Powers of Adjacency and Transition Matrices

Will Corcoran

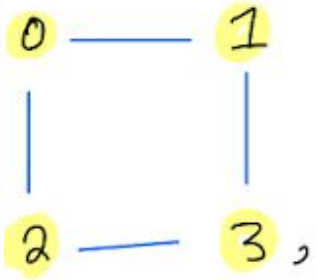
Adjacency Matrices

- Given a graph, G = (V, E), Adjacency Matrix, A, is defined by

$$a_{ij} = 1 \text{ if } i, j \in V : (i, j) \in E, \text{ else } 0$$

- Main uses of adjacency matrix power, A^k :
  - Counting the number of *k*-length walks from i to *j*
  - Shortest paths
    - if *k* is the first integer where a_{ij} is nonnegative, then *k* is the length of the shortest path
  - DAG (directed acyclic graph) detection
    - if there exists some *k* where A^k is the zero matrix (i.e. *A* is nilpotent), then G is a DAG
    - simple explanation: if you're counting *k*-length walks and you reach a sink vertex, the walk ends

Power of Adjacency Matrices (ex.)



$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

walks of length 3 (A^3):

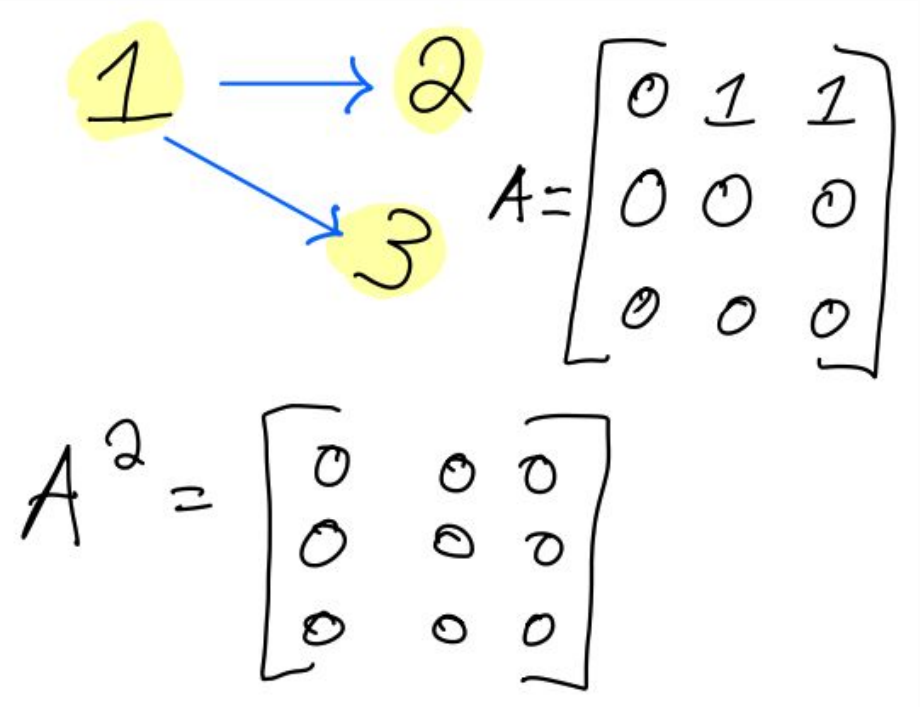
$$\begin{bmatrix} 0 & 4 & 4 & 0 \\ 4 & 0 & 0 & 4 \\ 4 & 0 & 0 & 4 \\ 0 & 4 & 4 & 0 \end{bmatrix}$$

example from (0, 1)

0-1-3-1 0-2-3-1

0-1-0-1 0-2-0-1

Power of Adjacency Matrices (ex. DAG)

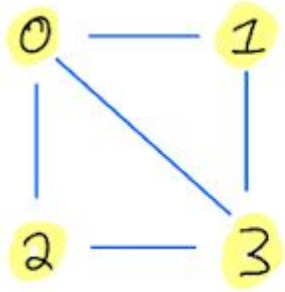


Transition Matrices

- Used for probabilistic methods (e.g. Markov chains/MCMC/random walks)
- Transition matrix defined as

$$t_{ij} = 1/d_i \text{ if } d_i > 0 \text{ and } (i, j) \in E \text{ else } 0$$

- t_{ij}^k is the probability of lying in state *j* after beginning in *i* and taking *k* steps



$$T = \begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/2 & 0 & 0 & 1/2 \\ 1/2 & 0 & 0 & 1/2 \\ 1/3 & 1/3 & 1/3 & 0 \end{bmatrix}$$

T^2 (ie probability of starting at i and ending in j after 2 steps)

$$\begin{bmatrix} 4/9 & 1/9 & 1/9 & 1/3 \\ 1/6 & 1/3 & 1/3 & 1/6 \\ 1/6 & 1/3 & 1/3 & 1/6 \\ 1/3 & 1/9 & 1/9 & 4/9 \end{bmatrix}$$

CXXGraph

- Basic information:
  - Open-source
  - Header-Only C++ library for graph representation and algorithms
  - Graph type supports four matrices: degree, laplacian, transition, and adjacency
  - Power functionality focuses on *transition* and *adjacency* matrices

- Specifications:
  - Build: *CMake 3.9*
  - Unit-Test: *Google Test*

```
CXXGraph::Node<int> a("a", 1);
CXXGraph::Node<int> b("b", 1);
CXXGraph::Node<int> c("c", 1);

CXXGraph::UndirectedEdge<int> e1(0, a, b);
CXXGraph::UndirectedEdge<int> e2(1, b, c);
CXXGraph::UndirectedEdge<int> e3(2, c, a);

CXXGraph::Graph<int> graph;
graph.addEdge(&e1);
graph.addEdge(&e2);
graph.addEdge(&e3);

auto adjMatrix = graph.getAdjMatrix();
```

Power Algorithm (fast exponentiation)

Input: M, a real matrix with values between $[0, 1]$, and a nonnegative integer, k
Output: M^k, the result of Mx M, repeated k times

1. Initialize A to the I_n identity matrix, B to M
2. If k is 0, return A
3. If k is odd, *A* = *A* * *B*
4. Then, set *B* = *B* * *B*
5. Right-shift k by 1
6. Repeat steps 2 - 5

The actual implementation requires some massaging to get the adjacency and transition matrices into a uniform form (which I will show).

Example of Repeated Squaring

Repeated squaring (with numbers):

M = 3, K = 5 (101)

0. A = 1, B = 3, K = 5
1. A = 3, B = 9, K = 2 (10)
2. A = 3, B = 81, K = 1 (1)
3. A = 243, B = 6561, K = 0

Done! return 243

Fast Exponentiation

```cpp
/**
 * @brief exponentation takes a matrix of arithmetic type and
 * raises it to the power of k.
 * @param mat a square matrix
 * @param k a nonnegative integer
 * @return M, the result of mat^k
 */
template <typename T>
std::vector<std::vector<T>> exponentiation(std::vector<std::vector<T>> mat,
                                           unsigned int k) {
  static_assert(std::is_arithmetic<T>::value,
                "Type T must be any arithmetic type");

  // validate size and shape of matrix
  if (mat.size() == 0 || mat.size() != mat[0].size()) {
    throw std::invalid_argument("Matrix must be square and at least 1x1.");
  }
  int n = static_cast<int>(mat.size());
  std::vector<std::vector<T>> res(n, std::vector<T>(n, 0));

  // build identity matrix
  for (int i = 0; i < n; i++) res[i][i] = 1;

  // fast exponentiation!
  while (k) {
    if (k & 1) res = matMult(res, mat);
    mat = matMult(mat, mat);
    k >>= 1;
  }

  return res;
}
```

Matrix Multiplication

```cpp
/**
 * @brief simple matrix multiplication of two matrices
 * A and B
 * @param a matrix A
 * @param b matrix B
 * @return A times B
 */
template <typename T>
std::vector<std::vector<T>> matMult(const std::vector<std::vector<T>> &a,
                                    const std::vector<std::vector<T>> &b) {
  static_assert(std::is_arithmetic<T>::value,
                "Type T must be an arithmetic type");

  // two square matrices both of size N x N where N > 0
  if (a.empty() || a[0].size() != b.size() || a.size() != a[0].size() ||
      b.size() != b[0].size()) {
    throw std::invalid_argument(
        "Matrix must have valid dimensions and be at least 1x1.");
  }
```

```cpp
  int n = static_cast<int>(a.size());  // N x N matrix
  std::vector<std::vector<T>> res(n, std::vector<T>(n, 0));

  // O(n^3) matrix multiplication
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        res[i][j] += a[i][k] * b[k][j];
      }
    }
  }

  return res;
}
```

Adjacency Matrix Implementation

```
template <typename T>
using AdjacencyMatrix = std::unordered_map<
    shared<const Node<T>>,
    std::vector<std::pair<shared<const Node<T>>, shared<const Edge<T>>>>,
    nodeHash<T>>;
```

```
struct PowAdjResult_struct {
  bool success = false;
  std::string errorMessage = "";
  std::unordered_map<std::pair<std::string, std::string>, unsigned long long, pair_hash>
      result = {};
};
typedef PowAdjResult_struct PowAdjResult;
```

```
/**
 * @brief This function raises the adjacency matrix to some k.
 * Best used for counting number of k-length walks from i to j.
 * @param k value by which to raise matrix
 * @return (success, errorMessage, matrix): where matrix is equivalent to A^k
 */
template <typename T>
const PowAdjResult matrixPow(const shared<AdjacencyMatrix<T>> &adj,
                             unsigned int k) {
  PowAdjResult result;
  result.success = false;
  result.errorMessage = "";

  // convert back and forth between user ids and index in temporary adj matrix
  std::unordered_map<std::string, int> userIdToIdx;
  std::unordered_map<int, std::string> idxToUserId;

  int n = 0;
  for (const auto &[node, _] : *adj) {
    userIdToIdx[node->getUserId()] = n;
    idxToUserId[n] = node->getUserId();
    n++;
  }
```

Adjacency Matrix Implementation

```
// adj int will store the temporary (integer) adjacency matrix
std::vector<std::vector<unsigned long long>> tempIntAdj(
    n, std::vector<unsigned long long>(n, 0));

// populate temporary adjacency matrix w/ edges
// can handle both directed and undirected
for (const auto &[_, edges] : *adj) {
  for (const auto &[_, edge] : edges) {
    const auto &[u, v] = edge->getNodePair();
    const auto uIdx = userIdToIdx[u->getUserId()];
    const auto vIdx = userIdToIdx[v->getUserId()];

    // if undirected, add both sides
    if (!(edge->isDirected().has_value() && edge->isDirected().value()))
      tempIntAdj[vIdx][uIdx] = 1;
    tempIntAdj[uIdx][vIdx] = 1;
  }
}

// calculate the power matrix
const auto powerMatrix = exponentiation(tempIntAdj, k);
```

```
  // remap values
  std::unordered_map<std::pair<std::string, std::string>, unsigned long long,
                     CXXGraph::pair_hash>
      powAdj;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      powAdj[std::make_pair(idxToUserId[i], idxToUserId[j])] =
          powerMatrix[i][j];
    }
  }

  result.result = std::move(powAdj);
  result.success = true;

  return result;
}
```

Transition Matrix Implementation

```
template <typename T>
using TransitionMatrix =
    std::unordered_map<shared<const Node<T>>,
                       std::vector<std::pair<shared<const Node<T>>, double>>,
                       nodeHash<T>>;
```

```
struct PowTransResult_struct {
  bool success = false;
  std::string errorMessage = "";
  std::unordered_map<std::pair<std::string, std::string>, double, pair_hash>
      result = {};
};
typedef PowTransResult_struct PowTransResult;
```

```
/**
 * @brief This function raises a transition matrix to some k.
 * Best used for finding equilibrium.
 * @param k value by which to raise matrix
 * @return (success, errorMessage, matrix): where matrix is equivalent to S^k
 */
template <typename T>
const PowTransResult matrixPow(const shared<TransitionMatrix<T>> &trans,
                               unsigned int k) {
  PowTransResult result;
  result.success = false;
  result.errorMessage = "";

  // get a map between index in adj matrix
  // and userId
  std::unordered_map<std::string, int> userIdToIdx;
  std::unordered_map<int, std::string> idxToUserId;

  int n = 0;
  for (const auto &[node, _] : *trans) {
    userIdToIdx[node->getUserId()] = n;
    idxToUserId[n] = node->getUserId();
    n++;
  }
```

Transition Matrix Implementation

```
std::vector<std::vector<double>> stochasticMatrix(n,
                                                  std::vector<double>(n, 0.0));

// given transition matrix, convert it to
// stochastic matrix
for (const auto &[u, edges] : *trans) {
  const int uIdx = userIdToIdx[u->getUserId()];

  for (const auto &[v, weight] : edges) {
    const int vIdx = userIdToIdx[v->getUserId()];
    stochasticMatrix[uIdx][vIdx] = weight;
  }
}

// exponentiate stochastic matrix
auto powerTransMatrix = exponentiation(stochasticMatrix, k);
```

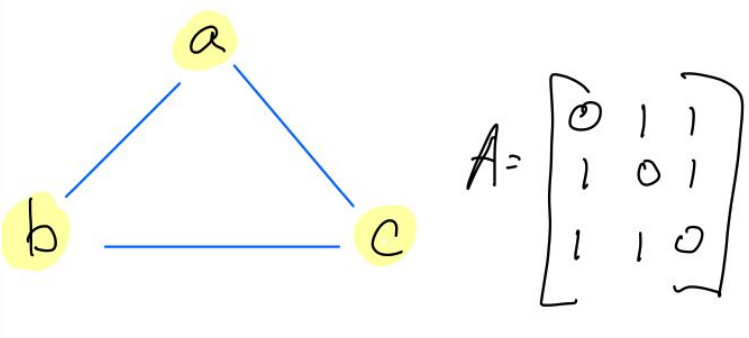
```
  // turn back into a map between nodes
  std::unordered_map<std::pair<std::string, std::string>, double,
                     CXXGraph::pair_hash>
      powTrans;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      powTrans[std::make_pair(idxToUserId[i], idxToUserId[j])] =
          powerTransMatrix[i][j];
    }
  }

  result.result = std::move(powTrans);
  result.success = true;

  return result;
}
```

Test Examples

```cpp
TEST(PowAdjTest, triangle) {
  CXXGraph::Node<int> a("a", 1);
  CXXGraph::Node<int> b("b", 1);
  CXXGraph::Node<int> c("c", 1);

  CXXGraph::UndirectedEdge<int> e1(0, a, b);
  CXXGraph::UndirectedEdge<int> e2(1, b, c);
  CXXGraph::UndirectedEdge<int> e3(2, c, a);

  CXXGraph::Graph<int> graph;
  graph.addEdge(&e1);
  graph.addEdge(&e2);
  graph.addEdge(&e3);

  CXXGraph::PowAdjResult res = matrixPow(graph.getAdjMatrix(), 5);

  ASSERT_TRUE(res.success);
  ASSERT_TRUE(res.result[std::make_pair("a", "a")] == 10);
  ASSERT_TRUE(res.result[std::make_pair("b", "b")] == 10);
  ASSERT_TRUE(res.result[std::make_pair("c", "c")] == 10);
  ASSERT_TRUE(res.result[std::make_pair("a", "b")] == 11);
  ASSERT_TRUE(res.result[std::make_pair("b", "a")] == 11);
  ASSERT_TRUE(res.result[std::make_pair("b", "c")] == 11);
  ASSERT_TRUE(res.result[std::make_pair("c", "b")] == 11);
  ASSERT_TRUE(res.result[std::make_pair("a", "c")] == 11);
  ASSERT_TRUE(res.result[std::make_pair("c", "a")] == 11);
}
```

```cpp
TEST(PowTransTest, transition_matrix) {
  CXXGraph::Node<int> n1("a", 1); // deg 2
  CXXGraph::Node<int> n2("b", 1); // get 3
  CXXGraph::Node<int> n3("c", 1); // deg 2
  CXXGraph::Node<int> n4("d", 1); // get 0

  CXXGraph::UndirectedEdge<int> e1(0, n1, n2);
  CXXGraph::UndirectedEdge<int> e2(1, n1, n3);
  CXXGraph::UndirectedEdge<int> e3(2, n2, n3);
  CXXGraph::DirectedEdge<int> e4(3, n2, n4);
  CXXGraph::DirectedEdge<int> e5(4, n4, n1);
  CXXGraph::DirectedEdge<int> e6(5, n4, n2);
  CXXGraph::DirectedEdge<int> e7(6, n4, n3);

  CXXGraph::Graph<int> graph;
  graph.addEdge(&e1);
  graph.addEdge(&e2);
  graph.addEdge(&e3);
  graph.addEdge(&e4);
  graph.addEdge(&e5);
  graph.addEdge(&e6);
  graph.addEdge(&e7);

  CXXGraph::PowTransResult res = matrixPow(graph.getTransitionMatrix(), 10);

  const double threshold = 1e-3;

  ASSERT_TRUE(res.success);
  ASSERT_NEAR(res.result[std::make_pair("a", "a")], 0.286, threshold);
  ASSERT_NEAR(res.result[std::make_pair("a", "b")], 0.321, threshold);
  ASSERT_NEAR(res.result[std::make_pair("c", "d")], 0.107, threshold);
  ASSERT_NEAR(res.result[std::make_pair("c", "c")], 0.286, threshold);
  ASSERT_NEAR(res.result[std::make_pair("d", "a")], 0.286, threshold);
  ASSERT_NEAR(res.result[std::make_pair("a", "d")], 0.107, threshold);
}
```

Example of Adjacency Matrix



Many ways to get a path length 5

$\overline{ab}$: abcacb, abcbab, ...

$$a^5 = \begin{bmatrix} 11 & 10 & 10 \\ 10 & 11 & 10 \\ 10 & 10 & 11 \end{bmatrix}$$

Sources

Directed Graphs - Transition Matrices, Cornell
Markov Chains, Pearson
Adjacency Matrix, Matrix Powers
Matrix Powers, *datafireball*
Powers of the Adjacency Matrix and the Walk Matrix, Andrew Duncan
Repeated Squaring, UNCG
Matrix Exponentiation, G4G