Chapter: Graph Traversals

Guilt-by-Association Principle

Guilt by association refers to the situation where an individual is assumed to be guilty of a crime because of their association with someone who has committed the crime.

Gene networks are commonly interpreted as encoding functional information in their connections. Guilt-by-association principle states that genes which are associated or interacting are more likely to share function, i.e, connected nodes tend to have similar functions.

- Proteins with known function + network topology ⇒ function assignment for unknown proteins.

Graph Traversals

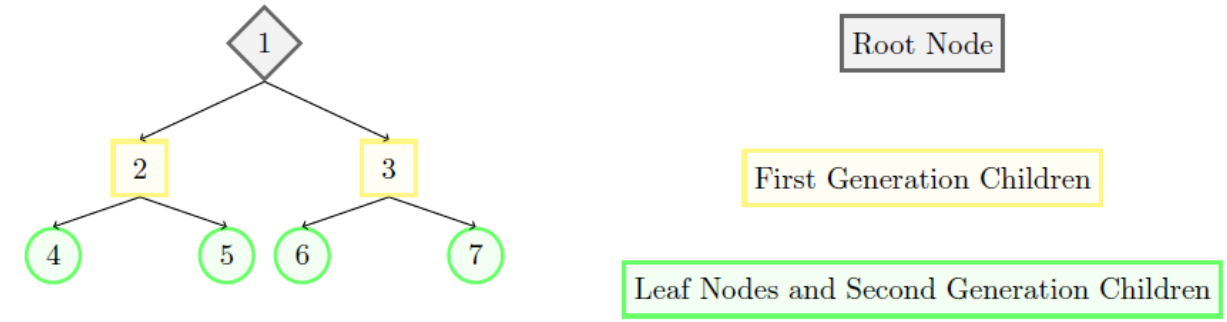
Graph traversals are also called graph search.

- Can be thought of as "walking around" in a graph with the goal of visiting every node (=vertex) in a graph.
- Core ingredient of most graph algorithms

There are three main strategies for traversing a graph:

1. Breadth-First Search
2. Depth-First Search
3. Random Walk

For the below examples, we will use a rooted tree graph, where one vertex is the root from which all the other nodes "sprout" directly or indirectly.



Breadth-First Search (BFS)

A graph traversal algorithm where we traverse the nodes level by level. So, all the "shallow nodes" will be visited before any "deep nodes".

- We use a queue, operating on First-in-first-out principle.

- We mark each visited node as *visited*, so we do not visit the same node twice.

Compared to the Depth First Search, the BFS is the more suitable approach if one wants to find a node that is closer to the starting vertex.

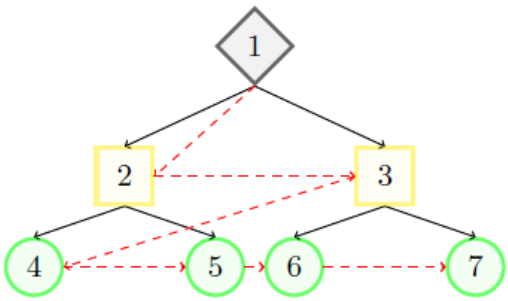


Figure 6.2: Example of a **Breadth First Search** in a binary rooted tree graph.

Table 6.1: Example of what the queue during the BFS in the above Figure would look like after each visiting step.

Visited node	Queue after visiting that node
–	1
1	2,3
2	3,4,5
3	4,5,6,7
4	5,6,7
5	6,7
6	7
7	–

BFS Algorithm

As breadth-first search is the process of traversing each node of the graph, a standard BFS algorithm traverses each vertex of the graph into two parts: 1) Visited 2) Not Visited. The purpose of the algorithm is to visit all the vertices while avoiding cycles.

BFS starts from a node, then it checks all the nodes at distance one from the beginning node, then it checks all the nodes at distance two, and so on. So as to recollect the nodes to be visited, **BFS uses a queue**.

The steps of the algorithm work as follow:

1. Start by putting any one of the graph's vertices at the back of the queue.
2. Now take the front item of the queue and add it to the visited list.

3. Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
4. Keep continuing steps two and three till the queue is empty.

BFS Pseudocode

```
create a queue Q
mark v (current node) as visited and put v into Q
while Q is non-empty
      remove the head u of Q
      mark and enqueue all (unvisited) neighbors of u
```

Depth-First Search (DFS)

A graph traversal algorithm, where we traverse as deeply as possible into the graph. So, all the "deep nodes" will be visited before any "shallow" nodes. It finds a path between two vertices by exploring each possible path as far as possible before backtracking.

- We use a stack, operating on Last-in-first-out principle
- General steps:
    - Follow path until you get stuck (reached a leaf node)
    - Backtrack until reaching unexplored neighbor (furthest from the beginning)
    - Recursively explore
    - Mark each visited node (not to repeat)

Compared to the BFS, the DFS is the more suitable approach if one wants to find a node that is quite far away from the starting vertex.

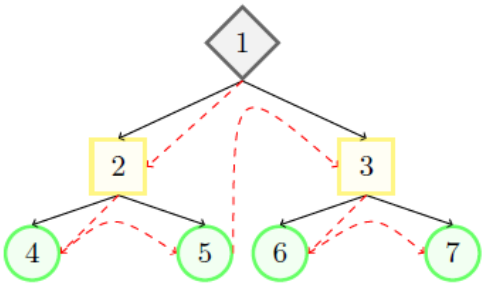


Figure 6.3: Example of a **Depth First Search** in a binary rooted tree graph.

Table 6.2: Example of what the stack during the DFS in the above Figure would look like after each visiting step.

Visited node	Stack after visiting that node
–	1
1	2,3
2	4,5,3
4	5,3
5	3
3	6,7
6	7
7	–

DFS Algorithm

The Depth-First Search (DFS) is a recursive algorithm that uses the concept of backtracking. It involves thorough searches of all the nodes by going ahead if possible, else it backtracks. Here, backtracking means once there are no more nodes along the present path to move forward, we progress backward on an equivalent path to seek out other nodes to traverse. All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

The DFS algorithm is **implemented using stack**. A standard implementation puts every vertex of the graph into one in all 2 categories: 1) Visited 2) Not Visited. The purpose of this algorithm is to visit all the vertices of the graph avoiding cycles.

The steps of the algorithm work as follow:

1. We will start by putting any one of the graph's vertices on top of the stack.
2. After that take the top item of the stack and add it to the visited list of the vertex.
3. Next, create a list of that adjacent node of the vertex. Add the ones which aren't in the visited list of vertices to the top of the stack.
4. Lastly, keep repeating steps 2 and 3 until the stack is empty.

Random Walks

- A random walk is a process for traversing a graph where at every step we follow an outgoing edge chosen uniformly at random.

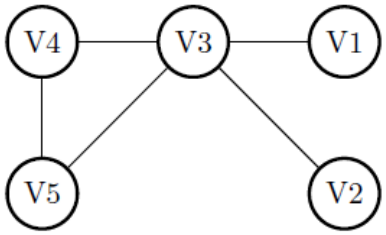
Let G = (V, E, w) be a weighted undirected graph. A random walk may start at any node of G with some probability. In each step, the walk may continue along any edge, in the direction of edges if the network is directed with some probability.

- The probability to which an edge is chosen can be distributed evenly or unevenly, when there are weights provided.
- A random walk has the Markov property: knowledge of previous states is irrelevant in predicting the probability of subsequent states. Therefore, the next state depends entirely on the current state.
- Going backwards and revisiting nodes multiple times is entirely possible. The nodes are not being marked as *visited*.

If there is no possibility of going backwards, it is no longer considered a random walk but instead a One-Way-Walk.

Math of Random Walks

The random walk is defined by the transition matrix M, which is a stochastic matrix where each element a_{ij} represents the probability of moving from i to j , with each row summing to 1. The rows represent from which node we are starting (i) and the columns represent the nodes that we could walk towards (j) from i.



$$M_{ij} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 0 \\ 3 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} \\ 4 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 5 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \end{array}$$

- Diagonal elements are 0, since there is no self-loops
- 0 in other than diagonal position means there is no edge existing between that pair of nodes
- Probability of traveling from node i to node j depends on d(i), degree of node i

Mathematically,

$$M_{ij} = \begin{cases} \frac{1}{d(i)} \; if \; (i,j) \in E(G) \\ 0 \; otherwise \end{cases}, \sum_{j \in N(i)} M_{ij} = 1$$

It reads: the probability for walking from i to j is 1 divided by the degree of i if there is an edge between i and j. Otherwise it is zero. The sum of each row is 1.

Probability Distributions

- Let P^t be a row vector with a component for each vertex specifying the probability mass of the vertex at time t and let P^{t+1} be the row vector of probabilities at time $t + 1$.
- The distribution one step later can be calculated as: $P^{t+1} = P^t M$
- If P^0 is some initial distribution, then: $P^t = P^0 M^t$

As the random walk continues, it is more likely to be on a higher-degree vertex than a lower-degree vertex at any given time.

Stationary Distribution

When the walker keeps walking for a long, long time, we get a stationary distribution.

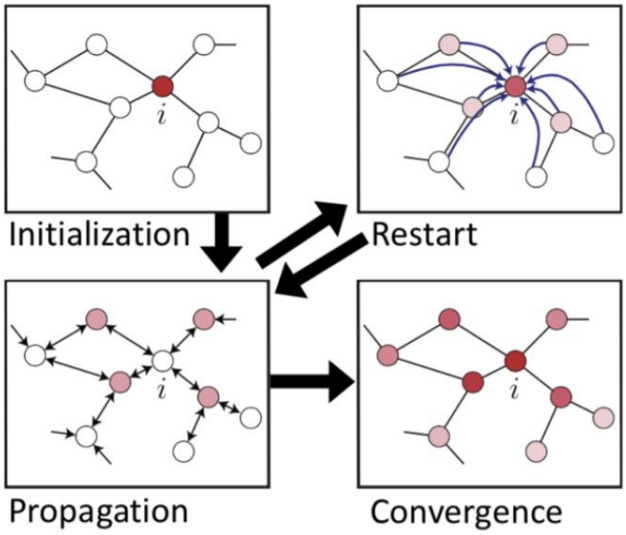
- A stationary distribution $\pi \in R^n$ of a Markov chain with transition matrix M is a probability distribution satisfying: $\pi = \pi M$
- In other words, if we draw a random vertex from π, then take a random step from the vertex, the distribution of our final endpoint is again π.

Random Walk with Restart (RWR)

By adding a Restart possibility into the Random Walk algorithm, the distribution of nodes where you land upon at the end of your walk will be shifted closer towards the source.

Intuition:

- Walker at a node either with probability $1 - r$, follows an outgoing edge
- With restart probability r, returns to node in restart set
- In long run, converges “Stationary distribution” of the walker over the nodes

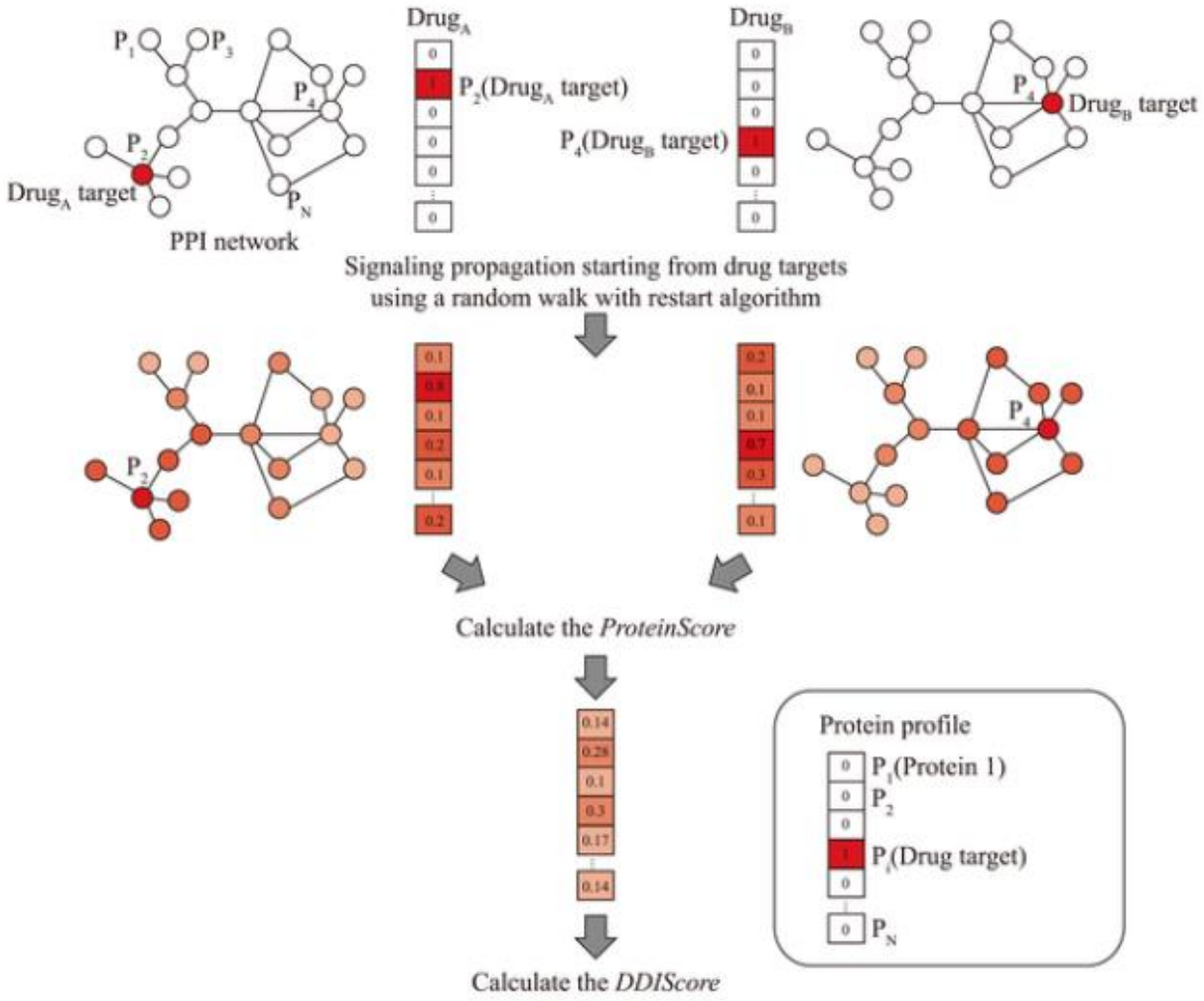


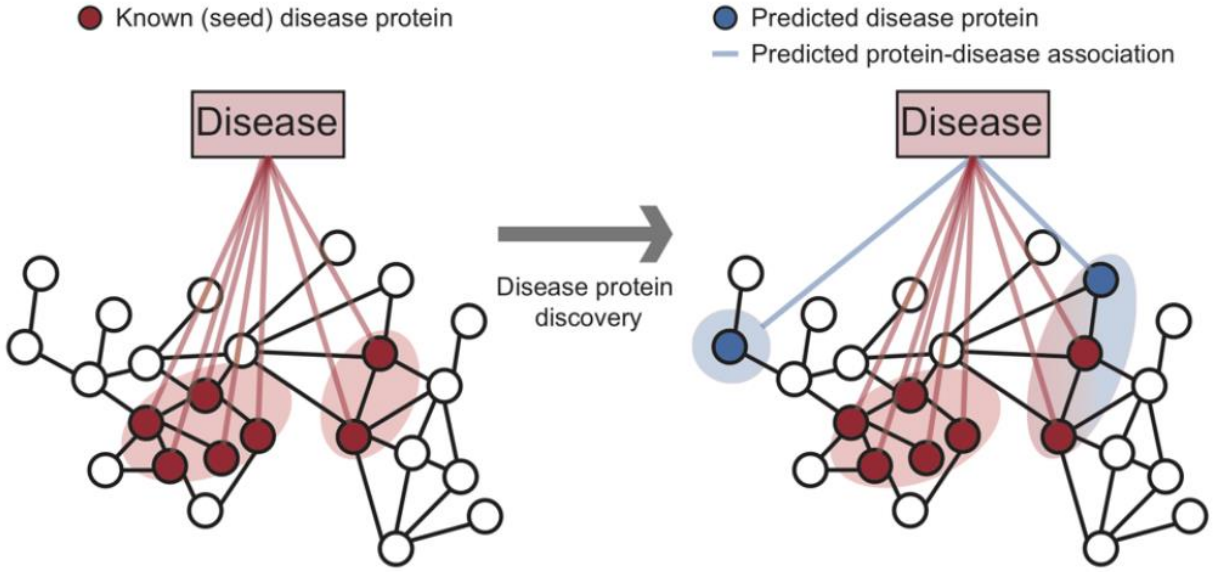
An example for such an application is Network-based gene prioritization, where you want to find out which genes in a network are most probable to be connected to a particular process, e.g. a disease.

Network-based Gene Prioritization

Gene prioritization is the task of ranking the most important genes for a particular process or system under study through integrative computational analysis of public and private genomic data. If we already have a set of genes that we know are important, we can derive other genes that are also important based on their proximity to the network (Guilt of Association principle).

- For Network-based gene prioritization, RWR is considered a state-of-the-art approach to identify and prioritize disease genes.
- By making the random walk start from known disease genes we obtain a global proximity measure from these known genes.
- A random walker moves from a known disease gene to a random neighbor within the human PPI network.


P1
P3
P4
P2
DrugA target
PN
PPI network
DrugA
P2(DrugA target)
DrugB
P4(DrugB target)
P4
DrugB target
Signaling propagation starting from drug targets
using a random walk with restart algorithm
P2
P4
Calculate the ProteinScore
Protein profile
P1(Protein 1)
P2
Pi(Drug target)
PN
Calculate the DDIScore


Known (seed) disease protein
Predicted disease protein
Predicted protein-disease association
Disease
Disease protein
discovery
Disease

Shortest Path Problem

Given a graph G = (V, E) with edges of length w(u, v) >0,
Goal: find paths with minimum length between sources s and destination t

Real-world example:

- Real life maps and trip planning
    - Can we get from one location (vertex) to another location (vertex) given the current available roads (edges)
    - What is the most efficient path (e.g. cost wise or time wise) to reach a location
- Family trees and checking ancestry
    - Are two people (vertices) related by some common ancestor
- Geometric analysis of proteins and their binding site
- Learning about protein functions by guilt of association

Dijkstra's Algorithm

It is a greedy algorithm (oftentimes a greedy strategy does not produce an optimal solution because it gets stuck on local optima instead of reaching the global optimum.) but will still find the optimal solution (shortest path between two nodes).

Solves the single-source shortest path problem:

- Find the shortest path from a single source to ALL nodes in the network
- Works on both directed and undirected networks
- Works on both weighted and non-weighted networks

Input: A weighted or unweighted graph G which can be directed or undirected. You also need to specify the starting node S.
Output: The shortest paths from S to all other nodes in G, respectively, are calculated.
Limits: Negative weights cannot be used.

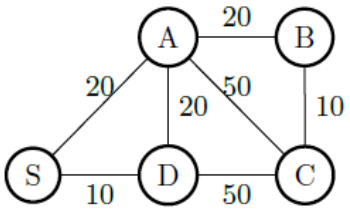
Initialization:

- Lists with visited and unvisited nodes are created. The visited-list is empty at first.
- Cost to reach starting point S is set to zero, since it takes zero steps to get to S from S.
- Cost to reach every other node from S is set to ∞.

Algorithm:

a. The node with the smallest cost that has not been visited yet (S in the beginning) is set as the current node. If there is no smallest cost because there are two nodes that have the same smallest cost, one is chosen at random
b. Traverse to the neighbors of the currently visited node. Calculate the cost of these nodes by adding the values of the weights of the respective edges to the cost of the current node
c. Check if the new cost is lower than the old cost. If so, overwrite the old cost with the new
d. Have all nodes been visited? If not, go back to step two and repeat the process. If yes, the algorithm is done.

Let's do an easy example to illustrate the workflow of Dijkstra's algorithm and also show how to extract the shortest path from a table. Given the following graph G = (V, E) with edges of weight w(u, v) > 0 (Figure below).



Node	Cost	Previous Node
S	0	S
A	∞	not set
B	∞	not set
C	∞	not set
D	∞	not set

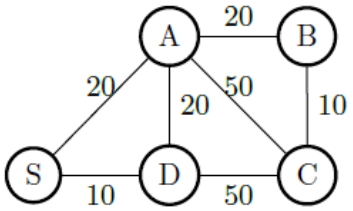
Figure 6.5: Left: Example Graph G with weighted edges. Right: Table for Dijkstra's algorithm. At the beginning the cost of S is 0 and all other costs are set to infinity. No nodes have been visited yet.

Below table shows this procedure for every iteration. In addition, the table on the right in the above figure is also filled in with the cost values and the node that was visited before the current node. The previous node gets also updated if the cost is reduced.

Table 6.3: The table shows the state in every iteration of dijkstra's algorithm. On the right side of the table the cheapest cost is noted. The red indicates a change in the cost that iteration.

Iteration	Current node	Visited	Not Visited	S	A	B	C	D
Initialization	-	-	S, A, B, C, D	0	∞	∞	∞	∞
1	S	S	A, B, C, D	0	20	∞	∞	10
2	D	S, D	A, B, C	0	20	∞	60	10
3	A	S, D, A	B, C	0	20	40	60	10
4	B	S, D, A, B	C	0	20	40	50	10
5	C	S, D, A, B, C	-	0	20	40	50	10

After five iterations we end up with a filled-in table (below figure). The smallest costs have been calculated for every node and a predecessor for every node has been written down. From this table it is now possible to calculate the shortest path from S to every other node. For this we just move backwards from the target node T. Let's say our target node T = C. We write this into a list shortest path =[C]. We look at the previous node, which is B and add it at the front of our list shortest path =[B,C]. We find the predecessor of B in the table, A, and add it to the front of the list shortest path =[A,B,C]. We do the same step one more time for A, find its predecessor is S and add it to the list shortest path =[S,A,B,C]. Then we stop because we have reached S and have found the shortest path from S to C.



Node	Cost	Previous Node
S	0	S
A	20	S
B	40	A
C	50	B
D	10	S

Figure 6.6: Left: Example Graph G with weighted edges. Right: Table for Dijkstra's algorithm. After all nodes have been visited, each node as a respective cost that is smaller than infinity. For each node a predecessor has been written down.