Data Structure lab8

吴嘉骜 21307130203

2023 年 11 月 23 日

Experiment environment

Windows 11 VsCode Python 3.11.5 64-bit

Critical Paths

The Program (or Project) Evaluation and Review Technique, commonly abbreviated PERT, is a statistical tool, used in project management, that is designed to analyze and represent the tasks involved in completing a given project. First developed by the United States Navy in the 1950s, it is commonly used in con- junction with the critical path method or CPM.

Critical paths in PERT chart analysis: Edges represent jobs to be performed, and edge weights represent the times required to perform particular jobs. If edge (u, v) enters vertex v and edge (v, x) leaves v, then job (u, v) must be performed prior to job (v, x). A path through this dag represents a sequence of jobs that must be performed in a particular order. A critical path is a longest path through the dag, corresponding to the longest time to perform an ordered sequence of jobs.

The PERT chart formulation given above is somewhat unnatural. It would be more natural for vertices to represent jobs and edges to represent sequencing constraints; that is, edge (u, v, w) would indicate that job u must be performed before job v. Weights would then be assigned to edges. Write a procedure that can finds a longest path in a directed acyclic graph with weighted edges.

Solution:

The code from `pert.py` is as follows:

```python
def toposort(graph):
    """
    Parameters:
        - graph: a dictionary representing a graph

    Returns:
        - a list of vertices in topological order
    """
    def dfs(v):
        # depth first search from vertex v
```

```
        visited.add(v)
        for u, _ in graph[v]:
            if u not in visited:
                dfs(u)
        result.append(v)

    visited = set() # visited vertices
    result = [] # topological order of vertices

    for v in graph:
        if v not in visited:
            dfs(v)

    return result[::-1] # reverse order

def dag_longest(graph):
    '''
    Parameters:
        - graph: a dictionary representing a graph

    Returns:
        - a list of vertices in the longest dist, and the length of the dist
    '''
    # Perform a Topological Sort on the graph
    topo_order = toposort(graph)
    # Initialize the maximum distance and corresponding dist
    max_dist = float('-inf')
    longest_path = []
    # Check each vertex as a starting point
    for s in topo_order:
        # Initialize the distance to each vertex to be -inf and the parent to be None
        dist = {v: (float('-inf'), None) for v in graph}
        dist[s] = (0, None)
        # Relax each edge in the topological order
        for v in topo_order:
            for u, w in graph[v]:
                if dist[v][0] + w > dist[u][0]:
                    dist[u] = (dist[v][0] + w, v)
        # Find the vertex with the largest distance from s
        max_dist_cur = max(dist.values(), key=lambda x: x[0])[0]
        # Update the maximum distance and corresponding dist
```

```
        if max_dist_cur > max_dist:
            max_dist = max_dist_cur
            max_v = max(dist, key=lambda x: dist[x][0]) # the vertex with the largest
                distance from s
            path_tmp = []
            while max_v:
                path_tmp.append(max_v)
                max_v = dist[max_v][1]
            longest_path = path_tmp[::-1]

    return longest_path, max_dist
```

Interpretation:

The `toposort` function performs a topological sort on the graph, and the `dag_longest` function finds the longest path in the graph.

The idea of the algorithm is similar to a DAG single-source shortest path algorithm, which is based on the topological sort of the graph. The difference is that we have to check each vertex as a starting point, and find the vertex with the largest distance from the starting point. The inner loop of the algorithm relaxes each edge in the topological order, and updates the distance and predecessor of each vertex.

The pseudocode of the algorithm is as follows:

Algorithm 1: Algorithm for finding the longest path in a DAG with weighted edges

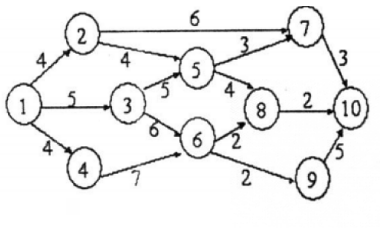
Result: The longest path in a DAG with weighted edges

Input: A directed acyclic graph $G = (V, E)$ with a weight $w(u, v)$ for each edge $(u, v) \in E$;
Perform a topological sort on G;
for *each vertex s as a starting point* **do**
    Initialization: Set $longestPath(v) = -\infty$ for each vertex v, except for the start vertex s which is set to 0. Create an array *predecessors* to store predecessors;
    for *each vertex v in topological order* **do**
        for *each edge (v, u) in E* **do**
            if $longestPath(u) + w(u, v) > longestPath(v)$ **then**
                Update $longestPath(v) = longestPath(u) + w(u, v)$;
                Update $predecessors[v] = u$;
            end
        end
    end
end
Find the end vertex v with the maximum value in $longestPath$;
Reconstruct the path from the end vertex v using *predecessors* and determine the length;
Output: The longest path and its length;

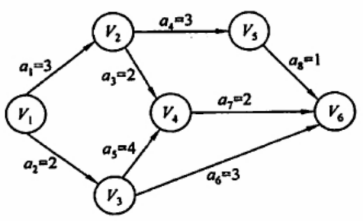
Results:

We test the algorithm on the examples in Figure 1. The answers are correct, and note that there may

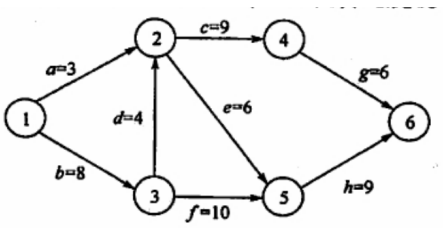
be multiple longest paths in a graph.



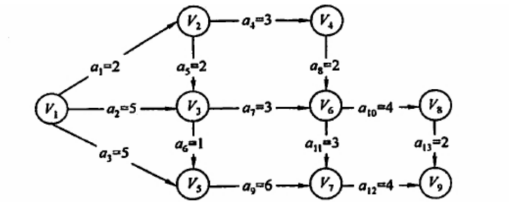
(a) Example Graph 1



(b) Example Graph 2



(c) Example Graph 3



(d) Example Graph 4

```
eg1. Longest path:  [1, 4, 6, 9, 10]  total weights:  18
eg2. Longest path:  [1, 3, 4, 6]  total weights:  8
eg3. Longest path:  [1, 3, 5, 6]  total weights:  27
eg4. Longest path:  [1, 3, 5, 7, 9]  total weights:  16
```

(e) Output of the answers

Figure 1: Examples of DAGs and their longest paths

Complexity Analysis:

To analyze the complexity of the algorithm above, we can consider the following components:

1. **Topological Sort**: The topological sort of a graph $G = (V, E)$ has a complexity of $O(V + E)$, as it requires visiting every vertex once and examining all edges once.

2. **Main loop**: The main loop of the algorithm iterates over all vertices V as a starting point. Then for each vertex, it iterates over all edges E to relax them. Therefore, the complexity of the main loop is $O(V \cdot (V + E))$.

3. **Finding the Longest Path**: Finding the vertex with the maximum distance takes $O(V)$, and tracing the longest path takes $O(V)$ in the worst case.

Therefore, the overall complexity of the algorithm is $O(V^2 + V \cdot E)$.
In most cases, $O(V \cdot E)$ will be the dominating term, except for very sparse graphs where $O(V^2)$ might be comparable.