching factor, there is a finite number of expansions required before the total path cost is equal to the path cost of the goal state. Hence, we will reach it in a finite number of steps.

  ↳ Cost가 모두 양수인 경우 보장되는 내용, Goal이 존재하면 유한 시간안에 찾을 수 있다는 내용

- Proof of Optimality given Completeness:
  - Assume UCS is not optimal.
  - Then, there must be a goal state with path cost smaller than the goal state which was found (invoking completeness)
  - However, this is impossible because UCS would have expanded that node first by definition.
  - Contradiction.

  ↳ Optimality 증명

Complexity of Uniform Cost

- Let C* be the cost of the optimal solution
- Assume that every step costs at least $\varepsilon > 0$

이렇게 바뀜

- Worst-case time and space complexity is:

전체 Path 비용

$$\boxed{O(\ b^{\,[1 + \mathrm{floor}(C^*/\varepsilon)]}\ )}$$

1개 edge 비용

$O(b^d)$

Why?

floor(C*/ε) ~ depth of solution if all costs are approximately equal

Comparison of Uninformed Search Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	O Yes	O Yes	X No	X No	O Yes
~~Time~~	$\cancel{O(b^{d+1})}$	$\cancel{O(b^{\lceil C^*/\epsilon \rceil})}$	$\cancel{O(b^m)}$	$\cancel{O(b^l)}$	$\cancel{O(b^d)}$
Space	X $O(b^{d+1})$	X $O(b^{\lceil C^*/\epsilon \rceil})$	O $O(bm)$	O $O(bl)$	O $O(bd)$
Optimal?	O Yes	O Yes	X No	X No	O Yes

→ time은 모두 같다고 볼 수 있다

Summary

- A review of search
  - a search space consists of states and operators: it is a graph
  - a search tree represents a particular exploration of search space
- There are various strategies for "uninformed search"
  - breadth-first
  - depth-first　　모두 장단점이 있다
  - iterative deepening　→ DFS 장점 + BFS 장점
  - Uniform cost search　→ 일반적인 BFS
- Various trade-offs among these algorithms
  - "best" algorithm will depend on the nature of the search problem
- Next up – heuristic search methods