CS 106X, Lecture 19
Trees

reading:

Programming Abstractions in C++, Chapters 16.1-16.4

Plan For Today

- Trees
- Announcements
- Binary Search Trees
  - Traversing
  - Adding
  - Removing

小结：

1. tree = ptr + recursion综合
2. 所有tree的问题：都与traverse order有关（思考的角度方向）
3. 不同节点的分类cases：nullptr（base case）、leaf（2边都没children）、“single” internal vertex（只有1边有child，出现最大最小值）、internal vertex（2边都有child）
4. tree branch arrow：可看作是一个指针
5. remove节点操作：分类：single vertex（调整arrow指向的终点）；internal vertex：replace it with right_min / left_max

第2点的补充：可以参考browser2的inrange函数如何从树种取出从小到大的元素，inorder traverse

6. 在heap中创建数据结构初始化的时候，即使是empty了记得也要加上nullptr防止dangling ptr；删除清空的时候也是如此，删完的指针设置成nullptr
7. Trie: 每条指针代表一个字母，节点的位置决定一个单词（参考leetcode一道模版题）
8. 平衡树的构建：hw6的displaying text：https://www.baeldung.com/cs/balanced-bst-from-sorted-list

CS 106X, Lecture 22
Graphs; BFS; DFS

图的总结参考class_code文件夹中的Graph Design.md文件
要点：
1. dfs，bfs：optimal, retrival(path record)
2. Dijkstra and A*(heuristics)
3. topological sort: dependency, Kahan’s algorithm
4. MST: Krukal algorithm: assignment 2: maze generator
5. Modeling problems using graph and tree

reading:

Programming Abstractions in C++, Chapter 18

DFS that finds path

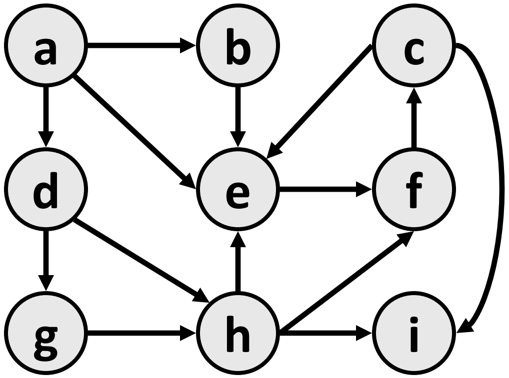
dfs from v_1 to v_2:

mark v_1 as **visited**, and **add to path**. 选对象

perform a **dfs** from each of v_1's unvisited neighbors n to v_2:

if **dfs**(n, v_2) succeeds: a path is found! yay!

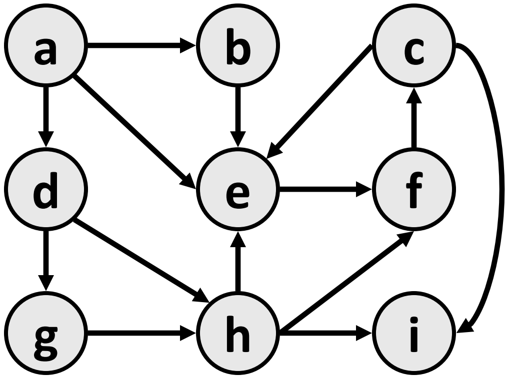
if all neighbors fail: **remove v_1 from path**.



- To retrieve the DFS path found, pass a collection parameter to each call and choose-explore-unchoose.

DFS observations

- *discovery*: DFS is guaranteed to find <u>*a*</u> path if one exists.



- *retrieval*: It is easy to retrieve exactly what the path is (the sequence of edges taken) if we find it
  - choose - explore - unchoose

- *optimality*: not optimal. DFS is guaranteed to find <u>a</u> path, not necessarily the best/shortest path
  - Example: dfs(a, i) returns {a, b, e, f, c, i} rather than {a, d, h, i}.

Breadth-First Search (BFS)

- Keep a Queue of nodes as our TODO list
- Idea: dequeue a node, enqueue all its neighbors
- Still will return the same nodes as reachable, just might have shorter paths

BFS Details

- In an n-node, m-edge graph, takes O(m + n) time with an adjacency list
  - Visit each edge once, visit each node at most once

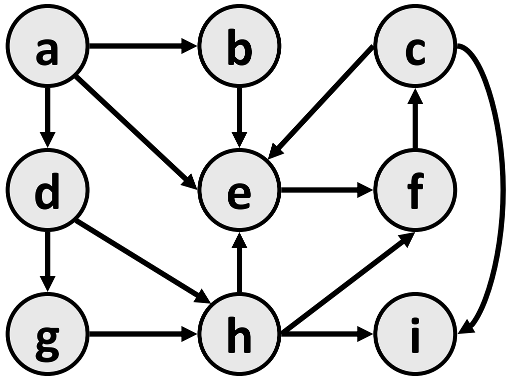
bfs from v_1 to v_2:
    create a *queue* of vertexes to visit,
        initially storing just v_1.
    mark v_1 as **visited**.

    while *queue* is not empty and v_2 is not seen:
        dequeue a vertex v from it,
        mark that vertex v as **visited**,
        and add each unvisited neighbor ***n*** of v to the *queue*.

- How could we modify the pseudocode to look for a specific path?

BFS observations

- *optimality*:
  - always finds the shortest path (fewest edges).
  - in unweighted graphs, finds optimal cost path.
  - In weighted graphs, *not* always optimal cost.



- *retrieval*: harder to reconstruct the actual sequence of vertices or edges in the path once you find it
  - conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a path array/list in progress
  - solution: We can keep track of the path by storing predecessors for each vertex (each vertex can store a reference to a *previous* vertex).

- DFS uses less memory than BFS, easier to reconstruct the path once found; but DFS does not always find shortest path. BFS does.

BFS that finds path

bfs from v_1 to v_2:

create a *queue* of vertexes to visit,

initially storing just v_1.

mark v_1 as **visited**.

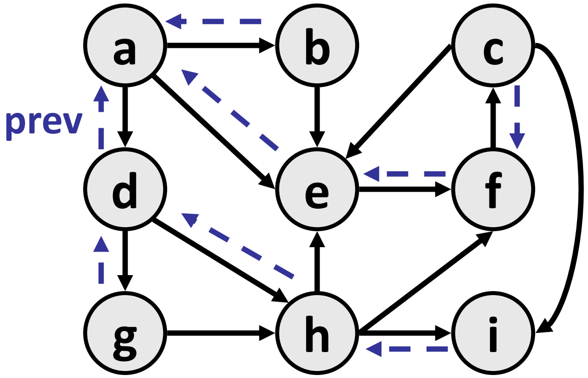
while *queue* is not empty and v_2 is not seen:

dequeue a vertex v from it,

mark that vertex v as **visited**,

and add each unvisited neighbor ***n*** of v to the *queue,*

while setting n's previous to v.



Dijkstra's Algorithm (18.6)

- **Dijkstra's algorithm**: Finds the minimum-weight path between a pair of vertices in a weighted directed graph.
  - Solves the "one vertex, shortest path" problem in weighted graphs.
  - *basic algorithm concept*: Create a table of information about the currently known best way to reach each vertex (cost, previous vertex), and improve it until it reaches the best solution.

- *Example:* In a graph where vertices are cities and weighted edges are roads between cities, Dijkstra's algorithm can be used to find the shortest route from one city to any other.

Dijkstra pseudocode

dijkstra(v_1, v_2):
consider every vertex to have a cost of infinity, except v_1 which has a cost of 0.
create a *priority queue* of vertexes, ordered by cost, storing only v_1 .

while the *pqueue* is not empty:
dequeue a vertex *v* from the *pqueue*, and mark it as **visited**.
for each of the unvisited neighbors *n* of *v*, we now know that we can reach this neighbor with a total **cost** of (*v*'s cost + the weight of the edge from *v* to *n*).
if the neighbor is not in the *pqueue*, or this is cheaper than *n*'s current cost, we should **enqueue** the neighbor *n* to the pqueue with this new cost, and with *v* as its previous vertex.

when we are done, we can **reconstruct the path** from v_2 back to v_1 by following the previous pointers.

Heuristics

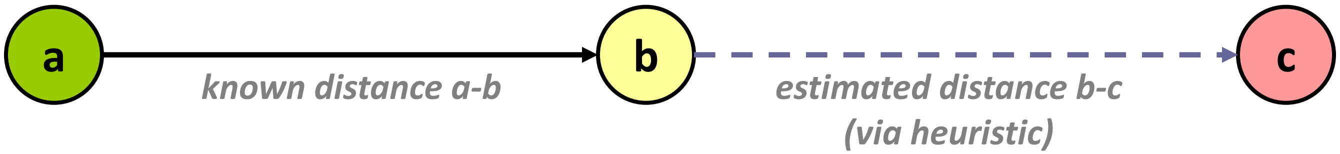
- **heuristic**: A speculation, estimation, or educated guess that guides the search for a solution to a problem.
  - *Example:* Spam filters flag a message as probable spam if it contains certain words, has certain attachments, is sent to many people, ...
  - In the context of graph searches: A function that approximates the distance from a known vertex to another destination vertex.
  - Example: Estimate the distance between two places on a Google Maps graph to be the direct straight-line distance between them.

very eager, AH <= real answer

- **admissible heuristic**: One that never overestimates the distance.
  - Okay if the heuristic underestimates sometimes (e.g. Google Maps).
  - Only ignore paths that in the *best case* are worse than your current path

The A* algorithm

- **A*** ("A star"): A modified version of Dijkstra's algorithm that uses a heuristic function to guide its order of path exploration.



关键在于如何准确定义heuristic

- Suppose we are looking for paths from start vertex *a* to *c*.
  - Any intermediate vertex *b* has two costs:
  - The known (exact) cost from the start vertex *a* to *b*.
  - The heuristic (estimated) cost from *b* to the end vertex *c*.

- *Idea:* Run Dijkstra's algorithm, but use this priority in the pqueue:
  - priority(*b*) = cost(*a*, *b*) **+ Heuristic(*b*, *c*)**
  - Chooses to explore paths with lower estimated cost.

A* pseudocode

astar(v_1, v_2):

consider every vertex to have a cost of infinity, except v_1 which has a cost of 0.

create a *priority queue* of vertexes, ordered by (cost+**heuristic**), storing only v_1 **with a priority of H(v_1, v_2)**.

while the *pqueue* is not empty:

dequeue a vertex *v* from the *pqueue*, and mark it as **visited**.

for each of the unvisited neighbors *n* of *v*, we now know that we can reach this neighbor with a total **cost** of (*v*'s cost + the weight of the edge from *v* to *n*).

if the neighbor is not in the *pqueue*, or this is cheaper than *n*'s current cost, we should **enqueue** the neighbor *n* to the pqueue with this new cost **plus H(*n*, v_2)**, and with *v* as its previous vertex.

when we are done, we can **reconstruct the path** from v_2 back to v_1 by following the previous pointers.

** (basically, add H(...) to costs of elements in PQ to improve PQ processing order)*

Kruskal's algorithm

- **Kruskal's algorithm**: Finds a MST in a given graph.

```
function kruskal(graph):
  Start with an empty structure for the MST
  Place all edges into a priority queue
      based on their weight (cost).
  While the priority queue is not empty:
    Dequeue an edge e from the priority queue.
    If e's endpoints aren't already connected,
        add that edge into the MST.
    Otherwise, skip the edge.
```

- **Runtime:** O(E log E) = O(E log V)

Kahn's Algorithm

map := {each vertex $\rightarrow$ its in-degree}.

queue := {all vertices with in-degree = 0}.

ordering := { }.

Repeat until queue is empty:

  Dequeue the first vertex *v* from the queue.

  ordering += *v*.

  Decrease the in-degree of all *v*'s neighbors by 1 in the *map*.

  queue += {any neighbors whose in-degree is now 0}.

- **This algorithm doesn't modify the passed-in graph! ☺**
- **Have we handled all edge cases?**
- **What is the runtime of this algorithm?**

3 Key Ideas for Improvement

- Keep a *queue of nodes with in-degree 0* so we don't have to search for nodes multiple times.
- A node's in-degree only changes when one of its prerequisites is completed. Therefore, when completing a task, check if any of its neighbors now has in-degree 0.
- Keep a *map of nodes' in-degrees* so we don't need to modify the graph.