Day4 – 과제 2

20기 예비 김도현

목차

BFS & DFS

프림

크루스칼

다익스트라

벨만-포드

A*

알고리즘

입력을 출력으로 변환하는 계산 절차

정렬 (sorting algorithm)

목록의 요소들을 기준에 맞게 재배열하는 알고리즘.

정렬된 데이터를 요구하는 알고리즘을 위해 사용.

요소의 탐색 효율 증가.

정렬된 목록을 요구하는 알고리즘을 위한 전 처리에 활용.

선택 정렬 (selection sort)

아직 정렬되지 않은 요소 중 기준에 부합하는 요소를 정렬되지 않은 요소 중 첫 번째 요소와 교환하는 과정을 반복한다.

Time complexity:

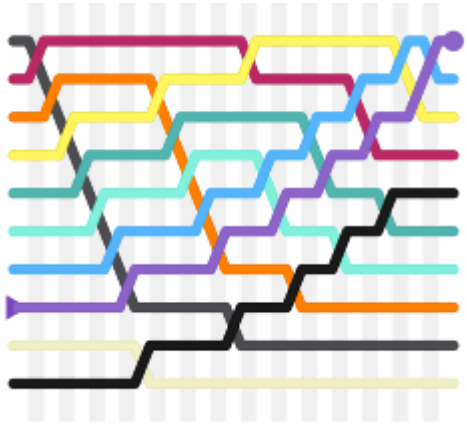
Worst case: $O(n^2)$

Best case: $O(n^2)$

Space complexity: $O(1)$

비효율적인 time complexity를 가지지만 메모리 효율성이 높음.

버블 정렬 (bubble sort)



두 요소의 순서를 비교하여 올바른 순서로 정렬하는 과정을 모든 요소가 정렬될 때까지 반복한다.

Time complexity:

Worst case: $O(n^2)$

Best case: $O(n)$

Space complexity: $O(n) + O(1)$

삽입 정렬 (insertion sort)

아직 정렬되지 않은 요소를 현재까지 정렬된 목록의 올바른 위치에 삽입한다.

Time complexity:

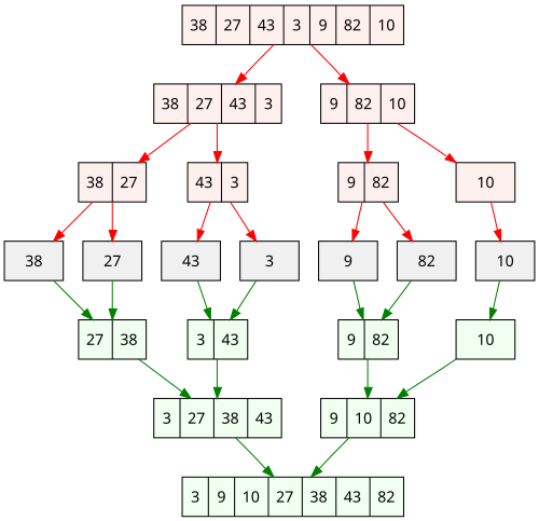
Worst case: $O(n^2)$

Best case: $O(n)$

Space complexity: $O(n) + O(1)$

선택 정렬이나 버블 정렬에 비해 효율적임.

병합 정렬 (merge sort)



목록을 하위 목록으로 분할한 후 각 목록을 정렬하여 병합한다.

Time complexity:

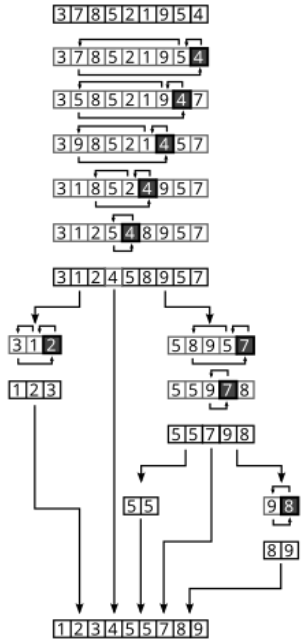
Worst case: $O(n \log n)$

Best case: $\Omega(n \log n)$

Space complexity: $O(n)$

효율성과 범용성이 높다.

퀵 정렬 (quick sort)



목록의 요소 중 하나를 pivot으로 지정하고 pivot을 기준으로 분류한다.

분류된 목록에서 새로운 pivot을 지정하고 정렬을 반복한다.

Time complexity:

Worst case: $O(n \log n)$

Best case: $O(n \log n)$

Space complexity: $O(n) + O(log\, n)$

넓게 분산된 목록을 효율적으로 정렬할 수 있다.

쉘 정렬 (shell sort)

목록을 분할하여 insertion sort를 실행하고 정렬된 목록을 다시 분할하는 과정을 각 목록별로 하나의 요소만을 포함하게 될 때까지 반복한다.

Time complexity:

Worst case: $O(n^2)$

Best case: $O(n \log n)$

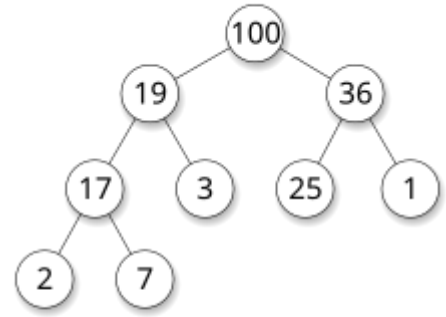
Space complexity: $O(n) + O(1)$

정렬시작 시 목록이 정렬되어 있을수록 효율적이다.

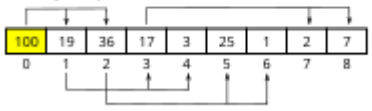
힙 정렬 (heap sort)

목록을 heap으로 변환하고 순서에 맞는 요소를 추출하는 과정을 반복한다.

Tree representation



Array representation



Heap은 tree 자료형의 일종으로 모든 parent node와 child node의 대소관계가 동일하다.

parent node가 더 크거나 같을 경우에는 max-heap 작거나 같을 경우에는 min-heap이다.

Time complexity:

Worst case: $O(n \log n)$

Best case: $O(n \log n)$

Space complexity: $O(n) + O(1)$

안정적이고 메모리 활용성과 효율적인 time complexity를 가진다.

탐색 (search algorithm)

목록에서 원하는 요소를 탐색하는 알고리즘

Linear search

목록의 시작부터 순차적으로 탐색하며 탐색중인 요소를 발견하거나 모든 요소를 탐색할 때까지 반복한다.

목록이 정렬되어 있을 경우 목표 값이 탐색 범위에서 벗어났을 경우 중간에 탐색을 중지할 수 있다.

Time complexity:

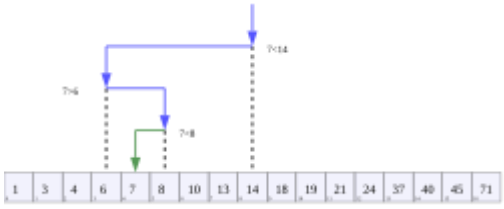
Worst case: $O(n)$

Best case: $O(1)$

Space complexity: $O(1)$

Binary search

정렬된 목록에서 사용할 수 있다.



배열의 중간과 목표 값을 비교하고 범위를 재조정한다.

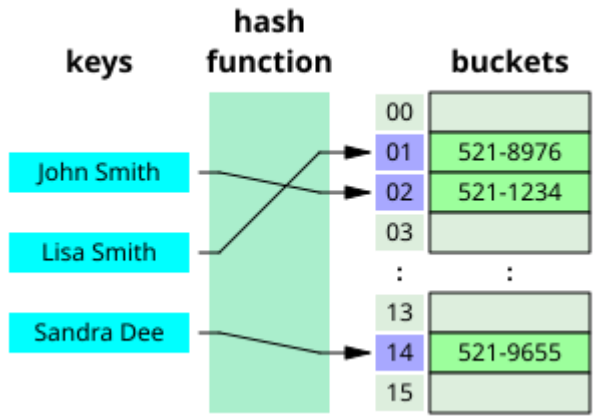
Time complexity:

Worst case: $O(\log n)$

Best case: $O(1)$

Space complexity: $O(1)$

Hash table



각 데이터에 key를 mapping한다.

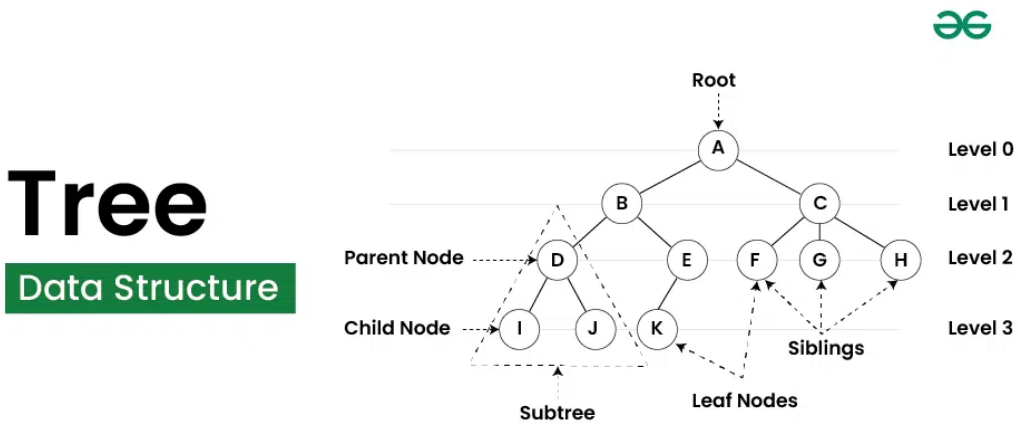
Time complexity:

Worst case: $O(1)$

Best case: $O(1)$

Space complexity: $O(n)$

트리 (tree)



계층적 자료구조

Node: 각 요소를 저장하는 단위.

Parent node ⇔ Child node: 상위/하위 node

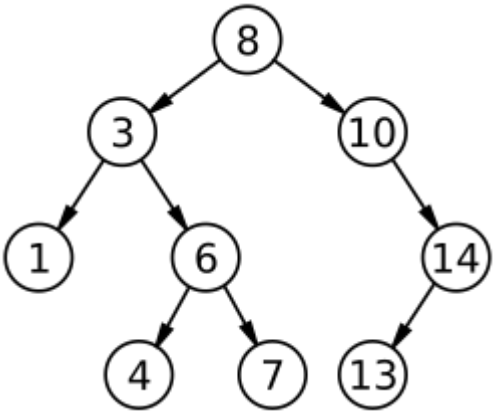
Root: tree의 가장 위에 위치하는 node.

Leaf: tree의 가장 아래에 위치하며 children node를 가지지 않는다.

Height: 해당 node로부터 가장 먼 leaf node와의 거리.

Depth: root node로부터의 거리

이진탐색 트리 (BST: Binary Search Tree)



각 node는 최대 두개의 child node를 가지며 해당 node와 비교하여 작으면 왼쪽, 크면 오른쪽 node로 넘기며 leaf에 도달할 때까지 반복한다.

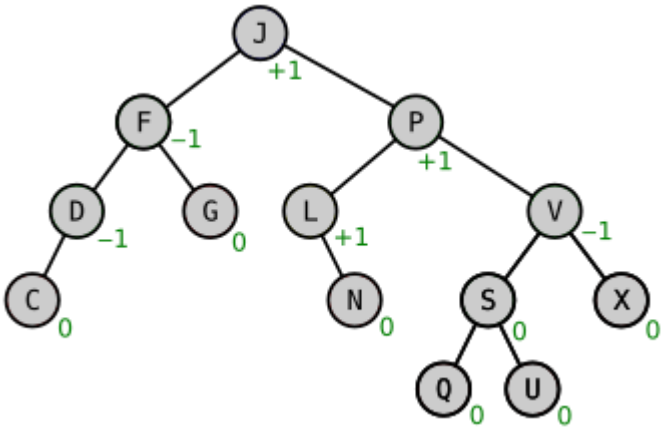
Time complexity:

Worst case: $O(log\ n)$

Best case: $O(n)$

Space complexity: $O(n)$

AVL 트리



모든 node에서 오른쪽 subtree와 왼쪽 subtree의 height차가 1이하가 되도록 정렬한 tree.

Time complexity:

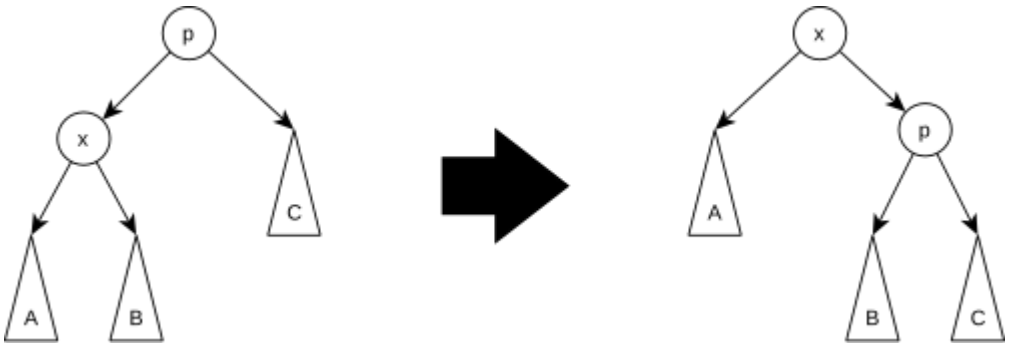
Worst case: $O(log\ n)$

Best case: $O(log\ n)$

Space complexity: $O(n)$

Best case와 worst case의 편차를 최소화한다.

스플레이 트리 (splay tree)



최근 접근한 node가 root node로 재배치하여 자주 접근하는 node에 빨리 접근할 수 있도록 한다.

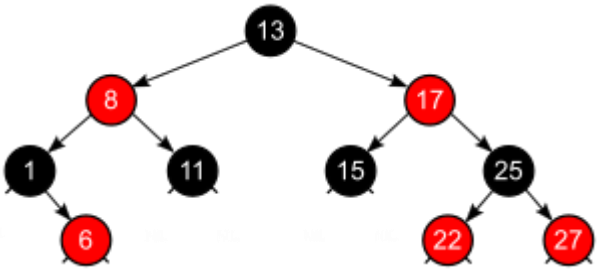
Time complexity:

Worst case: $O(log\ n)$

Best case: $O(log\ n)$

Space complexity: $O(n)$

레드블랙 트리 (red-black tree)



각 node를 red node와 black node로 구분하고 각 node의 subtree들이 동일한 black height(leaf node까지의 black node의 개수)를 가지도록 정렬한다.

Red node의 child는 black node여야 한다.

Time complexity:

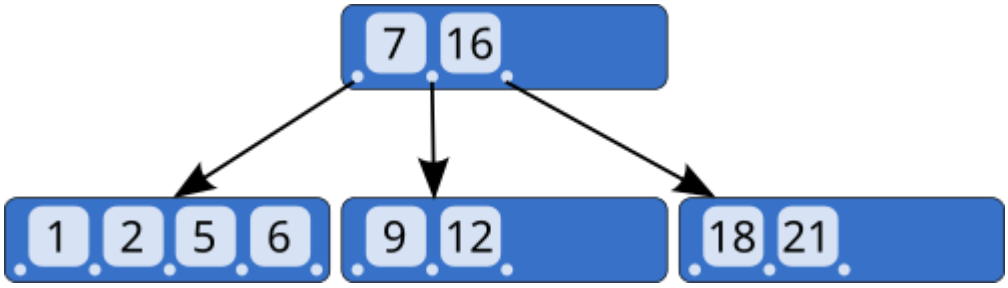
Worst case: $O(log\ n)$

Best case: $O(log\ n)$

Space complexity: $O(n)$

Best case는 black node만 반복되고, worst case에서는 red node와 black node가 번갈아 가며 반복되어 best case에 비해 두배의 탐색길이를 가진다.

B트리 (B-tree)



하나의 node가 두 개 이상의 child node를 가질 수 있다.

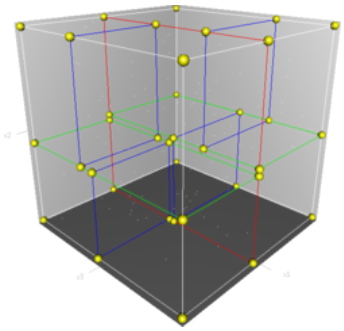
Time complexity:

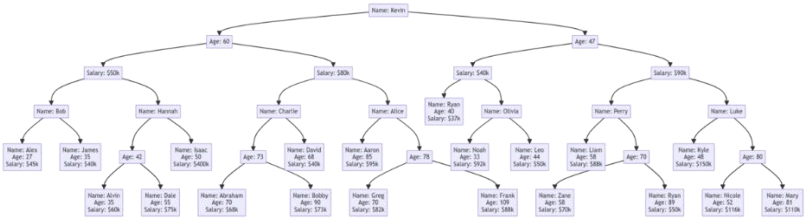
Worst case: $O(log\ n)$

Best case: $O(log\ n)$

Space complexity: $O(n)$

KD트리 (k-d tree)





k-dimensional space로 구성된 tree로 다차원상으로 인접한 node를 탐색한다.

Point cloud 등에 활용된다.

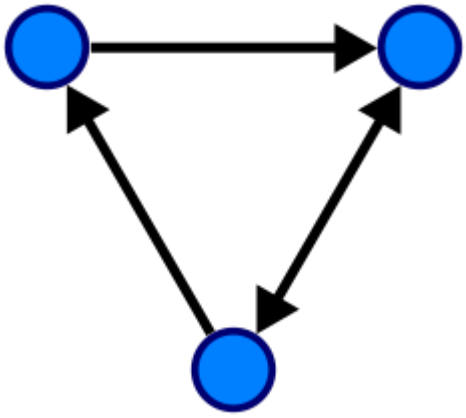
Time complexity:

Worst case: $O(log\ n)$

Best case: $O(log\ n)$

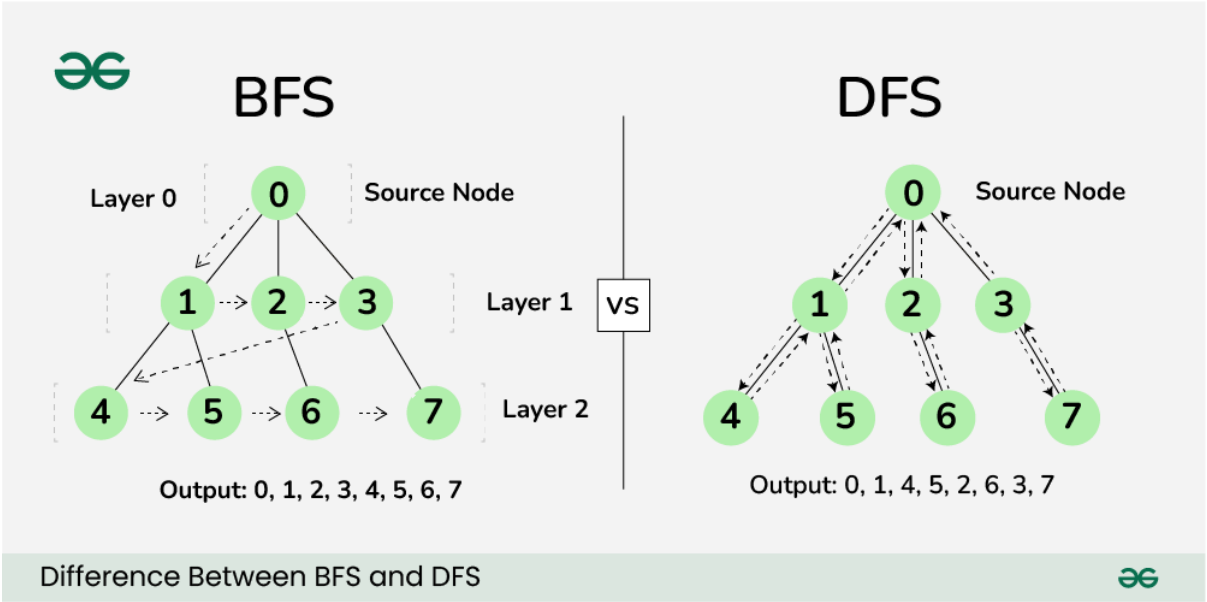
Space complexity: $O(n)$

그래프



Vertex와 edge로 구성되며 경로 탐색과 병렬처리 등에 사용된다.

BFS (breadth-first search) & DFS (depth-first search)



동작 방식:

BFS: 같은 depth를 가지는 node로 이동하며 같은 depth의 모든 node 방문 이후 하위 depth의 node로 이동.

DFS: Child node로 이동하며 subtree의 모든 node 방문 이후 parent node로 이동

자료 구조

BFS: Queue – FIFO

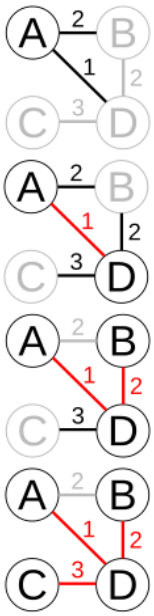
DFS: Stack – LIFO

BFS는 root와 가까이 위치한 node를 탐색하기 적합하며 최단거리 알고리즘 등에 활용된다.

DFS는 깊은 depth의 node를 탐색하기 적합하며 경로탐색 알고리즘 등에 활용된다.

프림 (Prim's algorithm)

Weighted undirected graph에서 minimum spanning tree(모든 vertex를 포함하는 edges의 집합)를 탐색하는 greedy algorithm이다.

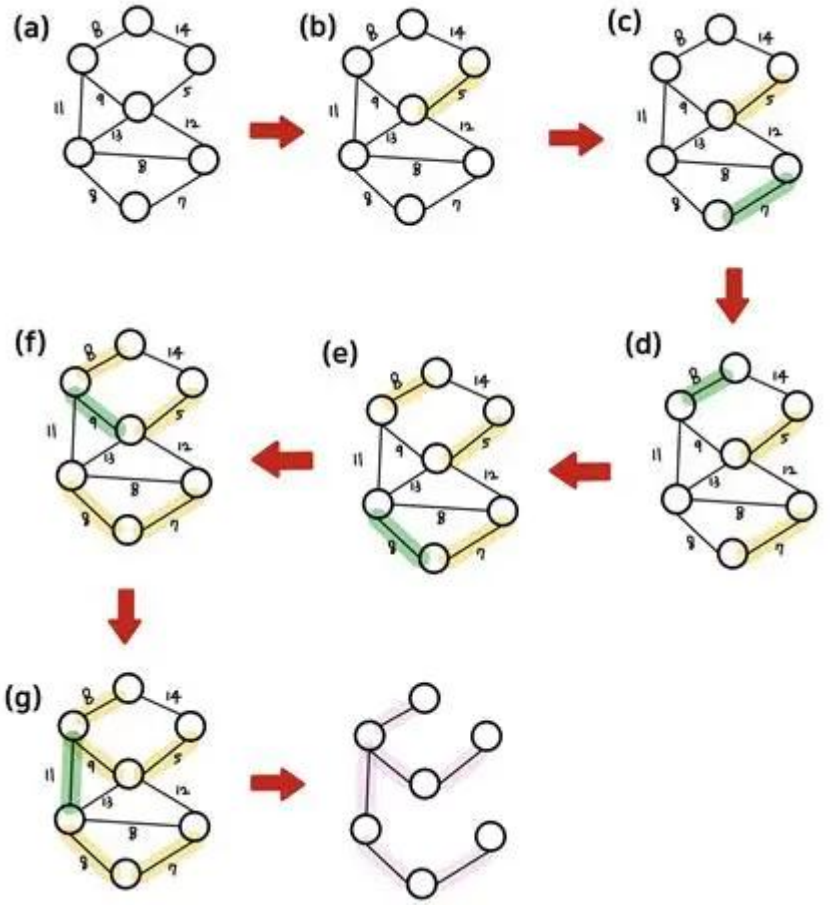


각 단계에서 가장 적은 비용의 경로를 연결한다.

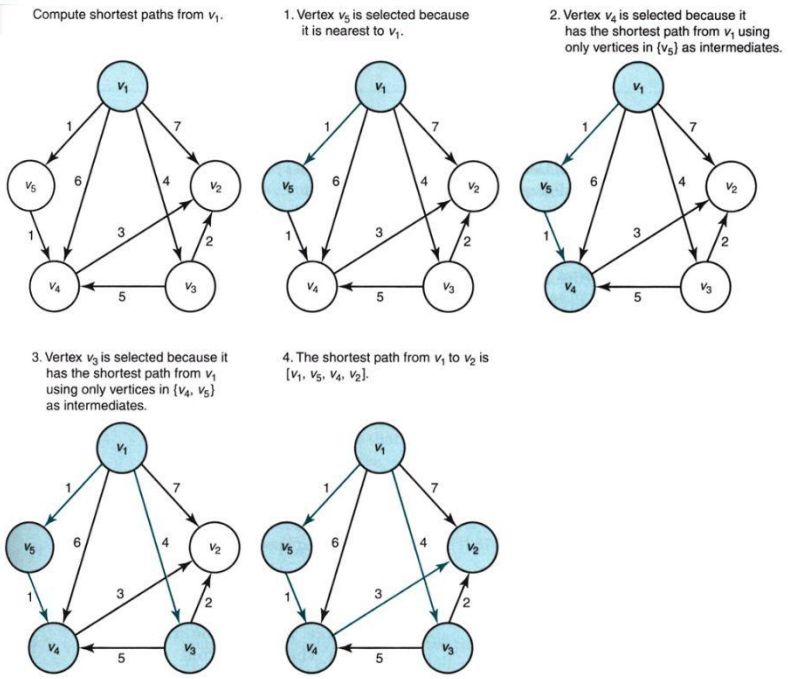
크루스칼 (Kruskal's algorithm)

Weighted undirected graph에서 minimum spanning tree를 탐색하는 greedy algorithm이다.

가장 낮은 weight의 edge부터 cycle을 생성하지 않도록 연결한다.



다익스트라 (Dijkstra's algorithm)



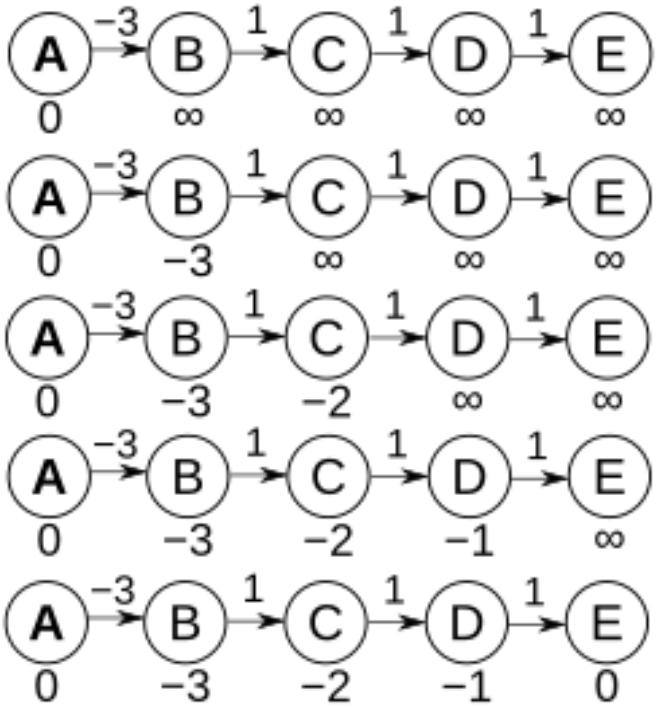
Weighted graph에서 최단경로를 계산

1. 출발 node를 기준으로 인접한 노드의 최소비용을 저장.
2. 방문하지 않은 node 중 가장 적은 비용의 node로 이동
3. 인접 node의 비용 갱신
4. 종료지점에 도달할 때까지 반복

벨만-포드(Bellman–Ford algorithm)

최단 경로 알고리즘.

모든 vertex의 거리를 무한으로 설정하고 각 edge를 탐색하며 거리를 업데이트한다.



양의 가중치만을 허용하는 Dijkstra's algorithm와는 달리 음의 가중치를 설정할 수 있다.

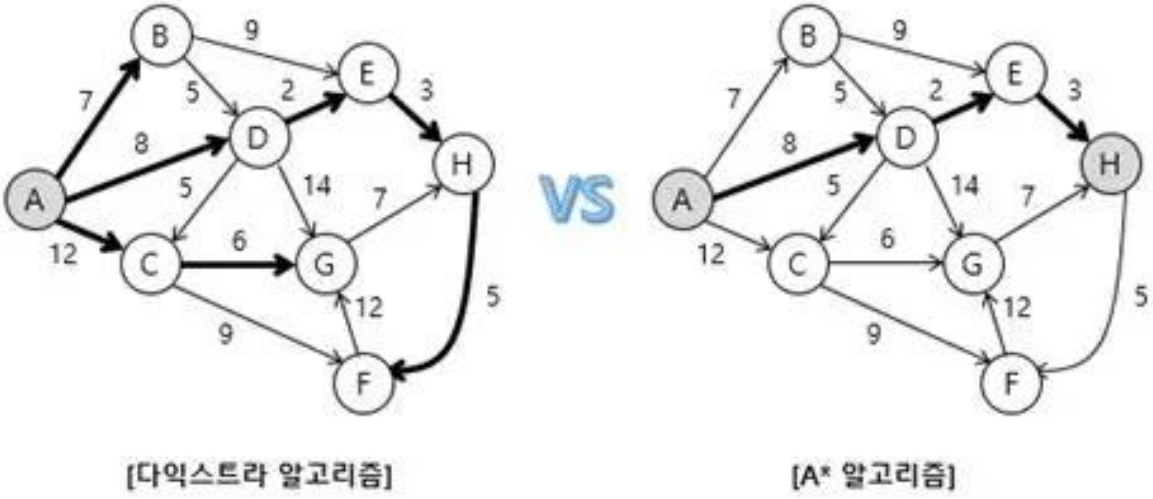
A*

Dijkstra's algorithm을 개선한 알고리즘.

다음으로 이동할 노드를 선택할 때, 목적지까지의 예상거리를 반영하여 결정한다.

1. 경로의 시작지점으로부터 출발
2. 해당 칸에 인접한 칸들의 경로의 시작지점으로부터의 거리 (g) 와 종료지점까지의 직선거리 (h)를 더한 예상거리 (f) 계산
3. 인접한 칸들의 기존의 예상 거리가 존재하지 않거나 새로운 예상거리가 기존의 예상 거리보다 짧을 때 해당 칸의 g를 바탕으로 주변 칸들의 거리 업데이트
4. 해당 칸의 거리를 결정짓고 나머지 칸들 중 예상 거리가 가장 짧은 칸으로 이동

5. 종료지점에 도달할 때까지 반복



[다익스트라 알고리즘] [A* 알고리즘]

Dijkstra's algorithm 에 비해 적은 비용으로 최단 경로를 도출할 수 있다.

출처

https://en.wikipedia.org/wiki/Selection_sort

https://en.wikipedia.org/wiki/Bubble_sort

https://en.wikipedia.org/wiki/Insertion_sort

https://en.wikipedia.org/wiki/Merge_sort

https://en.wikipedia.org/wiki/Quicksort

https://en.wikipedia.org/wiki/Shellsort

https://en.wikipedia.org/wiki/Heap_(data_structure)

https://en.wikipedia.org/wiki/Heapsort

https://en.wikipedia.org/wiki/Linear_search

https://en.wikipedia.org/wiki/Binary_search

https://en.wikipedia.org/wiki/Hash_table

https://en.wikipedia.org/wiki/Tree_(abstract_data_type)

https://en.wikipedia.org/wiki/Binary_search_tree

https://en.wikipedia.org/wiki/AVL_tree

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

https://en.wikipedia.org/wiki/B-tree

https://en.wikipedia.org/wiki/K-d_tree

https://en.wikipedia.org/wiki/Graph_(abstract_data_type)

https://www.geeksforgeeks.org/dsa/difference-between-bfs-and-dfs/

https://en.wikipedia.org/wiki/Prim%27s_algorithm

https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm