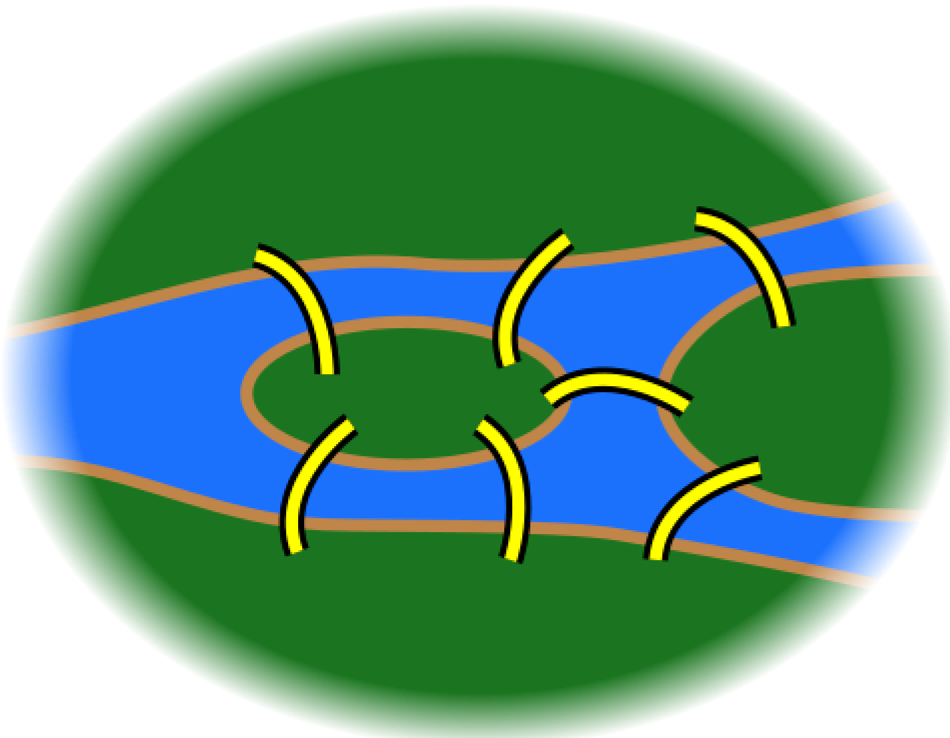
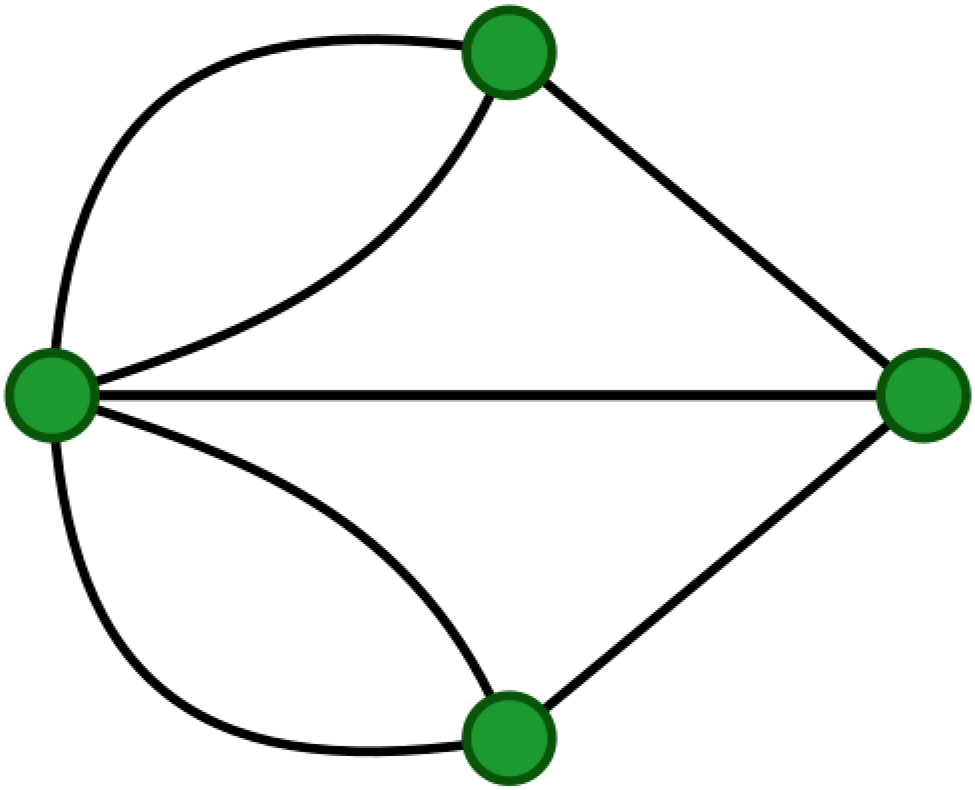
CS 5104

- ## Assignment 1

Solve Königsberg Bridge problem

[Wikipedia]





- **Problem**
  - In the 1700s, the city of Konigsberg had a river running through it.
  - The river split the city into **Four Land Areas** (two islands + two riverbanks).
  - These land area were connected by **seven bridges**.

*Is it possible to take a walk in which each bridge is crossed exactly once?

- **Euler's Analysis**
  - Euler shows that the possibility of a walk through a graph traversing each edge exactly once, depends on the degree of the nodes.
    ```
    The degree of a node is the number of edges touching it.
    ```
  - Euler's argument shows that a necessary condition for the walk of the desired form is the graph be connected and have exactly zero or two nodes of odd degree.

Entering and leaving a land mass

- Imagine you're walking across bridges.
- Every time you enter a land mass by one bridge, you must leave it by another bridge (unless its your start or end point).

- So, except for the start and end, the total number of bridges at a land mass must split evenly.
    - Half used to arrive.
    - Half used to leave.
- That means each land mass (except maybe the start and finish) must have an **even number or bridges** (degree even).

For Königsberg

- The four land masses have bridge counts (degrees).
    - One has 5.
    - The other three each have 3.
- So all 4 are odd.
- SO THERE IS NOT SOLUTION.

- Land Masses = Vertices
- Bridge = Edges

Edges here only serves to show the connected of vertices (land masses).

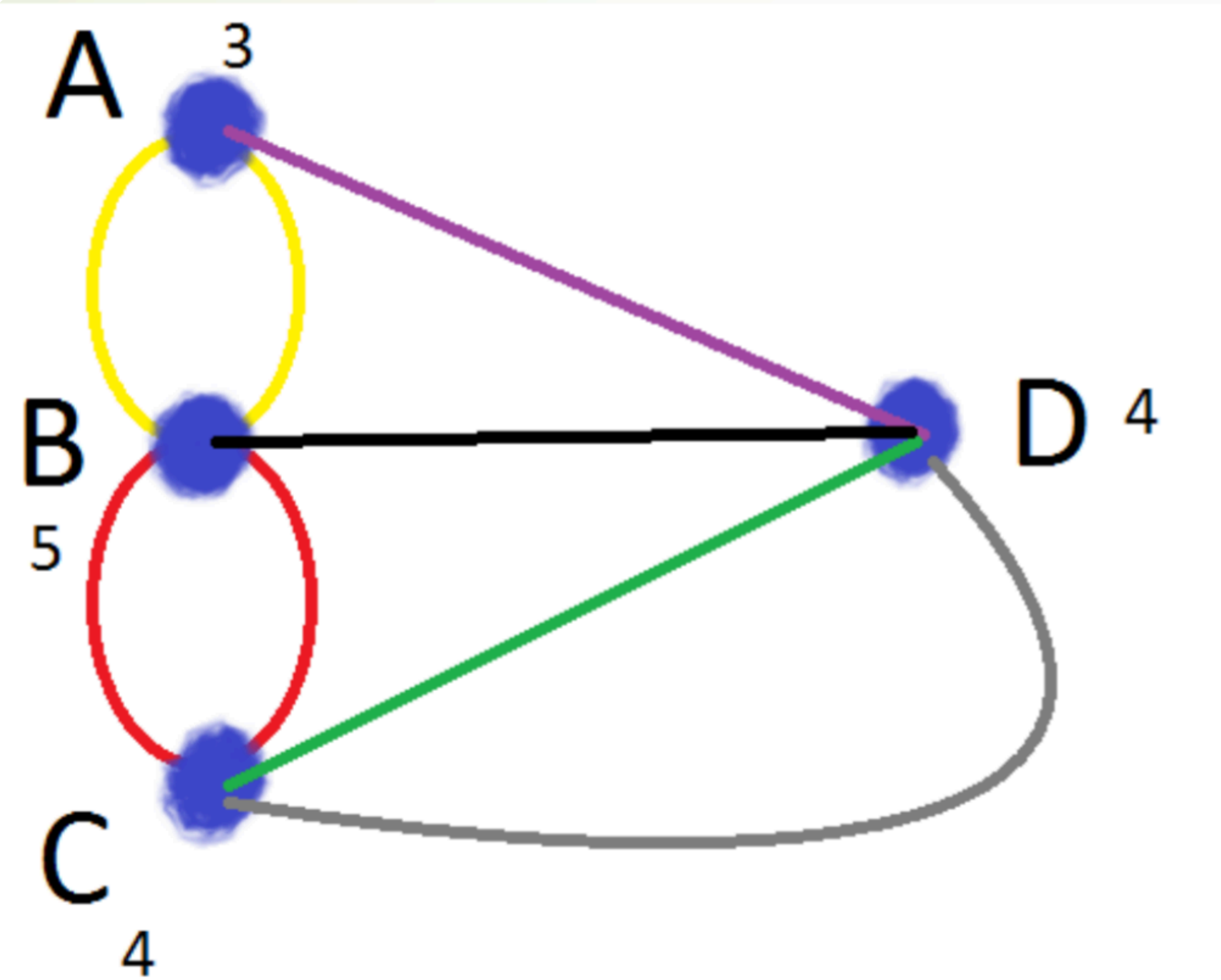
- Every land in Königsberg needs an even number of bridge connected to it, except for maybe the start and the end land mass, think about this, every land needs to bridges, one to enter it and one to get out of it, now the start and end only needs one bridge, one to get out of the starting land mass and the other to get into the end land mass.
- If any land mass has odd number of bridges connected to it, then it would not work because then you will have to enter the land mass through bridge 1, get out of the land mass through bridge 2 and then enter again using bridge 3, at the end, you would be stuck inside the land mass.
- Only the starting and ending land mass can have odd number of bridges.

In the Königsberg bridge problem, every land mass has an odd number of bridges, so it’s impossible to walk across all bridges exactly once.

Solution

A solution exists if we modify the graph, we have to understand that there is not solution for the original problem without making any modifications, because in the original solutions, all our vertices (land mass) has odd number of edges (bridges), so it is inevitable to get trapped in a land mass.

We have to modify (add or remove) edges in the graph to reach a solution.



Here if we connect vertex C and D with one more edge, we will have 2 vertices with even number of edges, and 2 vertices with odd number of edges, so now this will satisfy the Euler rule for Euler Trail, now we can travel each bridge once but we will not end on the land mass we started from.

```
{
  "nbformat": 4,
  "nbformat_minor": 0,
  "metadata": {
    "colab": {
      "name": "Königsberg.ipynb",
      "provenance": [],
      "collapsed_sections": [],
      "authorship_tag": "ABX9TyMYV1loPde4DxAMtDQ8NQWF"
```

```
    },
    "kernelspec": {
      "name": "python3",
      "display_name": "Python 3"
    }
  },
  "cells": [
    {
      "cell_type": "code",
      "metadata": {
        "id": "q2QD2EjUWa4R"
      },
      "source": [
        "BRIDGES = [\n",
        "    \"AaB\",\n",
        "    \"AbB\",\n",
        "    \"AcC\",\n",
        "    \"AdC\",\n",
        "    \"AeD\",\n",
        "    \"BfD\",\n",
        "    \"CgD\",\n",
        "]"
      ],
      "execution_count": null,
      "outputs": []
    },
    {
      "cell_type": "code",
      "metadata": {
        "id": "26TA_wL4W4Ug"
      },
      "source": [
        "def get_walks_starting_from(area, bridges=BRIDGES):\n",
        "    walks = []\n",
        "\n",
        "    def make_walks(area, walked=None, bridges_crossed=None):\n",
        "        walked = walked or area\n",
        "        bridges_crossed = bridges_crossed or ()\n",
        "        # Get all of the bridges connected to `area`\n",
        "        # that haven't been crossed\n",
        "        available_bridges = [\n",
        "            bridge\n",
        "            for bridge in bridges\n",
        "            if area in bridge and bridge not in bridges_crossed\n",
        "        ]\n",
        "\n",
```

```
        "        # Determine if the walk has ended\n",
        "        if not available_bridges:\n",
        "            walks.append(walked)\n",
        "\n",
        "        # Walk the bridge to the adjacent area and recurse\n",
        "        for bridge in available_bridges:\n",
        "            crossing = bridge[1:] if bridge[0] == area else
bridge[1::-1]\n",
        "            make_walks(\n",
        "                area=crossing[-1],\n",
        "                walked=walked + crossing,\n",
        "                bridges_crossed=(bridge, *bridges_crossed),\n",
        "            )\n",
        "\n",
        "    make_walks(area)\n",
        "    return walks"
      ],
      "execution_count": null,
      "outputs": []
    },
    {
      "cell_type": "code",
      "metadata": {
        "id": "HCoqxIi0XOPx",
        "colab": {
          "base_uri": "https://localhost:8080/"
        },
        "outputId": "18a1a33c-2b34-434c-f1e4-74060000ebd9"
      },
      "source": [
        "walks_starting_from = {area: get_walks_starting_from(area) for area
in \"ABCD\"}\n",
        "num_total_walks = sum(len(walks) for walks in
walks_starting_from.values())\n",
        "print(num_total_walks)"
      ],
      "execution_count": null,
      "outputs": [
        {
          "output_type": "stream",
          "text": [
            "372\n"
          ],
          "name": "stdout"
        }
      ]
```

```
    },
    {
      "cell_type": "code",
      "metadata": {
        "id": "ojYcCnAuceCg",
        "colab": {
          "base_uri": "https://localhost:8080/"
        },
        "outputId": "4fe6ac1c-6714-484d-d710-31d1bb807124"
      },
      "source": [
        "walks_starting_from[\"A\"][:3]"
      ],
      "execution_count": null,
      "outputs": [
        {
          "output_type": "execute_result",
          "data": {
            "text/plain": [
              "['AaBbAcCdAeDfB', 'AaBbAcCdAeDgC', 'AaBbAcCgDeAdC']"
            ]
          },
          "metadata": {
            "tags": []
          },
          "execution_count": 4
        }
      ]
    },
    {
      "cell_type": "code",
      "metadata": {
        "colab": {
          "base_uri": "https://localhost:8080/"
        },
        "id": "a6hI7RC2sHvf",
        "outputId": "0f826491-533f-4c4c-adc4-5f1ef353b688"
      },
      "source": [
        "from itertools import chain\n",
        "all_walks = chain.from_iterable(walks_starting_from.values())\n",
        "solutions = [walk for walk in all_walks if len(walk) == 15]\n",
        "print(len(solutions))"
      ],
      "execution_count": null,
      "outputs": [
```

```
        {
          "output_type": "stream",
          "text": [
            "0\n"
          ],
          "name": "stdout"
        }
      ]
    }
  ]
}
```


- It does a depth-first search (recursively) from each area and collects **all** walks that never reuse a bridge.
- It prints the total number of walks found (372) and then checks for any walk that uses **all 7 bridges** (would produce a string length of `1 + 2*7 = 15`. There are 0 such Eulerian walks - as expected for the classic Konigsberg setup.)

Cell 1 - bridge data

```
BRDIGES = [
"AaB",
"AbB",
"AcC",
"AdC",
"AeD",
"BfD",
"CgD",
]
```

- Each string encodes **one bridge**. Format: `XyZ` where `X` and `Z` are uppercase area labels (`A,B,C,D`) and `y` is a lowercase letter that uniquely identifies the bridge.
  - Example: `"AaB"` means a bridge `a` connecting area `A` and area `B`.
- The list has 7 bridges total.

Cell 2 - -`get_walks_starting_from(area, bridges=BRIDGES)

This is the main part of the program. It returns a list of all possible walks that start at the given `area` and never reuses a bridge.

```python
def get_walks_starting_from(area, bridges=BRIDGES):
    walks = []
    def make_walks(area, walked=None, bridges_crossed=None):
        walked = walked or area
        bridges_crossed = bridges_crossed or {}

```

- `walks` collects finished walk strings.
- `make_walks` is a recursive nested function.
- `walked` is the string representation of the path so fair. Initially it is the starting `area` (single uppercase letter).
- `bridges_crossed` is a tuple of bridge ids already crossed (so we don't reuse them).

Available bridges from the current `area`.

```python

available_bridges = [
    bridge
    for bridge in bridges
    if area in bridge and brdige not in bridges_crossed
]
```

- This finds every bridge that contains the current `area` (either as the first uppercase or the third char) and that hasn't been crossed yet.

```python
if not available_bridges:
    walks.append(walked)
```

- If there are no further unused bridges connected to the current area, the recursion stops and the current `walked` string is appended to `walks`.

```python
for bridge in available_bridges:
    crossing = bridge[1:] if bridge[0] == area else bridge[1::-1]

    make_walks{
        area=crossing[-1],
        walked=walked + crossing,
        bridges_crossed=(bridge, *bridges_crossed),
    }
```

- `bridge[1:]` is the substring `"<lowercase><otherArea>"`. Example `"AaB"[1:] == "aB"`.
  - Used when the bridge string's first char (`bridge[0]`) equals the current `area` (i.e, traversing in the direction the bridge string is written).
  - `bridge[1::-1]` reverses the first two chars producing `"<lowercase><thisArea>` (used when starting from the other side). Example: `"AaB"[1::-1] == "aA"`.
- The `crossing` is appended to `walked`. Because `walked` starts with an uppercase area, the walk string alternates area (uppercase) and bridge id (lowercase) and area, etc.
- Example final walk chunk: `'A' + 'aB' + 'bA' + 'cC'` etc.
- `bridges_crossed=(bridge, *bridges_crossed)` prepend the newly used bridge to the tuple of used bridges (so membership checks later exclude it).

Finally `make_walks(area)` is called once to start, and `walks` in returned.

Cell 3 - compute walks from every area and total count

```
walks_starting_from = {area: get_walks_starting_from(area) for area in
"ABCD"}
num_total_walks = sum(len(walks) for walks in walks_starting_from.values())
print(num_total_walks)
```

- Builds a dictionary mapping each start area `'A', 'B', 'C', 'D'` to the list of all possible walks from that start.
- Sums lengths to get the **total number of enumerated walks** (printed as `372`)

Cell 4 - peek at some walks from "A"

```
walks_starting_from["A"][:3]
# e.g. ['AaBbAcCdAeDfB', 'AaBbAcCdAeDgC', 'AaBbAcCgDeAdC']
```

- Each returned string encodes the walks. How to read `'AaBbAcCdAeDfB`:
  - Break into tokens: `A a B b A c C d A e D f B`
  - Interpreted as: start at A, cross bridge `a` -> arrive at `B`, cross bridge `b` -> arrive at `A`, cross `c` -> arrive `C`, cross `d` -> back to `A`, cross `e` -> `D`, cross `f` -> `B`.
  - So the sequence of nodes is `A -> B -> A -> C -> A -> D -> B` using bridges `a,b,c,d,e,f` in that order.

Cell 5 checks for walks that use all bridges exactly once

```
from itertools import chain
all_walks = chain.from_iterable(walks_starting_from.values())
solutions = [walk for walk in all_walks if len(walk) == 15]
print(len(solutions)) # print 0
```

- A walk that uses all 7 bridges has length `1 + 2*7 = 15` characters in this encoding (initial area + 2 chars per bridge).
- The code filed `0` such walks - there is no walk that crosses every bridge exactly once in this graph.

Reason (graph theory): for an undirected graph an Eulerian trail (a trail that uses every edge exactly once) exists only if **0 or 2 vertices have odd degree.** Here the degree counts are:

- `A`: 5 bridges(odd)
- `B`: 3 (odd)
- `C`: 3 (odd)
- `D`: 3 (odd)
  -> 4 vertices with odd degree -> impossible to have an Eulerian Trail. That is why `solutions` is empty.

Complexity

- This code does a full DFS enumerating every possible trail with *no repeated bridges*. The number of such trails grows exponentially with number of edges; enumerating them all is expensive but fine for 7 bridges.

Brute force solution with visuals

```
#include <bits/stdc++.h>
using namespace std;

// Bridges represented as strings "AaB"
vector<string> BRIDGES = {
    "AaB", "AbB", "AcC", "AdC", "AeD", "BfD", "CgD"
};

// One step in the walk
```

```
struct Step {
    char from;
    char bridge;
    char to;
};

void make_walks(char area, vector<Step> walked, unordered_set<string>
bridges_crossed,
                vector<vector<Step>>& walks) {
    vector<string> available_bridges;
    for (auto& bridge : BRIDGES) {
        if ((bridge[0] == area || bridge[2] == area) &&
            bridges_crossed.find(bridge) == bridges_crossed.end()) {
            available_bridges.push_back(bridge);
        }
    }

    if (available_bridges.empty()) {
        walks.push_back(walked);
        return;
    }

    for (auto& bridge : available_bridges) {
        char next_area;
        char bridge_label = bridge[1];
        Step step;

        if (bridge[0] == area) {
            next_area = bridge[2];
            step = {area, bridge_label, next_area};
        } else {
            next_area = bridge[0];
            step = {area, bridge_label, next_area};
        }

        auto new_crossed = bridges_crossed;
        new_crossed.insert(bridge);
        auto new_walked = walked;
        new_walked.push_back(step);

        make_walks(next_area, new_walked, new_crossed, walks);
    }
}

vector<vector<Step>> get_walks_starting_from(char area) {
    vector<vector<Step>> walks;
```

```
    make_walks(area, {}, {}, walks);
    return walks;
}

void print_walk(const vector<Step>& walk) {
    if (walk.empty()) return;
    cout << walk[0].from;
    for (auto& step : walk) {
        cout << " -(" << step.bridge << ")-> " << step.to;
    }
    cout << "\n";
}

// Export a walk as a DOT file for Graphviz with direction arrows
void export_walk_to_dot_directed(const vector<Step>& walk, const string&
filename) {
    ofstream out(filename);
    out << "digraph Walk {\n";
    out << "  node [shape=circle, style=filled, fillcolor=lightblue];\n";

    // Print all steps as directed edges
    for (int i = 0; i < (int)walk.size(); i++) {
        auto& step = walk[i];
        out << "  " << step.from << " -> " << step.to
            << " [label=\"" << step.bridge << " (" << i+1 << ")\",
color=red, penwidth=2];\n";
    }

    // Highlight start and end nodes
    out << "  " << walk[0].from << " [fillcolor=green];\n";
    out << "  " << walk.back().to << " [fillcolor=orange];\n";

    out << "}\n";
    out.close();

    cout << "DOT file with directions written to " << filename
         << ". Use `dot -Tpng " << filename << " -o walk.png` to render.\n";
}


int main() {
    map<char, vector<vector<Step>>> walks_starting_from;
    string areas = "ABCD";
    for (char area : areas) {
        walks_starting_from[area] = get_walks_starting_from(area);
    }
```

```
    cout << "Total number of walks: ";
    int num_total_walks = 0;
    for (auto& [area, walks] : walks_starting_from) {
        num_total_walks += walks.size();
    }
    cout << num_total_walks << "\n";

    cout << "Example walk from A:\n";
    auto example = walks_starting_from['A'][0];
    print_walk(example);

    // Export this walk to Graphviz DOT file
    export_walk_to_dot_directed(example, "walk.dot");

    return 0;
}
```

- Steps to run
- `g++ konigsberg.cpp -o konigsberg`
- `./konigsberg`
- `dot -Tpng walk.dot -o walk.png`

FINAL SOLUTION

```
#include <bits/stdc++.h>
using namespace std;

/*
 * BRIDGE TRAVERSAL PROBLEM (Based on Königsberg Bridge Problem)
 *
 * This program solves the classic problem of finding a path that crosses
 * every bridge exactly once. This is equivalent to finding an Eulerian
 * path in a multigraph (graph with multiple edges between nodes).
 *
 * Nodes represent land masses (islands, riverbanks)
 * Edges represent bridges connecting these land masses
 */

// Define the bridges as pairs of connected nodes (land masses)
```

```
// Multiple bridges between same nodes are allowed (multigraph)
vector<pair<char, char>> BRIDGES = {
    {'A', 'B'},  // Bridge 1: A to B
    {'A', 'B'},  // Bridge 2: A to B (parallel bridge)
    {'A', 'C'},  // Bridge 3: A to C
    {'A', 'C'},  // Bridge 4: A to C (parallel bridge)
    {'A', 'D'},  // Bridge 5: A to D
    {'B', 'D'},  // Bridge 6: B to D
    {'C', 'D'}   // Bridge 7: C to D
};

/**
 * Build adjacency list representation of the multigraph
 * Each node maps to a list of its neighbors (including duplicates for
parallel edges)
 *
 * @param bridges: Vector of bridge connections
 * @return: Adjacency list where each node has a list of connected nodes
 */
unordered_map<char, vector<char>> build_graph(const vector<pair<char,
char>>& bridges) {
    unordered_map<char, vector<char>> graph;

    // For each bridge, add connections in both directions (undirected
graph)
    for (const auto& [u, v] : bridges) {
        graph[u].push_back(v);  // u connects to v
        graph[v].push_back(u);  // v connects to u
    }
    return graph;
}

/**
 * Check if an Eulerian path exists in the graph
 *
 * EULERIAN PATH RULES:
 * - Eulerian Circuit (closed path): ALL nodes have even degree
 * - Eulerian Path (open path): EXACTLY 2 nodes have odd degree
 * - No Eulerian path: More than 2 nodes have odd degree
 *
 * @param graph: Adjacency list representation
 * @param odd_vertices: Output parameter - nodes with odd degree
 * @return: True if Eulerian path exists, false otherwise
 */
bool has_eulerian_path(const unordered_map<char, vector<char>>& graph,
vector<char>& odd_vertices) {
```

```
    odd_vertices.clear();

    // Count degree (number of connections) for each node
    for (const auto& [node, neighbors] : graph) {
        if (neighbors.size() % 2 == 1) {  // Odd degree
            odd_vertices.push_back(node);
        }
    }

    // Eulerian path exists if 0 or 2 nodes have odd degree
    return (odd_vertices.size() == 0 || odd_vertices.size() == 2);
}

/**
 * Find Eulerian path using Hierholzer's Algorithm
 *
 * ALGORITHM STEPS:
 * 1. Start from a node with odd degree (if any), otherwise any node
 * 2. Use DFS with backtracking to traverse edges exactly once
 * 3. Use a stack to handle dead ends and backtracking
 * 4. Build path in reverse order, then reverse at end
 *
 * @param graph: Mutable adjacency list (edges removed during traversal)
 * @return: Vector representing the Eulerian path
 */
vector<char> find_eulerian_path(unordered_map<char, vector<char>>& graph) {
    // Choose starting node: prefer odd degree node if available
    char start = graph.begin()->first;  // Default start
    for (const auto& [node, neighbors] : graph) {
        if (neighbors.size() % 2 == 1) {  // Found odd degree node
            start = node;
            break;
        }
    }

    vector<char> path;        // Final Eulerian path
    stack<char> st;          // DFS stack for traversal
    st.push(start);

    // Convert to multiset for efficient edge removal
    // multiset allows multiple edges between same nodes
    unordered_map<char, multiset<char>> adj;
    for (const auto& [u, neighbors] : graph) {
        for (const auto& v : neighbors) {
            adj[u].insert(v);
        }
```

```
    }

    // Hierholzer's algorithm main loop
    while (!st.empty()) {
        char u = st.top();

        if (!adj[u].empty()) {
            // Current node has unvisited edges
            char v = *adj[u].begin();  // Pick any adjacent node

            // Remove edge u-v from both directions (undirected graph)
            adj[u].erase(adj[u].begin());
            adj[v].erase(adj[v].find(u));

            st.push(v);  // Continue DFS from v
        } else {
            // Dead end: no more edges from current node
            // Add to path and backtrack
            path.push_back(u);
            st.pop();
        }
    }

    // Path was built in reverse order, so reverse it
    reverse(path.begin(), path.end());
    return path;
}

/**
 * Export enhanced graph visualization to DOT format for Graphviz
 * Creates a rich, informative visualization showing:
 * - Original graph structure
 * - Eulerian path (if exists) with step numbers
 * - Node degrees and classifications
 * - Bridge usage statistics
 * - Comprehensive legend
 *
 * @param bridges: Original bridge connections
 * @param path: Eulerian path (empty if none exists)
 * @param filename: Output DOT file name
 */
void export_to_dot(const vector<pair<char, char>>& bridges,
                   const vector<char>& path,
                   const string& filename) {
    ofstream out(filename);
```

```cpp
    // DOT file header with graph properties
    out << "graph BridgeTraversal {\n";
    out << "  // Graph layout and styling\n";
    out << "  layout=neato;\n";
    out << "  overlap=false;\n";
    out << "  splines=curved;\n";
    out << "  bgcolor=\"#f8f9fa\";\n";
    out << "  fontname=\"Arial Bold\";\n";
    out << "  fontsize=16;\n\n";

    // Calculate node degrees for labeling
    unordered_map<char, int> degrees;
    for (const auto& [u, v] : bridges) {
        degrees[u]++;
        degrees[v]++;
    }

    // Style nodes based on their role and degree
    out << "  // Node styling with degree information\n";
    for (const auto& [node, degree] : degrees) {
        string color = "lightblue";
        string shape = "circle";
        string extra_info = "";

        // Color code based on degree (odd/even)
        if (degree % 2 == 1) {
            color = "lightcoral";  // Odd degree nodes
            extra_info = " (odd)";
        } else {
            color = "lightgreen";  // Even degree nodes
            extra_info = " (even)";
        }

        // Special styling for start/end nodes in path
        if (!path.empty()) {
            if (node == path.front()) {
                color = "gold";
                shape = "doublecircle";
                extra_info += " START";
            } else if (node == path.back() && path.front() != path.back()) {
                color = "orange";
                shape = "doublecircle";
                extra_info += " END";
            }
        }
```

```
        out << "  " << node << " [label=\"" << node << "\\ndegree:" <<
degree
            << extra_info << "\", fillcolor=\"" << color << "\", shape=" <<
shape
            << ", style=filled, fontname=\"Arial Bold\", fontsize=12];\n";
    }
    out << "\n";

    // Draw all bridges with default styling
    out << "  // All bridges (default styling)\n";
    unordered_map<string, int> edge_count;  // Count parallel edges

    for (size_t i = 0; i < bridges.size(); i++) {
        char u = bridges[i].first;
        char v = bridges[i].second;

        // Create unique edge key (sorted to handle undirected edges)
        string edge_key = string(1, min(u, v)) + string(1, max(u, v));
        edge_count[edge_key]++;

        out << "  " << u << " -- " << v
            << " [color=\"#cccccc\", penwidth=2, style=solid, "
            << "id=\"bridge_" << (i+1) << "\"];\n";
    }

    // Highlight Eulerian path with step-by-step progression
    if (!path.empty()) {
        out << "\n  // Eulerian path highlighting\n";

        // Create color gradient for path steps
        for (size_t i = 0; i + 1 < path.size(); i++) {
            // Calculate color intensity based on step position
            double progress = (double)i / (path.size() - 2);  // 0 to 1
            int red = 255;  // Keep red constant
            int green = (int)(255 * (1 - progress * 0.7));  // Fade from 255
to ~76
            int blue = (int)(255 * (1 - progress));         // Fade from 255
to 0

            char color_hex[8];
            sprintf(color_hex, "#%02x%02x%02x", red, green, blue);

            out << "  " << path[i] << " -- " << path[i+1]
                << " [color=\"" << color_hex << "\", penwidth=5, "
                << "label=\" " << (i+1) << " \", fontcolor=\"black\", "
                << "fontname=\"Arial Bold\", fontsize=10, "
```

```
                << "labeltooltip=\"Step " << (i+1) << ": "
                << path[i] << " to " << path[i+1] << "\"];\n";
        }
    }

    // Add comprehensive title and statistics
    out << "\n  // Title and statistics\n";
    string title = path.empty() ?
        "Bridge Traversal Problem\\n✖ NO EULERIAN PATH EXISTS" :
        "Bridge Traversal Problem\\n□ EULERIAN PATH FOUND";

    out << "  labelloc=t;\n";
    out << "  label=\"" << title << "\\n\\n";
    out << "Total Bridges: " << bridges.size() << "\\n";
    out << "Land Masses: " << degrees.size() << "\\n";

    if (!path.empty()) {
        out << "Path Length: " << path.size() << " nodes\\n";
        out << "Steps: " << (path.size() - 1) << " bridge crossings";
    } else {
        // Count odd degree nodes for explanation
        int odd_count = 0;
        for (const auto& [node, degree] : degrees) {
            if (degree % 2 == 1) odd_count++;
        }
        out << "Odd degree nodes: " << odd_count << " (need exactly 0 or
2)";
    }
    out << "\";\n";

    // Enhanced legend
    out << "\n  // Enhanced legend\n";
    out << "  subgraph cluster_legend {\n";
    out << "    style=filled;\n";
    out << "    fillcolor=\"#ffffff\";\n";
    out << "    color=\"#666666\";\n";
    out << "    penwidth=2;\n";
    out << "    label=\"□ LEGEND\";\n";
    out << "    fontname=\"Arial Bold\";\n";
    out << "    fontsize=14;\n\n";

    // Legend nodes
    out << "    legend_unused [label=\"Unused Bridge\", shape=plaintext,
fontname=\"Arial\"];\n";
    out << "    legend_used [label=\"Traversed Bridge\\n(numbered by
step)\", shape=plaintext, fontname=\"Arial\"];\n";
```

```
    out << "    legend_even [label=\"Even Degree\\n(green)\",
shape=plaintext, fontname=\"Arial\"];\n";
    out << "    legend_odd [label=\"Odd Degree\\n(red)\", shape=plaintext,
fontname=\"Arial\"];\n";
    out << "    legend_start [label=\"Start Node\\n(gold)\",
shape=plaintext, fontname=\"Arial\"];\n";
    out << "    legend_end [label=\"End Node\\n(orange)\", shape=plaintext,
fontname=\"Arial\"];\n\n";

    // Legend edges
    out << "    legend_unused -- legend_used [color=\"#cccccc\",
penwidth=2];\n";
    out << "    legend_used -- legend_even [color=\"#ff6666\", penwidth=5,
label=\" 1 \"];\n";
    out << "    legend_even -- legend_odd [style=invis];\n";
    out << "    legend_odd -- legend_start [style=invis];\n";
    out << "    legend_start -- legend_end [style=invis];\n";
    out << "  }\n";

    out << "}\n";
    out.close();

    // Provide user instructions
    cout << "\n□ Enhanced visualization created!" << endl;
    cout << "□ DOT file: " << filename << endl;
    cout << "\n□ To generate images:" << endl;
    cout << "  PNG: dot -Tpng " << filename << " -o bridge_graph.png" <<
endl;
    cout << "  SVG: dot -Tsvg " << filename << " -o bridge_graph.svg" <<
endl;
    cout << "  PDF: dot -Tpdf " << filename << " -o bridge_graph.pdf" <<
endl;
    cout << "\n□ The visualization shows:" << endl;
    cout << "  • Node degrees (odd=red, even=green)" << endl;
    cout << "  • Bridge traversal order (numbered steps)" << endl;
    cout << "  • Start/end points highlighted" << endl;
    cout << "  • Color gradient showing traversal progression" << endl;
}

/**
 * MAIN FUNCTION
 * Orchestrates the entire bridge traversal analysis:
 * 1. Build graph from bridge connections
 * 2. Check if Eulerian path exists
 * 3. Find the path (if possible)
 * 4. Generate visualization
```

```
 * 5. Display results
 */
int main() {
    cout << "□ BRIDGE TRAVERSAL PROBLEM SOLVER" << endl;
    cout << "=====================================" << endl;
    cout << "Based on the famous Königsberg Bridge Problem" << endl;
    cout << "Goal: Cross every bridge exactly once\n" << endl;

    // Step 1: Build graph representation
    cout << "□ Building graph from " << BRIDGES.size() << " bridges..." <<
endl;
    auto graph = build_graph(BRIDGES);

    // Display graph structure
    cout << "\n□ Graph Structure:" << endl;
    for (const auto& [node, neighbors] : graph) {
        cout << "  Land mass " << node << ": degree " << neighbors.size()
             << " (connected to: ";
        for (size_t i = 0; i < neighbors.size(); i++) {
            cout << neighbors[i];
            if (i + 1 < neighbors.size()) cout << ", ";
        }
        cout << ")" << endl;
    }

    // Step 2: Check Eulerian path existence
    cout << "\n□ Checking Eulerian path existence..." << endl;
    vector<char> odd_vertices;
    bool path_exists = has_eulerian_path(graph, odd_vertices);

    cout << "Nodes with odd degree: ";
    if (odd_vertices.empty()) {
        cout << "None (Eulerian circuit possible)" << endl;
    } else {
        for (size_t i = 0; i < odd_vertices.size(); i++) {
            cout << odd_vertices[i];
            if (i + 1 < odd_vertices.size()) cout << ", ";
        }
        cout << endl;
    }

    if (!path_exists) {
        cout << "\n✗ NO EULERIAN PATH EXISTS" << endl;
        cout << "Reason: " << odd_vertices.size() << " nodes have odd
degree" << endl;
        cout << "(Need exactly 0 or 2 nodes with odd degree)" << endl;
```

```
        cout << "\nThis is like the original Königsberg problem -
unsolvable!" << endl;

        // Still generate visualization for educational purposes
        export_to_dot(BRIDGES, {}, "bridge_graph.dot");
        return 0;
    }

    // Step 3: Find Eulerian path
    cout << "\n□ EULERIAN PATH EXISTS!" << endl;
    cout << "□ Finding optimal route..." << endl;

    auto path = find_eulerian_path(graph);

    // Step 4: Display results
    cout << "\n□ SOLUTION FOUND!" << endl;
    cout << "=================" << endl;
    cout << "Eulerian path: ";
    for (size_t i = 0; i < path.size(); i++) {
        cout << path[i];
        if (i + 1 < path.size()) cout << " → ";
    }
    cout << endl;

    cout << "\n□ Statistics:" << endl;
    cout << "  • Total steps: " << (path.size() - 1) << endl;
    cout << "  • Bridges crossed: " << BRIDGES.size() << "/" <<
BRIDGES.size() << " (100%)" << endl;
    cout << "  • Starting point: " << path.front() << endl;
    cout << "  • Ending point: " << path.back() << endl;

    if (path.front() == path.back()) {
        cout << "  • Path type: Eulerian Circuit (closed loop)" << endl;
    } else {
        cout << "  • Path type: Eulerian Path (open trail)" << endl;
    }

    // Step 5: Generate visualization
    cout << "\n□ Generating enhanced visualization..." << endl;
    export_to_dot(BRIDGES, path, "bridge_graph.dot");

    cout << "\n✨ Analysis complete! Check the generated visualization for a
detailed view." << endl;
    return 0;
}
```

- Steps to run
- `g++ konigsberg.cpp -o konigsberg`
- `./konigsberg`
- `dot -Tpng walk.dot -o walk.png`

References

- https://tomrocksmaths.com/wp-content/uploads/2024/07/the-konigsberg-bridge-problem-and-graph-theory-1-beatrice-lawrence.pdf
- https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg
- https://www.cs.kent.edu/~dragan/ST-Spring2016/The%20Seven%20Bridges%20of%20Konigsberg-Euler%27s%20solution.pdf
- https://youtu.be/WWhGcwICoXE?feature=shared33
- https://www.cs.cornell.edu/courses/cs280/2006sp/280wk13.pdf
- https://graphviz.org/documentation/
- https://www.youtube.com/watch?v=GNr98tA1TaA