Algorithm Design and Analysis(H)
Southern University of Science and Technology
Mengxuan Wu
12212006

Assignment 2

Mengxuan Wu

Question 1: Algorithm Description

We introduce an efficient implementation of Edmond's algorithm for MDST problem, with the presumption that readers are familiar with the basic concepts of the algorithm.

Some Observations

With observation of the algorithm, we can see that the algorithm is mainly composed of three tasks:

1. find the set of minimum weight edges
2. find the cycle in the set of minimum weight edges
3. contract the cycle

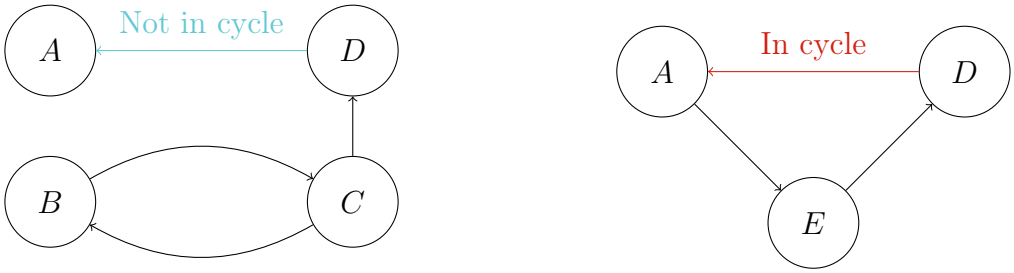
Among these tasks, the first one and the third one can be implemented efficiently by using a meldable heap. To find the minimum weight entering edge, we can simply store all entering edges of each node in a min heap. For efficiently maintaining the heap property when contracting the cycle, we can use a meldable heap.

The second task, however, is more challenging to implement efficiently. Finding cycle can be a time-consuming task. In the naive implementation, when the algorithm stop to check whether the cycle is found, it checks all selected edges in the graph. This means we may repeatedly check the same edge multiple times even if the edge is not in the cycle.

The efficient implementation we are going to introduce will focus on finding the cycle efficiently and expanding the cycles correctly. And the other two tasks will be implemented similarly as the naive implementation, but with a more efficient data structure introduced in the following sections.

Finding Cycle

As mentioned above, the cause for the inefficiency is that we may repeatedly check the same edge multiple times. This is because after a contract operation, an edge that is not in any cycle before may become part of a cycle as figure 1 shows.



(a) Before contracting the cycle (b) After contracting the cycle

Figure 1: An edge that is not in any cycle before may become part of a cycle

For this reason, every time after we contract a cycle, we still need to check all selected edges in the graph to ensure that we can find all cycles.

However, there are some observations that can help us to avoid this inefficiency. We can find that for these nodes that are not in the cycle, their minimum weight entering edge will not change after the cycle is contracted (node A and D in figure 1). If the new cycle contains some nodes v that are not in the cycle before, it means that we choose an edge $v \to s$ for the super node s formed by contracting old cycle (the edge $A \to E$ in figure 1).

Hence, the natural idea is to traverse the graph in a DFS manner by going to the starting node of the minimum weight entering edge chosen. When we traverse the graph, we can mark the nodes as visited as we include them in the current path, and unmark them when we backtrack. If we encounter a node that has been visited, it means that we find a cycle, and we can backtrack to identify all nodes involved and contract them. Then we continue the DFS traversal from the newly contracted node.

In this way, we don't need to check the edges we have checked before. Since if these edges form a cycle after some contraction, we will be able to find them again in the DFS traversal when checking the new super node. For example, let us assume that in figure 1, we start the DFS traversal from node A. After we contract the first cycle, we continue the DFS traversal from node E. Then we will find the cycle formed by the edge $A \to E$.

However, there is a problem with this approach. With the original idea of Edmond's algorithm, we do not include the root node when we find the minimum weight entering edge for each node. This may cause the DFS traversal starting from the root node fail to find the cycle as figure 2 shows.

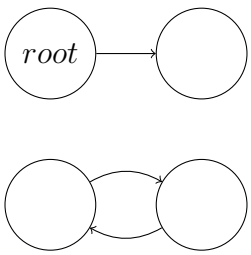


Figure 2: DFS traversal starting from the root node fails to find the cycle

To solve this problem, we add multiple edges to make this graph strongly connected and ignore the special treatment of the root node. This will guarantee that the DFS traversal starting from any node will find all cycles in the graph. However, by doing so we complicate the algorithm in two ways:

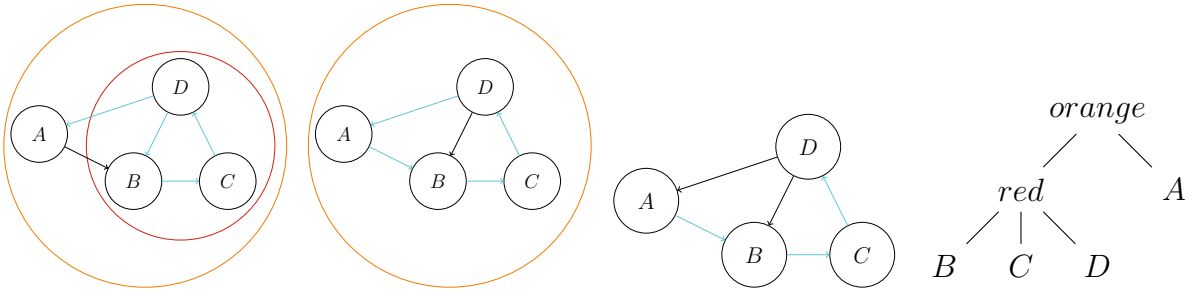
1. We may choose the newly added edges in the MDST, which is not allowed.
2. Since the graph is strongly connected, it will be contracted into a single node at the end of the algorithm.

To solve the first problem, we add new edges with weight $\left(\sum_{v \in V} w(v)\right) + 1$. And we can check the overall weight of the MDST at the end of the algorithm. If the weight of the MDST is greater than $\sum_{v \in V} w(v)$, it implies that we have chosen the newly added edges in the MDST, and there is no solution to the MDST problem. The second one is rather an advantage. It gives us a sign that we have found the MDST, and we can stop the algorithm when only one node is left in the graph.

Expand Cycles

To expand the cycles correctly, we need to maintain the information of which edge we choose for each (super) node. This is because we avoid cycles by updating weight for all other entering edges of each involved node when contracting the cycle, instead of actually removing one edge from the cycle. Hence, for all edges chosen there will be cycles formed by them.

With this information, we can expand the cycles with a bottom-up approach. First, we build a contraction tree, where super nodes are parents of all nodes that are contracted to form them. Then, we start from each node at the bottom of contraction tree, and go up to super nodes once all children are expanded. For each node, we look at the edge we choose for it, and use this edge as the entering edge of the edge's destination node as figure 3 shows. After all nodes are expanded, we remove the entering edge of the root node, and the MDST is found.



(a) Choose edges for leaf nodes in the contraction tree first

(b) Update the entering edge for A and B when expanding the red super node

(c) Remove the entering edge for root node A when expanding the final orange super node

(d) Contraction tree

Figure 3: Expand cycles with a bottom-up approach

Question 2: Meldable Heap

For task 1 and task 3, we need a data structure that can efficiently carry out the following operations:

1. Find the minimum element
2. reduce all elements by a constant

3. meld two of these into one

It is natural to use a heap for operation 1. But an array-based heap is slow for other two operations. We will need to traverse all elements in the heap, and the tree structure of the heap is not utilized.

A better choice is to use a heap based on pointers instead of arrays, which allows us to utilize the tree structure of the heap. We will introduce the leftist tree as the data structure for the heap.

Leftist Tree Definition

We first define the distance of each node in the leftist tree as follows:

- A node the has no children has $n.dist = -1$.
- A node that has only one child has $n.dist = 0$.
- A node that has two children has $n.dist = \min(n.left.dist, n.right.dist) + 1$.

The leftist tree is a binary tree that satisfies the leftist property: The distance of the left child is greater than or equal to the distance of the right child. And for this reason, we can always find $n.dist = n.right.dist + 1$ in a leftist tree.

By this property, a leftist tree with n nodes has a distance of at most $\lceil \log(n+1) \rceil - 2$ for its root node.

Proof.

We proof this by induction.

Base case:

For a root node whose $dist = -1$, the minimum number of node that can be in the tree is 1, which is the root node. For a root node whose $dist = 0$, the minimum number of node that can be in the tree is 3, which is the root node and its two children.

Inductive step: For a root node whose $dist = k \geq 1$, we can see that if we want the tree to have the minimum number of nodes, both left and right child should have distance $k-1$ and should be the minimum tree of that distance. Hence, the minimum tree will be a full binary tree. And we can draw the conclusion easily. □

Leftist Tree Operations

Merge

The most important operation of the leftist tree is the merge operation. It takes two leftist trees and merges them into one leftist tree. The merge operation is defined as algorithm 1.

The merge operation is a recursive operation. It first compares the root of two trees, and choose the smaller one as the root of the new tree. Then it recursively merges the right child of the tree with the other tree. After the merge operation, it swaps the left and right child if the distance of the left child is less than the distance of the right child, to maintain the leftist property.

Algorithm 1: Merge

Input: Two leftist trees T_1 and T_2
Output: A leftist tree that contains all elements in T_1 and T_2
if T_1 *is empty* **then**
  return T_2;
end
if T_2 *is empty* **then**
  return T_1;
end
if *The root of* T_1 *is greater than the root of* T_2 **then**
  Swap T_1 and T_2;
end
$T_1.right \leftarrow Merge(T_1.right, T_2)$;
if *The distance of* $T_1.left$ *is less than the distance of* $T_1.right$ **then**
  Swap $T_1.left$ and $T_1.right$;
end
return T_1;

The key idea of the merge operation is that we always recursively merge the right child of the tree. Since the property mentioned above, one of the trees' distance will be reduced by 1 after each merge operation. And the merge operation will stop once one of the tree is empty. Hence, we can guarantee the merge operation takes $O(\log n + \log m)$ time, where n and m are the number of nodes originally in T_1 and T_2 respectively.

In the leftist tree, most operations are replaced by the merge operation. For example, to insert an element, we can merge the tree with a single node with the tree. To delete the minimum element, we can merge the left and right child of the root node.

Pushdown

When we need to reduce all elements in the leftist tree by a constant, we can use the pushdown operation. Since adding, reducing all elements in a heap by a constant will not change the order of the elements, we don't need to update the key of each element until it needs to be compared in the merge operation. We do this by adding a tag to each node, which represents the constant we need to reduce for all of its children. When we need to compare the key of a node, we first reduce the key by the tag of the node, add this tag to the children of the node, and reset the tag of current node to 0.

Algorithm 2: Pushdown operation

Input: A node n
if n *is not empty* **then**
  if $n.left$ *is not empty* **then**
    $n.left.tag \leftarrow n.left.tag + n.tag$;
    $n.left.key \leftarrow n.left.key - n.tag$;
  end
  if $n.right$ *is not empty* **then**
    $n.right.tag \leftarrow n.right.tag + n.tag$;
    $n.right.key \leftarrow n.right.key - n.tag$;
  end
  $n.tag \leftarrow 0$;
end

Algorithm 3: Merge with pushdown

Input: Two leftist trees T_1 and T_2
Output: A leftist tree that contains all elements in T_1 and T_2
if T_1 *is empty* **then**
  return T_2;
end
if T_2 *is empty* **then**
  return T_1;
end
Pushdown($T_1.root$);
Pushdown($T_2.root$);
if *The root of* T_1 *is greater than the root of* T_2 **then**
  Swap T_1 and T_2;
end
$T_1.right \leftarrow Merge(T_1.right, T_2)$;
if *The distance of* $T_1.left$ *is less than the distance of* $T_1.right$ **then**
  Swap $T_1.left$ and $T_1.right$;
end
return T_1;

Question 3: Time Complexity Analysis

The full algorithm is defined as algorithm 4.

Algorithm 4: MDST

Input: A graph $G = (V, E)$, a root node r
Output: The minimum directed spanning tree of G rooted at r
for *i from 1 to* $|V|$ **do**
  $E.add(v_i, v_{i+1}, \sum_{v \in V} w(v) + 1)$;
end
foreach $e \in E$ **do**
  $leftistTree[e.destination].push(e)$;
end
$S \leftarrow$ empty stack;
$S.push(r)$;
while *leftistTree[S.top] is not empty* **do**
  $e \leftarrow leftistTree[S.top].pop()$;
  if *e.destination is in the same super node as e.source* **then**
    continue;
  end
  if $visited[e.destination] = false$ **then**
    $visited[e.destination] \leftarrow true$;
    $S.push(e.destination)$;
    continue;
  end
  super node $s \leftarrow$ new node;
  while $visited[e.source] = true$ **do**
    assign $S.top$ to s;
    reduce all entering edges of $S.top$ by $leftistTree[S.top].min()$;
    $leftistTree[S.top].pop()$;
    meld $leftistTree[S.top]$ with $leftistTree[s]$;
    $visited[S.top] \leftarrow false$;
    $S.pop()$;
  end
end

When we contract a cycle, we need to meld all nodes in the cycle. Since there are at most $|V|$ nodes in the graph, we need to do at most $|V| - 1$ meld operations. For each node, we at most have $|V| - 1$ entering edges in the heap (duplicate edges can be removed and keep only the minimum weight edge). For this part, the time complexity is $O((|V| - 1) \log(|V| - 1))$.

Also, we need to pop each edge in the heap. Similarly, we need to do $|E|$ pop operations, and each pop operation takes $O(\log(|V| - 1))$ time. Hence, the time complexity for this part is $O(|E| \log(|V| - 1))$.

The overall time complexity of the algorithm is $O((|E|+|V|) \log |V|)$ or simply $O(|E| \log |V|)$, which is equivalent to $O(m \log n)$.

Question 4: Space Complexity Analysis

The leftist tree will be taking most of the space. We need to store each edge as a node in the leftist tree, and also add $|V|$ new edges to the graph to make it strongly connected. Hence, there will be at most $|E|+|V|$ nodes in the leftist tree. Other parts of the algorithm are about storing information about each (super) node, which takes $O(2|V|)$ space.

Hence, the space complexity of the algorithm is $O(|E| + |V|)$, which is equivalent to $O(m + n)$.