Spanning Leafy Tree: The Report

Junzhe Huang Siyuan Hong Qiyu Chen
jmh8504@psu.edu sjh6636@psu.edu qfc5019@psu.edu
Fall 2022

1 INTRODUCTION

In this project, to find a spanning tree with maximum number of leaves, there are 4 representative algorithms in total:

- a randomized algorithm;
- a 2-approximation algorithm [1];
- a greedy 3-approximation algorithm [2];
- a brute force algorithm for graphs with small numbers of edges (less than 25).

2 CODE

We use Python as our programming language.

2.1 The randomized algorithm

```
def random_tree(graph):

  # Fill out graph attributes
  graph.search()
  nodes = get_nodes(graph)
  edges = get_edges(graph)

  # Bests so far
  most_leaves = 0
  best_tree = None

  # Run N iterations of randomized algorithm, save the best
  for i in range(0, NUMBER_OF_RANDOM_RUNS):

    # Add all vertices of graph to disjoint set
    disjoint_set = UnionFind()
    disjoint_set.insert_objects(nodes)

    # Shuffle edges to make function stochastic
    shuffle(edges)

    num_edges = 0
    current_tree = Graph(MAXIMUM_NUMBER_OF_NODES)

    # Build graph
    for edge in edges:
      u, v = edge.ends

      # Add edge if it doesn't create a cycle
      if disjoint_set.find(u) != disjoint_set.find(v):
        disjoint_set.union(u, v)
        current_tree.add_edge(edge)
        num_edges += 1
```

```
    # Check leaves when tree is complete, |E| = |V| - 1
    if num_edges == len(nodes) - 1:
      num_leaves = len(get_leaves(current_tree))

      # Update best_tree if better num_leaves
      if num_leaves > most_leaves:
        most_leaves = num_leaves
        best_tree = current_tree

      break

return best_tree
```

2.2 The 2-approximation algorithm

```
# Implements the Roberto algorithm
def expansion_rule_tree(graph):

  def expand_based_on_rules(E, T_i_edges):
    G = make_graph(E)
    continue_loop = True

    while(True):
      continue_loop = False
      use_lowest_priority_expansion_set = False

      T_i = make_graph(T_i_edges)
      T_i_nodes = get_nodes(T_i)
      leaves = get_leaves(T_i)
      lowest_priority_expansion_set = set()
      for x in leaves:
        # if leaf x has at least two neighbors not in T_i then all neighbors of x not
                                            in T_i are placed as its children
        x_neighbors = G.neighbors[x]
        x_potential_children = set()
        for neighbor in x_neighbors:
          if neighbor not in T_i_nodes:
            x_potential_children.add(neighbor)
        if len(x_potential_children) >= 2:
          T_i_edges.update({Edge(x, t) for t in x_potential_children})
          use_lowest_priority_expansion_set = False
          continue_loop = True
          break
        # if x has only one neighbor y that does not belong to T_i and at least two
                                           neighbors of y are not in T_i, put y as
                                           the only child of x and all neighbors of y
                                            as children of y
        elif len(x_potential_children) == 1:
          y = x_potential_children.pop()
          y_neighbors = G.neighbors[y]
          y_potential_children = set()
          for neighbor in y_neighbors:
            if neighbor not in T_i_nodes:
              y_potential_children.add(neighbor)
          if len(y_potential_children) > 2:
            T_i_edges.add(Edge(x, y))
            T_i_edges.update({Edge(y, t) for t in y_potential_children})
            use_lowest_priority_expansion_set = False
            continue_loop = True
            break
          # expansion rule with lowest priority
          elif len(y_potential_children) == 2:
            if (len(lowest_priority_expansion_set) == 0):
              lowest_priority_expansion_set.add(Edge(x, y))
              lowest_priority_expansion_set.update({Edge(y, t) for t in
                                           y_potential_children})
              use_lowest_priority_expansion_set = True
              continue_loop = True

      # only use the expansion rule with priority 1 when no other expansion rules
                                           available
      if use_lowest_priority_expansion_set:
        T_i_edges.update(lowest_priority_expansion_set)

      if not continue_loop:
        break

    return T_i_edges

  def maximally_leafy_forest_Roberto(graph):
    # Initialization
```

```
  G_prime = create_copy(graph)
  F = set()
  continue_loop = True

  while (True):
    continue_loop = False

    E_prime = set(get_edges(G_prime))
    V_prime = set(get_nodes(G_prime))
    for v in V_prime:
      if len(G_prime.neighbors[v]) < 3:
        continue
      else:
        continue_loop = True
        T_i_edges = {e for e in E_prime if v in e.ends}
        T_i_edges = expand_based_on_rules(E_prime, T_i_edges) # expand the tree T_i
                                          based on the expansion rule

        # concatenate T_i to F_i and then remove from G all vertices in T_i and all
                                          edges incident to them
        F.update(T_i_edges)
        V_prime_i = get_nodes(make_graph(T_i_edges))
        for v_i in V_prime_i:
          for t in G_prime.neighbors[v]:
            if Edge(v_i, t) in E_prime:
              E_prime.remove(Edge(v_i, t))
        G_prime = make_graph(E_prime)
        break

    if not continue_loop:
      break

  return make_graph(F)

# Takes a leafy forest (a Graph instance composed of one or more disjoint trees)
                                          and
# a list of unused edges in the original graph.
# Returns a leafy spanning tree of the original graph.
def create_spanning_tree_from_forest(forest, unused_edges):

  def is_leaf(node):
    return len(forest.neighbors[node]) == 1

  spanning_tree = create_copy(forest)

  nodes = get_nodes(forest)
  edges = get_edges(forest)

  # Initialize meta-graph
  connected_components = UnionFind()
  connected_components.insert_objects(nodes)
  for edge in edges:
    connected_components.union(edge.ends[0], edge.ends[1])

  # Sort unused edges by tier as follows:
  # 1. Edge from internal node to internal node
  # 2. Edge from internal node to leaf
  # 3. Edge from leaf to leaf
  internal_to_internal_edges = []
  internal_to_leaf_edges = []
  leaf_to_leaf_edges = []
  for edge in unused_edges:
    u, v = edge.ends
    if not is_leaf(u) and not is_leaf(v):
      internal_to_internal_edges.append(edge)
    elif is_leaf(u) and is_leaf(v):
      leaf_to_leaf_edges.append(edge)
    else:
      internal_to_leaf_edges.append(edge)
```

```
    unused_edges = internal_to_internal_edges
    unused_edges.extend(internal_to_leaf_edges)
    unused_edges.extend(leaf_to_leaf_edges)

    # Add edges (by tier) if it doesn't induce a cycle
    for edge in unused_edges:
      u, v = edge.ends
      if connected_components.find(u) != connected_components.find(v):
        spanning_tree.add_edge(edge)
        connected_components.union(u, v)

    return spanning_tree

  leafy_forest = maximally_leafy_forest_Roberto(graph)

  unused_edges = get_edge_difference(graph, leafy_forest)
  leafy_spanning_tree = create_spanning_tree_from_forest(leafy_forest, unused_edges)

  return leafy_spanning_tree
```

2.3 The 3-approximation algorithm

```
# Implements the Lu-Ravi algorithm in the paper "Approximating Maximum Leaf
# Spanning Trees in Almost Linear Time"
def lu_tree(graph):

  def maximally_leafy_forest(graph):
    # Initialization
    G = create_copy(graph)
    E = get_edges(G)
    V = get_nodes(G)
    S = UnionFind()
    d = {}
    F = set()

    for v in V:
      S.find(v)
      d[v] = 0

    for v in V:
      S_prime = {}  # Maps vertex to union-find set index
      d_prime = 0
      for u in G.neighbors[v]:
        if S.find(u) != S.find(v) and S.find(u) not in S_prime.values():
          d_prime += 1
          S_prime[u] = S.find(u)
      if d[v] + d_prime >= 3:
        for u in S_prime:
          F.add(Edge(u,v))
          S.union(u, v)
          d[u] += 1
          d[v] += 1

    return make_graph(F)

  # Takes a leafy forest (a Graph instance composed of one or more disjoint trees)
                                                and
  # a list of unused edges in the original graph.
  # Returns a leafy spanning tree of the original graph.
  def create_spanning_tree_from_forest(forest, unused_edges):

    def is_leaf(node):
      return len(forest.neighbors[node]) == 1

    spanning_tree = create_copy(forest)

    nodes = get_nodes(forest)
    edges = get_edges(forest)

    # Initialize meta-graph
    connected_components = UnionFind()
    connected_components.insert_objects(nodes)
    for edge in edges:
      connected_components.union(edge.ends[0], edge.ends[1])

    # Sort unused edges by tier as follows:
    # 1. Edge from internal node to internal node
    # 2. Edge from internal node to leaf
    # 3. Edge from leaf to leaf
    internal_to_internal_edges = []
    internal_to_leaf_edges = []
    leaf_to_leaf_edges = []
    for edge in unused_edges:
      u, v = edge.ends
      if not is_leaf(u) and not is_leaf(v):
        internal_to_internal_edges.append(edge)
      elif is_leaf(u) and is_leaf(v):
        leaf_to_leaf_edges.append(edge)
```

```
      else:
        internal_to_leaf_edges.append(edge)
    unused_edges = internal_to_internal_edges
    unused_edges.extend(internal_to_leaf_edges)
    unused_edges.extend(leaf_to_leaf_edges)

    # Add edges (by tier) if it doesn't induce a cycle
    for edge in unused_edges:
      u, v = edge.ends
      if connected_components.find(u) != connected_components.find(v):
        spanning_tree.add_edge(edge)
        connected_components.union(u, v)

    return spanning_tree

  leafy_forest = maximally_leafy_forest(graph)

  unused_edges = get_edge_difference(graph, leafy_forest)
  leafy_spanning_tree = create_spanning_tree_from_forest(leafy_forest, unused_edges)

  return leafy_spanning_tree
```

2.4 The Brute Force method

```
def brute_force(graph):
  edges = get_edges(graph)
  edge_num = len(edges)
  nodes = get_nodes(graph)
  node_num = len(nodes)
  subgraph_num = 2 ** edge_num

  best_tree = None
  most_leaves = 0

  for k in range(subgraph_num):
    edges_new = set()
    remain  = k
    for i in range (edge_num):
      #exist = (subgraph_num // (10**i)) % 10
      exist = remain % 2
      remain = remain // 2
      if exist == 1:
        edges_new.add(edges[i])
      if remain == 0:
        break

    graph_new = make_graph(edges_new)
    # check if new graph is a spanning tree
    if (len(get_nodes(graph_new)) == node_num) and (is_tree(graph_new)):
      leaf_number = len(get_leaves(graph_new))
      if leaf_number > most_leaves:
        most_leaves = leaf_number
        best_tree = graph_new

  #print(most_leaves)
  return best_tree
```

References

[1] Solis-Oba, R., Bonsma, P., and Lowski, S. (2015). A 2-Approximation Algorithm for Finding a Spanning Tree with Maximum Number of Leaves. Algorithmica, 77(2), 374–388. doi:10.1007/s00453-015-0080-0.

[2] Lu, H.-I., and Ravi, R. (1998). Approximating Maximum Leaf Spanning Trees in Almost Linear Time. Journal of Algorithms, 29(1), 132–141. doi:10.1006/jagm.1998.0944.