Programming Assignment 5 Roadmap

CS260 Data Structures

Galen Long

Winter 2024-25

k is the number of rows and columns. A board with k rows/columns will have k^2 spaces, labeled with the numbers $0, 1, ..., k^2 - 1$. 0 represents the empty space.

Contents

General algorithm

See Recitation 9 solutions for the general algorithm.

Use a 1D array to represent boards, convert from 1D to 2D to get neighbors

We recommend using a 1D array to represent boards, as a 2D array would require k mallocs to create and 2 dereferences to access its elements, while a 1D array requires 1 malloc to create and 1 dereference to access its elements.

Since you need to find the tiles horizontally and vertically adjacent to the empty tile, this will require you to do some arithmetic to convert a 1D index into a 2D (row, column) pair, and vice versa. If i is the 1D index and (row, col) are the 2D row and column indices:

- To go from 1D to 2D: $row = \lfloor i/k \rfloor, col = i\%k$
  - Integer division is floored in C automatically
- To go from 2D to 1D: $i = row * k + col$

For example, if $k = 3$, $7 \rightarrow (\lfloor 7/3 \rfloor, 7\%3) = (2, 1)$, and $(2, 1) \rightarrow 2 * 3 + 1 = 7$:

0	1	2	3	4	5	6	7	8

	0	1	2
0	0	1	2
1	3	4	5
2	6	7	8

To get a (row, col) pair's neighbors:

- Up: $(row - 1, col)$
- Down: $(row + 1, col)$
- Left: $(row, col - 1)$
- Right: $(row, col + 1)$

You'll need to make sure each neighbor is within the bounds of the board. The easiest way to do it is to simply check if row and col are both ≥ 0 and $< k$. We recommend avoiding special case logic ("if it's the top left corner", "bottom right corner", "top row but not a corner", etc.), as this is more error prone to get right.

Tip: Make sure this code is correct before proceeding, since this is easy to mess up. Check all cases: corners, outer rows/columns, middle elements, etc.

How to check if a board is unsolvable in $\Theta(k^2)$ time

Not all sliding tile puzzle boards are solvable. Luckily, there's a $\Theta(k^2)$ time method to check if a board is solvable before you even run BFS.

Begin by counting the number of inverted pairs (excluding the empty space), making sure not to double count pairs. Say we had an array A and two indices, i and j. i and j are an inverted pair if $i < j$ but $A[i] > A[j]$. For example, the board [1, 2, 3, 0, 4, 6, 8, 5, 7] has 3 pairs (ignoring the empty space, 0) out of order: (6, 5), (8, 5), and (8, 7).

A board is unsolvable if either of the following are true:

- k is odd and number of inversions is odd
- k is even and (number of inversions + row index of empty space) is even

Examples:

- [1, 2, 3, 4, 5, 6, 8, 7, 0] is unsolvable
  - $k = 3$ is odd
  - Inverted pairs: (8, 7)
  - 1 inverted pair, which is odd
- [1, 2, 3, 4, 5, 6, 0, 8, 9, 10, 7, 11, 13, 14, 12, 15] is unsolvable
  - $k = 4$ is even
  - Inverted pairs: (8, 7), (9, 7), (10, 7), (13, 12), (14, 12)
  - 5 inverted pairs
  - Row of empty space: 1
  - $5 + 1 = 6$, which is even
- [0, 1, 3, 4, 2, 5, 7, 8, 6] is solvable
  - $k = 3$ is odd
  - Inverted pairs: (3, 2), (4, 2), (7, 6), (8, 6)
  - 4 inverted pairs, which is even

It's very important to use this formula to check if a board is unsolvable. If you try to figure it out by exploring the graph of boards, you'll end up exploring every possible board and get the worst possible performance.

Construct the graph as you go, not all at once in the beginning

There are $(k^2)!$ possible boards. If $k = 3$, there's 362,880 possible boards. If $k = 4$, there's 20,922,789,888,000 possible boards. If your code constructs the graph of all possible boards and adjacency relationships in the beginning before even trying to run BFS, it'll be hopelessly slow. Instead, you should construct the board nodes as you go:

- Start from your initial board.
- Find the empty tile in the current board, then generate the new boards by swapping each tile adjacent to the empty tile with the empty tile. If the new board hasn't been previously generated, add it to the queue.
- Stop when you've found the goal board, which will always be $[1, 2, ..., k^2 - 1, 0]$.

Use a hash table set to track visited boards

The BFS pseudocode we've seen in class assumes the graph has been previously created, so it stores whether a node has been visited or not inside the node itself. However, you're going to construct your graph as you explore each board, so you need a way to track if a new board you just created has been previously generated.

> It's not enough to simply check if the newly generated board is different than its predecessor. It's possible to have two different chains of swaps that end up at the same board. If you don't properly check if a board has already been visited, your code will end up in an infinite loop.

We have an ADT for this: a set, implemented with a hash table data structure. Each time you add a board to the queue, to mark it as visited, you'll add it to the set. When generating a board, you'll check if it's in the set before you add it.

> You should add a board to the set as soon as you discover it. If you only add it once it's been dequeued, you might end up re-adding the same board to the queue multiple times. In other words, when you'd normally mark the node as "gray", you should add it to the set.

You can use your hash table implementation from PA3, tweaking it to use boards as keys instead of strings.

Use an efficient hash function and large, hard-coded capacity for your hash table (don't resize)

A hash table requires a hash function. We can hash a board similar to how we'd hash a string, only our array elements will be integers, not characters.

It's very important to ensure your hash table is efficient:

- Choose a good hash function that uniformly distributes boards throughout the table. The Java stdlib hash function is a good choice.
- Choose a capacity that works well with your hash function. The Java stdlib works nicely with powers of 2 capacities.
- We recommend hardcoding different capacities for different values of k and never resizing.
  - For example, we chose the capacities $2^{12}, 2^{14}, 2^{16}$ for $k = 3, 4, 5$, respectively.
  - Resizing is slow, as it would require re-hashing every element in your table.
  - Don't worry about keeping your load factor below 1 – this would require a capacity larger than the total number of possible boards you might explore, which would take up too much memory. Just make

sure your capacity is reasonably large, and try making it bigger if you find your buckets are getting too large.

Feel free to experiment with different hash functions/capacities to see what runs faster in practice. To evaluate a hash/capacity strategy, try running your code on an example and then printing:

a. How many elements were inserted into the hash table (n)
b. What percentage of buckets were filled (have at least one element)
c. What the average filled bucket size is

Ideally, the percentage of buckets filled should be large (approaching 100%, at least around 75%) and the average filled bucket size should approach $\frac{n}{capacity}$. If you find that some buckets are huge while others are small, you may want to revisit your choice of hash function and capacity.

If you've picked a decent hash/capacity strategy and your runtime is still too slow, you probably have other issues with your code and should check for logic errors or other areas of slowness (like the ones described in the next sections).

BFS should stop as soon as the target board is discovered, before it's dequeued

When generating neighboring boards, you should check if they're visited. If they're not, you should then check if they're the goal board, and if they are, stop searching immediately. Don't wait until you dequeue it to check if it's the target board – check as soon as you discover/generate it.

> It's possible to wait until a board is dequeued to check if it's the goal board, but if you do this, your implementation will be much slower in certain circumstances. Remember that BFS explores layer by layer and won't begin exploring the next layer until the current layer is finished. If you only check when you dequeue, if a node early in the layer has the goal board as a neighbor, you'll only find out when you dequeue the goal board, which will only happen after every other node in the current layer has been dequeued and explored. And since we can have a quadrillion (!) possible boards for even $k = 4$, there may be tons of nodes sitting on the queue, waiting to be explored.

Use `memcmp` to check if two boards are equal

You can check if two boards are equal the same way you check if two strings are equal: by comparing them tile-by-tile. However, you can also use a C stdlib function called `memcmp` to do this, as `memcmp` is implemented very efficiently.

Use an efficient queue implementation

Since there are tons of possible boards, your queue sizes can get very large. Make sure you choose an efficient queue implementation so you have $\Theta(1)$ enqueue and dequeue. See the Week 2 slides.

Reconstructing the moves needed to reach the target

To output the moves, you'll need to save how each board was generated so you can traverse backwards from the target board to the initial board to reconstruct which tiles were shifted.

When discovering/generating a new board, you can store this information inside its node. You can either:

1. Store the tile (or the current index of the tile) that was moved to create the board, e.g. if the current board is [1, 2, 3, 0, 4, 5, 6, 7, 8] and we slide 4 into the empty space to create the new board [1, 2, 3, 4, 0, 5, 6, 7, 8], we'd store 4 (or its index in the new board, 3) in the new board's node.
    - To reconstruct the original board, you'd need to retrieve what tile was swapped, create a "new" board by swapping it back with the empty space, then look up the "new" board in your hash table to find what tile was swapped to make it, and so on.
2. Store a pointer to the predecessor board that it was generated from.
    - To reconstruct the original board, you'd need to compare the current board with its predecessor board to see which tile (excluding the empty space) is out of place, swap that tile with the empty space to generate a "new" board, look up that board in the hash table, then follow its predecessor pointer, and so on.