МИНОБРНАУКИ РОССИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

ОТЧЕТ

По лабораторной работе № 02

по дисциплине

«Параллельные методы и алгоритмы»

РУКОВОДИТЕЛЬ:

_____ __Кулясов П.С.

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

_____ Каллаб Д. БибиковА.А.___

(подпись) (фамилия, и.,о.)

__М24-ИВТ-1_____

(шифр группы)

Нижний Новгород

2024

Задание к лабораторной работе №2

Для выполнения лабораторной работы необходимо написать программу на языке Elixir для генерации графа и проведения поиска по глубине (Depth-First Search) по этому графу. В программе должна быть реализована возможность настройки следующих параметров генерируемого графа:

- количество вершин;

- количество ребер (либо коэффициент ветвления);

- ориентированный/неориентированный граф;

- взвешенный/невзвешенный граф.

Кроме того, генерируемый граф должен быть связным (не должно быть вершин, не соединенных ни с одной другой вершиной, - недостижимых вершин) – нужно либо предусмотреть это в алгоритме генерации графа, либо выполнять проверку и корректировку уже сгенерированного графа.

Графический интерфейс не требуется, программа может быть консольной. Обязательно должна быть реализована возможность отрисовки сгенерированного графа с сохранением в файл. Также программа должна выводить матрицу смежности сгенерированного графа. Наконец, пользователь должен иметь возможность задать номер узла, который требуется найти, и программа должна найти этот узел при помощи алгоритма DFS.

Алгоритм должен быть проанализирован и выделены те его шаги, которые можно распараллелить. Алгоритм необходимо реализовать в двух вариантах: в «обычном» и с распараллеливанием вычислений на тех шагах, где это возможно.

Необходимо провести несколько экспериментов: сгенерировать не менее 5 графов с различным количеством вершин и ребер (обязательно должен быть пример с малым количеством вершин и несколько примеров с большим количеством вершин, не менее 1000), затем на каждом из них решить задачу (согласно выбранному варианту задания), используя обе реализации алгоритма («обычную» и с распараллеливанием), выполнить замеры времени расчетов и требуемого для вычислений объема памяти.

Полученные результаты необходимо сравнить, представить в виде таблицы и сделать вывод о том, какое преимущество дает применение распараллеливания в зависимости от количества вершин и ребер графа.

Решение:

Цель работы - сгенерировать граф с помощью языка программирования elixir, на результате должен полученная матрица смежности, файл dot и рисунок сгенерированного графа с весами. Затем требуется найти заданный узел в графе с помощью DFS.

Необходимые зависимости: установили Erlang и Elixir для выполнения команд, Graphviz для построения графиков/визуализации.

Основные используемые команды:

mix deps.get

iex.bat -S mix

Генерация графиков в Elixir

Алгоритм генерации графиков описан в документе первой лабораторной работы, поэтому для краткости в данной работе его описание сокращено. Напомним, что граф представляется через двумерную матрицу смежности, в которой каждая строка и столбец представляют собой вершину. В зависимости от отношений между этими вершинами, между ними рисуются рёбра.

Результирующий график сохраняется в формате DOT-файла, который может быть обработан инструментами Graphviz (например, dot) для создания визуального графика в виде изображения.

Функции программы

Функция generate_spanning_tree создаёт остовное дерево графа для соединения всех его узлов.

```
@doc """
Generates a spanning tree to ensure the graph is connected.
"""
defp generate_spanning_tree(vertices, directed, weighted) do
  # Start with the first vertex and add edges to connect all vertices
  Enum.reduce(2..vertices, [], fn j, acc ->
    i = Enum.random(1..(j - 1))
    w = if weighted, do: Enum.random(1..10), else: 1
    if directed do
      [{i, j, w} | acc]
    else
      [{i, j, w}, {j, i, w} | acc]
    end
  end)
end
```

Функция is_connected проверяет, соединены ли все узлы в графе (т.е. проверяет результат работы предыдущей функции). Мы считаем, что данная функция наилучше всего подходит для параллелизации, т.к. в таком случае не нарушается целостность данных.

```
@doc """
Checks if all nodes in the graph are connected (parallel version).
"""
def is_connected_parallel?(%GraphGenerator{} = graph) do
  if graph.directed do
    # Parallel DFS for directed graphs
    1..graph.vertices
    |> Enum.map(&Task.async(fn -> dfs(graph, &1) end))
    |> Enum.all?(fn task ->
      visited = Task.await(task)
      MapSet.size(visited) == graph.vertices
    end)
  else
    # Single DFS for undirected graphs
    visited = dfs(graph, 1)
    MapSet.size(visited) == graph.vertices
  end
end
```

Функция dfs проводит поиск по глубине, и возвращает набор узлов, по которым алгоритм прошёлся. Вместе с этим, функция build_adjacency_list строит список смежности из рёбер графа.

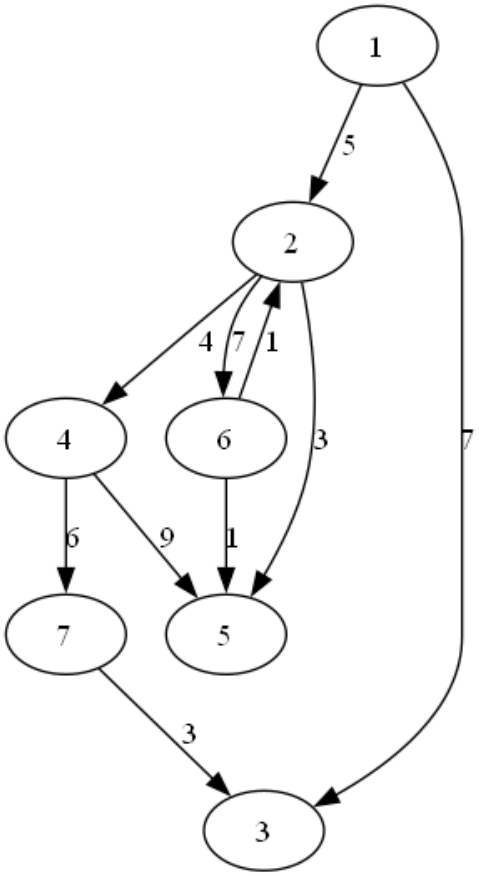
```
@doc """
Performs Depth-First Search (DFS) on the graph and returns the set of visited nodes.
"""
def dfs(%GraphGenerator{vertices: _vertices, edges: edges, directed: directed}, start_vertex) do
  adj_list = build_adjacency_list(edges, directed)
  visited = MapSet.new()
  dfs_helper(adj_list, start_vertex, visited)
end

defp dfs_helper(adj_list, vertex, visited) do
  if MapSet.member?(visited, vertex) do
    visited
  else
    visited = MapSet.put(visited, vertex)
    Enum.reduce(adj_list[vertex] || [], visited, fn neighbor, acc ->
      dfs_helper(adj_list, neighbor, acc)
    end)
  end
end

@doc """
Builds an adjacency list from the graph's edges.
"""
defp build_adjacency_list(edges, directed) do
  Enum.reduce(edges, %{}, fn {i, j, _w}, acc ->
    acc = Map.update(acc, i, [j], &(&1 ++ [j]))
    if not directed, do: Map.update(acc, j, [i], &(&1 ++ [i])), else: acc
  end)
end
```

Наконец, функции save_to_dot(_and_generate_image) сохраняют получившийся в граф формате .dot, который затем можно визуализировать при помощи библиотеки Graphviz.

```
  @doc """
  Saves the graph to a DOT file and generates an image using Graphviz.
  """
  def save_to_dot_and_generate_image(graph, filename) do
    save_to_dot(graph, filename <> ".dot")
    {output, status} = System.cmd("dot", ["-Tpng", filename <> ".dot", "-o", filename <> ".png"])
    IO.inspect(output)
    IO.inspect(status)
    IO.puts("Graph image saved as #{filename}.png")
  end

  @doc """
  Saves the graph to a DOT file.
  """
  def save_to_dot(%GraphGenerator{edges: edges, directed: directed, weighted: weighted}, filename) do
    graph_type = if directed, do: "digraph", else: "graph"
    edge_connector = if directed, do: "->", else: "--"

    content = ["#{graph_type} G {"] ++
              Enum.map(edges, fn {i, j, w} ->
                if weighted, do: "  #{i} #{edge_connector} #{j} [label=\"#{w}\"]", else: "  #{i} #{edge_connector} #{j}"
              end) ++
              ["}"]

    File.write!(filename, Enum.join(content, "\n"))
  end
end
```

Пошаговый пример использования

Задан граф, в котором 7 вершин и 10 рёбер. Граф является ориентированным и взвешенным. В результате генерации графа и преобразования его в изображение, получаем следующее:



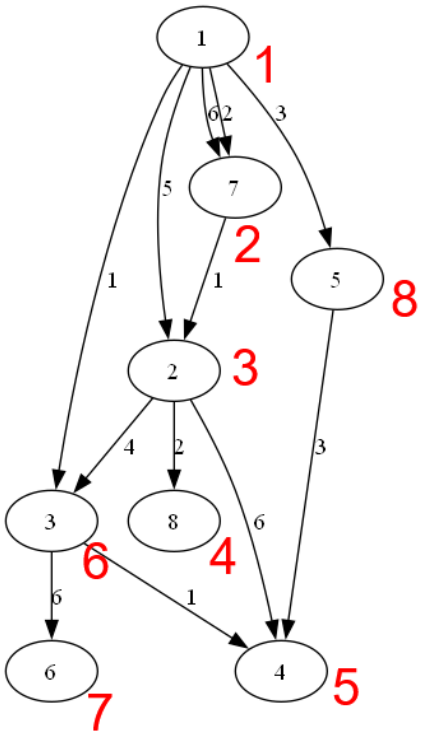
В результате вывод следующий:

```
Testing graph with 7 vertices, 10 edges, directed: true, weighted: true
Graph Edges: [
  {4, 7, 6},
  {2, 6, 7},
  {2, 5, 3},
  {2, 4, 4},
  {1, 3, 7},
  {1, 2, 5},
  {6, 2, 1},
  {7, 3, 3},
  {4, 5, 9},
  {6, 5, 1}
]
Sequential: Connected? false, Time: 0.0 seconds
Parallel: Connected? false, Time: 0.005427 seconds
DFS Result starting from vertex 1: [2, 3, 4, 5, 6, 7]
Memory usage: 46.165256 MB
Graph image saved as graph_7_10_true_true.png
```

Как можно увидеть, из узла 1 можно прийти к узлам с остальными индексами.

Шаги решения:

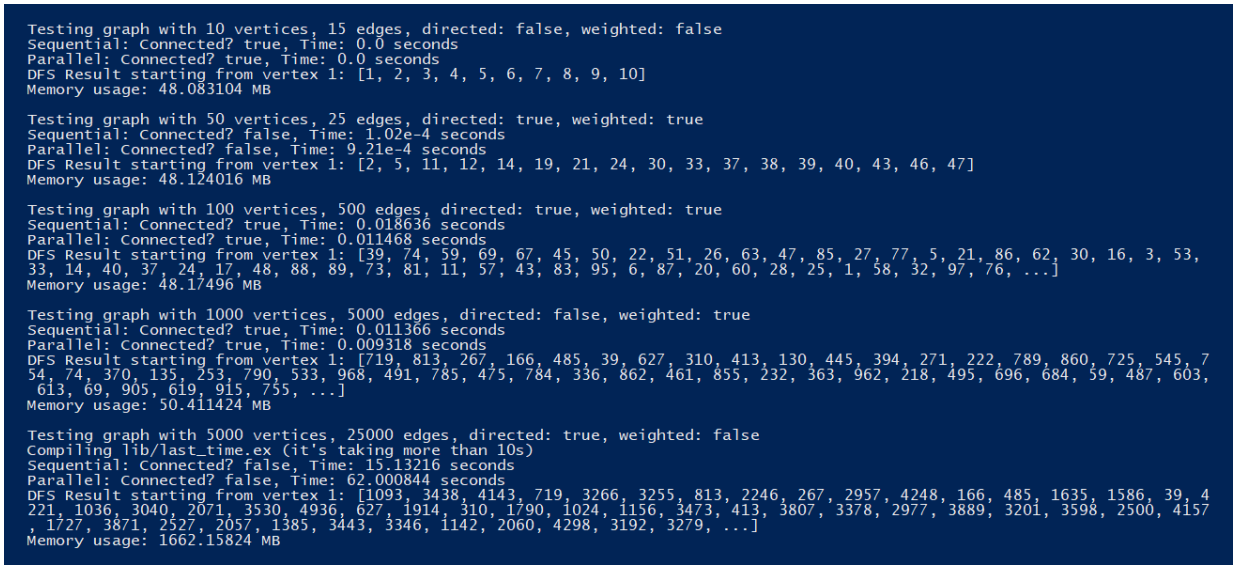
1) Запускаем функцию *generate*, которая подверждает правильность ввода данных, создаёт остовное дерево для соединения узлов, делает проверку на соединённость графа, и рандомно генерирует дополнительные соединения если требуется. На выводе получается смежная матрица соединений в графе (с весами).
2) Запускаются проверки на соединенность – параллельная и непараллельная.
    a. Проверка проходит для каждого узла (сколько узлов, столько и проверок). Если у узла нет соединения, то ему присваивается дополнительное рандомное, и затем присваивается переменная “visited”.
    b. Данный шаг можно распараллелить, разделив проверки по потокам.
3) Время проверки сравнивается для последовательного и параллельного выполнения.
4) DFS:
    a. Генерируется список связностей, по которому алгоритм знает, какие узлы соединены с какими.
    b. Создаётся MapSet под названием “visited”
    c. Запускается рекурсивная функция dfs_helper
        i. Если узел (член MapSet’а) был ранее посещён, то ничего не делаем)
        ii. Иначе, добавляем узел в MapSet “visited”.
        iii. Рекурсивно запускаем dfs_helper, в качестве начального узла передавая ближайшего соседа (по глубине). Если нельзя, то возвращаемся на уровень выше и пробуем снова.
5) Сохраняем граф в формате .dot, по возможности генерируем изображение с помощью Graphviz.



Эксперименты

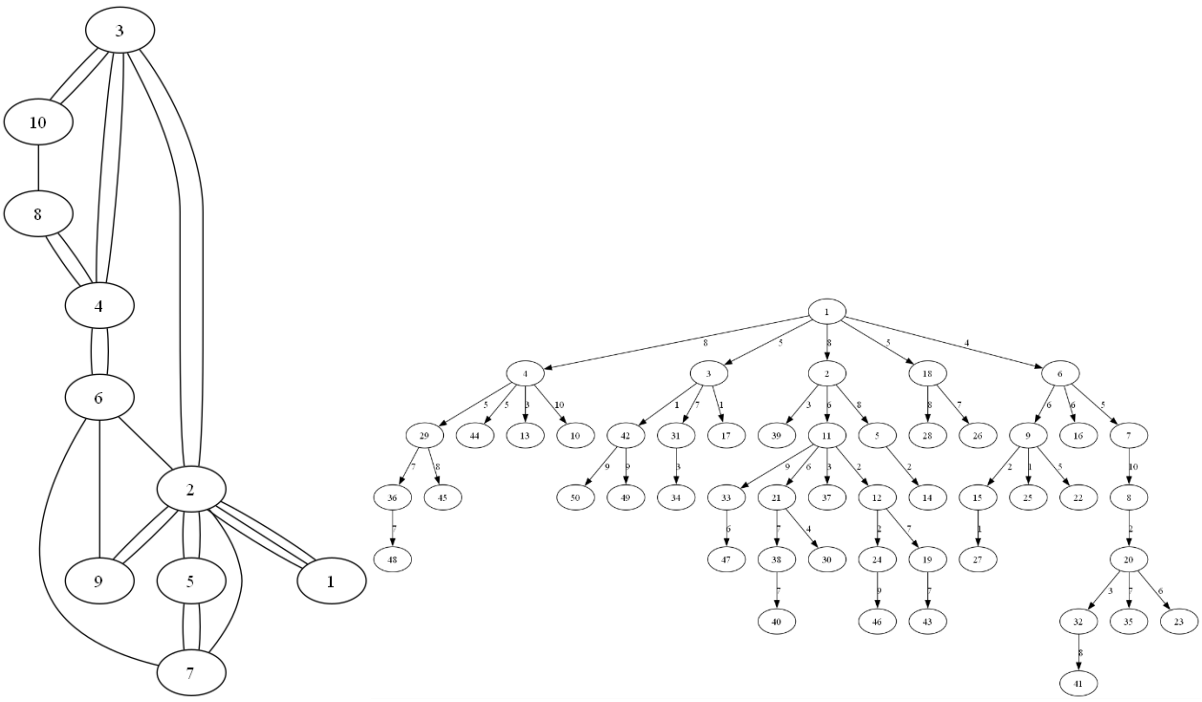
По заданию лабораторной работы, были проведены эксперименты, в которых сравнивались параллельное и непараллельное выполнения программы.

Были сгенерированы 5 графов с различным количеством вершин и ребёр. Было измерено время, требуемое для выполнения задачи в параллельном/непараллельном режиме, а также рассчитан требуемый для вычислений объём памяти.



На приложенном скриншоте можно увидеть параметры графов, время выполнения, массив DFS, и объём требуемой памяти в мегабайтах. Следует отметить, что параллельная проверка соединений, как правило, выигрывает по времени, но незначительно. Исключением является случай с 5000 узлов, где проверка заняла в 4 раза больше времени. Это показывает, что с увеличением размера графа, параллельная обработка начинает проигрывать.

Ниже приведены изображения, демонстрирующие два первых тест-кейса. Изображения последующих тест-кейсов невозможно поместить в данный документ.



Вывод

В результате лабораторной работы №2 был написан алгоритм на языке Elixir, который генерирует граф по заданным параметрам, а затем проводит поиск по глубине в заданном графе.