Graphs and Data Structure Design

Nicolò Felicioni[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano

nicolo . felicioni @ polimi . it

June 3, 2021

[1] Based on Nicholas Mainardi's material.

Graphs

Definition

A graph is a pair of sets V, E, where:

- V is the set of vertexes
- E is the set of edges, which are connections among vertexes
- Formally, $E \subseteq V^2$, since each edge is a pair of vertexes
- If the order of pairs in an edge is relevant, the graph is called directed

- Trees are a particular, constrained type of graphs: acyclic undirected connected graphs
- Graphs are generally really useful to model a wide variety of real world situations
  - In particular, edges usually are easily mapped to relationships
  - There are no constraints on the topology of the graph, differently from trees (indeed suitable for hierarchical relationships)

Graphs

Memory Representation

- There are 2 main strategies to represent a graph in memory:
  1. Adjacency lists: the graph is an array of vertexes, where each one has a pointer to a list of vertexes which are connected to it through an edge
  2. Adjacency matrix: the graph is a $V \times V$ matrix A, where $A[i][j] = 1 \iff (i, j) \in E$
- List is more compact: requires $\Theta(V + E)$
- Matrix is always $\Theta(V^2)$
- Therefore, if the graph is highly connected, that is there are a lot of edges, there is no big difference between the 2 representations, thus matrix may be better because some operations are faster
- Instead, a list is way more compact if the graph has few edges

Graphs

Graph Operations Recap

- BFS: Breadth First Search $\Rightarrow O(V + E)$
- DFS: Depth First Search $\Rightarrow O(V + E)$
- Finding shortest path in a weighted graph : $O(VE)$ with Bellman-Ford, $O(V \log(V) + E)$ with Dijkstra (applicable only if all the weights are non-negative)
- Topological Sorting on direct acyclic graphs: $\Rightarrow O(V + E)$

Minimum Path with Prohibited Nodes

Problem

Given a graph $G = (V, E)$, 2 nodes $s, t \in V$, and a set of prohibited nodes $P \subseteq V$, design an algorithm able to find, if exists, the minimum path between s and t which does not include any prohibited node

- Looking at the problem from another perspective: it is equivalent to searching t starting from s, without including prohibited nodes in the search path
- Therefore, we may use a search algorithm which is modified to avoid prohibited nodes
- In particular, nodes in P are marked with a different color (e.g. black, which is used for nodes being already visited in search algorithms) which allows to avoid considering these nodes

Minimum Path with Prohibited Nodes

Which Search Algorithm?

- Recall that, if the path exists, we have to return the minimum
- BFS try first all the paths with length 1, then paths with length 2 and so on
- Therefore, if we use BFS, the first solution found is already the minimum path
- Indeed, if there had been a shorter path, it would have been already found by the algorithm
- Instead, with DFS we may find a longer path as the first one
- Time complexity?
- BFS code requires only slight modifications, actually just black coloring nodes in $P \Rightarrow O(P) \in O(V)$
- Therefore, the time complexity is the same as BFS: $O(V + E)$

Universal Sinks

Problem

Given a directed graph $G = (V, E)$, a sink s is a node with no outgoing edges, that is $\nexists (i, j) \in E (i = s)$. A sink is called universal if it is connected to all other nodes. Consider the problem of searching a universal sink in a graph G. Choose the memory representation which is best suitable to efficiently solve this problem, and describe an algorithm which searches a universal sink in G.

Universal Sinks

Choosing Memory Representation

- Let's focus on the verification for a candidate universal sink
- To verify it is a sink, we have to look at outgoing edges:
  - ↪ With a list, we just need to verify that the list of adjacent nodes is empty $\Rightarrow O(1)$
  - ↪ With a matrix, we need to verify that all elements are 0 in the row corresponding to the candidate node $\Rightarrow O(V)$
- To verify it is universal, we have to look at ingoing edges:
  - ↪ With a list, we need to search for the candidate sink in all the adjacent lists, for each node $\Rightarrow O(V + E)$
  - ↪ With a matrix, we need to verify that all elements are 1 in the column corresponding to the candidate node, except for the element on the diagonal (which means there are no self-loops on the candidate node) $\Rightarrow O(V)$
- Thus, verification costs: $O(V)$ for matrix, $O(V + E)$ for list

Universal Sinks

Algorithm Design

- Given the graph G is better represented with an adjacency matrix, we need to find a fast algorithm to verify existence of universal sinks in the graph
- Naive idea: verifying each node!
  - ↪ Each verification costs $O(V)$
  - ↪ $O(V)$ verifications a most
  - ↪ This simple solution has $O(V^2)$ time complexity
- Let's see what we can infer by looking at a generic element of the adjacency matrix $A[i][j]$
  1. If $A[i][j] = 1$, we know that the node associated to row i is not a sink
  2. Conversely, if $A[i][j] = 0$, we know that the node associated to column j cannot be a universal sink (except for the case $i = j$)
- Thus, by looking at a single cell we can discard 1 node

Universal Sinks

Algorithm Design

- Now, we need to define a policy to visit cells in matrix A
- Suppose we start from the cell $A[1][2]$ ($i = j = 1$ is a corner case which will be handled later)
- If $A[1][2] = 1$, then we can discard node 1, thus we can consider $A[2][2]$
- Conversely, we can move to $A[1][3]$, discarding node 2
- With such a policy, calling $m = max(i, j)$, it means that we have already discarded all nodes $< m$, except for the node i or j which is different from m
- Therefore, it is legitimate updating an index i to $m + 1$
- In this way, we avoid considering nodes already being discarded

Universal Sinks

Algorithm Design

- The corner case $i = j$ needs to be handled: if $A[i][j] = 1$, we need to discard i, moving to the next element in the diagonal. Instead, default behavior is ok if $A[i][j] = 0$, since we retain i as a candidate universal sink, but we consider $j + 1$ element for further analysis
- Since now we have just handled corner case $i = j$, we can also start with $A[1][1]$ our visit
- Summing up, after V comparison, $m + 1 > V$, and thus the only node not being discarded is the only one which can still be a universal sink
- We can test it in $O(V)$ time
- Therefore, we need V comparisons to find the candidate universal sink, and $O(V)$ time to verify it: $\Rightarrow \Theta(V)$ time

Universal Sinks

Pseudocode

FIND-UNIVERSAL-SINK(A)

```
 1  i ← j ← max ← 1
 2  while max ≤ A.n
 3        do max ← max + 1
 4           if A[i][j] = 0
 5              then j ← max
 6                   sink ← i
 7              else  if i = j
 8                       then i ← j ← sink ← max ▷ Corner case for A[i][i] = 1
 9                       else  i ← max
10                             sink ← j
11  if max = sink ▷ All nodes already discarded since sink = max = A.n + 1
12     then return NIL
13  for j ← 1 to A.n ▷ Verify if candidate sink is a sink
14        do if A[sink][j] = 1
15              then return NIL
16  for i ← 1 to A.n ▷ Verify if candidate sink is a universal sink
17        do if A[i][sink] = 0 ∧ i ≠ sink
18              then return NIL
19  return sink
```

Colored (Sub)Graph

Problem

Consider an undirected graph G where the nodes may have 2 colors, yellow and red. We want to split the graph in 2 connected subgraphs, one with all and only yellow nodes, the other one with all and only red nodes

Thinking About a Solution

- Idea: the graph is already split, we have just to check that all nodes are connected
- In other words, we have already the 2 subgraphs, we have to visit them to verify all the nodes are reachable
- Thus, we can employ a slightly modified graph visit algorithm $\text{VISIT-COLOR}(G, \textit{color})$

Colored (Sub)Graph

Algorithm Design

- In particular, it visits only nodes of one color, replacing this color with another one to mark the nodes as visited
- Note that if nodes with a different color are met, they are ignored
- Therefore, if there is a yellow node which is reachable just by a red one, $\mathrm{VISIT\text{-}COLOR}(G, \mathit{yellow})$ will not visit it
- The algorithm works as follows:
  1. Call $\mathrm{VISIT\text{-}COLOR}(G, \mathit{yellow}) \Rightarrow O(V + E)$
  2. Call $\mathrm{VISIT\text{-}COLOR}(G, \mathit{red}) \Rightarrow O(V + E)$
  3. Check if there are still yellow or red nodes, implying at least one of the subgraphs is unconnected $\Rightarrow O(V)$
- Time complexity is dominated by $\mathrm{VISIT\text{-}COLOR}$ algorithm $\Rightarrow O(V + E)$

Company Employees

Problem

A company has a hierarchical organization for its employees. In particular, each one has a supervisor (except for the CEO) and may have one or more person he/she is responsible for. The last level of the hierarchy is represented by workers, which do not have any people they are responsible for. The CEO wants to verify if this hierarchy is not too long, which means he wants to ensure that the levels between him and any worker are less than $\frac{1}{20}$ of the total number of employees of the company. Propose a solution to the CEO problem and get hired by this company

Data Structure

- We need to represent hierarchical relationships
- A tree seems the best data structure

Company Employees

Algorithm Design

- However, it is not a binary tree, neither a search one
- The children pointers are all stored in an array in the parent node
- Let's analyze what problem we need to solve on a tree
- The maximum levels between the CEO and any worker are equal to the height of the tree h
- The total number of employees are the nodes of the tree n
- Thus, we need to verify that $\frac{h}{n} \leq \frac{1}{20}$
- We can do it with a visit algorithm which computes the max depth and count the nodes

Company Employees

Pseudocode

```
CountNodesMaxDepth(node)
1  max ← 0
2  count ← 1
3  for i ← 1 to node.children.length
4        do (c, m) ← CountNodesMaxDepth(node.children[i])
5           count ← count + c
6           if m > max
7              then max ← m
8  return (count, max + 1)
```

- $(n, h) \leftarrow$ CountNodesMaxDepth(Ceo), where Ceo is the root of the tree
- Complexity: $O(1)$ processing on each node, and every node is visited once $\Rightarrow O(n)$

Boxers

Problem

There is a set of n boxers, where some of them are rivals. Each boxer may have an arbitrary number of rivals. Design a data structure and an algorithm able to partition the set in 2 teams, with the constraint of no rival boxers in the same team

Choosing the Data Structure

- We need to represent relationships between a set of elements
- These relationships are not hierarchical
- A graph is the perfect data structure
- Which type of graph?
- The rivalry is a symmetric relationship: undirected graph

Boxers

Algorithm Design

- The constraint on the teams basically means that 2 adjacent nodes cannot be in the same team
- Therefore, if a node is already in team A, all its neighbors must be in team B
- Then, all the neighbors of each neighbors must be in team A, and so on
- Basically, if we start from a node v placing it in team A, all nodes at distance 1 are placed in team B, all nodes at distance 2 in team A, and so on
- Moreover, for each node being assigned, we have to check that no one of its neighbors is in the same team

Boxers

Algorithm Design

- Therefore, we need to visit the graph, assign the node to the correct team depending on its distance from the source, and check all the neighbors are in a different team
- Which visit algorithm? Teams are assigned depending on the distance, thus Breadth First Search seems a viable solution
- We can introduce 2 new colors to assign elements to one team or another
- The problem is thus equivalent to color the nodes of the graph such that there are no adjacent nodes with the same color $\Rightarrow$ Graph 2 Coloring Problem
- Which node do we start with? Does it affect the coloring of the graph?

Boxers

Graph 2 Coloring

- Suppose that, by start coloring from node v, we do not find a suitable coloring, even if this coloring exists
- Fact: for each node, all the nodes at odd distance have different colors, while all the nodes at even distance have the same color
- If we have not found a coloring, it means that there are 2 adjacent nodes x, y with the same color
- But if they share the same color, then their distances from v are either both even or both odd
- However, if there exists a coloring, then these nodes must have different colors in such a coloring. There are 2 cases:
  1. Their distances from v are odd
  2. Their distances from v are even

Boxers

Graph 2 Coloring - Case 1

Since the distance between x and v is odd, then their colors must be different, and the same holds for y. Therefore,
$x.col \neq v.col \wedge y.col \neq v.col \Rightarrow x.col = y.col \Rightarrow$ absurd

Graph 2 Coloring - Case 2

Since the distance between x and v is even, then their colors must be the same, and the same holds for y. Therefore,
$x.col = v.col \wedge y.col = v.col \Rightarrow x.col = y.col \Rightarrow$ absurd

- In conclusion, if we do not find a suitable coloring starting from a node v, then this coloring does not exist
- Therefore, the choice of the node v as a starting point for BFS does not matter
- Complexity: BFS with some additional checks $\Rightarrow O(V + E)$

Real Numbers

Problem

We want to design a data structure to represent a set of real numbers. We need to perform these 3 operations:

1. $\mathrm{INSERT}(S, r)$: check if the value is in the set, if not r is added to S
2. $\mathrm{DELETE}(S, r)$: check if the value is in the set and delete it
3. $\mathrm{CLOSER\text{-}TO\text{-}AVG}(S)$: find the element in S which is closer to the average of all elements of S

Design the data structure which minimizes the complexity of these 3 operations

- The average of the set can be easily updated during INSERT and DELETE in $O(1)$ with a couple of operations
- Thus, we can store this average in a variable

Real Numbers

Data Structure

- The usual structure to represent a set is an hash table
- There are nice hash functions for real numbers which may be employed
- However, hash tables are suitable for $\mathrm{SEARCH}, \mathrm{INSERT}$ and DELETE. What about aggregate functions like $\mathrm{CLOSER\text{-}TO\text{-}AVG}$?
- We would like to avoid iterating over the whole table to analyze all elements in the set
- We may use 2 pointers in each record of the table which points to the previous and next elements in the table
- Sort of doubly linked list with $O(1)$ access to each element
- To compute $\mathrm{CLOSER\text{-}TO\text{-}AVG}$, we can find the closest element by visiting all of them using the pointers $\Rightarrow \Theta(n)$

Real Numbers

Improving Hash Tables?

- Is $\Theta(n)$ the best we can obtain for $\textsc{closer-to-avg}$?
- If the set is sorted, we can surely stop as soon as the elements become greater than the average and find the closest one
- Which data structure allows to benefit from such a structured set? A BST!
- In a BST, if we search for the average, the closest value will surely be visited in the search path. Indeed:
  - If the average is in the set, we will find it with $\textsc{search}$
  - If it is not, then we will reach the place where the node with the average value should be inserted. Then, $\textsc{closer-to-avg}$ can be computed as the closer between the predecessor and the successor of this average node. By definition, since this node has no children, its predecessor and its successor will be in the search path, and thus they have already been visited (and hence saved)

Real Numbers

Pseudocode

CLOSER-TO-AVG(S)

```
 1  node ← S.root
 2  pred ← succ ← NIL
 3  while node ≠ NIL
 4        do if S.avg = node.key
 5              then return node
 6           if S.avg < node.key
 7              then succ ← node
 8                   node ← node.left
 9              else  pred ← node
10                   node ← node.right
11  if pred = NIL
12     then return succ
13  if succ = NIL
14     then return pred
15  if S.avg − pred.key > succ.key − S.avg
16     then return succ
17  return pred
```

Real Numbers

Time Complexities Comparison:

Algorithm	Doubly Linked Hash Table	Balanced BST
INSERT	$O(1)$	$O(\log(n))$
DELETE	$O(1)$	$O(\log(n))$
CLOSER-TO-AVG	$O(n)$	$O(\log(n))$

BST is preferable since the complexity is at most $O(\log(n))$ for each operation, while the hast table incurs $O(n)$ overhead for CLOSER-TO-AVG, despite being faster for INSERT and DELETE