Our standing at graph algorithms:

1. Search Strategies (Chapter)
2. Combinatorial Search(Chapter)
3. Advanced Graph Algorithm(Current)
4. Graph Problem Patterns(Future Chapter)

This chapter is more to apply the basic search strategies and two advanced algorithm design methodologies–Dynamic Programming and Greedy Algorithms– on a variety of classical graph problems:

- Cycle Detection (Section -1.1), Topological Sort(Section -1.2), and Connected Components(Section -1.3) which all require a through understanding to properties of basic graph search, especially Depth-first graph search.
- On the other hand, Minimum Spanning Tree (MST) and Shortest Path Algorithm on the entails our mastering of Breath-first Graph Search.
- Moreover, to achieve better efficiency, Dynamic Programming and Greedy Algorithms has to be leveraged in the graph search process. For example, Bellman-Ford algorithm uses the Dynamic Programming to avoid recomputing intermediate paths while searching the shortest paths from a single source to all other targets. The classical Prim's and Kruskal's MST algorithm both demonstrates how greedy algorithm can be applied, each in a different way.

-1.1 Cycle Detection

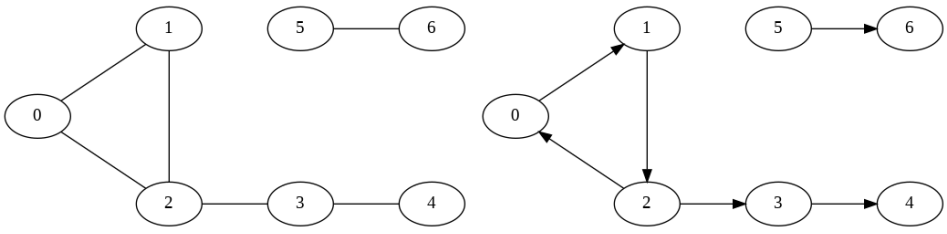


Figure 1: In both the Undirect and Directed Graph, $(0, 1, 2, 0)$ is a cycle.

Problem Definition Detect cycles in both directed and undirected graph. Specifically, given a path with $k+1$ vertices, denoted as $v_0, v_1, ..., v_k$ in graph G:

1. When G is directed: a cycle is formed if $v_0 = v_k$ and the path contains at least one edge. For example, there is a cycle $0, 1, 2, 0$ shown in the directed graph of Fig. 1.

2. When G is undirected: the path forms a cycle only if $v_0 = v_k$ and the path length is at least three (i.e., there are at least three distinct vertices within the path). For example, in the undirected graph of Fig. 1, we couldn't say $(0, 2)$ is a cycle even though there is a path $0, 2, 0$, but the path $0, 2, 1, 0$ is as the path length ≥ 3.

DFS to Solve Cycle Detection Recall the process of DFS graph search where a vertex has three possible states–white, gray, and black. A back edge appears while we reach to an adjacent vertex v_i which is in gray state from current vertex u. If we connect u back to its ancestor v_i, we find our cycle if the graph is directed. When the graph is undirected, we have one more step to do: avoiding cycle of length one which is any existing edge within the graph. We can easily achieve this by tracking the predecessor p of the exploring vertex during the search, and making sure the predecessor is not the same as the current vertex: $p \neq u$.

Implementation We define a function `hasCycle` with g as the adjacent list of graph, *state* as a list to track state for each vertex, and s as the exploring vertex. The function returns a boolean value to indicate if there is a cycle or not. The function is essentially a DFS graph search along with an extra condition check on the back edge.

```
def hasCycle(g, s, state):
  '''convert dfs to check cycle'''
  state[s] = STATE.gray # first be visited
  for v in g[s]:
    if state[v] == STATE.white:
      if hasCycle(g, v, state):
        print(f'Cycle found at node {v}.')
        return True
    elif state[v] == STATE.gray: # aback edge
      print(f'Cycle starts at node {v}.')
      return True
    else:
        pass
    state[s] = STATE.black # mark it as complete
    return False
```

Because a graph can be disconnected with multiple components, we run `hasCycle` on each unvisited vertex within the graph in a main function `cycleDetect`.

```
def cycleDetect(g):
  '''cycle detect in directed graph'''
  n = len(g)
  state = [STATE.white] * n
  for i in range(n):
    if state[i] == STATE.white:
      if hasCycle(g, i, state):
        return True
  return False
```

In the case of undirected graph, we add another variable `p` to track the predecessor along search in `hasCycle`. p will first be initialized to -1 because the root in the rooted search tree has no predecessor (or ancestor).

```
def hasCycle(g, s, p, state):
  '''convert dfs to check cycle'''
  state[s] = STATE.gray # first be visited
  for v in g[s]:
      if state[v] == STATE.white:
        if hasCycle(g, v, s, state):
          print(f'Cycle found at node {v}.')
          return True
      elif state[v] == STATE.gray and v != p: # aback edge
          print(f'Cycle starts at node {v}.')
          return True

  state[s] = STATE.black # mark it as complete
  return False
```

Please check the source code for full implementation and try out the examples.

How to find all cycles? First, we need to enumerate all paths while searching in order to get all cycles. This requires us to retreat to less efficient search strategy: depth-first tree search. Second, for each path, we find where the cycle starts by comparing each v_i with current vertex u: in directed graph, once $v_i == u$, the cycle is $v_i, v_{i+1}, ..., v_k, v_i$; in undirected graph, the cycle is found only if the length of $v_i, ..., v_k \geq 3$.

-1.2 Topological Sort

Problem Definition In a given Directed Acyclic Graph (DAG) $G = (V, E)$, *topological sort/ordering* of is a linear ordering of the vertices V, such that for each edge $e \in E, e = (u, v)$, u comes before v. If a vertex represents a task to be completed and each directed edge denotes the order between two tasks, then topological sort is a way of linearly ordering a number of tasks in a completable sequence.

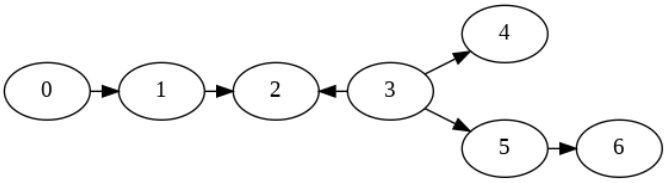


Figure 2: DAG 1

Every DAG has at least one topological ordering. For example, the topological ordering of Fig 2 can be `[0, 1, 3, (2, 4, 5), 6]`, where $(2, 4, 5)$ can be of any order, i.em., $(2, 4, 5), (2, 5, 4), (4, 2, 5), (4, 5, 2), (5, 2, 4), (5, 4, 2)$.

A topological ordering is only possible if there is no cycle existing in the graph. Thus, a cycle detection should be applied first when we are given a possible cyclic graph.

Kahn's algorithm (1962)

In topological sort, the first vertex is always ones with in-degree 0 (a vertex with no incoming edges). A naive algorithm is to decide the first node (with in-degree 0), add it in resulting order S, and remove all outgoing edges from this node. Repeat this process until:

- $V - S$ is empty, i.e., $|S| = |V|$, which indicates we found valid topological ordering.
- no node with 0 in-degree found in the remaining graph $G = (V - S, E^{'})$ where $E^{'}$ are the remaining edges from E after the removal, i.e., $|S| < |V|$, indicating a cycle exists in $V - S$ and no valid answer exists.

For example, with the digraph in Fig. 2, the process is:

```
S       Removed Edges
0, 3 are the in-degree 0 nodes
Add 0      (0, 1)
1, 3 are the current in-degree 0 node
Add 1      (1, 2)
3  is the only in-degree 0 node
Add 3      (3, 2), (3, 4), (3,5)
2, 4, 5 are the in-degree 0 nodes
Add 2
Add 4
Add 5     (5, 6)
6 is the only in-degree 0 node
Add 6
V-S empty, stop
```

In this process, we see that in some time 2, 4, 5 are no in-degree 0 nodes, that is why their orderings can be permutated, resulting multiple topological orderings.

In implementation, instead of removing edges from the graph explicitly, a better option is to track $V - S$ with each vertex's in-degree: whenever a in-degree 0 vertex u is added into S, $\forall v, u \rightarrow v$, decrease the in-degree of v by one. We also keep a queue of the all nodes with in-degree zero Q. Whenever a vertex in $V - S$ is detected with zero in-degree, add it into Q. Accumulatively, the cost of decreasing the in-degree for vertices in $V - S$ is $|E|$ as from the start to end,"all edges are removed." The cost of removing of vertex from $V - S$ is $|V|$ as all nodes are removed at the end. With the initialization of the in-degree for vertices in $V - S$, we have a total of $O(2|E| + |V|)$, i.e., $O(|E| + |V|)$ as the time complexity. Python code:

```
from collections import defaultdict
import heapq
def kahns_topo_sort(g):
  S = []
  V_S =[(0, node) for node in range(len(g))] # initialize node
    with 0 as in-degree
  indegrees = defaultdict(int)
  # Step 1: count the in-degree
  for u in range(len(g)):
    indegrees[u] = 0
  for u in range(len(g)):
    for v in g[u]:
      indegrees[v]+= 1
  print(f'initial indegree : {indegrees}')
  V_S = [(indegree, node) for node, indegree in indegrees.items
    ()]
  heapq.heapify(V_S)

  # Step 2: Kan's algorithm
  while len(V_S) > 0:
    indegree, first_node = V_S.pop(0)
    if indegree != 0: # cycle found, no topological ordering
      return None
    S.append(first_node)
    # Remove edges
    for v in g[first_node]:
      indegrees[v] -= 1
    # update V_S
    for idx, (indegree, node) in enumerate(V_S):
      if indegree != indegrees[node]:
        V_S[idx] = (indegrees[node], node)
    heapq.heapify(V_S)
  return S
```

Calling the function using graph in Fig. 2 gives result:

```
initial indegree : defaultdict(<class 'int'>, {0: 0, 1: 1, 2: 2,
    3: 0, 4: 1, 5: 1, 6: 1})
[0, 1, 3, 2, 4, 5, 6]
```

Linear Topological Sort with Depth-first Graph Search

In depth-first graph search, if there is an edge $u \to v$, the recursive search from v will always be completed ahead of the search of u. With a simple reverse of the finishing ordering of vertices in depth-first graph search, the topological ordering takes $O(|E| + |V|)$ time. The time complexity equates to that of Kahn's algorithm, but this process is more efficient as it does not require the counting and updates of node in-degrees. The whole process is exactly the same as Cycle Detection with additional complete ordering tracking.

First, the code of the DFS is:

```
def dfs(g, s, colors, complete_orders):
  colors[s] = STATE.gray
  no_cycle = True
  for v in g[s]:
    if colors[v] == STATE.white:
      no_cycle = no_cycle and dfs(g, v, colors, complete_orders)
    elif colors[v] == STATE.gray: # a cycle appears
      print(f'Cycle found at node {v}.')
      return False
  colors[s] = STATE.black
  complete_orders.append(s)
  return no_cycle
```

Then main function is:

```
def topo_sort(g):
  n = len(g)
  complete_orders = []
  colors = [STATE.white] * n
  for i in range(n): # run dfs on all the node
    if colors[i] == STATE.white:
      ans = dfs(g, i, colors, complete_orders)
      if not ans:
        print('Cycle found, no topological ordering')
        return None
  return complete_orders[::-1]
```

Call `topo_sort` on the graph, we will have the sorted ordering as:

```
[3, 5, 6, 4, 0, 1, 2]
```

which is another linear topological ordering.

Example: Course Schedule (L210, m)

There are a total of n courses that you have to take. Some courses may have prerequisites, for example course 1 has to be taken before course 0, which is expressed as $[0, 1]$. Given the total number of courses and the prerequisite pairs, return the ordering of courses you should take to finish all courses. If it is impossible to finish, return an empty array.

Analysis Viewing a pair $[u, v]$ as an directed edge $v \rightarrow u$, we have a directed graph with n vertices and we solve the ordering of courses as getting the topological sort of vertices in the resulting digraph.

-1.3 Connected Components

Problem Definition In graph theory, a *connected component*(or simply component) is defined as a subgraph where all vertices are mutually connected, i.e., where there exists a path between any two vertices in it. A graph $G = (V, E)$ is thus composed of separate connected components(sets) which are mutually exclusive and include all the vertices, .i.e., $V = V_0 \cup V_1 \cup ... \cup V_{m-1}, V_i \cap V_{j \neq i} = \emptyset$. A connected component algorithm should be able to cluster vertices of each single connected component. For example,

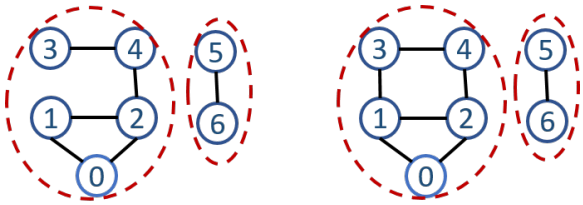


Figure 3: The connected components in undirected graph, each dashed read circle marks a connected component.

the undirected graph in Fig. 3 has two connected components: $\{0, 1, 2, 3, 4\}$ and $\{5, 6\}$.

Given a directed graph,

- the term *Strongly Connected Component (SCC)* or *diconnected* is used to refer to the same definition– where in a SCC any two vertices are reachable to each other by paths. In the leftest directed graph shown in Fig. 4, there is a total of five SCCs: $\{0, 1, 2\}$, $\{3\}$,$\{4\}$, $\{5\}$, and $\{6\}$. Vertex 5 and 6 is only connected in one way, resulting into two separate SCCs.

- ignoring the direction of edges, a *weakly connected component (WCC)* equates to a connected component in the resulting undirected graph.

Cycles and Strongly Connected Components A directed graph is acyclic if and only it has no strongly connected subgraphs with more than one vertex. We call SCCs with at least two vertices nontrivial SCCs. Nontrial SCCs contains at least one directed cycle, and more specifically, nontrivial SCCs is composed of a set of directed cycles as we have observed that there are two directed cycles in our above example and they share at

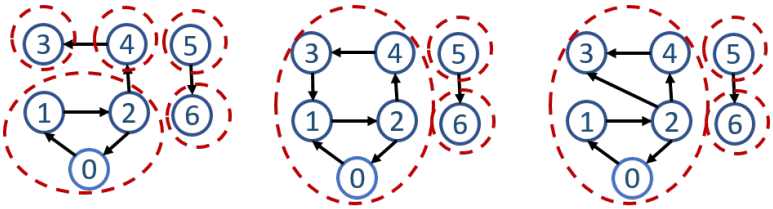


Figure 4: The strongly connected components in directed graph, each dashed read circle marks a strongly connected component.

least one common vertex. The shared common vertex act as "transferring stop" between these directed cycles thus they all compose to one component. Therefore, SCCs algorithms can be indirectly used to detect cycles. If there exists nontrivial SCC, directed graph contains cycles.

-1.3.1 Connected Components Detection

In general, there are two ways to detect connected components in an undirected graph: graph search and union-find, each suits different needs.

Graph Search and Search Tree In undirected graph G, executing a BFS or DFS starting at some vertex u will result in a rooted search tree. As the edges are undirected or bidirectional, all vertices in the search tree belong to the same connected component. To find all connected components, we simply loop through all vertices V, for each vertex u:

- if u is not visited yet, we start a new DFS/BFS. Mark all vertices along the traversal as the same component.
- otherwise, u is already included in a previously found connected component, continue.

The time complexity will be $O(|V| + |E|)$ and the space complexity will be $O(|V|)$. Since the code is trivial, we only demonstrate it in the notebook.

Union Find

We represent each connected component as a set. For the exemplary graph in Fig. 3, we have two sets: $0, 1, 2, 3, 4$ and $5, 6$. Unlike the graph-search based approach, where the edges are visited in certain order,in union-find approach, the ordering of edges to be visited can be arbitrary. The algorithm using union-find is:

- Initialize in total $|V|$ sets, one for each vertex V.
- For each edge (u, v) in E, `union` the two sets where vertex u and v previously belongs to.

Implementing it with Python:

```
from collections import defaultdict
def connectedComponent(g):
  n = len(g)
  # initialize disjoint set
  ds = DisjointSet(n)

  for i in range(n):
    for j in g[i]: # for edge i<->j
      ds.union(i, j)
  return ds.get_num_sets(), ds.get_all_sets()
```

How we implement the union-find data structure decides the complexity of this approach. For example, if we use linked list based structure, the complexity will be $O(|E| \times |V|)$ as we traversal $|V|$ edges and each step in worst case can take $O(|V|)$ to find the set that it belongs to. However, if path compression and union by rank is used for optimization, the time complexity could be lowered to $O(|E| \times \log |V|)$.

Dynamic Graph Since union-find has worse time complexity compared with graph search, then why do we care about it? The answer is: if we use graph search, whenever new edges and vertices are added to the graph, we have to rerun the graph search algorithm. Imagine that if we double $|V|$ and E, the worst time complexity will be $O(|V| \times (|V| + |E|)$, bringing up the complexity to polynomial of the number of edges. However, for each additional edge, union-find adds only a single merge operation to address the change, keeping the time complexity unchanged.

In detail, we adapt the union-find structure dynamically. Set up a `dict` to track vertex and its index in the union find. Set `index=0`. When a new edge (u, v) comes, union find includes:

- check if u and v exists in `dict`. If not, (a) add a key-value into the node tracker, (b) append `index` into the list of vertex-set, (c) `index+=1`.
- `find` the sets where u and v belongs to.

Implementation Here we demonstrate how to implement a dynamic connected component detection algorithm. First, convert the graph representation from adjacent list to a list of edges:

```
ug_edges = [(0, 1), (0, 2), (1, 2), (2, 4), (4, 3), (4, 3), (5,
    6)]
```

Then, we implement a class `DynamicConnectedComponent` offering all functions needed.

```
class DynamicConnectedComponent():
  def __init__(self):
    self.ds = DisjointSet(0)
```

```
    self.node_index= defaultdict(int)
    self.index_node = defaultdict(int)
    self.index = 0

  def add_edge(self, u, v):
    if u not in self.node_index:
      self.node_index[u], self.index_node[self.index] = self.index, u
      self.ds.p.append(self.index)
      self.ds.n += 1
      self.index += 1

    if v not in self.node_index:
      self.node_index[v], self.index_node[self.index] = self.index, v
      self.ds.p.append(self.index)
      self.ds.n += 1
      self.index += 1
    u, v = self.node_index[u], self.node_index[v]
    self.ds.union(u, v)
    return

  def get_num_sets(self):
    return self.ds.get_num_sets()

  def get_all_sets(self):
    sets = self.ds.get_all_sets()
    return {self.index_node[key] : set([self.index_node[i] for i in list(value)]) for key, value in sets.items()}
```

Now, to find the connected components dynamically based on incoming edges, we can run:

```
dcc = DynamicConnectedComponent()
for u, v in ug_edges:
  dcc.add_edge(u, v)
dcc.get_num_sets(), dcc.get_all_sets()
```

The output is consistent with previous result, which is:

```
(2, {3: {0, 1, 2, 3, 4}, 6: {5, 6}})
```

Examples

1. 547. Number of Provinces(medium)
2. 128. Longest Consecutive Sequence (hard), union find solution: `https://leetcode.com/problems/longest-consecutive-sequence/discuss/1109808/Python-Clean-Union-Find-with-explanation`

Implement WCC detection algorithm in directed graph?

-1.3.2 Strongly Connected Components

In graph theory, two nodes $u, v \in V$ are called strongly connected iff v is reachable from u and u is reachable from v. If we contract each SCC into a single vertex, the resulting graph will be a DAG. Denoting the contracted DAG as $G^{SCC} = (V^{SCC}, E^{SCC})$, V^{SCC} are vertices of SCCs and E^{SCC} are defined as follows:

(C_1, C_2) is an edge in G^{SCC} iff $\exists u \in C_1, v \in C_2$. (u, v) is an edge in G.

In other words, if there is an edge in G from any node in C_1 to any node in C_2, there is an edge in G^{SCC} from C_1 to C_2.

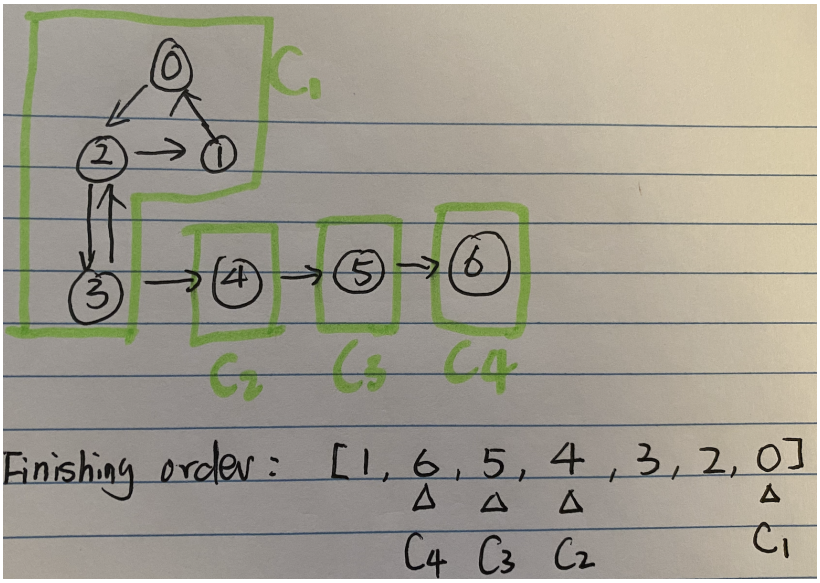


Figure 5: A graph with four SCCs.

Kosaraju's Algorithm If we were to do a DFS in G, and $C_1 \to C_2$ is an edge in G^{SCC}, then at least one vertex in C_1 will finish after all vertices in C_2 being finished. If we first start with vertex 0, the finishing order of of all vertices is [1, 6, 5, 4, 3, 2, 0]. 0 finished later than 4 from C_2, satisfying the claim. If we look purely at the last node from each SCC to turn dark, we get a topological sort of G^{SCC} in reverse([1, 6, 5, 4, 3, 2, 0]), which is $[C_4, C_3, C_2, C_1]$. How to find the last node in each SCC? We can reverse the dfs finishing order, having [0, 2, 3, 4, 5, 6, 1].

If we reverse the order, we have [0, 2, 3, 4, 5, 6, 1]. What happens if we do another round of DFS on the given ordering? First, starting from 0 (last node), we can (1) reach to all vertices in C_1 as they are connected, (2) reach to vertices in C_2 if there exists no edge or edges only from C_1 to C_2 in between. If we can reverse the edges in between, then we can avoid (2) and still keeps (1). The way we do this is: reverse the direction of all edges in graph G. Run DFS on the reversed finishing ordering, then a SCC will include any vertex along the traversal that hasn's been put into a SCC yet. In our example, the process is:

```
0:  find {0, 1, 2, 3}
4: find {4}
5: find {5}
6: find {6}
```

We formalize Kosaraju's algorithm into three steps:

1. Retrieve a reversed finishing order of vertices during DFS L. This step is similar to topological sort in an DAG.

2. Transpose the original graph G to G^T by reversing the directional of edges in G.

3. Run another DFS in L_1 ordering on G^T, any df-search tree starting from a vertex that hasn't been put into a SCC yet make up to another SCC.

Implementation The main function `scc` calls two functions: `topo_sort_scc` and `reverse_graph` to get L and G^T. The topological ordering like function:

```
# DFS traversal with reversed complete orders
def dfs(g, s, colors, complete_orders):
  colors[s] = STATE.gray
  for v in g[s]:
    if colors[v] == STATE.white:
      dfs(g, v, colors, complete_orders)
  colors[s] = STATE.black
  complete_orders.append(s)
  return

# topologically sort in terms of the last node of each scc
def topo_sort_scc(g):
  v = len(g)
  complete_orders = []
  colors = [STATE.white] * v
  for i in range(v): # run dfs on all the node
    if colors[i] == STATE.white:
      dfs(g,i, colors, complete_orders)
  return complete_orders[::-1]
```

The main scc is straightforward:

```python
# get conversed graph
def reverse_graph(g):
  rg = [[] for i in range(len(g))]
  for u in range(len(g)):
    for v in g[u]:
      rg[v].append(u)
  return rg

def scc(g):
  rg = reverse_graph(g)
  orders = topo_sort_scc(g)

  # track states
  colors = [STATE.white] * len(g)
  sccs = []

  # traverse the reversed graph
  for u in orders:
    if colors[u] != STATE.white:
      continue
    scc = []
    dfs(rg, u, colors, scc)
    sccs.append(scc)
  return sccs
```

Try to take a look at Tarjans' algorithm for SCC

Examples

1. 1520. Maximum Number of Non-Overlapping Substrings (hard): set up 26 nodes for all letters. A node represents a substray from start to end. Given a string abacdb, for a(0-2), add an edge between a -> to any other letter between start and end.Then we will have a directed graph. There is a scc (loop) between a and d, meaning a substring a has occurence of b and b substring has occurence of a, which is conflicting condition 2, so that they have to be combined. all results are sccs that are leaves in the contracted scc graph. We can think the scc graph is acyclic which is a forest. If we choose an internal node, we cant choose any of the leaves. Which making choosing the number of leaves maximum. Another solution is using two pointers: `https://zxi.mytechroad.com/blog/greedy/leetcode-1520-maximum-number-of-non-overlapping-substri`

-1.4 Minimum Spanning Trees

Problem Definition A *spanning tree* in an undirected graph $G = (V, E)$ is a set of edges, with no cycles, that connects all vertices. There can exist

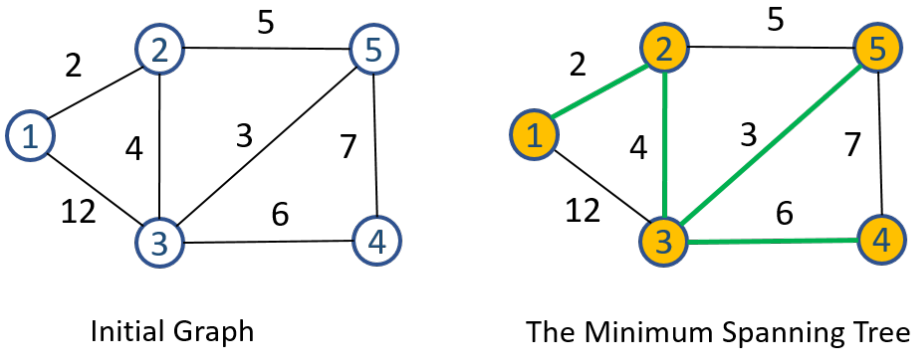


Figure 6: Example of minimum spanning tree in undirected graph, the green edges are edges of the tree, and the yellow filled vertices are vertices of MST (change this to a graph with multiple spanning tree, and highlight the one with the minimum ones.

many spanning trees in a graph. Given a weighted graph, we are particularly interested with the *minimum spanning tree (MST)*–a spanning tree with the least total edge cost.

One example is shown in Fig. 6. This graph can represent a collection of houses, and possible wires that we can lay. How we lay wires to connect all houses with the least total cost is equivalently a MST problem.

Spanning Tree To obtain a tree from a graph, the essence is to select edges iteratively until we have $|V| - 1$ edges which form a tree connecting V. We have two general approaches:

- Start with a forest consists of $|V|$ trees and contains only one node. We design a method to merge these trees into a final connected MST by selecting one edge at a time. This is the path taken by the Kruskal's algorithm.
- Start with a root node which can be any vertex selected from G, grow the tree by spanning to more nodes iteratively. In the process, we maintain two disjoint sets of vertices: one containing vertices that are in the growing spanning tree S and the other to track all remaining vertices $V - S$. This is the path taken by the Prim's algorithm.

We denote the edges in the growing as A. In this section, we explain two greedy algorithms to find MST.

-1.4.1 Kruskal's Algorithm

Kruskal's algorithm starts with $|V|$ trees that each has only one node. The main process of the algorithm is to merge these trees into a single one by

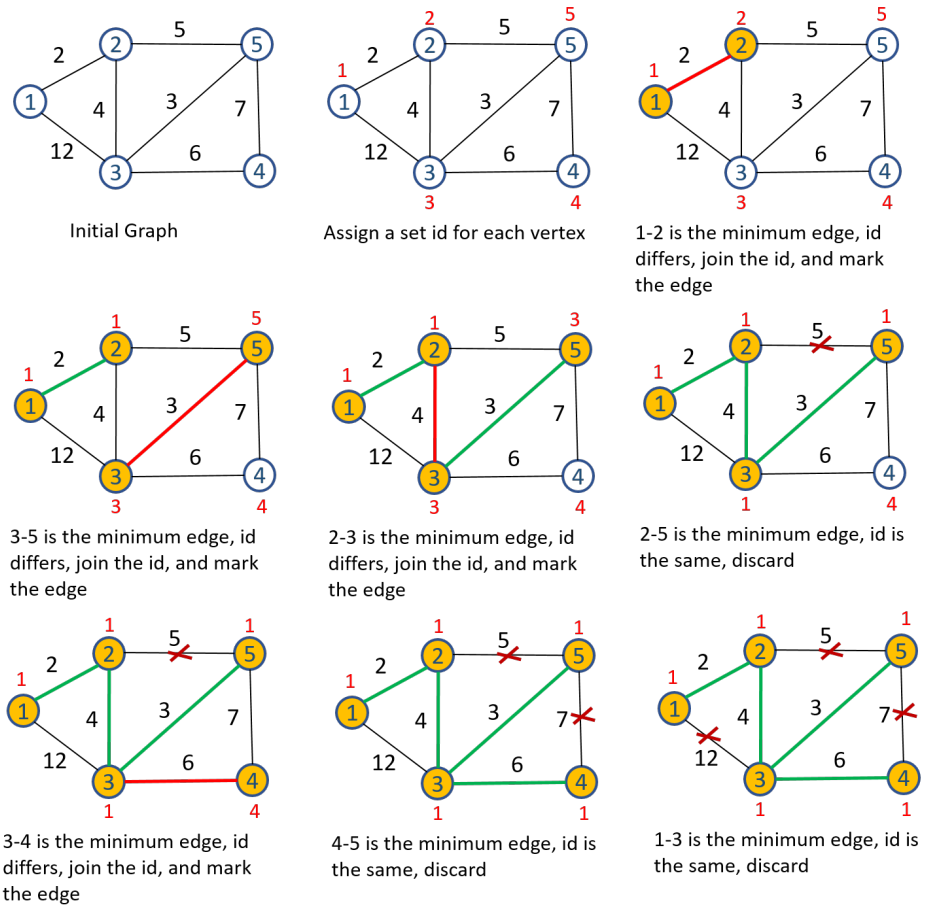


Figure 7: The process of Kruskal's Algorithm

iterating through all edges.

Generate Spanning Tree with Union-Find For each edge (u, v):

- if u and v belongs to the same tree, adding this edge will form a cycle, thus we discard this edge.
- otherwise, combine these two trees and add this edge into A.

This process will result in a single spanning tree. In implementation wise, we can do this easily by using union-find data structure. A tree is a set. Adding one edge is to merge two sets/trees into a single one if they belong to different sets.

Being Greedy with MST At each step i, we have $|E| - i$ edges to choose from. Applying the principle of greedy algorithm, maybe we can try

to choose the edge with the minimum cost among $|E| - i$ options. That is to say, we iterate edges in increasing order of its weight in the process of generating a spanning tree. Doing so will ensure us to have the MST, and this algorithm is the so called Kruskal's algorithm.

Fig. 7 demonstrates the run of Kruskal's on the input undirected graph. Here, the edges are ordered increasingly, i.e., [(1,2), (3, 5), (2, 3), (2, 5), (3, 4), (4, 5), (1, 3)]. As initialization, we assign a set id for each vertex that is marked in read and placed above its corresponding vertex. The process is:

```
edge      logic                                  action
(1,2)     1's set_id 1 != 2's set_id 2       merge set 2 to set 1
(3,5)     3's set_id 3 != 5's set_id 5       merge set 5 to set 3
(2,3)     2's set_id 1 != 3's set_id 3       merge set 3 to set 1
(2,5)     2's set_id 1 == 5's set_id 1       continue
(3,4)     3's set_id 1 != 4's set_id 4       merge set 4 to set 1
(4,5)     4's set_id 1 == 5's set_id 1       continue
(1,3)     1's set_id 1 == 3's set_id 1       continue
```

This process produces edges $[(1,2),(3,5),(2,3),(3,4)]$ as the edges of the final MST. We can have slightly better performance if we can stop iterating through edges once we have selected $|V| - 1$ edges. The implementation is as simply as:

```python
from typing import Dict
def kruskal(g: Dict):
  # g is a dict with node: adjacent nodes
  vertices = [i for i in range(1, 1 + len(g))]
  vertices = g.keys()
  n  = len(vertices)
  ver_idx = {v: i for i, v in enumerate(vertices)}

  # initialize a disjoint set
  ds = DisjointSet(n)

  # sort all edges
  edges = []
  for u in vertices:
    for v, w in g[u]:
      if (v, u, w) not in edges:
        edges.append((u, v, w))
  edges.sort(key=lambda x: x[2])

  # main section
  A = []
  for u, v, w in edges:
    if ds.find(ver_idx[u]) != ds.find(ver_idx[v]):
      ds.union(ver_idx[u], ver_idx[v])
      print(f'{u} -> {v}: {w}')
      A.append((u, v, w))
  return A
```

For the exemplary graph, we denote an weighted edge as a (key, value) pair, where the value is a tuple of two with the first item being the other endpoint

from the key vertex and the second item being the weight of the edge. The graph will thus be represented by a dictionary, {1:[(2, 2), (3, 12)], 2:[(1, 2), (3, 4), (5, 5)], 3:[(1, 12), (2, 4), (4, 6), (5, 3)], 4:[(3, 6), (5, 7)], 5:[(2, 5), (3, 3), (4, 7)]}. Running `kruskal(a)` will return the following edges:

```
[(1, 2, 2), (3, 5, 3), (2, 3, 4), (3, 4, 6)]
```

Complexity Analysis The sorting takes $O(|E| \log |E|)$ big oh time. The cost of checking each edge's belonging set id and merging two trees into a single one is decided by the complexity of the disjoint set, it can range from $O(\log |V|)$ to $O(|V|)$. Therefore, we can conclude the time complexity will be bounded by the sorting time, i.e., $O(|E| \log |E|)$.

-1.4.2 Prim's Algorithm

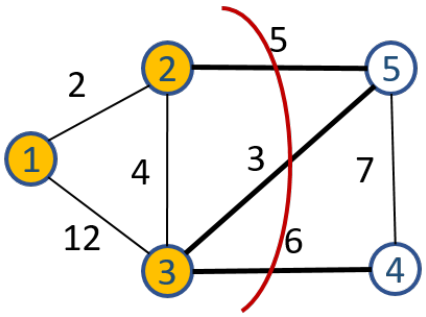


Figure 8: A cut denoted with red curve partition V into {1,2,3} and {4,5}.

In graph theory, a *cut* is a partition of V into S and $V - S$. For example, in Fig. 8 a cut is marked by red curve, removing three edges $(2, 5), (3, 5), (3, 4)$ partitions the set into two subgraph with subsets $\{1, 2, 3\}$ and $\{4, 5\}$. A *cross edge* $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoint is in S and the other is in $V - S$. A *light edge* is the minimum edge among all cross edges, such as edge $(3, 5)$ is the light edge in our example. We say, a cut *respects* a set of edges A if no edge in A crosses the cut, such as the marked cut in the example respects the set of edges $(1, 2), (2, 3), (1, 3)$.

Prim's algorithm starts with a randomly chosen root node and be put into a set S, leaving us with two sets of vertices, S and $V - S$. Next, it iteratively grows the partial and connected MST by adding an edge from the cross edges between the cut of $(S, V - S)$. Prim's algorithm is greedy in the sense that it chooses a light edge among its options to form the final MST. This process simulates the uniform-cost search which compose the Dijkstra's shortest path algorithm.

Fig. 9 demonstrates the process of Prim's algorithm. We start from vertex 1. with the set A, S, $V - S$, the cross edges at each step are denoted

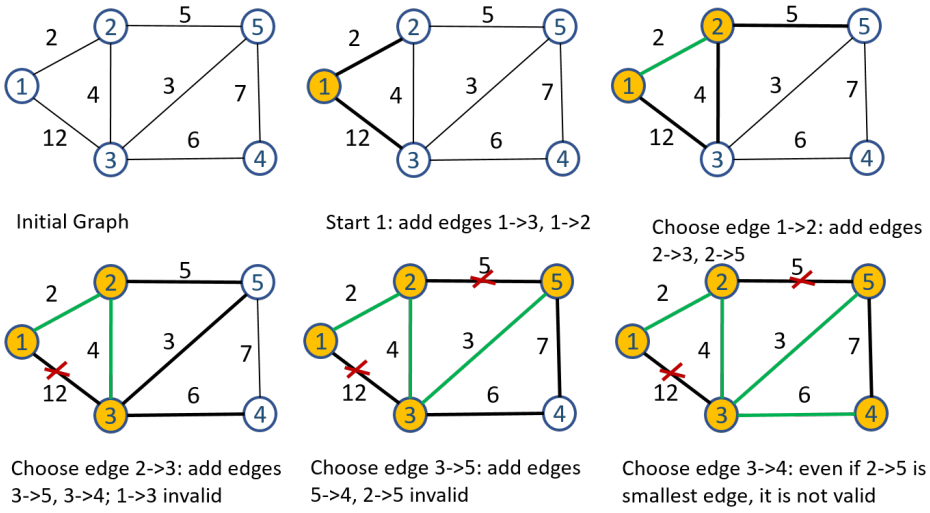


Figure 9: Prim's Algorithm, at each step, we manage the cross edges.

as CE, and a decision valid if it does not form a cycle within A, we list the process as:

A	S	V–S	CE	light edge
	1	2,3,4,5	(1,2), (1,3)	(1,2)
(1,2)	1,2	3,4,5	(1,3),(2,3),(2,5)	(2,3)
(1,2),(2,3)	1,2,3	4,5	(3,4),(3,5),(2,5)	(3,5)
(1,2),(2,3),(3,5)	1,2,3,5	4	(3,4),(5,4)	(3,4)
(1,2),(2,3),(3,5),(3,4)	1,2,3,4,5			

Implementation

One key step is to track all valid cross edges and be able to select the minimum edge from the set. Naturally, we use priority queue `pq`. `pq` can be implemented in two ways:

- **Priority Queue by Edges**–Considering the set S as a frontier set, `pq` maintains all edges expanded from the frontier set.
- **Priority Queue by Vertices**–`pq` maintains the minimum cross edge cost between vertices in S to the current vertex which is in $|V - S|$. This is an optimization over the first approach as it reduces multiple cross edges between S and current vertex v into a single cost – the minimum.

Priority Queue by Edges For example shown in Fig. 9, at first, the frontier set has only 1, then we have edges $(1,2),(1,3)$ in `pq`. Once edge $(1,2)$ is popped out as it has the smallest weight, we explore all outgoing

edges of vertex 2 to nodes in $V - S$, adding $(2, 3), (2, 5)$ in `pq`, resulting $pq = (2, 3), (2, 5), (1, 3)$. Then we pop out edge $(2, 3)$, and explore outgoing edges of vertex 3 and add $(3, 4), (3, 5)$ into `pq`, with $pq = (2, 5), (1, 3), (3, 4), (3, 5)$. At this moment, we can see that edge $(1, 3)$ is no longer a cross edge. Therefore, whenever we are about to add the light edge into the expanding tree, we check if both of its endpoints are in set S already. If true, we skip this edge and use the next valid light edge. Repeat this process will get us the set of edges A forming a MST. The Python code is as:

```
1  import queue
2
3  def _get_light_edge(pq, S):
4    while pq:
5      # Pick the light edge
6      w, u, v = pq.get()
7      # Filter out non-cross edge
8      if v not in S:
9        S.add(v)
10       return (u, v, w)
11   return None
12
13 def prim(g):
14   cur = 1
15   n = len(g.items())
16   S = {cur} #spanning tree set
17   pq = queue.PriorityQueue()
18   A = []
19
20   while len(S) < n:
21     # Expand edges for the exploring vertex
22     for v, w in g[cur]:
23       if v not in S:
24         pq.put((w, cur, v))
25
26     le = _get_light_edge(pq, S)
27     if le:
28       A.append(le)
29       cur = le[1] #set the exploring vertex
30     else:
31       print(f'Graph {g} is not connected.')
32       break
33   return A
```

In line 24, we use a 3 item tuple representing the edge cost, the first endpoint in the set S and the second endpoint in $V - S$ to align with the fact that the `PriorityQueue()` uses the first item of a tuple as the key for sorting. The `while` loop is similar to our breath-first-search and can be terminated in the following two conditions:

- when the set S is as large as the set V by checking the size of set S
- when we can not find a light edge which happens when the graph is

not connected.

Call `prim(a)` will return us the following A:

```
[(1, 2, 2), (2, 3, 4), (3, 5, 3), (3, 4, 6)]
```

Complexity Analysis The main cost of this implementation is on the priority queue, which has a maximum of $|E|$ items. In the worst case we have to enqueue and dequeue all edges, making the complexity as $O(|E| \log |E|)$. In a graph, generally, $E < V^2$, the complexity become $O(|E| \log V)$.

Priority Queue by Vertices Instead of tracking cross edges in the priority queue explicitly, we reduce all cross edges that reaches to a vertex in $V - S$ into the smallest cost and a predecessor which is to track the node in S that resulted in the smallest cost, saving us some additional space and time in the queue operations.

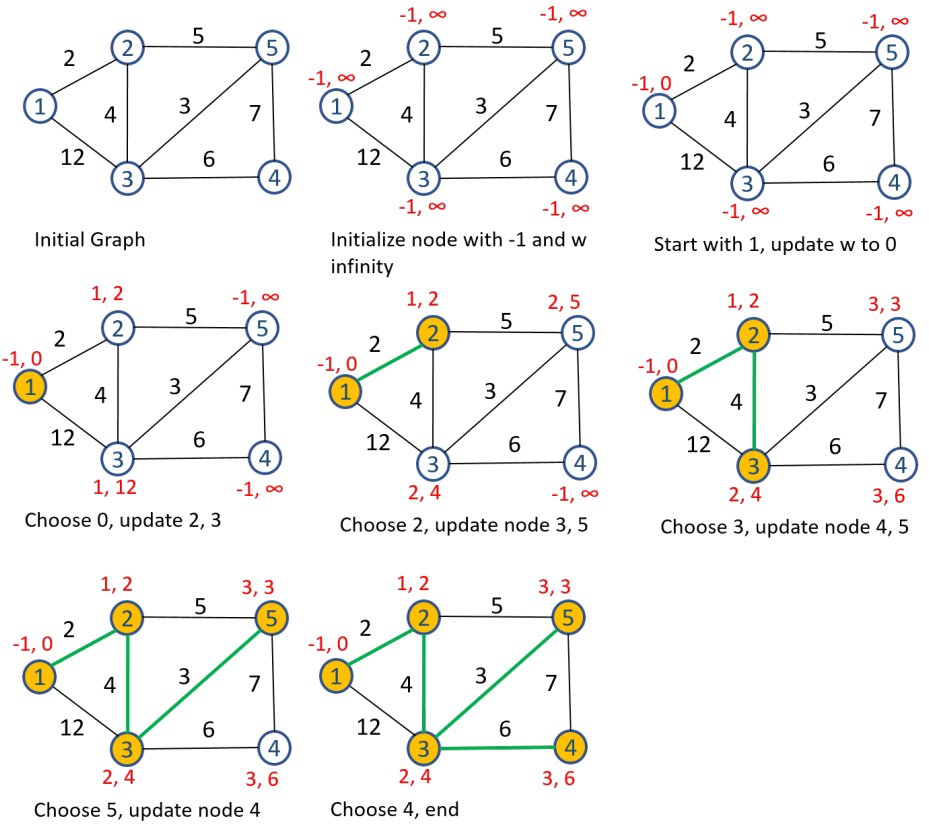


Figure 10: Prim's Algorithm

As shown in Fig. 10, we first initialize a priority queue with $|V|$ items, each has a task id same as the vertex id, a predecessor vertex $p = -1$, and

a cost initialized with ∞. We start by pointing vertex 1 as the root node, setting $S = 1$ and modify the task 1's cost to 0 and points its predecessor to itself. Then, we repeatedly pop out the vertex in the queue that has the smallest weight, along with the predecessor of this node, we are choosing the light edge. With this chosen node, we are able to reach out to adjacent nodes that are still in $V - S$ and see if we are able to find an even "lighter" edge. Applying this process on the given example:

1. First, we have the start vertex 1 with the smallest cost, pop it out, and explore edges $(1, 2), (1, 3)$, resulting in (a) modifying task 2 and 3's cost to 2 and 12, respectively and (b) set 2 and 3's predecessor to 1.

2. Pop out vertex 2, explore edges $(2, 3), (2, 5)$, resulting in (a) modifying task 3 and 5's cost to 4 and 5, respectively and (b) set 3 and 5's predecessor to 2.

3. Pop out vertex 3, explore edges $(3, 5), (3, 4)$, resulting in (a) modifying task 5 and 4's cost to 3 and 6, respectively and (b) set 3 and 5's predecessor to 3.

4. Pop out vertex 5, explore edges $(5, 4)$: since the new cross edge $(5, 4)$ has larger cost compared with previous reduced cross edge to reach to vertex 4, the vertex 4 in the queue is not modified.

5. Pop out vertex 4, no more new edges to expand, terminate the program.

This process results in the exactly same MST compared with the implementation by edges. However, it adds additional challenges into the implementation of the priority queue: We have to modify an enqueued item's record during the life cycle of the queue. In the Python implementation, we use the our customized `PriorityQueue()` in Section. **??**(also included in the notebook). The main process of the algorithm is:

```
def prim2(g):
  n = len(g.items())
  pq = PriorityQueue()
  S = {}
  A = []
  # Initialization
  for i in range(n):
    pq.add_task(task=i+1, priority=float('inf'), info=None) #
    task: vertex, priority: edge cost, info: predecessor vertex

  S = {1}
  pq.add_task(1, 0, info=1)

  while len(S) < n:
```

```
    u, p, w = pq.pop_task()
    if w == float('inf'):
      print(f'Graph {g} is not connected.')
      break
    A.append((p, u, w))
    S.add(u)
    for v, w in g[u]:
      if v not in S and  w < pq.entry_finder[v][0]:
        pq.add_task(v, w, u)

  return A

```

Calling function `prim2(a)` will output the following A:

```
[(1, 1, 0), (1, 2, 2), (2, 3, 4), (3, 5, 3), (3, 4, 6)]

```

Examples

1. 1584. Min Cost to Connect All Points (medium)
2. 1579. Remove Max Number of Edges to Keep Graph Fully Traversable (hard)

Try to prove the correctness of Kruskal's Algorithm.

-1.5 Shortest-Paths Algorithms

Problem Definition Given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to R$ that maps edges to real-valued weights, the weight of a path $p = (v_0, v_1, ..., v_k)$ is the summation over its constituent edge weights, denoted as $w(p)$:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{1}$$

The shortest path problem between v_i and v_j is to find the shortest path weight $\sigma(v_i, v_j)$ along with the shortest path p.

$$\sigma(v_i, v_j) = \begin{cases} \min\{w(p) : v_i \xrightarrow{p} v_j\} & \text{if there is a path from } v_i \text{ to } v_j \\ \infty & \text{otherwise} \end{cases} \tag{2}$$

For example, for the graph shown in Fig. 11, the shortest-path weight and its corresponding shortest-path between s to any other vertex in V is listed as:

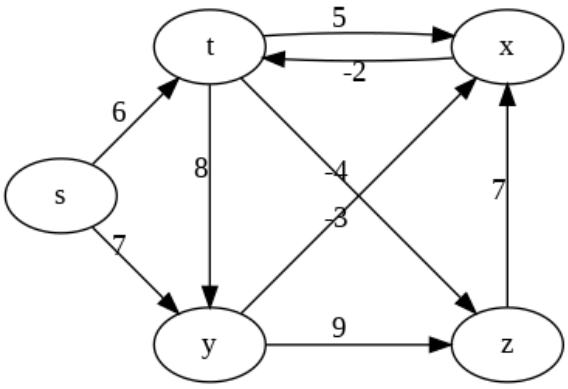


Figure 11: A weighted and directed graph.

(source, target)	shortest−path weight	shortest path
(s, s)	0	s
(s, y)	7	(s, y)
(s, x)	4	(s, y, x)
(s, t)	2	(s, y, x, t)
(s, z)	−2	(s, y, x, t, z)

Variants of Shortest-path Problems Generally, there exists a few variants of shortest path problems:

1. *Single-source shortest-path:* Find a shortest path from a given source s vertex to each vertex $v \in V$.
2. *Single-target shortest-path:* Find a shortest path to a given target t from each vertex $v \in V$. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source shortest-path problem.
3. *Single-pair shortest-path problem:* Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem too.
4. *All-pairs shortest-path problem:* Find a shortest path from u to v for every pair of vertices u and v in V if there exists one. Although we can solve this problem by running a single-source algorithm once for each vertex, we usually can solve it faster with algorithms addressed in Section (Sec. -1.5.4).

-1.5.1 Algorithm Design

In this section, we discuss the shortest path problem, and analyze it by using both graph theory and the fundamental algorithm design principle–Dynamic Programming.

Shortest path and Cycle From our experience in Combinatorial Search, we have to detect cycles within a path in the graph-based tree search to avoid being stuck in infinite recursion. So, how will cycle affect the detection of shortest paths? For example, in Fig. 11, a path $p = (s, t, x, t)$ contains the cycle (t, x, t). Because the cycle has a positive path weight $5+(-2)$, the path (s, t) remains smaller than the path that comes with the cycle. However, if we switch the weight of edge (t, x) with that of (x, t), then the same cycle (t, x, t) will have negative path weight $(-5) + 2$, repeating the cycle within the path infinitely we will have a cost of $-\infty$. Therefore, for a graph where the weights can be both negative and positive, one requirement posed on the single-source shortest-path algorithm, recursive or iterative, is to detect the negative-weight cycle that is reachable from the source. Once we get rid of all negative-weight cycles, the remaining of the algorithm can focus on only shortest-paths of at most $|V|-1$ edges, and the resulting shortest-paths will not contain neither negative- nor positive-weight cycles.

Exponential Naive Solution

Assume the given graph has no negative-weight cycle, a naive solution to obtain the shortest path and its weight is simply through a tree-search which starts from a source vertex s and enumerates all possible paths between s to any other vertex in V. The search tree will have a maximum height of $|V| - 1$, making the time complexity of this naive solution to be $O(b^{|V|})$, where b is the maximum branch of a vertex. Recall the path enumeration in Search Strategies, we implement this solution as:

```
def all_paths(g, s, path, cost, ans):
  ans.append({'path': path[::], 'cost': cost})
  for v, w in g[s]:
    # Avoid cycle
    if v in path:
      continue
    path.append(v)
    cost += w
    all_paths(g, v, path, cost, ans)
    cost -= w
    path.pop()
```

To obtain all possible paths, we call the function `all_paths()` with the following code:

```
g = {
    't':[('x', 5), ('y', 8), ('z', -4)],
    'x':[('t',-2)],
    'y':[('x',-3), ('z',9)],
    'z':[('x',7)],
    's':[('t', 6), ('y', 7)],
}
ans = []
all_paths(g, 's', ['s'], 0, ans)
```

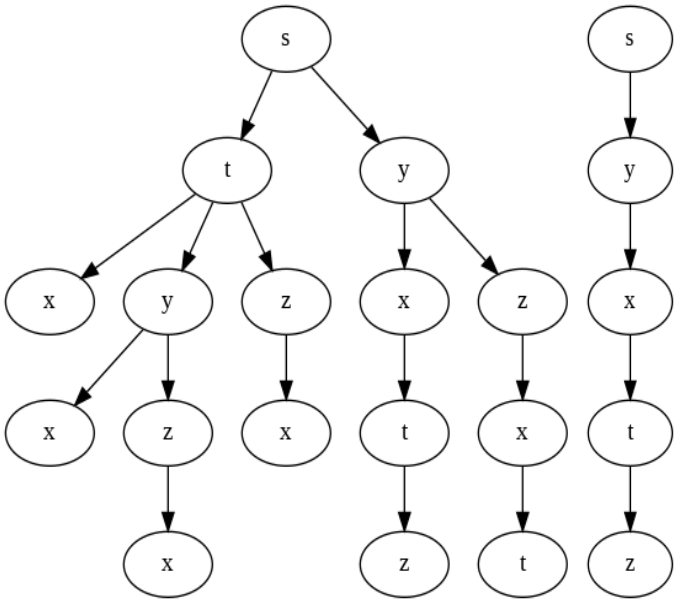


Figure 12: All paths from source vertex s for graph in Fig. 11 and its shortest paths.

Shortest-paths Tree We visualize all paths in `ans` in a tree structure shown in Fig. 12. We can easily extract the shortest paths between s to any other vertex from this result, which is shown on the right side of Fig. 12. All possible paths starting from source vertex can be viewed as a tree, and the shortest paths from source to all other vertices within the graph will be a subtree of the former tree structure, known as the *shortest-paths tree*. Formally, a shortest-paths tree rooted at s is a directed subgraph $G^{'} = (V^{'}, E^{'})$, where $V^{'} \in V$ and $E^{'} \in E$, such that

1. $V^{'}$ is the set of vertices reachable from s in G,
2. for each $v \in V^{'}$, the unique simple path from s to v in $G^{'}$ is a shortest path from s to v in G.

Predecessor Rule The shortest-paths tree makes it possible for us to track shortest paths with the predecessor rule: Given a graph $G = (V, E)$, and in the single-source shortest path problem, we maintain for each vertex $v \in V$ a predecessor π that is either another vertex or empty as for the root node. The shortest-paths between s and another vertex v can be obtained by iterating the chained predecessors starting from v and all the way backward to the source s. To summarize, each vertex v in the graph stores two values, $d(v)$ and $\pi(v)$, which (inductively) describe a tentative shortest path from s to v.

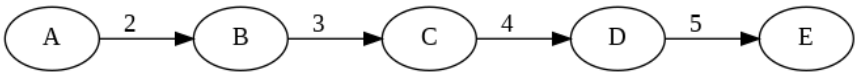
Optimization

As we see, shortest path problem is a truly combinatorial optimization problem, making them the best demonstration examples of the algorithm design principles–Dynamic Programming and Greedy Algorithm. On the other hand, depending on the characteristics of targeting graph, either they are dense or spares, directed acyclic graph (DAG) or not DAG, we can further optimize the efficiency besides of the design principle. However, in this chapter, we focus on the gist: *how to solve all-pair shortest path problems with dynamic programming?*

First, we use an adjacency matrix to represent our weight matrix W of size $|V| \times |V|$. In the process, we track shortest-path weight estimate D and additionally the predecessor Π. Both D and Π are of same size as W. w_{ij} indicates the weight of each edge with startpoint i and endpoint j,

$$W(i,j) = \begin{cases} 0 & \text{if } i = j \\ w_{ij} & \text{if } i \neq j, \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j, \text{ and } (i,j) \notin E \end{cases} \tag{3}$$

With this definition, we show a naive directed graph in Fig. 13 along with its W.



W

	A	B	C	D	E
A	0	2	∞	∞	∞
B	∞	0	3	∞	∞
C	∞	∞	0	4	∞
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

Figure 13: The simple graph and its adjacency matrix representation (changing it to lower letter)

Overlapping Subproblems and Optimal Substructures For all-pair shortest paths, we have $|V|^2$ optimal subproblems, each subproblem $D(i,j)$ is defined as the shortest-path between v_i and v_j. Optimal Substructures states "*the optimal solution to a problem has the optimal solutions to subproblems in it.*" All of this boils down to how to define the "subproblem" and how a larger subproblem is divided into smaller subproblems (the recurrence relation).

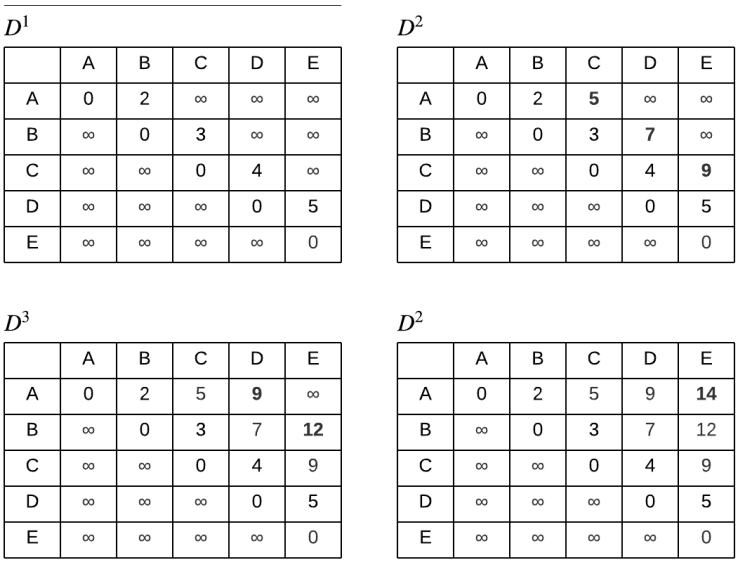
With our naive directed graph, the shortest path between a and d come from the shortest path between a to an intermediate node x or the shortest

path between a and d found so far. First, we define the subproblem as the shortest path between a and d with maximum path length(MPL) m. With this definition, we show two possible ways of dividing the subproblem:

1. We divide a subproblem with MPL m into a subproblem with MPL $m-1$ and an edge. Therefore, the shortest path at this maximum length m is either the shortest path found so far or equals to the shortest path between a and x plus the weight of edge (x, d), our recurrence relation is:

$$D^m(a,d) = \min_x(D^{m-1}(a,d), D^{m-1}(a,x) + W(x,d)) \tag{4}$$

   As we can see, each update for an item in distance matrix D takes $O(|V|)$ time as it has to check all possible intermediate nodes. Furthermore, it takes $|V|-1$ passes to update D^0 all the way to $D^{|V|-1}$. Therefore, this approach has a time complexity of $O(|V|^4)$. We demonstrate the update process in Fig. 14 for our naive example.



D^1	A	B	C	D	E
A	0	2	∞	∞	∞
B	∞	0	3	∞	∞
C	∞	∞	0	4	∞
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

D^2	A	B	C	D	E
A	0	2	**5**	∞	∞
B	∞	0	3	**7**	∞
C	∞	∞	0	4	**9**
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

D^3	A	B	C	D	E
A	0	2	5	**9**	∞
B	∞	0	3	7	**12**
C	∞	∞	0	4	9
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

D^2	A	B	C	D	E
A	0	2	5	9	**14**
B	∞	0	3	7	12
C	∞	∞	0	4	9
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

Figure 14: DP process using Eq. 4 for Fig. 13

2. We divide a subproblem with MPL m into two equal sized subproblem, each with MPL $m/2$. Therefore, the shortest path at this maximum length m is either the shortest path found so far or equals to the shortest path between a and x of length $m/2$ plus the shortest path between x and d of length $m/2$. With recurrence relation:

$$D^m(a,d) = \min_x(D^{m/2}(a,d), D^{m/2}(a,x) + D^{m/2}(x,d)) \tag{5}$$

D^1

	A	B	C	D	E
A	0	2	∞	∞	∞
B	∞	0	3	∞	∞
C	∞	∞	0	4	∞
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

D^2

	A	B	C	D	E
A	0	2	**5**	∞	∞
B	∞	0	3	**7**	∞
C	∞	∞	0	4	**9**
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

D^4

	A	B	C	D	E
A	0	2	5	**9**	**14**
B	∞	0	3	7	**12**
C	∞	∞	0	4	9
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

Figure 15: DP process using Eq. 5 for Fig. 13

Similarly, each update takes $|V|$ time. Differently, it only takes $\log |V|$ updates to get the final optimal subproblems. Thus, this approach gives a better time complexity, $O(|V|^3 \log |V|)$. The process is demonstrated in Fig. 15.

Alternatively, we define the subproblem as the shortest path between a and d with x as an intermediate node along the path, the number of intermediate node is $|V|$. Here, we use k to index the intermediate node, and i, j to index the start and end node. Then a subproblem $D^k(i, j)$ can be either the shortest path between i and j with intermediate nodes $0, 1, ..., k-1$ or the shortest path between i and k with all previous intermediate nodes plus the shortest path between k and j with all previous intermediate nodes. The recurrence relation is:

$$D^k(i, j) = \min(D^{\{0,...,k-1\}}(i, j), D^{\{0,...,k-1\}}(i, k) + D^{\{0,...,k-1\}}(k, j)) \quad (6)$$

As we see, each recurrence update only takes constant time. At the end, after we consider all possible intermediate nodes, we reach out to the optimal solution. This approach results in the best time complexity, $O(|V|^3)$ so far. We demonstrate the update process in Fig. 16. At pass C, using C as intermediate node, we end up only use C-th row and C-th column to update our matrix.

As we shall see later, the first way is similar to Bellman-Ford, the second is a repeated squaring version of Bellman-Ford, and the third is Floyd-warshall algorithm.

D^A

	A	B	C	D	E
A	0	2	∞	∞	∞
B	∞	0	3	∞	∞
C	∞	∞	0	4	∞
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

D^B

	A	B	C	D	E
A	0	2	**5**	∞	∞
B	∞	0	3	∞	∞
C	∞	∞	0	4	∞
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

D^c

	A	B	C	D	E
A	0	2	5	9	∞
B	∞	0	3	**7**	∞
C	∞	∞	0	4	∞
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

D^D

	A	B	C	D	E
A	0	2	5	9	**14**
B	∞	0	3	7	**12**
C	∞	∞	0	4	**9**
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

D^E

	A	B	C	D	E
A	0	2	5	9	14
B	∞	0	3	7	12
C	∞	∞	0	4	9
D	∞	∞	∞	0	5
E	∞	∞	∞	∞	0

Figure 16: DP process using Eq. 6 for Fig. 13

Greedy algorithms For $|V|^2$ subproblems, solving each subproblem takes at least $|V|$ using Floyd-warshall algorithm. Greedy approach would think of ways to decide the optional solution to each subproblem in one try, making it $|V|^2$ or $|V| + |E|$. We will see Dijkstra algorithm which is only applicable on all positive weighted W.

In the following section, we start with going through algorithms solving single-source shortest path problem before we put up more details to the all-pair shortest path algorithms introduced above.

-1.5.2 The Bellman-Ford Algorithm

Bellman-ford algorithm addresses single-source shortest path problem using a single-source version of DP approach one.

Dynamic Programming Representation Given a single source node s in graph G, we define D and Π as just a one-dimensional vector instead of a matrix in all-pair shortest paths. D_i^m represents the shortest path between

s and i with maximum path length m. When $m = 0$, there is a shortest path from s to v with no edge iff $s = v$.

$$D_i^0 = \begin{cases} 0 & \text{if } s = i \\ \infty & \text{otherwise} \end{cases} \tag{7}$$

Similarly, Π^0 is initialized as `None`. Our simplified recurrence relation is:

$$D_i^m = \min(D_i^{m-1}, \min_{k,k\in[0,n-1]}(D_k^{m-1} + W(k,i))) \tag{8}$$

which can be further simplified to:

$$D_i^m = \min_{k,k\in[0,n-1]}(D_k^{m-1} + W(k,i)) \tag{9}$$

In Eq. 9, once an intermediate node is found to have smaller tentative path weight than the current's value, we set $\Pi(i) = k$.

Implementation In function `bellman_ford_dp`, W is an $n \times n$ adjacency matrix. In the first `for` loop, we run recurrence relation in Eq. 9 for $|V| - 1$ passes, giving the fact that other than the negative-weight cycle, there will be at most $|V| - 1$ edges for all paths within the graph.

```
def bellman_ford_dp(s, W):
  n = len(W)
  # D, pi
  D = [float('inf') if i!=s else 0 for i in range(n)] # * n
  P = [None] * n
  for m in range(n-1):
    newD = D[:]
    for i in range(n): # endpoint
      for k in range(n): # intermediate node
        if D[k] + W[k][i] < newD[i]:
          P[i] = k
          newD[i] = D[k] + W[k][i]

    D = newD
    print(f'D{m+1}: {D}')
  return D, P
```

Now, to retrieve the path from source s to other vertices, we implement a recursive function named `get_path` that starts from the target u and backtraces to the source s through Π. The code is as:

```
def get_path(P, s, u, path):
  path.append(u)
  if u == s:
    print('Reached to the source vertex, stop!')
    return path[::-1]
  elif u is None:
    print(f"No path found between {s} and {u}.")
    return []
  else:
    return get_path(P, s, P[u], path)
```

For the graph in Fig. 11, the updating on D using s as source is visualized in Fig. 17. Connecting all red arrows along with the shaded gray nodes, we

	0	1	2	3	4
	t	x	y	z	s
D^0	∞	∞	∞	∞	0
D^1	6	∞	∞	7	0
D^2	6	4	7	2	0
D^3	2	4	7	2	0
D^4	2	4	7	-2	0

Figure 17: The update on D for Fig. 11. The gray filled spot marks the nodes that updated its estimate value, with its precessor indicated by incoming red arrow.

have a tree structure, each update on D, we expand the tree by one more level, updating the best estimate reaching to target node with one more possible edge. We visualize this tree structure in Fig. 20. We explain the tree like this: if we are at most one edge away from s, we get t as small as 6, if we are three edges away, t is able to gain a smaller value through its predecessor x which is at most 2 edges away. After the last round of update, when the tree reaches to height $|V| - 1$, the predecessor vector Π will gives out the shortest-path tree: each edge in the shortest path tree can be obtained by connecting each predecessor with vertices in the graph. The shortest-path tree is marked in Fig. 20 in red color.

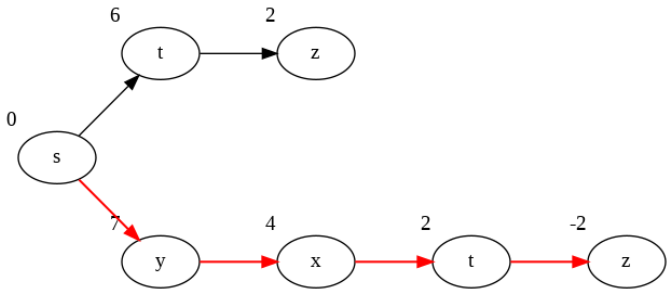


Figure 18: The tree structure indicates the updates on D, and the shortest path tree marked by red arrows.

Formal Bellman-Ford Algorithm In the above implementation, at each round, we made a copy of D, which is named `newD`. However, we can actually reuse the original D and update directly on it. The difference is: we would update `D[i]` at step m with other `D[k]` at step m instead of at step $m-1$, making some nodes' optimal estimate even more optimal. In the previous implementation, we can guarantee that after iteration i, for all $i \in [0, n-1]$, D_i is at most the weight of every path from s to i using at most m edges. In the new version, we end up reaching to the optimal value even earlier, but still it takes $n-1$ passes to guarantee.

Second, the inner two `for` loops are equivalently enumerating edges: for each possible edge (k, i), we update the best estimate for node i. With such two points modified, we get our official the Bellman-Ford algorithm, which states:

1. Initialize D and Π as D^0 and Π^0.

2. Run a relaxation process for $|v| - 1$ passes. Within each pass, go through each edge $(u, v) \in E$, with Eq. 9, if using u as an intermediate node, the tentative shortest path has smaller value, update D and Π.

Implement Bellman-Ford by checking edges from adjacency list as defined in `g` for Fig. 11. We should notice that different ordering of vertices /edges to be relaxed leads to different intermediate results in D, though the final result is the same.

Implementation We give out an exemplary implementation

```
def bellman_ford(g: dict, s: str):
  n = len(g)
  # Assign an enumerial index for each key
  V = g.keys()
  # Key to index
  ver2idx = dict(zip(V, [i for i in range(n)]))
  # Index to key
  idx2ver = dict(zip([i for i in range(n)], V))
  # Initialization the dp matrix with d estimate and predecessor
  si = ver2idx[s]
  D = [float('inf') if i!=si else 0 for i in range(n)] # * n
  P = [None] * n

  # n-1 passes
  for i in range(n-1):
    # relax all edges
    for u in V:
      ui = ver2idx[u]
```

```
      for v, w in g[u]:
        vi = ver2idx[v]
        # Update dp's minimum path value and predecessor
        if D[vi] > D[ui] + w:
          D[vi] = D[ui] + w
          P[vi] = ui
    print(f'D{i+1}: {D}')
  return D, P, ver2idx, idx2ver
```

During each pass, we relax on the estimation D with the following ordering:

```
    's':[('t', 6), ('y', 7)],
    't':[('x', 5), ('y', 8), ('z', -4)],
    'x':[('t',-2)],
    'y':[('x',-3), ('z',9)],
    'z':[('x',7)],
```

Printing out on the updates of D, we can see that it converges to the optimal value faster than the previous strict Dynamic programming version.

Time Complexity The first dynamic programming solution takes $O(|V|^3)$, and the formal Bellman-Ford takes $O(|V||E|)$. The later would be more efficient than the first if our graph is dense.

Detect Negative-weight Cycle If the graph contains no negative-weight cycle, after $|V| - 1$ passes of relaxation, D will reach to the minimum path value. Thus, if we run additional pass of relaxation, no vertex would be updated further. However, if there exists at least one negative-weight cycle, the $|V|^{th}$ update will have at least one vertex in D with decreased value.

Special Cases and Further Optimization

From the perspective of optimization, there are at least two approaches we can try to further boost the time efficiency, such as

1. special linear ordering of vertices to relax its leaving edges that leads us to its shortest-paths in just one pass of the Bellman-Ford algorithm,

2. and some greedy approach that takes only one pass of relaxation which can be similar to breath-first graph search or the Prim's algorithm.

In Fig. 17, suppose we are relaxing leaving edges of vertices in linear order $[s, t, y, z, x]$, the process will be as follows:

```
vertex          edges       relaxed vertices
s             (s,t),(s,y)   {t:6, y:7}
t    (t,x),(t,y),(t,z)      {x:11, z:2, t:6, y:7}
y          (y,x), (y,z)      {x:4, z:2, t:6, y:7}
z              (z,x)         {x:4, z:2, t:6, y:7}
x              (x,t)         {t:2, x:4, z:2, y:7}
```

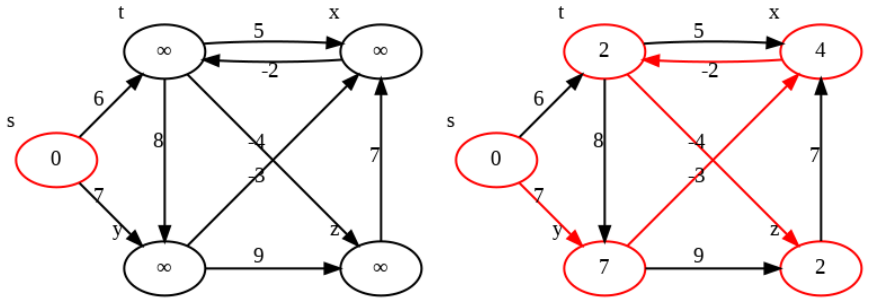


Figure 19: The execution of Bellman-Ford's Algorithm with ordering $[s, t, y, z, x]$.

The process is also visualized in Fig. 19. We see that only vertex z did not find its shortest-path weight. Why? From s to z, there are paths: $(s, t, z), (s, t, z), (s, t, y, z), (s, y, x, t, z)$. If we want to make sure after one pass of updates, vertex z reaches to its minimum shortest-path weight, we have to make sure its predecessors all reach to its minimum-path weight too which are vertex y and t. Same rule applies to its predecessors. In this graph, the ordering

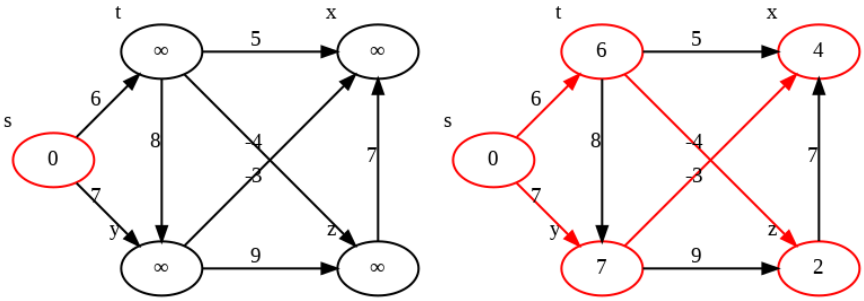
```
vertex   predecessor
s          None
t         s, x
y         s, t
x       t, y, z
z      y, t
```

From the listing, we see that the pair t and x conflicts each other: t needs x as predecessor and x needs t as predecessor. Tracking down this clue, we will find out that it is due to the fact that t and x coexist in a cycle.

Order Vertices with Topological Sort Taking away edge (x, t), we are able to obtain a topological ordering of the vertices, which is $[s, t, y, z, x]$. Relaxing vertices by this order of its leaving edges will guarantee to reach to the global-wise shortest-path weight that would otherwise be reached in $|V|-1$ passes in Bellman-Ford algorithm using arbitrary ordering of vertices. The shortest-paths tree is shown in Fig. 20.

So far, we discovered a $O(|V| + |E|)$ linear algorithm for single-source shortest-path problem when the given graph being directed, weighted, and acyclic. The algorithm consists of two steps: topological sorting of vertices in G and one pass of Bellman-Ford algorithm using the reordered vertices instead of arbitrary ordering. Calling the `topo_sort` function from Section. -1.2, we have our Python code:

```
1 def bellman_ford_dag(g, s):
```



(a) Initialization (b) After 1^{st} Pass

Figure 20: The execution of Bellman-Ford's Algorithm on DAG using topologically sorted vertices. The red color marks the shortest-paths tree.

```
  s = s
  n = len(g)
  # Key to index
  ver2idx = dict(zip(g.keys(), [i for i in range(n)]))
  # Index to key
  idx2ver = dict(zip([i for i in range(n)], g.keys()))
  # Convert g to index
  ng = [[] for _ in range(n)]
  for u in g.keys():
    for v, _ in g[u]:
      ui = ver2idx[u]
      vi = ver2idx[v]
      ng[ui].append(vi)
  V = topo_sort(ng)
  # Initialization the dp matrix with d estimate and predecessor
  si = ver2idx[s]
  dp = [(float('inf'), None) for i in range(n)]
  dp[si] = (0, None)

  # relax all edges
  for ui in V:
    u = idx2ver[ui]
    for v, w in g[u]:
      vi = ver2idx[v]
      # Update dp's minimum path value and predecessor
      if dp[vi][0] > dp[ui][0] + w:
        dp[vi] = (dp[ui][0] + w, ui)
  return dp
```

-1.5.3 Dijkstra's Algorithm

From Prim's to Dijkstra's In Breath-first Search, it hosts a FIFO queue, and whenever the vertex finishes exploring and turns into BLACK color, it is guaranteed to have the shortest-path length from the source. Similarly, in Prim's algorithm, it maintains a priority queue of cross edges

between the spanning tree set S and the remaining set $V - S$, whenever a vertex is added into S, it is a part of the MST.

In the shortest-path problem, using the same initialization in Bellman-Ford algorithm, that source vertex has 0 estimate to the source and all other vertices take ∞. Following the process of Prim's algorithm, we set a set S to save vertices that has found its shortest-path weight and predecessor, which is empty initially. Then, the algorithm starts the from the "lightest" vertex in $V - S$ to add to the set S, which is source vertex s at first, and it relax on the shortest-path estimate of vertices that are the endpoints of edges leaving the lightest vertex. This process is repeated in a loop until $V - S$ is empty. This devised approach indeed follows the principle of greedy algorithm just as Prim's algorithm does, this algorithm is called *Dijkstra's*.

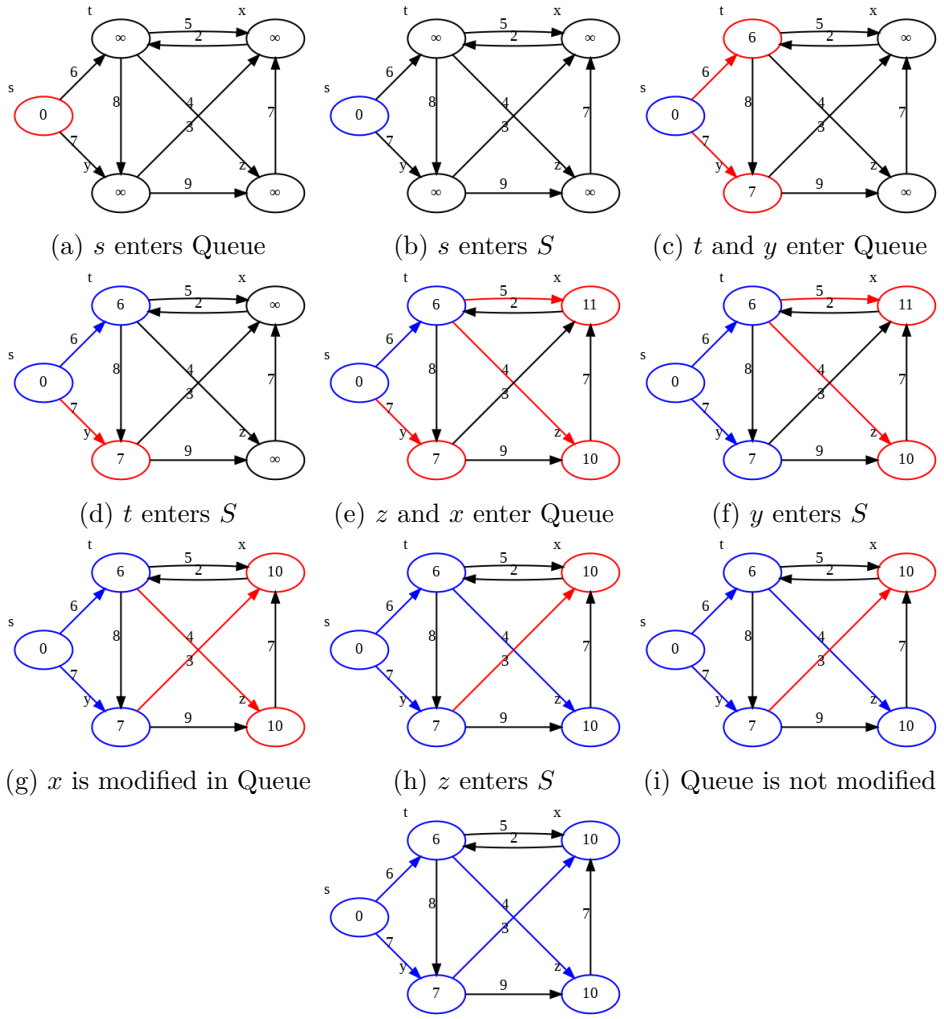
How is it greedy? Dijkstra's is the "greedy" version of Bellman-ford Algorithm. At each step, dynamic programming uses Eq. 9 to update D_i^m by trying all possible edges that extend the paths between s and i one at a time. Bellman-ford can only guarantee to achieve the optimal solution at the very end of running all passes. However, in Dijkstra algorithm, it reaches to the optimal solution in only one step–whenever a vertex is added into S, it adds a vertex in the shortest-path tree with only "local" information.

Correctness Condition: Non-negative Weight But, how to make sure that whenever the vertex was added into set S, it reaches to its shortest-path weight? Specifically, how to ensure our locally optimal decision is global optimal? This also means after this step, no matter how many additional paths with larger path length can reach to i, they shall never have less distance. This requires all of graph edges to be non-negative.

Implementation The implementation relies on the `PriorityQueue()` customized data structure once again, where we can modify an existing item in the queue. There are two ways to apply the priority queue:

- Add all vertices into the queue all at once at the beginning. Then only `deque` and modification operations are needed.
- Add vertex in the queue only when it is relaxed and has a non-∞ shortest-path estimate. The process of Dijkstra algorithm on a non-negative weighted graph that takes this approach of queue is demonstrated in Fig. 21 and the code is as follows:

```
1 def dijkstra(g, s):
2   Q = PriorityQueue()
3   S = []
4   # task: vertex id, priority: shortest-path estimate, info:
      predecessor
```



(a) s enters Queue (b) s enters S (c) t and y enter Queue

(d) t enters S (e) z and x enter Queue (f) y enters S

(g) x is modified in Queue (h) z enters S (i) Queue is not modified

(j) x enters S

Figure 21: The execution of Dijkstra's Algorithm on non-negative weighted graph. Red circled vertices represent the priority queue, and blue circled vertices represent the set S. Eventually, the blue colored edges represent the shortest-paths tree.

```
  Q.add_task(task=s, priority=0, info=None)
  visited = set()
  while not Q.empty():
    # Use the light vertex
    u, up, ud = Q.pop_task()
    visited.add(u)
    S.append((u, ud, up))

    # Relax adjacent vertice
    for v, w in g[u]:
```

```
15      # Already found the shortest path for this id
16      if v in visited:
17        continue
18
19      vd, vp = Q.get_task(v)
20      # First time to add the task or already in the queue, but
      need update
21      if not vd  or ud + w < vd:
22        Q.add_task(task=v, priority=ud + w, info=u)
23  return S
```

Complexity Analysis Once again, the complexity of Dijkstra's relies on the specific implementation of the priority queue. In our implementation, we used a customized `PriorityQueue()` which takes $|V|$ to initialize the queue. In this queue, we did not really remove the task from the queue but instead marked it as "REMOVED," so we can end up having maximum of $|E|$ vertices in the queue, making the cost of extracting the minimum item be $O(\log |E|)$. For the update, the main cost comes from inserting an new vertex through `heappush-like` operation, which is $O(\log |E|)$ too. In all, we have $|V|$ times of `pops` and $|V|$ times of updates, ending up with a worst-case time complexity of $O(|V| \log |E|)$.

Try to prove the correctness of Dijkstra's using greedy algorithms' two approaches on proving.

-1.5.4 All-Pairs Shortest Paths

In this section, we first summarize the solutions to singe-source shortest-path problem due to the fact that the problem of finding all-pairs shortest-path problem can be naturally decomposed into $|V|$ such single sourced subproblems. Next, we systematically build into three all-pair paths algorithms we are about to learn:

Summary to Single-source Shortest-Path Algorithms The solutions vary to the type of weighted graph G that we are dealing with:

- if (1) each weight $w \in R$ and (2) only non-negative cycle, we can apply the generalist dynamic programming approach–Bellman-Ford Algorithm,
- if each weight is non-negative, i.e., $w \in R^+$, we take the greedy approach–"Dijkstra's Algorithm"
- and (1) if the graph is acyclic and (2) only have non-negative cycles, we can run one pass of Bellman-Ford algorithm with vertices being relaxed in topologically sorted liner ordering.

Depends on which category the given graph G falls into, a naive and nature solution to all-pairs shortest-path problem can be addressed by running the corresponding algorithm $|V|$ passes–once for each vertex viewed as source in a complexity scaled by $|V|$ times.

Extended Bellman-Ford's Algorithm

We leverage the first DP approach in Section -1.5.1. Define weight matrix W, shortest-path weight estimate matrix D, and predecessor matrix Π. We have recurrence relation:

$$D^m(i,j) = \min_{k\in[0,n-1]}(D^{m-1}(i,k) + W(k,j)), \tag{10}$$

$\Pi^m(i,j)$ is updated by:

$$\Pi^m(i,j) = \begin{cases} \text{None}, & \text{if } D^m(i,j) = 0 \text{ or } D^m(i,j) = \infty, \\ \text{argmin}_{k\in[0,n-1]}(D^{m-1}(i,k) + W(k,j)), & \text{otherwise.} \end{cases} \tag{11}$$

with initialization:

$$D^0(i,j) = \begin{cases} 0, & \text{if } i = j, \\ \infty, & \text{otherwise.} \end{cases} \tag{12}$$

$$\Pi^0(i,j) = None \tag{13}$$

In detail, out extended Bellman-ford algorithm consists of these main steps:

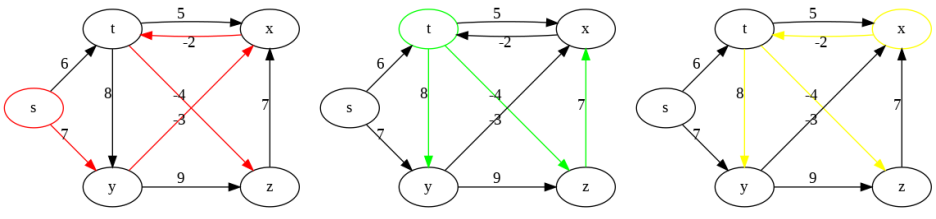
1. Initialization: we initialize d and π using Eq. 12 and 13.
2. For every pair of vertices i and j, we update the d and π using recurrence relation in Eq. 10 and 13, respectively, for $|V| - 1$ passes.
3. Run the $|V|^{th}$ pass to decide if any negative-weight cycle exist in each rooted shortest-path tree.

To notice that after one pass of update on D since it is initialized, $D^{(1)} = W$, thus, in our implementation, only $|V| - 2$ passes of updates are needed actually. Assume we have converted the graph shown in Fig. 11 into a W adjacency matrix representation and a dictionary `key2idx` that maps each key to a numerical index from 0 to $|V| - 1$. This extended Bellman-ford algorithm is implemented in main function `extended_bellman_ford_with_predecessor` which calls a subfunction `bellman_ford_with_predecessor` that does one pass of relaxation and does not detect non-negative cycle. The code is as:

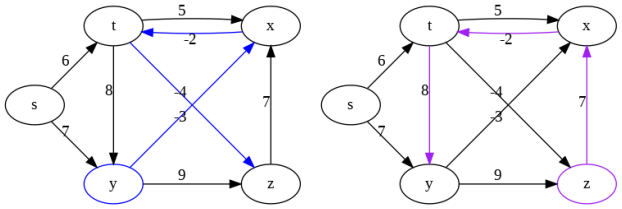
```python
import copy
def bellman_ford_with_predecessor(W, L, P):
  n = len(W)
  for i in range(n): # source
    for j in range(n): # endpoint
      for k in range(n): # extend one edge
        if L[i][k] + W[k][j] < L[i][j]:
          L[i][j] = L[i][k] + W[k][j] # set d
          P[i][j] = k # set predecessor

def extended_bellman_ford_with_predecessor(W):
  n = len(W)
  # initialize L, first pass
  L = copy.deepcopy(W)
  print(f'L1 : {L} \n')
  P = [[None for _ in range(n)] for _ in range(n)]
  for i in range(n):
    for j in range(n):
      if L[i][j] != 0 and L[i][j] != float('inf'):
        P[i][j] = i
  # n-2 passes
  for i in range(n-2):
    bellman_ford_with_predecessor(W, L, P)
    print(f'L{i+2}: {L} \n')
  return L, P
```

The L matrix will be having all zeros along the diagonal, in this case, it is

```
[ [0, 2, 4, 7, -2],
  [inf, 0, 3, 8, -4],
  [inf, -2, 0, 6, -6],
  [inf, -5, -3, 0, -9],
  [inf, 5, 7, 13, 0]],
```



(a) s as source (b) t as source (c) x as source



(d) y as source (e) z as source

Figure 22: All shortest-path trees starting from each vertex.

We reconstruct the shortest-path trees and visualize them in Fig. 22.

Repeated Squaring Extended Bellman-Ford Algorithm

We leverage the second DP approach in Section -1.5.1. This approach bears resemblance to the repeated squaring optimization in matrix multiplication. Repeated squaring is a general method for fast computation of exponentiation with large powers of a number or more generally of a polynomial or a square matrix. The underlying algorithm design methodogy is divide and conquer. Assume our input is x^n, where x is an expression, repeat squaring computes this in $O(\log n)$ steps by repeatedly squaring an intermediate result. Repeating Squaring method is actually used a lot in some advanced algorithm. Another one we will see in String algorithms.

Repeated Squaring Applied on Extended Bellman-Ford Algorithm

If we observe the `bellman_ford_one_pass`, it has three for loops, and it shows similar pattern with matrix multiplication. Suppose A and B are both $n \times n$ matrix, and we compute $C = A \times B$, the formulation is $c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}$ which has the same pattern as of Eq. 10. If we use $\cdot$ to mark `bellman_ford_one_pass` operation on L and W, we will have the following relations:

$$\begin{aligned} L^1 &= L^0 \cdot W = W, \\ L^2 &= L^1 \cdot W = W^2, \\ L^3 &= L^2 \cdot W = W^3, \\ &\vdots \\ L^{n-1} &= L^{n-2} \cdot W = W^{n-1} \end{aligned} \tag{14}$$

With repeated squaring technique, we can compute L^{n-1} with only $\log(n-1)$ round of one pass operation

$$\begin{aligned} L^1 &= W, \\ L^2 &= W \cdot W, \\ L^4 &= W^2 \cdot W^2, \\ &\vdots \end{aligned} \tag{15}$$

The above repeation stops when our $m \geq n-1$. The implementation is:

```
import copy
import math
def bellman_ford_repeated_square(L):
  n = len(W)
  for i in range(n): # source
    for j in range(n): # endpoint
```

```
      for k in range(n): # double the extending length
        L[i][j] = min(L[i][j], L[i][k]+L[k][j])

def extended_bellman_ford_repeated_square(W):
  n = len(W)
  # initialize L, first pass
  L = copy.deepcopy(W)
  print(f'L1 : {L} \n')
  # log n passes
  for i in range(math.ceil(math.log(n))):
    bellman_ford_repeated_square(L)
    print(f'L{2^(i+1)}: {L} \n')
  return L
```

The Floyd-Warshall Algorithm

We leverage the third DP approach in Section -1.5.1, this approach is called *The Floyd-Warshall Algorithm*. We directly put the code here:

```
def floyd_warshall(W):
  L = copy.deepcopy(W) #L0
  n = len(W)
  for k in range(n): # intermediate node
    for i in range(n): # start node
      for j in range(n): # end node
        L[k][i] = min(L[k][i], L[k][j] + L[j][i])
  return L
```