SOME GRAPH EXPLORATION

1. Search

I have at least two questions:

(1) How does the fact that nodes have 2, 3, or 4 neighbors manifest in the adjacency matrix?
(2) Can we exploit this structure to parallelize exploration?

But first, some prerequisites.

1.1. **Graphs.** For our purposes, graphs will have no self-loops and no parallel edges.

Definition 1. *An* **undirected graph** *is a pair $G = (V, E)$ where V is a finite set of* **nodes** *or* **vertices**, *and E is a subset of $V \times V$ of* **edges** *such that*

(1) If $(u, v) \in E$, then $(v, u) \in E$,
(2) For every $v \in V$, it is the case that $(v, v) \notin E$.

For $u, v \in V$, a **path of length** *k from u to v is a $(k+1)$-tuple $(w_0, w_1, \ldots, w_k)$ with $w_i \in V$, and $(w_i, w_{i+1}) \in E$ for $i = 0, \ldots, k$. We call G* **connected** *if for every $u, v \in V$, there is a path from u to v.*

In practice since V is finite, we usually denote its elements, without loss of generality, by $1, \ldots, n = |V|$. Dropping condition (1) gives a **directed** graph, and dropping condition (2) gives a graph with **self-loops**.

Many discrete structures with relationships between the constituent parts can be represented as graphs. For example:

- Pixels or collections of pixels in image processing [2]
- Objects in fusion categories related multiplicatively [1]

In practice, one is often not presented with a graph $G = (V, E)$ as such. One might, for example, be given V and a function

$$f(u, v) \coloneqq \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{if } (u, v) \notin E \end{cases}$$

along with an algorithm to compute values of f. Two of the most common problems involving graphs are

(1) Exploration: Given a collection $V = \{1, \ldots, n\}$ of vertices and an algorithm for computing f, find all pairs $(u, v) \in E$.
(2) Pathfinding: Given a graph $G = (V, E)$ and two nodes $u, v \in V$, find a path from u to v.
(3) Optimal pathfinding: Given a graph $G = (V, E)$ and two nodes $u, v \in V$, find a path of length k from u to v such that every path from u to v has length at least k.

There are many algorithms to solve these problems. One algorithm to solve the first is *breadth-first search* or BFS. An algorithm that solves the second is *heuristic search*. An algorithm to solve the third is *A**. We'll briefly touch on these three algorithms. Henceforth, fix a connected graph $G = (V, E)$, where $|V| = n$.

1.1.1. *BFS.* The idea of BFS is fairly straightforward. For now we will assume we have a modified version of the function f above. Suppose that we have an algorithm to compute the function N which is defined by

$$N(v) = \{u \in V \mid (u, v) \in E\}.$$

Suppose you begin at a vertex 1, knowing nothing about E; suppose you would like to visit every vertex in V.
.....

1.2. **Structure of the graph.** The first goal here is to write down the adjacency matrix for the 100 or so nodes surrounding the 3×3 solved state. This is to start addressing question (1) above. See `explore_puzzle_space.py` for progress.

We'll be considering, e.g., a 3×3 puzzle configuration as a permutation of the set

$$\{1, 2, 3, 4, 5, 6, 7, 8, 0\}.$$

There are some particularities about where 0 goes in the null/solved permutation, but that shouldn't be too important now. Permutations will be denoted by σ.

Let $\sigma_0, \ldots, \sigma_{99}$ be the configurations of the 100 nodes surrounding the solved state. Let $\lambda(\sigma)$ denote the Lehmer encoding of the permutation σ. So $\lambda(\sigma) \in \{0, \ldots, 99\}$. Let $i_0, \ldots, i_{99}$ be such that $\lambda(\sigma_{i_0}) < \cdots < \lambda(\sigma_{i_{99}})$, and define f by $f(\lambda(\sigma_{i_k})) \coloneqq k$.

Now, whenever a new node n is discovered from parent p, keep track of number $\lambda(n.\sigma)$ as well as the pair

$$(\lambda(n.\sigma), \lambda(p.\sigma)).$$

The size of the set $\{n.\sigma\}$ will tell us how big to make the adjacency matrix, and the pairs $(\lambda(n.\sigma), \lambda(p.\sigma))$ will tell us which entries are nonzero.

2. Graph Spectral Theory

Let $G = (V, E)$ be a graph with $V = \{1, \dots, n\}$. That is, $E \subseteq V \times V$ is such that if $(i, j) \in E$ then $(j, i) \in E$. We will assume G has no **self-loops**, i.e., edges of the form (i, i). We call j a **neighbor** of i if $(i, j) \in E$. The **adjacency matrix** of G is the matrix $A = A_G \in M_n(\mathbb{N})$ given by

$$A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{else} \end{cases}.$$

Proposition 1. *Define $\mathbb{1}$ to be the $n \times 1$ matrix (i.e., column vector) with 1 in every slot; define $D = D_G \coloneqq A\mathbb{1}$. Then $D_{i,i}$ is the number of neighbors of i.*

Proof. Exercise. □

Definition 2 (Laplacian)**.** *We define the* **graph Laplacian** *of G to be the matrix $L = L_G \coloneqq D - A$.*

The following proposition summarizes the entries of L.

Proposition 2. $L_{i,j} = \begin{cases} D_{i,i} & i = j \\ -1 & (i,j) \in E \\ 0 & \textit{else} \end{cases}$

Proof. Exercise. □

Since we defined the set V of nodes to be the set $\{1, \dots, n\}$, we may consider a real-valued vector $v \in \mathbb{R}^n$ to be an assignment of real numbers to the nodes of G; i.e., a vector v may be seen as a function on G.

This begs the question of the action of L on such a vector (i.e., function) $v \in \mathbb{R}^n$. The answer is given by the following lemma.

Lemma 1. *Let $v \in \mathbb{R}^n$, and define $w \coloneqq Lv$. Then*

$$w_i = \sum_{\substack{j: \\ (i,j) \in E}} (v_i - v_j).$$

Proof. Exercise. □

So for each i, w_i measures the net difference between v_i and its neighbors. We turn this local information into global measure of "how constant v is" by multiplying L by v on both sides.

Proposition 3. *For $v \in \mathbb{R}^n$, the quantity $v'Lv$ is the sum of squares of differences between neighboring nodes' v-values.*[1] *That is,*

$$v'Lv = \sum_{\substack{(i,j) \in E \\ i<j}} (v_i - v_j)^2.$$

Proof. Exercise.
Hint: to get at the $i < j$ condition, remember that you only want to count each edge **once**! □

3. Results

Not a true "results" section, but rather just explaining what we see. The file `spec_emb_2x2_-sqrt3-sqrt3.pdf` contains a spectral embedding of the state space of the 2×2 sliding puzzle along the two eigenvectors with eigenvalue $-\sqrt{3}$. Not very good. Also, because of the overcounting used, there's an extra point at the origin with no neighbors. That, and the crazy look of the graph are not good. This was done in Mathematica just to get something done; will convert to Python later.

[1] Here v' denotes the usual transpose of v.

Meeting notes

1/13/2026. We agreed the following tasks should be done: (not ranked)

- Change the `PuzzleState` class around, maybe splitting it into two different classes. E.g., `Puzzle` and `State`
- Rewrite `explore_puzzle_space.py` to match `w_a-star_variants.py` with use of classes, arg parsing, etc.
- Add some information about search algorithms to the writeup. For instance, why does A* need a `.d` attribute?

Classes. The following class structures might work

`Puzzle` Att/method	Functionality
`.current_state`	A `State` class: now
`.sol_state`	A `State` class: goal
`.is_solved()`	Checks `State` class
`.d`	Measures distance from parent

`State` Att/method	Functionality
`.config`	A 2-d array of integers
`.linear()`	Returns a copied 1-d array of integers

References

[1] Caleb Kennedy Hill. *Type G_2 Quantum Subgroups from Graph Planar Algebra Embeddings*. 2026. arXiv: 2601.05381 [math.QA]. URL: https://arxiv.org/abs/2601.05381.

[2] Jeremy Juybari et al. "Context-guided segmentation for histopathologic cancer segmentation". In: *Scientific Reports* 15 (Feb. 2025). DOI: 10.1038/s41598-025-86428-7.