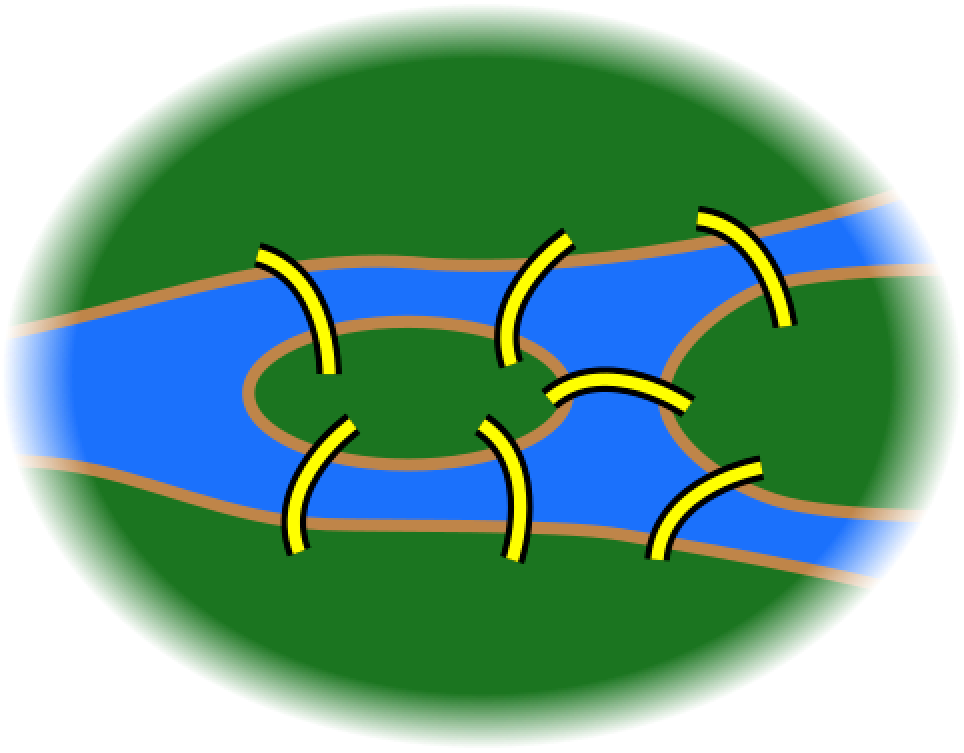
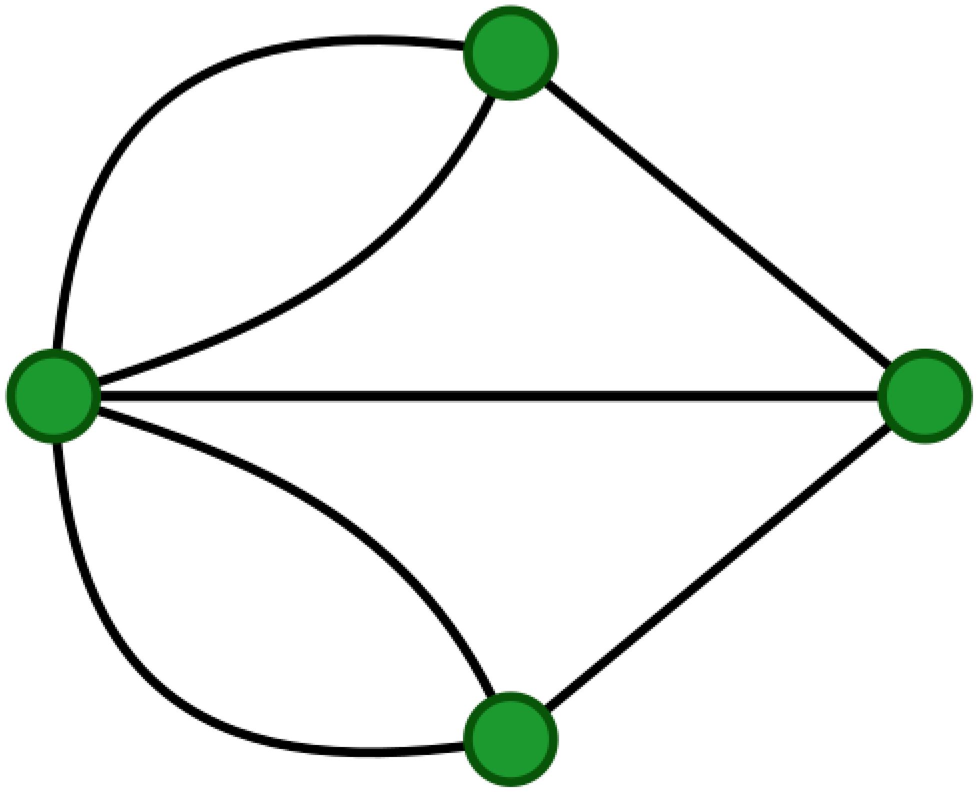
CS 5104

- **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).

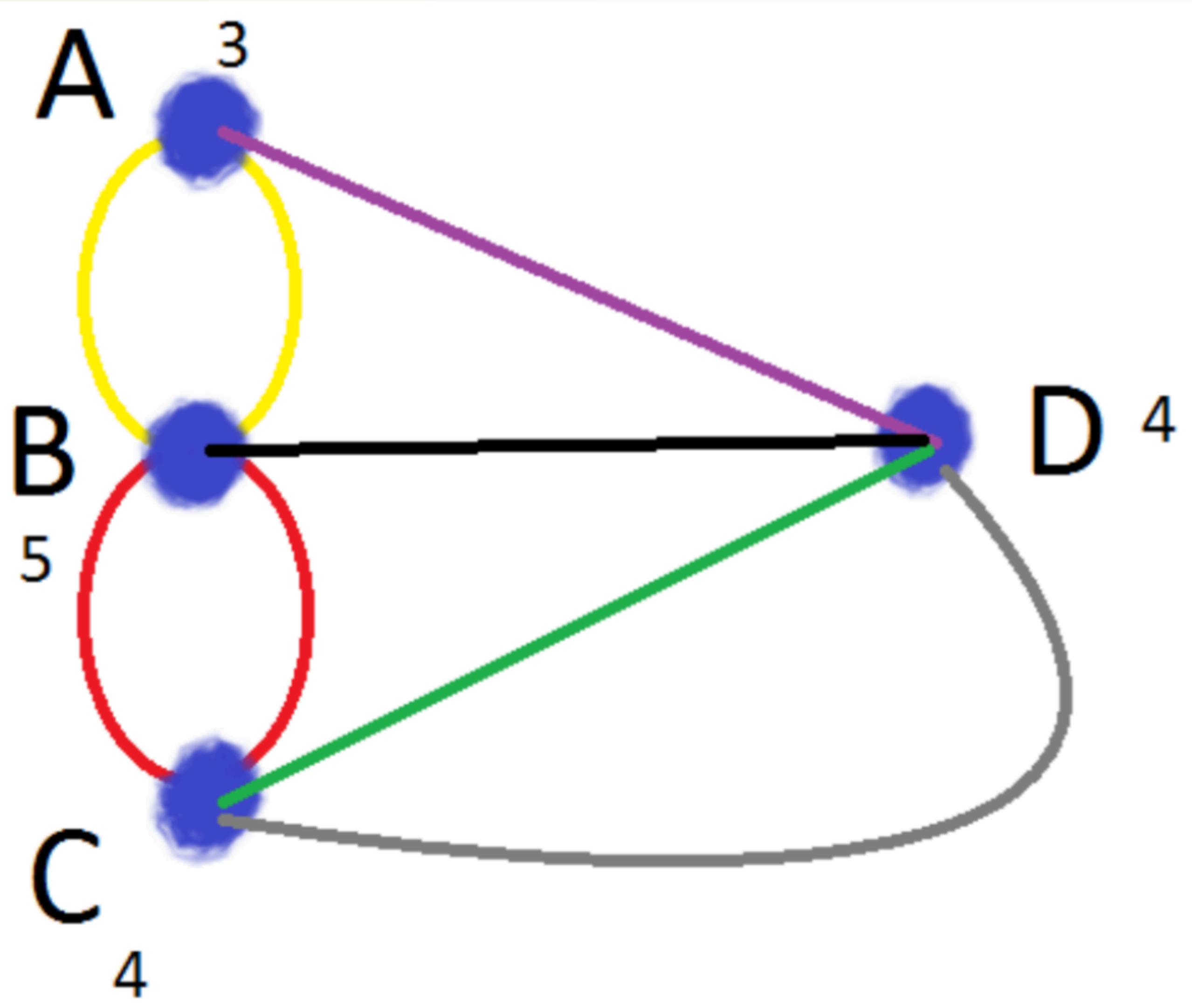
- Every land in Königsberg needs an even number of bridge connected to it, except for maybe the start and the end land mass, think about this, every land needs to bridges, one to enter it and one to get out of it, now the start and end only needs one bridge, one to get out of the starting land mass and 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.

CODE IN C++

```

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

// Bridges represented as strings "AaB", meaning
// Area 'A' connected to Area 'B' by bridge 'a'.
vector<string> BRIDGES = {
    "AaB",
    "AbB",
    "AcC",
```

```cpp
    "AdC",
    "AeD",
    "BfD",
    "CgD"
};

// Walks are stored as a sequence of (Area, bridgeLabel, Area)
struct Step {
    char from;
    char bridge;
    char to;
};

// Recursive helper to generate walks
void make_walks(
    char area,                                  // current area
    vector<Step> walked,                        // path so far
    unordered_set<string> bridges_crossed,      // already used bridges
    vector<vector<Step>>& walks                 // result container
) {
    // Collect available bridges connected to current area
    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 no more bridges are available, the walk ends
    if (available_bridges.empty()) {
        walks.push_back(walked);
        return;
    }

    // Try walking across each available bridge
    for (auto& bridge : available_bridges) {
        char next_area;
        char bridge_label = bridge[1];
        Step step;

        if (bridge[0] == area) {
            // Move from bridge[0] -> bridge[2]
            next_area = bridge[2];
            step = {area, bridge_label, next_area};
        } else {
            // Move from bridge[2] -> bridge[0]
            next_area = bridge[0];
            step = {area, bridge_label, next_area};
        }

        // Add this bridge to crossed set
        auto new_crossed = bridges_crossed;
        new_crossed.insert(bridge);

        // Extend walked path
        auto new_walked = walked;
        new_walked.push_back(step);

        // Recurse deeper
        make_walks(next_area, new_walked, new_crossed, walks);
    }
}

// Wrapper: get all walks starting from a given area
vector<vector<Step>> get_walks_starting_from(char area) {
    vector<vector<Step>> walks;
    make_walks(area, {}, {}, walks);
    return walks;
}

// Pretty-print a walk
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";
}

int main() {
    // Compute walks starting from each area A, B, C, D
    map<char, vector<vector<Step>>> walks_starting_from;
    string areas = "ABCD";
    for (char area : areas) {
        walks_starting_from[area] = get_walks_starting_from(area);
    }

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

    // Print first 3 walks starting from A
    cout << "Some walks starting from A:\n";
    for (int i = 0; i < min(3, (int)walks_starting_from['A'].size()); i++) {
        print_walk(walks_starting_from['A'][i]);
    }
    cout << "\n";

    // Check for Eulerian walks (using all 7 bridges exactly once)
    // Each walk must have exactly 7 steps
    int solutions = 0;
    for (auto& [area, walks] : walks_starting_from) {
        for (auto& walk : walks) {
            if ((int)walk.size() == 7) {
                solutions++;
            }
        }
    }
    cout << "Number of Eulerian walks (using all bridges): " << solutions <<
"\n";

    return 0;
}
```

```
// output

Total number of walks: 372

Some walks starting from A:
A -(a)-> B -(b)-> A -(c)-> C -(d)-> A -(e)-> D -(f)-> B
A -(a)-> B -(b)-> A -(c)-> C -(d)-> A -(e)-> D -(g)-> C
A -(a)-> B -(b)-> A -(c)-> C -(g)-> D -(e)-> A -(d)-> C

Number of Eulerian walks (using all bridges): 0
```

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`

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/WWhGcwlCoXE?feature=shared33
- https://www.cs.cornell.edu/courses/cs280/2006sp/280wk13.pdf