Lecture 13: Graphs I: Breadth First Search

Lecture Overview

- Applications of Graph Search
- Graph Representations
- Breadth-First Search

그래프 탐색 알고리즘:

Depth Frist Search: 깊이 우선 탐색 알고리즘

BFS는 Breadth First Search의 약자로, 너비 우선탐색 알고리즘 , 체계적인 방식으로 그래프나 트리를 순화하거나 검색하는데 사용하는 알고리즘이다.

그래프는 여러 개체들이 연결되어 있는 자료 구조인데 탐색과정에서 특정 개체를 찾기 위한 알고르즘

1. DFS: 하나를 몰아본다
2. BFS: 어떤것글이 연속해서 이어질 때, 모두 확인하는 방법, 여러개를 하나씩 본다 (맹목적인 탐색을 할 때 사용할 수 있는 탐색기법, 최단 경로를 찾아 주며, 시작 노드에 인접한 노드부터 탐색하는 알고리즘.

Graph: Vertex (어떤 것) + Edge (이어지는 것)

Graph $G = (V, E)$

- V = set of vertices (arbitrary labels)
- E = set of edges i.e. vertex pairs (v, w)
  - ordered pair ⇒ directed edge of graph
  - unordered pair ⇒ undirected

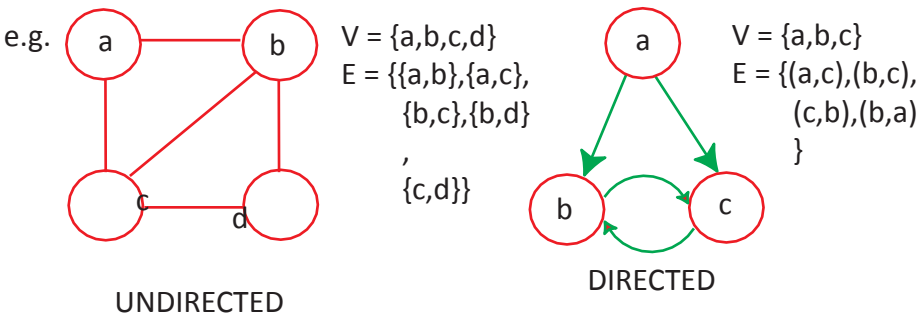


Figure 1: Example to illustrate graph terminology

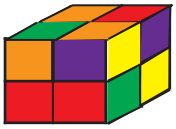
응용:

여러 응용 사례들이 있습니다.

- 웹 크롤링 (구글이 페이지를 찾는 방법)
- 소셜 네트워킹 (페이스북이 친구 찾기를 사용하는 법)
- 네트워크 브로드캐스트 라우팅
- 가비지 컬렉션
- 모델 검사 (무한 상태 기계)
- 수학적 추측 확인하기
- 퍼즐이나 게임 풀기

포켓 큐브:

2 x 2 x 2 루빅 큐브



Configuration Graph:

- vertex for each possible state
  - edge for each basic move (e.g., 90 degree turn) from one state to another
- undirected: moves are reversible

Diameter ("God's Number")

11 for $2 \times 2 \times 2$, 20 for $3 \times 3 \times 3$, $\Theta(n^2/\lg n)$ for $n \times n \times n$ [Demaine, Demaine, Eisenstat Lubiw Winslow 2011]

- 표기법 Θ는 주어진 입력 크기에 대한 최상의 경우와 최악의 시나리오 측면에서 시간 복잡도를 나타냅니다. 이 경우 Θ(n2/lg n)은 시간 복잡도가 n2/lg n에 비례하는 함수에 의해 위와 아래 모두 제한됨을 의미합니다.

- n2/lg n이라는 용어는 알고리즘이나 문제가 해결되는 방식에서 파생됩니다. 입력 크기 n의 제곱을 n의 로그로 나눈 값만큼 알고리즘을 완료하거나 문제를 해결하는 데 필요한 시간이 증가함을 나타냅니다.

- 전반적으로 이 코드는 특정 입력 크기에 대한 특정 알고리즘 또는 문제의 시간 복잡도에 대한 정보를 제공하므로 다른 알고리즘의 효율성을 분석 및 비교하거나 다른 방법으로 동일한 문제를 해결하는 데 유용합니다.

Graph Representations: (data structures)

Adjacency lists:

Array Adj of $|V|$ linked lists

- for each vertex $u \in V$, $Adj[u]$ stores u's neighbors, i.e., $\{v \in V \mid (u, v) \in E\}$ (u, v) are just outgoing edges if directed. (See Fig. 2 for an example.)

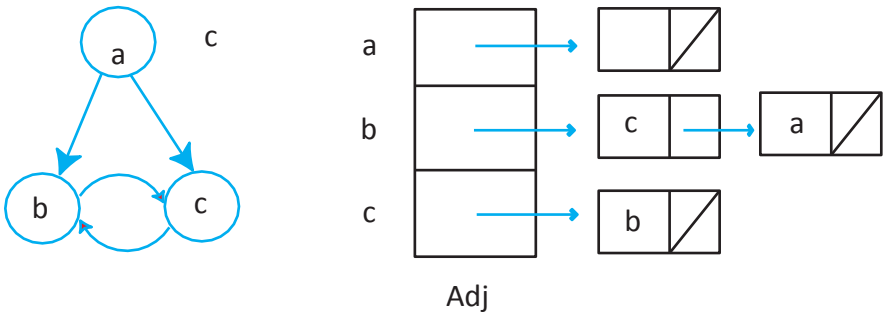


Figure 2: Adjacency List Representation: Space $\Theta(V + E)$

- in Python: Adj = dictionary of list/set values; vertex = any hashable object (e.g., int, tuple)

- advantage: multiple graphs on same vertices

Implicit Graphs:

Adj(u) is a function — compute local structure on the fly (e.g., Rubik's Cube). This requires "Zero" Space.

Breadth-First Search

Explore graph level by level from s

- level 0 = {s}
- level i = vertices reachable by path of i edges but not fewer

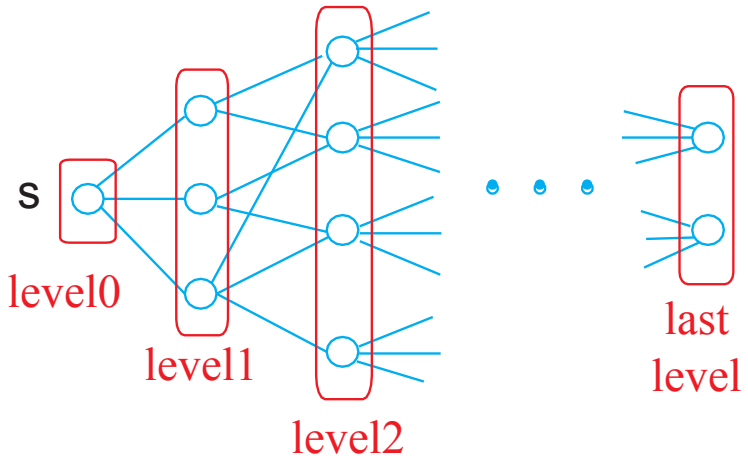


Figure 3: Illustrating Breadth-First Search

- build level $i > 0$ from level $i - 1$ by trying all outgoing edges, but ignoring vertices from previous levels.

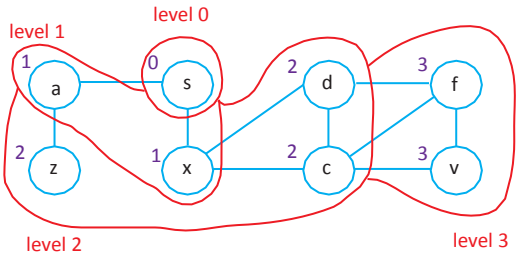
https://www.youtube.com/watch?v=CJiF-muKz30&list=PLVsNizTWUw7H9_of5YCB0FmsSc-K44y81&index=19

Breadth-First-Search Algorithm

```
BFS (V,Adj,s):                                  See CLRS for queue-based
    level = { s: 0 }                            implementation
    parent = {s : None }
    i = 1
    frontier = [s]                               # previous level, i − 1
    while frontier:
        next = [ ]                               # next level, i
        for u in frontier:
            for v in Adj
            [u]:                                 # not yet seen
                if v not in level:               level[v] = level[u] +
                    level[v] = i                 1
                    parent[v] = u
                    next.append(v
                    )
        frontier = next
        i += 1
```

- The algorithm takes three inputs: a set of vertices V, an adjacency list Adj, and a starting vertex s.
- The algorithm works by maintaining two data structures - a level dictionary and a parent dictionary. The level dictionary keeps track of the level of each vertex in the graph or tree, and the parent dictionary keeps track of the parent of each vertex.
- The algorithm starts by setting the level of the starting vertex s to 0, and its parent to none. Then it initializes a variable i to 1, and creates a list called frontier containing only the starting vertex s.
- The algorithm then enters a loop that continues until the frontier list is empty. Within the loop, the algorithm creates an empty list called next. For each vertex u in the frontier list, the algorithm examines all of its neighbors v using the adjacency list Adj[u].
- For each neighbor v, if it has not yet been assigned a level, the algorithm sets its level to i and its parent to u, and appends it to the next list.
- Once all the neighbors of all vertices in the frontier list have been examined, the frontier list is updated to be equal to the next list, and i is incremented by 1. The algorithm then repeats the process of examining all the neighbors of the vertices in the new frontier list until all vertices in the graph or tree have been visited.
- At the end of the algorithm, the level dictionary and parent dictionary contain the level and parent of each vertex in the graph or tree with respect to the starting vertex s.

Example



$frontier_0 = \{s\}$
$frontier_1 = \{a, x\}$
$frontier_2 = \{z, d, c\}$
$frontier_3 = \{f, v\}$
(not x, c, d)

Figure 4: Breadth-First Search Frontier

Analysis:

- vertex v enters next (& then frontier) only once (because level[v] then set)

  base case: $v = s$

  6

- $\Rightarrow$ Adj[v] looped through only once

$$\text{time} = \sum_{v \in V} |Adj[v]| = \begin{cases} |E| \text{ for directed graphs} \\ 2|E| \text{ for undirected graphs} \end{cases}$$

- $\Rightarrow$ $O(E)$ time

- $O(V + E)$ ("LINEAR TIME") to also list vertices unreachable from v (those still not assigned level)

Shortest Paths:

cf. L15-18

- for every vertex v, fewest edges to get from s to v is

$$\begin{cases} \text{level}[v] & \text{if } v \text{ assigned level} \\ \infty & \text{else (no path)} \end{cases}$$

- parent pointers form shortest-path tree = union of such a shortest path for each v
  $\Rightarrow$ to find shortest path, take v, parent[v], parent[parent[v]], etc., until s (or None)

Here is the simple implementation of BFS:

- Create a queue and enqueue the starting vertex
- Mark the starting vertex as visited
- While the queue is not empty, dequeue a vertex and process it
- For each unvisited neighbor of the processed vertex, mark it as visited, enqueue it, and mark its parent as the processed vertex
- Repeat steps 3-4 until the queue is empty

- At the end of the algorithm, you would have visited all the vertices in the graph reachable from the starting vertex, and you would have recorded their parent-child relationships as well.

- BFS is like exploring a maze. Imagine you're in a maze and you want to find your way to the exit. You start at a specific point and you want to explore the maze level by level.

- You explore all the paths from your starting point that are one step away. Then you explore all the paths that are two steps away, and so on. You keep track of the distance from your starting point to each point you visit, and you also keep track of which point you came from to get to each point.

- That's essentially what BFS does - it starts at a specific point in a graph and explores all the vertices that are adjacent to it. Then it explores all the vertices that are adjacent to those vertices, and so on. It keeps track of the distance from the starting point to each vertex, and it also keeps track of which vertex was visited first to reach each vertex.

- By exploring the graph in this way, BFS can find the shortest path from the starting point to any other vertex in the graph.