DESIGN OF ALGORITHMS

A Recursion Perspective

Liming Cai

August 27, 2024

Contents

Chapter 4

Graph Algorithms

Definition 2. A *graph* G consists of a pair of two sets (V, E), where V is the set of *vertices* and $E \subseteq V \times V$ is the set of *edges*.

Graphs defined above should not contain *self-loop edges*. That is, for every edge $(u, v) \in E$, $u \neq v$. The following briefly gives definitions to some basic terminologies in graph theory, which will be extensively used in this and other chapters.

Terminologies of graphs

Let $G = (V, E)$ be a graph. G is a *directed graph* if the relation E is not symmetric; otherwise, it is called a *non-directed graph*. G is a *weighted graph* if it is associated with an *edge-weight* function $w : E \to \mathbb{R}$, where $\mathbb{R}$ is the real number set. Let $(u, v) \in E$, then vertex v is called a *neighbor* of vertex u. The *degree* of a vertex v is the number of its neighbors. In particular, for directed graphs, the degree of vertex u can be broken down as the sum of *out-degree* and *in-degree*. The former is the number of neighbors of v for which $(u, v) \in E$ and the latter is the number of neighbors of v for which $(v, u) \in E$.

Definition 3. Given graph $G = (V, E)$, a *subgraph* of G is a graph $H = (U, F)$ for some $U \subseteq V$ and $F \subseteq E \cap (U \times U)$, denoted with $H \subseteq G$. In addition, H is called an *induced subgraph* if for every pair $u, v \in U$, $(u, v) \in F$ if and only if $(u, v) \in E$.

A *path* in graph $G = (V, E)$ is a collection of k edges $\{(u_1, v_1), \ldots, (u_k, v_k)\}$

in G, for some $k \geq 1$, such that $v_i = u_{i+1}$, for all $1 \leq i < k$. A path in a directed graph is called a *directed path*. The path is a *simple path* if all u_i are different, $1 \leq i \leq k$. This path is said to *connect* vertices u_1 and v_k. The path is called a *cycle* if $u_1 = v_k$. A non-directed graph is a *tree* if there is no cycle present in the graph. A directed graph without a directed cycle is termed with *directed acyclic graph* (DAG). A subgraph H is a *connected component* if there is a path connect every pair of vertices in H. A subgraph H in a directed graph is a *strongly connected component* (SCC) if there is a directed path from every vertex to every other vertex.

A graph is *complete graph* if there is an edge between every pair of vertices. A complete graph of n vertices has $\frac{n}{2}(n+1)$ edges, denoted with K_n. A graph $G = (V, E)$ is a *bi-partite graph* if V can be partitioned into two disjoint sets $V_1 \cup V_2$ such that $E \subseteq V_1 \times V_2$. When $E = V_1 \times V_2$, the graph is called a *complete bi-partitie* graph, denoted with K_{n_1,n_2}, where $n_1 = |V_1|$ and $n_2 = |V_2|$. A graph is called *planar* if it can be embedded on the plane without cross edges. Complete graph K_4 is planer while complete graph K_5 is not. Complete bi-partite graph $K_{3,3}$ is not a planar graph.

Representation of graphs

Graphs, to be processed by algorithms, need to be represented in data structures accessible by the algorithms. There are two types data structures for graphs: adjacency list and adjacency matrix.

Let $G = (V, E)$ be a graph. An *adjacency list* representation of G is a list L_G of lists, such that each element $L_G(u)$ corresponding to the vertex u in the graph, such that $L_G(u) = (v_1, \ldots, v_{k_u})$, where $v_i \in V$, $1 \leq i \leq k_u$, are all neighbors of u. Edges emitting from vertex u can be identified by random access to $L_G(u)$ from which edges (u, v_i), $1 \leq i \leq k_u$, can be sequentially found.

Another representation of graphs is *adjacency matrix*. For graph $G = (V, E)$, this is a $|V| \times |V|$ matrix M_G with entry $M_G(u, v) \in \{0, 1\}$, where the boolean value represents if edge (u, v) is present in the graph. In the case of weighted graph, $M_G(u, v)$ can represent the numerical value for the weight of corresponding edge (u, v). In contrast to the adjacency list, any edge (u, v) can be found in time $O(1)$ time, while it may take more than a constant time to find an edge in the adjacency list. However, adjacency list has advantage over adjacency matrix to reduce memory space for graphs that are sparse, e.g., $|E| = O(|V|)$.

4.1 Recursive definitions of graphs

Graphs are mathematical objects, whose "size" is not limited by a certain number. So much like mathematical object lists, graphs they can be recursively defined. We do this starting from a simpler type of graphs: trees.

Definition 4. $\mathcal{T}$ is the set of pairs defined only by the following generating rules:

(1) Pair $(\{v\}, \emptyset) \in \mathcal{T}$;
(2) If pair $(V, E) \in \mathcal{T}$, $u \in V$, and $v \notin V$, then $\big(V \cup \{v\}, E \cup \{(u, v)\}\big) \in \mathcal{T}$.

In the above definition, each pair (V, E) in $\mathcal{T}$ intends to represent a tree with vertex set V and edge set E. Rule (1) generates a single vertex as a tree without any edge; while rule (2), in a recursive fashion, adds a new vertex v along with a new edge (u, v) to an existing tree to generate a new tree, where vertex u can be considered as the "parent" of vertex v (if you want to record how every edge is generated). This way every vertex in every genrated tree has exactly one "parent" except for the first vertex which does not have a "parent". This "parenting" pattern ensures every pair of vertex set and edge set forms indeed a tree (understood in the conventional way).

Proposition 1. $\mathcal{T}$ *in* Definition 4 *is exactly the set of all non-directed trees.*

How would you modify the Definition 4 for directed trees?

Now we extend the recursive definition for trees to a definition for graphs.

Definition 5. $\mathcal{G}$ is the set of pairs defined only by the following generating rules.

(1) Pair $(\{v\}, \emptyset) \in \mathcal{G}$;
(2) If pair $(V, E) \in \mathcal{G}$, $U \subseteq V$, and $v \notin V$, then pair $(V', E') \in \mathcal{G}$, where $V' = V \cup \{v\}, E' = E \cup \{(u, v) : u \in U\}$.

In the above definition, each pair intends to represents a *graph* with vertex set V and edge set E. Different from Definition 4, to generate a new graph, a new vertex v is added to an existing graph along with zero, one, or more new edges, i.e., $\{(u, v) : u \in U\}$ established between v and all vertices u in U, where U us the chosen subset of vertices in the existing graph.

Proposition 2. $\mathcal{G}$ *in* Definition 5 *is exactly the set of all non-directed graphs.*

How would you modify the Definition 5 for directed graphs?

Rigorous proofs for Propositions 1 and 2 can be done by induction on the cardinality $n = |V|$. These are left as exercises.

Now we turn out attention to subgraphs of desired properties within a given graph $G = (V, E)$. There are two kinds of subgraphs, one called *spanning subgraphs* if their vertex set is required to be the same as V, and the other not necessarily. For example, the set of all trees within a given graph can be defined recursively as well.

Let $G = (V, E)$ be a graph, the set of all trees within G, denoted with $\mathcal{T}_G$ can be defined by the following rules.

(1) If $v \in V$, then pair $(\{v\}, \emptyset) \in \mathcal{T}_G$;
(2) If pair $(U, F) \in \mathcal{T}_G$, $u \in U$, $v \in V \backslash U$, $(u, v) \in E$, then
pair $\big(V \cup \{v\}, E \cup \{(u, v)\}\big) \in \mathcal{T}_G$.

Definition 6. Let pair $(U, F) \in \mathcal{T}_G$. (U, F) is a *spanning tree* if $U = V$.

Can you give a recursive definition for all simple paths in a graph?

Recursive definitions for graphs and subgraphs may play an important rules in algorithmic solutions to finding subgraphs of some desired property. This is because a recursive definition of such subgraphs from sub-cases can lead to effective methods for algorithm design (such as dynamic programming and even greedy algorithms), if a desired property is quantifiable.

4.2 Depth-first search

We now discuss how to traverse a graph. That is, to visit vertices or edges of desired properties or to explore the structure of the whole graph. Unlike search through a list, however, traversal of a graph can be tricky due to the non-linear nature of graphs. Algorithmically we would like to avoid making unnecessary revisit vertices or edges that have already been visited. Simple approach to this issue is possible via the recursive definition of graphs.

According to Definition 5, any defined graph (V', E') can be decomposed into the following components:

(1) a subgraph graph (V, E);
(2) a vertex $v \in V' \backslash V$;

(3) a subset $U \subseteq V$;
(4) a subset of edges $\{(v, u) : u \in U\} \subseteq E' \backslash E$

Therefore, a simple strategy for traversal on graph (V', E') is to take the following steps recursively

(1) visiting vertex v;
(2) for all edges $(v, u) \in E' \backslash E$, walk through edge (v, u), then visit u recursively, if u has not been visited.

Depending on how step (2) "for all edges $(v, u) \in E' \backslash E$" is implemented, there are two different approaches for the recursive traversal idea. One approach is to walk through all edges (v, u) one after another and then move on from every visited u, recursively, which is known as *breadth-first-search* (BFS). Another approach is to walk through one edge (v, u) and then move on from vertex u, recursively, which is termed as *depth-first search* (DFS). We will focus our discussion on DFS.

The follow pseudocode outline the typical DFS traversal on graph `G`. Boolean variable `visited` asserts if the vertex in the argument has been visited. The `for` loop in line 2 and steps within its body shows that the visiting process always has the priority to go "deeper" instead of going "broader", which is the strategy of BFS.

```
function dfs(G, x);
 1. visited(x) = true;
 2. for each edge (x, y) in G
 3.   if not visited(y)
 4.      dfs(G, y);
```

Note that function `dfs` starts traversal from the given vertex `x` and will eventually visit all (and only) vertices reachable from `x`. This means, if the graph `G` is not connected, not all disconnected components can be visited. Therefore, we need the following main process.

```
function dfs-main(G);  \\ main function
 1. for each vertex x in G
 2.   visited(x) = false;
 3. for each vertex x in G
 4.   if not visited(x)
 5.      dfs(G, x);
```

Additional information can be made available from the traversal process, often information about the time when a vertex is visited can be acquired. The following

enhanced `dfs` function gives time stamps before and after every vertex `v` is visited. Time stamps can be used to determined the topological relationships among vertices thus the structure of the graph being visited. The variable `parent` is used to to construct a DFS tree.

```
function dfs(G, x);   \\ dfs enhanced with time stamps
 1. visited(x) = true;
 2. pre(x) = time_stamp;
 3. time_stamp = time_stamp + 1;
 4. for each edge (x, y) in G
 5.   if not visited(y)
 6.     parent(y) = x;
 6.     dfs(G, y);
 7. post(x) = time_stamp;
 8. time_stamp = time_stamp + 1;
```

DFS tree on non-directed graphs

We first consider using the above DFS pseudocode onto non-directed graphs.

On the given graph `G`, the *DFS tree* is a tree *with directed edges* over vertices in `G`, such that it consists of the directed edge `(x,y)` if and only if `visited(y)` is checked with the false value in line 5 of function `ts-dfs`. By `parent(y)=x`, the search gives the information of the corresponding tree edge `(x,y)`. The root of the DFS tree is the first vertex `x` whose `visited(x)` is checked with the false value in line 4 of function `nfs-main`. In the case when graph `G` consists of more than one disconnected component, the DFS is actually a it DFS forest consisting DFS trees whose number is the same as that of disconnected components in graph `G`. The root of every such tree is a vertex whose `visited(u)` is checked with the false value in step 4 of function `nfs-main`.

It also needs to be pointed out that the DFS tree (forest) depends on which vertex is first visited and which neighboring vertex is visited next to break a tie - if there is. For example, vertices can be visited in the alphabetical order when there are more than one option.

Time stamps can be used to monitor the dynamic status of vertices during a DFS process. A vertex without `pre` and `post` time stamps suggests that it has not been visited. A vertex with a `pre` value but not a `post` value says that the traversal has gone through this vertex but not all of those reachable from this vertex. A valid

`post` value asserts that all reachable vertices from it have been visited. Therefore, `pre` and `post` time stamps of vertices provide information about relationships of vertices in the DFS tree.

Proposition 3. *Let x and y be two vertices in a DFS forest. Then*

(1) Interval $[\texttt{pre}(x), \texttt{post}(x)]$ contains interval $[\texttt{pre}(y), \texttt{post}(y)]$ if and only if x is an ancestor of y within a DFS tree of the DFS forest;
(2) Interval $[\texttt{pre}(x), \texttt{post}(x)]$ disjoins with interval $[\texttt{pre}(y), \texttt{post}(y)]$ if and only if x is a sibling of y within a DFS tree or across two DFS trees of the DFS forest.

Edges present on the DFS forest are derived from some edges in the original graph that the DFS is applied on. These edges are called *tree edges*. The rest of the edges in the graph are called *back edges*. Intuitively, these are edges that are probed but not traveled through by the DFS process as they would lead to vertices already visited.

DFS tree on directed graphs

DFS on directed graphs can use the algorithm `dis-main` as well. Because edges in a directed graph are of directions, it sometimes causes confusion with its DFS forest whose edges are all directed. In particular, a DFS forest for a directed graph is still constructed to contain all (directed) edges that are probed and followed through at lines 4 and 5 of function `dfs`. Edges in the directed graph are partitioned into four categories of edges:

(1) Tree edges: edges forming the DFS forest;
(2) Back edges: edges that would connect a descendent to an ancestor in a DFS tree in the DFS forest;
(3) Forward edges: edges that would connect an ancestor to a descendent in a DFS tree in the DFS forest;
(4) Cross edges: edges that would connect two siblings in a DFS tree or across two DFS trees of the DFS forest;

It is interesting to ask what relationships hold between the `pre` and `post` time stamps intervals of the two vertices involved in each of these four types of edges. In addition, `post` values can be used to determine if there is a directed path between two vertices in a directed graph. This is particularly useful to answer such a question in directed acyclic graphs (DAGs).

Proposition 4. *Let x and y be two vertices in directed acyclic graph G upon which function* `dfs` *is applied. If there is a directed path from x to y, then* $\texttt{post}(x) > \texttt{post}(y)$.

Proof: Let $x \rightsquigarrow y$ denote a directed path from x to y. We should consider both cases that vertex x is visited before and after vertex y is visited.

(1) $\texttt{pre}(x) < \texttt{pre}(y)$, i.e., x is visited before y is. Since by assumption of the directed path $x \rightsquigarrow y$, y is reachable from x. Therefore $\texttt{post}(x) > \texttt{post}(y)$.

(2) $\texttt{pre}(y) < \texttt{pre}(x)$, i.e., y is visited before x is. If $\texttt{post}(x) < \texttt{post}(y)$, then interval $[\texttt{pre}(y), \texttt{post}(y)]$ contains interval $[\texttt{pre}(x), \texttt{post}(x)]$ and y is an ancestor of x in the DFS forest by Proposition 3. Thus there must be a directed path $y \rightsquigarrow x$ in G, which, together with the condition that there is a directed path $x \rightsquigarrow y$ from x to y, contradicting that G is a DAG. □

We now estimate the time complexity for `dfs-main` together with `dfs`. Let the input graph be $G = (V, E)$. The main body uses linear time $O(|V|)$ to initialize variable $\texttt{visited}(v)$ to `false` for every vertex v. Variable $\texttt{visited}(v)$ is set `true` once also. Though every vertex v may be probed a number of times, this number sum across all vertices is proportional to the number of edges because each probe is via an edge and every edge is probed only once (or twice if the graph is non-directed). So the total time for `dfs-main` together with `dfs` is $O(|V|+|E|)$, a linear function in both number of vertices and number of edges.

4.3 Applications of DFS

DFS is a simple graph traversal algorithmic strategy yet can be very effective in solving a number of computational problems on graphs. For example, with a slight modification, the function `dfs` can be used to solve the following problems (in the order of increasing difficulty) and their variants:

(1) determine if the input graph is connected;
(2) determine if two given vertices belong to the same connected components;
(3) determine if the input graph contains a cycle;
(4) find a topological order of vertices on the input DAG;
(5) find the strongly connected components of the input graph;
(6) find single-source shortest paths on the input DAG;

Specifically, to solve problem (1), it suffices to see if function `dfs` is called more than once from line 5 within `dfs-main`. Problem (2) is to determine, given

two vertices, e.g., s and t in a non-directed graph, if $s \rightsquigarrow t$. This can be determined by calling `dfs`(s) and adding checking statement `if y equals t` to line 5 in `dfs` or examining if `visited`(t) = `true` after the search terminated.

For problem (3), in general, a back edge during DFS process suggests the existence of a cycle in the graph. For non-directed graphs, a back edge can simply be identified once the outcome of line 5 in `dfs` is true, with exception, however, that `y` is the parent of `x`. The last condition can be simply checked with a statement `if y equals parent(x)`. On directed graphs, encountering a vertex that has been visited does not necessarily imply a back edge, as it could be a forward or a cross edge. Elimination of both these possibilities can be done by checking time stamps of the involved vertices. Specifically, `(x,y)` is a back edge if `pre(y)<pre(x)` and `post(y)` has no value.

In the rest of this chapter, we will give details to solve problems (4) – (6): topological sort of DAG, strongly connected components, and single-source shortest paths on DAGs.

Topological sort of DAG

A topological order of vertices in a DAG $G = (V, E)$ is a sorted order of vertices such that for every two vertices $u, v \in V$, u proceeds v if there is a directed path $u \rightsquigarrow v$. Note that the directed path is a sufficient, not a necessary condition, for the precedence to hold. That is, u can precedes v in the sorted order even if neither a path $u \rightsquigarrow v$ nor a path $v \rightsquigarrow u$ present in the graph.

Since a directed path from one vertex to another determines their sorted positions, by Proposition 4, this appears to pertain to the relationship of their `post` time stamps. Indeed, once the algorithm `dfs-main` is applied on the input DAG, one can use `post` time stamps to arrange vertices to a topological order.

Theorem 3. *Let G be a DAG. A decreasing order of* `post` *time stamps yields a topological sort order of the vertices.*

Proof: Let x and y be two vertices in G and `post`$(x) >$ `post`(y). Suppose arranging x before y violates their topological relationship; then there must be a directed path $y \rightsquigarrow x$. However, by Proposition 4, this implies `post`$(y) >$ `post`(x), contradicts. □

Application of the DFS to find topological order of an DAG $G = (V, E)$ has the same time complexity as the DFS itself. That is $O(|V| + |E|)$.

Strongly connected components SCC

In graph theory, a *connected component* in a non-directed graph is a maximal subgraph in which every pair vertices are connected with paths. Because the number of connected components corresponds to the number of trees in a DFS forest, such components can be easily determined with a slight modification to the main body `dfs-main` of the DFS where the each vertex x identified with $\texttt{visited}(x) = \texttt{false}$ in the `for` loop leads a DFS tree (thus a connected component).

For directed graphs, the notion of *strongly connected component* (SCC) is more interesting. An SCC is a maximal subgraph of a directed graph in which there is a directed path for every vertex to every other vertex. The above idea for determining connected components in non-directed graphs may not be applicable to SCC as the DFS only traverses a a non-directed graph via one-way, not two-way, connectivity. However, DFS can still be adopted to find SCCs from directed graphs and we discuss how to accomplish this in the following.

Assume the DFS is applied on the input directed graph $G = (V, E)$. Let T be a DFS tree in the corresponding DFS forest, with its root r having the highest `post` time stamp. It is clear that there is a path $r \rightsquigarrow x$ from the root r to every other vertex x in tree T. In addition, vertices in T may belong to one or more SCC. It is also worth noting that any SCC of graph G containing vertex r can only contains vertices from T.

Let C be an SCC in the graph G that contains r. Then there should be directed path from every vertex $x \in C$ to vertex r which is the root of T. Let G^T be the transposed graph of G (i.e., graph G with all edge directions reversed). We claim:

Proposition 5. *An application of the DFS on G^T starting from vertex r will produce an DFS tree T' that contains exactly the vertices in SCC C.*

To see the correctness of this claim, we point that all vertices in C should be included in T', simply because all vertices in C have directed paths to r in G and these paths allow them to be included in DFS tree T' rooted at r when the DFS is applied on the transposed graph G^T.

On the other hand, if T' includes any vertex $x \notin C$, it implies that $x \rightsquigarrow r$ in graph G and $\texttt{post}(x) > \texttt{post}(r)$, contradicting that r is the root of a DFS tree with the highest `post` time stamps in the DFS forest.

The above discussion leads to the following correct algorithm for finding SCCs on directed graphs.

```
function strongly-connected component(G);
   1. call dfs-main(G);
   2. let G^T be the transposed graph of G;
   3. in decreasing order of post(x), x in G
   4.    call dfs-main(G^T);
   5. return DFS forest of G^T;
```

The *post* time stamp values in line 3 are from the first call to `dfs-main`(G). The second call to `dfs-main`(G^T) prefer vertices with higher *post* time stamp values when there are options. For every DFS tree obtained in line 4, the subgraph in G formed by all the vertices in the DFS tree is an SCC.

To transpose graph G into G^T by reversing all edge directions, the following pseudocode, running in linear time $O(|V| + |E|)$, does the work by storing the graph G in an adjacency list `L` instead of an adjacency matrix that would have incurred complexity $\Theta(|V|^2)$. The transposed graph G^T is then stored in adjacency list `L'`.

```
function transpose-adjacency-list(L,L',n);
   1. for i = 1 to n     // for every vertex i
   2.   if L[i] is not empty
   3.      j = first vertex of L[i];
   4.      remove first vertex from L[i];
   5.      append i to last vertex of L'[j];
```

As the DFS process takes linear time $O(|V|+|E|)$, the total time complexity of the algorithm to find strongly connected components remains linear $O(|V|+|E|)$.

4.4 Shortest paths problems

Let $G = (V, E)$ be a graph. Recall a path on the graph is a sequence of k connected edges $(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_{k+1})$ for some $k \geq 1$. Specifically, this is a path from v_1 to v_{k+1}. If there are no $i < j$, $v_i = v_j$, then the path is a simple path. It is not difficult to see that there may be more than one path between two vertices and the number of paths between two given vertices in the graph can be exponential in n, the number of vertices.

As graphs are models for binary relations (represented as edges) among the objects (represented by vertices). So a path is a collection of such relationships that

represent certain higher-order relationship among the involved objects. Sometimes the interest is only to detect if a path exists, e.g., whether two non-directly-related objects can be related in some other way. The path detection is a relatively simple problem since it can be solved by a DFS-like algorithm as we discussed earlier.

More often, however, a path problem may desire finding a path that meets some quality requirement on a quantity associated with the path. Typically, finding a *shortest* (or *longest*) path from one vertex to another vertex (or paths to all other vertices) has been extensively investigated. For this purpose, weighted graphs are investigated instead, whose edges (or vertices or both) are associated numerical values. Note that the length (or weight) of a path we refer to in this setting does not necessarily mean geometric distance. It can also be cost, profit, or other numerical value in path problems formulated from industry, science, or society.

A graph $G = (V, E)$ is *edge-weighted* if it is associated with a function $w : E \to R$. Let $s, t \in V$ be two specific vertices and a path from s to t be $p : s \rightsquigarrow t$ involving edges $\{(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_{k+1})\}$, with $s = v_1$, $t = v_{k+1}$, and $(v_i, v_{i+1}) \in E$ for all $i = 1, 2, \ldots, k$. The *distance* of the path p is defined as $w(p) = \sum_{i=1}^{k} w(v_i, v_{i+1})$, the total weight of edges on the path. Path p is a *shortest path* if $w(p) = \min_{p': s \rightsquigarrow t} w(p')$. That is, $w(p)$ is the minimum over all paths from s to t.

While finding a shortest path from a *source* vertex (e.g., s) to a *target* vertex (e.g., t) may be the most demanded task, the following property of shortest paths suggests that this task actually need to involve finding shortest paths between other vertices.

Proposition 6. *The subpath of a shortest path is also a shortest path.*

Specially, consider a shortest path $p : x \rightsquigarrow y \rightsquigarrow z$ from vertex x to vertex z going through an intermediate vertex y. We name subpath q for the segment $x \rightsquigarrow y$ on p and subpath r for the segment $y \rightsquigarrow z$ on p such that $w(p) = w(q) + w(r)$. The above proposition claims that the subpath q is a shortest path from x to y and the subpath r is a shortest path from y to z.

Otherwise, there must be a path $q' : x \rightsquigarrow y$ with weight $w(q') < w(q)$. Combining the paths $q' : x \rightsquigarrow y$ and $r : y \rightsquigarrow z$ results in a path from x to z of weight $w(q') + w(r) < w(q) + r(w) = w(p)$, contradicting the assumption p is a shortest path from x to z.

Definition 7. Let $G = (V, E)$ be a graph with edge weight function $w : E \to R$ and let $s \in V$ be a vertex. The *single-source shortest problem* is to find shortest

paths from s to all other vertices in G.

There are a few algorithms that deal with different types of graphs, including graph without cycles, graphs without negative weights, and graph that may contain negative weighted cycles. We first discuss for the most restricted graphs – DAGs.

Single-source shortest paths on DAG

Without loss of generality, we assume that the source vertex s has a directed path to every other vertices, as we will see that those vertices not reachable from s will result in the ∞ value for their distances from s. We use variable $\texttt{dist}(v)$ to record the distance from s to every vertex v computed, $v \neq s$. Specifically, the value of $\texttt{dist}(v)$ can be computed with the following recursive scheme

$$\texttt{dist}(v) = \min_{(x,v)\in V} \Big\{ \texttt{dist}(x) + w(x,v) \Big\} \tag{4.1}$$

with $\texttt{dist}(s)$ = 0. The recurrence holds because because one of these vertices x has to be the immediate predecessor of v on a shortest path from s to v. Note that the operand $\min$ in (4.1) requires to loop through all edges `(x,v)` by any algorithm. This in general is done with the following scheme.

Definition 8. Let some path from source s to vertex v be such that it has the distance $\texttt{dist}(v)$. Edge (x, v), if exists, is called being *relaxed* if $\texttt{dist}(v)$ is updated with

$$\texttt{dist}(v) = \min \begin{cases} \texttt{dist}(v) \\ \texttt{dist}(x) + w(x,v) \end{cases}$$

The term *relaxation* may sound a bit misleading but it should become intuitively clear in the following context. A path from s to v without considering going through an edge (x, v) is actually *constrained*; once the option of going through edge (x, v) is considered, the constraint is relaxed (through relaxing edge (x, v)).

Definition 9. For every vertex $v \in V$, $d^*(v)$ denotes the real shortest distance between the source s and v. The variable $\texttt{dist}(v)$ is called *finalized* if $\texttt{dist}(v) = d^*(v)$.

According to equation (4.1), after $\texttt{dist}(x)$ is finalized for all vertices x, $\texttt{dist}(v)$ is finalized with the $\min$ operand. Therefore, the computation of `dist` values for vertices may proceed in a topological sort order.

We will show several different algorithmic techniques to update `dist` for all vertices depending on how the `dist` value for one vertex is used for computing `dist` value of another vertex, as shown in the following. All these algorithms assume the initializations of $\texttt{dist}(v) = \infty$ and $\texttt{dist}(s) = 0$.

The first algorithm follows a topological sort order.

```
function DAG-shortest-paths(G,s)
 1. pick x in the topological sort order;
 2.   for every edge (x,v)
 3.      if dist(v) > dist(x) + w(x,v)
 4.         dist(v) = dist(x) + w(x,v);
 5.         parent(v) = x;
```

where variable $\texttt{parent}(y)$ points to y's predecessor in the found path (so far).

Function `shortest-paths-DAG` relaxes edges (x, y) after encountered with vertex x. The order of edge relaxations can be subtly different, as shown in the following algorithm.

```
function DAG-shortest-paths-DP(G,s)
 1. pick v (not s) in the topological sort order;
 2.   for every edge (x,v)
 3.      if dist(v) > dist(x) + w(x,v)
 4.         dist(v) = dist(x) + w(x,v);
 5.         parent(v) = x;
```

Algorithm `shortest-paths-DAG-DP` actually bears the spirit of dynamic programming where a data object is computed from other computed data objects according to a recursive formula, such as equation (4.1). Therefore, such a process can also be accomplished recursively. This leads to a different version of algorithm to compute `dist` for vertices. It, however, does not rely on a topological sort order of vertices! Assume that initially variable $\texttt{finalized}(s) = 0$ and $\texttt{dist}(s) = 0$.

```
function DAG-shortest-paths-DP-recursive(v)
 1. for every edge (x,v)
 2.    if finalized(x) equals false
 3.       DAG-shortest-paths-DP-recursive(x);
 4.    if dist(v) > dist(x) + w(x,v)
 5.       dist(v) = dist(x) + w(x,v);
 6.       parent(v) = x;
```

```
7. finalized(v) = true;
```

where variable `finalized`(v) is used to remember if distance `dist`(v) has been finalized or not. Algorithm `DAG-shortest-paths-DP-recursive` is called a *memoized* function, a type of algorithm especially useful for implementation of dynamic programming solutions.

It is not difficult to see that all the three outlined algorithms run in linear time $O(|E| + |V|)$ simply because in all algorithms, every edge is examined once or at most twice in the case of non-directed graphs.

Dijkstra's algorithm

More common scenarios of graphs are that they may contain cycles, which describes perhaps more sophisticated relationships among objects modeled in the graph. To solve the single-source shortest paths problem in this case, none of the previous three algorithms is applicable. This is because they have all relied on the topological precedency of vertices made possible by DAGs. However, the idea built in these algorithm may inspire one for solving the problem on general graphs but without negative edge weights.

Recall the previous three algorithms on DAGs are made simply by following the topological order to finalize their `dist`values. The strategy is sound because the `dist` value computed for a vertex will not have an impact on any of its predecessors in the topological order. For graphs with non-negative edge weights, a similar strategy may exist. That is to follow the non-decreasing order of the finalized `dist`values of vertices. Consider two vertices x and y with $d^*(x) \leq d^*(y)$. Then the order to pick vertices to update their `dist`values should have x precede y instead of the other way around since the shortest path from s to x should not have y being an intermediate vertex on the path. But how do we know which vertex has higher finalized `dist` value before it is finalized? The following lemma offers a technical solution.

Lemma 1. *Let $A \subseteq V$ be a subset of all vertices x, for which* `dist`$(x) = d^*(x)$ *and all edges emitting x have been relaxed. Let $B = V \backslash A$. Let $v \in B$. If* `dist`$(v) \leq$ `dist`(u) *for every $u \in B$, then* `dist`$(v) = d^*(v)$.

Proof: Assume `dist`(v) is not finalized. Then there must be an edge (w, z), with both $w, z \in B$, whose relaxation will decrease `dist`(v), implying a path

$$s \rightsquigarrow x \rightarrow y \rightsquigarrow w \rightarrow z \rightsquigarrow v$$

where (x, y) is an edge with $x \in A$ and $y \in B$. According to the given condition, (x, y) has been relaxed. In order for the current value $\texttt{dist}(v)$ to reduce via this path, we need $\texttt{dist}(y) + w(p) < \texttt{dist}(v)$, where subpath $p : y \rightsquigarrow w \rightarrow z \rightsquigarrow v$. However, because all edges have non-negative weights, the above inequality requires $\texttt{dist}(y) <\texttt{dist}(v)$, which contradicts the property that v satisfies, i.e., the current value $\texttt{dist}(v)$ is the smallest value among vertices in B. □

Now we are ready for the Dijkstra's algorithm.

```
function Dijkstra(G,s)
   1. for all v in G
   2.    dist(v) = infinity;
   3. dist(s) = 0;
   4. Q = V;                // Q is a priority queue
   5. while Q not empty     // with dist as key
   6.    x = dequeue(Q);
   7.    for every edge (x,v) in G
   8.       if dist(v) > dist(x) + w(x,v);
   9.          dist(v) = dist(x) + w(x,v);
   10.         parent(v) = x;
   11.return parent;
```

The algorithm implements the idea in the Lemma 1 that finalizes the `dist` for a vertex that has the smallest value among those yet to be finalized. Note that set B in the lemma is the set of vertices in Q, a priority queue for which the dequeue operation for vertices is based on their `dist` values. In particular, the first vertex de-queued is the source vertex s that has `dist` value 0. Therefore we concludes with the following theorem without giving a proof:

Theorem 4. *Algorithm* `Dijkstra` *correctly computes shortest paths from the source s to all other vertices. In particular, the order that vertices to finalize their* `dist` *values are in the non-decreasing order of the* `dist` *values.*

Algorithm `Dijkstra` works correctly on graphs without negative-weighted edges. This is because at each step, it finalizes `dist` for the vertex that has the smallest of `dist` at that step with the assumption that this value will not be improved at a later step. Validity of the assumption is ensured by the graph that does not contain negative weight edges.

It is natural to ask if algorithm *Dijkstra* is still valid on graphs with negative-weighted edges. The answer is "no". Without doubt, any vertex already de-queued may still be updated on its `dist` value (with line 8). In particular, negative edges entering to the de-queued vertex would keep the vertex's `dist` value updated. However, since a de-queued vertex cannot be en-queued and de-queued again, the updated `dist`value of the vertex will fail to be used to update `dist`values for its neighbors. To correctly accommodate negative-weighted edges, we need a different algorithm.

Finally, we examine the time complexity of algorithm `Dijkstra`. It is clear that the total time is $O(|V| \times T_{deq}(|V|) + |E|)$, where $T_{deq}(|V|)$ is the time for dequeuing a vertex from the priority queue Q originally storing V, thus of size $|V|$. In a later section, we will discuss time complexities of various operations on a priority queue based on its implementations. For now, we may assume that $T_{deq}(n) = O(\log_2 n)$ on a priority queue of size at most n. So the time complexity of algorithm `Dijkstra` is $O(|V| \log_2 |V| + |E|)$.

Dealing with negative weights and cycles

When the input graph contains negative weighted edges, finding algorithms for the shortest paths problem may require a different perspective. We first consider shortest paths that are simple paths.

Let $p : s \rightsquigarrow v$ be a shortest path with $\texttt{dist}(v) = d^*(v)$. If p is a simply path, it consists of at most $|V| - 1$ edges. Let these edges are

$$(v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, v_{k+1})$$

where $s = v_1$ and $v_{k+1} = v$. It is clear that each one of these edges needs to be relaxed before $\texttt{dist}(v) = d^*(v)$ – it is just that we may not have control on the order of these edges being relaxed. However, if all edges are relaxed, the first edge (v_1, v_2) on the path p, where $v_1 = s$, will be relaxed and the `dist`value for v_2 be finalized. Afterward, if all edges are relaxed again, the second edge (v_2, v_3) will be relaxed and the `dist`value for v_2 be finalized. Now it is clear that with $|V| - 1$ rounds of relaxations, each relaxing all edges, should get `dist` values of all vertices finalized.

Lemma 2. *$n-1$ rounds of edge relaxations settle $\texttt{dist}(v) = d^*(v)$ for all vertices v in a graph of n vertices without negative cycles.*

This idea has lead to Bellman-Ford algorithm.

```
function Bellman-Ford(G,s);
  1. for every vertex v in G
  2.    dist(v) = infinity;
  3. dist(s) = 0;
  4. for round = 1 to |V|-1
  5.    for every edge (u, v) in G
  6.       if dist(v) > dist(u) + w(u, v)
  7.          dist(v) = dist(u) + w(u, v);
  8.          parent(v) = u;
  9. If dist(v) > dist(u) + w(u, v) for any (u,v)
  10.    return false;
  11.else
  12.   return true;
```

Figure K illustrates algorithm `Bellman-Ford`, with dynamic changes of `dist` values for all vertices at different rounds of relaxations. And it is not difficult to see that the time complexity of algorithm `Bellman-Ford` is $O(|V||E|)$, as every round relaxes all edges.

Since the input graph may contain negative-weighted edges, they may potentially form a negative cycle, which is a cycle of total weight below 0. If such a cycle is reachable from the source s, some of the vertices may have paths from s with `dist` values approaching $-\infty$ by looping through the cycle as many times as possible. Line 9 detects if a negative cycle may present in the graph. It does so by checking if, after $|V| - 1$ rounds of relaxation, not all vertices have their `dist` values finalized.

Theorem 5. *Line 10 in algorithm* `Bellma-Ford` *is executed if and only if there is a negative cycle in the graph* `G`*.*

Proof: Assume that there is a negative cycle consisting of edges $(v_0, v_1), \ldots, (v_{k-1}, v_k)$, where $v_0 = v_k$, such that the total weight

$$\sum_{i=0}^{k-1} w(v_i, v_{i+1}) < 0 \tag{4.2}$$

If line 10 is not executed, then $\texttt{dist}(u) + w(u, v) \geq \texttt{dist}(v)$ for every edge $(u, v) \in E$. Apply the this inequality to all edges $(v_0, v_1), \ldots, (v_{k-1}, v_k)$ in the

cycle, the following set of inequalities can be obtained:

$$\begin{cases} \texttt{dist}(v_0) + w(v_0, v_1) \geq \texttt{dist}(v_1) \\ \texttt{dist}(v_1) + w(v_1, v_2) \geq \texttt{dist}(v_2) \\ \ldots \\ \texttt{dist}(v_{k-1}) + w(v_{k-1}, v_k) \geq \texttt{dist}(v_k) \end{cases}$$

Because $v_0 = v_k$, sum of the above inequalities lead to

$$\sum_{i=0}^{k-1} w(v_i, v_{i+1}) \geq 0$$

contradicting inequality (4.2).

On the other hand, assume there is no negative cycle in the graph. By Lemma 2, there should not inequality $\texttt{dist}(u) + w(u, v) \geq \texttt{dist}(v)$ for any (u, v). after $|V| - 1$ rounds of relaxations. Thus line 10 in algorithm `Bellman-Ford` will not be executed. □