CS106B: Personal Project

Musical Modulations

Rishubh Thaper

December 3, 2020

§1 Problem Description

In music, a *key* is a collection of pitches, defined by an underlying *scale*, that serves as the harmonic pivot of a piece. It is denoted by a single pitch, representing the *tonality*, and a characterization of either *minor* or *major* that describes the *modality* of the composition. For example, Beethoven's famous Fifth Symphony is in the key of "C minor," meaning that is fundamentally built upon the C minor scale and thus organizes its motives around the center of C. Since there are 12 total pitches in Western music and 2 modalities, a total of 24 keys are possible.

In classical music, especially, a notion of *related keys* is used to delineate various common modulations. Every key has an associated set of six frequently-used related keys, defined by their *scale degree* (distance from the tonic, or starting key). They are displayed in the following table,

Distance From Tonic	Major Tonic	Minor Tonic
0	Parallel	Parallel
±1	Supertonic	Subtonic
±2	Mediant	Submediant
±3	Subdominant Above	Dominant Below
±4	Dominant Above	Subdominant Below
±5	Relative	Relative

The descriptions of each scale degree are as follows:

- **Parallel**: The key with same tonality but opposite modality,
- **Supertonic**: A whole-tone above the tonic and with opposite modality,
- **Subtonic**: A whole-tone below the tonic and with opposite modality,

- **Mediant**: A third above the tonic and with opposite modality,
- **Submediant**: A third below the tonic and with opposite modality,
- **Subdominant**: A fourth from the tonic and with same modality,
- **Dominant**: A fifth from the tonic and with same modality,
- **Relative**: The key with same signature but opposite modality

For instance, the key of C major has the related keys D minor, E minor, F major, G major, A minor, and C minor, and the key of A minor has the related keys G major, F major, E minor, D minor, C major, and A major. Note that the interval distance of each related key is positive for a major tonic and negative for a minor tonic. Therefore, when finding the related keys of a major tonic, one moves upwards by successive scale degrees, and when finding the related keys of a minor tonic, one moves downward.

We also present the idea of *enharmonic equivalents*, which are the same pitches with different spellings. For example, C-sharp and D-flat are enharmonic equivalents.

With these preliminaries out of the way, we turn to the problem you must solve. Suppose you are a prolific composer and are writing a piece. Beginning from the home key of your piece, you want to traverse through a series of successive modulations to reach a target key. However, you are quite finicky (all artists are!), and you have chosen a specific subset of related keys that you can travel to at each modulation. Thus, the problem statement is as follows:

Given a starting key, an ending key, and a set of allowed related keys, determine the shortest path your composition can take from the starting key to the ending key by moving only to a related key in the allowed set at each step. As an example, suppose you start in C major and you want to end up in A-flat major, but you can only traverse the related keys that are 0, 3, and 5 scale steps from the current tonic. Then a shortest path you can take is

{C major, F major, F minor, Ab major}.

Your task is to write a function

```
Stack<string> modulate(string startKey, string endKey, Set<int> allowed)
```

that takes in two `strings` containing starting and ending keys and a `Set<int>` containing the allowed distances (in scale degrees) from the tonic, and returns a `Stack<string>` containing the shortest path through related keys from the start key to the end key.

When writing your function, take into consideration the following:

1. If the starting and ending keys are the same, your function should return a path of size 1 consisting only of the starting key. If there is no path from the starting key to the ending key, your function should return an empty `Stack<string>`.

2. You can be assured that the input arguments will be in the form "X[♯/♭] [major/minor]," where X is a letter from the set $\{A, B, C, D, E, F, G\}$, and the regular expression [♯/♭] represents the set of accidentals. A flat will be designated by a lowercase 'b' in the string.

3. Although some foreign modulations like $1 \rightarrow 7$ (from the tonic to the seventh scale degree) can occur in music, for this problem, you can expect that the set of allowed transitions will be a subset of the related keys in the table above.

4. You must output only one path; that is, if there are multiple paths of the same minimal length, your function need only return one of them.

5. Both starting and ending keys in your returned `Stack<string>` should be the same as the inputs; i.e., if your function explores them as enharmonics, you must transform them in your final result.

6. Avoid outputting keys in non-standard notation. For example, A# major is more commonly written as Bb minor. This will require a significant amount of string processing and data structure exploitation.

7. To follow up the above point, you can expect the inputs to be in standard form as per musical tradition. Thus, an input like C# major is impossible.

8. Because the input sizes and search space are relatively small, your function does not need to have any particular time complexity, but it should run in about a second.

```
Stack<string> modulate(string startKey, string endKey) {
  /* TODO: Fill me in! */
  return {};
}
```

§2 Solutions and Test Cases

§2.1 *Solution I: Breadth-First Search*

We first define three global constant data structures to aid in generating related keys and switching between enharmonic equivalents.

```
const Vector<string> keys = {"A", "A#", "B", "C", "C#", "D", "D#",
"E", "F", "F#", "G", "G#"};

const Map<string, string> enharmonics = {{"Bb","A#"}, {"Db","C#"},
{"Eb","D#"}, {"Gb","F#"}, {"Ab","G#"}};

const Map<string, string> standard = {{"A# major", "Bb major"},
{"A# minor", "Bb minor"}, {"D# major", "Eb major"},
{"G# major", "Ab major"}, {"C# major", "Db major"}};
```

From here, we implement three helper functions to convert a sharp key to its flat standard, generate the set of related keys, and check if a key has already been explored in a path.

The longest function to write is `relatedKeys`, which employs a significant amount of string processing to extract the tonality and modality of a key and then jumps modular intervals on the `Vector<string>` of keys to find all related keys.

```
string standardize(string key) {
  if (standard.containsKey(key)) {
    return standard.get(key);
  }
  return key;
}

Set<string> relatedKeys(string key, Set<int> allowed) {
  Set<string> related;
  string tonality = key.substr(0, 2);
  string modality;
  if (!isspace(tonality[1])) {
    if (enharmonics.containsKey(tonality)) {
      tonality = enharmonics.get(tonality);
    }
    modality = key.substr(3);
  }
  else {
    tonality = tonality.substr(0, 1);
    modality = key.substr(2);
  }
  int index = keys.indexOf(tonality);
  if (modality == "major") {
    if (allowed.contains(0)) {
```

```
        related.add(standardize(tonality + " minor"));
      }
      if (allowed.contains(1)) {
        related.add(standardize(keys[(index + 2) % keys.size()]
        + " minor"));
      }
      if (allowed.contains(2)) {
        related.add(standardize(keys[(index + 4) % keys.size()]
        + " minor"));
      }
      if (allowed.contains(3)) {
        related.add(standardize(keys[(index + 5) % keys.size()]
        + " major"));
      }
      if (allowed.contains(4)) {
        related.add(standardize(keys[(index + 7) % keys.size()]
        + " major"));
      }
      if (allowed.contains(5)) {
        related.add(standardize(keys[(index + 9) % keys.size()]
        + " minor"));
      }
    }
    else {
      if (allowed.contains(0)) {
        related.add(standardize(tonality + " major"));
      }
      if (allowed.contains(1)) {
        related.add(standardize(keys[(index + keys.size() - 2)
        % keys.size()] + " major"));
      }
      if (allowed.contains(2)) {
        related.add(standardize(keys[(index + keys.size() - 4)
        % keys.size()] + " major"));
      }
      if (allowed.contains(3)) {
        related.add(standardize(keys[(index + keys.size() - 5)
        % keys.size()] +  " minor"));
      }
      if (allowed.contains(4)) {
        related.add(standardize(keys[(index + keys.size() - 7)
```

```
        % keys.size()] + " minor"));
      }
      if (allowed.contains(5)) {
        related.add(standardize(keys[(index + keys.size() - 9)
        % keys.size()] + " major"));
      }
    }
    return related;
}

bool containsMusicalKey(Stack<string> path, string key) {
    while (!path.isEmpty()) {
      string check = path.pop();
      if (check == key) {
        return true;
      }
    }
    return false;
}
```

Now, we implement breadth-first search to search for paths extending radially outward from the starting key until the ending key is reached, ensuring that the shortest path is found.

```
Stack<string> modulate(string startKey, string endKey,
                       Set<int> allowed) {
    Stack<string> path = {startKey};
    Queue<Stack<string>> paths;
    paths.enqueue(path);

    while (!paths.isEmpty()) {
      path = paths.dequeue();
      string enharmonicEndKey = enharmonics.get(endKey.substr(0, 2))
       + endKey.substr(2);
      if (path.peek() == endKey) {
        return path;
      }
      else if (path.peek() == enharmonicEndKey) {
        path.pop();
        path.push(endKey);
        return path;
      }
      else {
```

```
        Set<string> related = relatedKeys(path.peek(), allowed);
        Set<string> valid;
        for (string key : related) {
          if (!containsMusicalKey(path, key)) {
            valid.add(key);
          }
        }
        for (string key : valid) {
          Stack<string> explore = path;
          explore.push(key);
          paths.enqueue(explore);
        }
      }
    }
    return path;
  }
```

Let k be the length of the shortest path from the starting key to the ending key. Since there are 6 related keys to explore at each step in the search, the Big-O of this function is $O(6^k)$. The space complexity is also $O(6^k)$.

§2.2 *Solution II: Iterative Deepening*

Breadth-first search, in general, consumes a large amount of space. Furthermore, since the input and path sizes are fairly small, we can trade space for time by implementing a version of depth-first search using recursive backtracking. This algorithm explores as far as possible down a single branch of the decision tree of related key choices before considering other paths, updating the current path at each step rather than storing all possible paths in a data structure. However, the decision tree has theoretically infinitely many levels; nevertheless, we know that any path from one key to another cannot have length more than 24, since that is the total number of keys available. Thus, we can prune the algorithm using *iterative deepening*, whereby we explore up to a maximum depth of $i = 0, 1, 2, \ldots, 24$ until a path is found. This is similar to breadth-first search but saves much more space.

```
int modulate(string startKey, string endKey, Set<int> allowed,
Stack<string>& bestPath,
Stack<string>& currentPath, int maxDepth) {
  if (maxDepth < 0) {
    return false;
  }
  string enharmonicEndKey;
```

```
  for (string enharmonic : enharmonics.keys()) {
    if (enharmonics.get(enharmonic) == endKey.substr(0, 2)) {
      enharmonicEndKey = enharmonic + endKey.substr(2);
    }
  }
  if (currentPath.peek() == endKey ||
   currentPath.peek() == enharmonicEndKey) {
    bestPath = currentPath;
    bestPath.pop();
    bestPath.push(endKey);
    return true;
  }
  Set<string> related = relatedKeys(currentPath.peek(), allowed);
  Set<string> valid;
  for (string key : related) {
    if (!containsMusicalKey(currentPath, key)) {
      valid.add(key);
    }
  }
  for (string key : valid) {
    currentPath.push(key);
    if (modulate(startKey, endKey, bestPath, currentPath,
    maxDepth - 1)) {
      return true;
    }
    currentPath.pop();
  }
  return false;
}

Stack<string> modulate(string startKey, string endKey,
Set<int> allowed) {
  Stack<string> currPath;
  Stack<string> best;
  currPath.push(startKey);
  for (int i = 0; i <= 24; i++) {
    if (modulate(startKey, endKey, allowed, best, currPath, i)) {
      return best;
    }
  }
  return best;
```

```
}
```

The iterative deepening depth-first search uses a helper function that keeps track of the current path, a "best" path that is updated to equal to the current path when the target key is reached, and a maximum depth parameter that limits the search to a certain level.

The time complexity of this function is, like BFS, $O(6^k)$, where k is the length of the found path, but the space complexity is much smaller at $O(k)$ since that the is the maximum possible length of the path being explored.

§2.3 *Test Cases*

I tested both functions on the following test cases (after thoroughly checking the helper function `relatedKeys` first, of course).

```
STUDENT_TEST("Test modulate") {
  /* Same start and end key */
  EXPECT_EQUAL(modulate("C major", "C major", {0,1,2,3,4,5}).size(), 1);

  /* Enharmonic manipulation required, minor end key */
  Stack<string> result = modulate("C major", "Bb minor", {0,1,2,3,4,5});
  EXPECT_EQUAL(result.size(), 4);
  EXPECT_EQUAL(result.pop(), "Bb minor");

  /* Restricted set of allowed related keys */
  EXPECT_EQUAL(modulate("Db major", "A minor",{3,5}).size(), 7);

  /* No path from start to end */
  EXPECT_EQUAL(modulateBFS("G minor", "Ab major",{1}).size(), 0);
}
```

Since there can be multiple shortest paths to the end key, I decided to check the size of the path returned by the functions rather than its explicit construction, since any path that I find manually may not necessarily be the one found by the function. However, for an input in which the ending key was a flat key, I made sure that the final string in the path was also the same key since my helper functions do convert flat keys to sharp keys for ease of access in the global constant `Vector<string>` of keys.

§3 Problem Motivation

§3.1 Concept Coverage

This problem hones strategies for using efficient data structures (in this case, Vector, Stack, Queue, Set, and Map) and implementing algorithms that operate on these containers. In particular, this question requires a sophisticated level of comprehension in the following areas:

- Instantiating abstract data structures and accessing, modifying, and iterating over their elements.
- Manipulating nested data structures (like a Queue containing Stacks of strings).
- Designing and writing efficient algorithms (such as BFS and DFS) that take advantage of the facilities provided by abstract containers.
- Passing data structures by reference and modifying them inside a function.
- Decomposing a large task into reasonable subtasks using suitable helper functions.
- Processing strings efficiently.

§3.2 Personal Significance

My favorite part of CS106B was its emphasis on algorithmic development, and data structures are an integral part of this skill. Specifically, Assignments 2 and 3 applied abstract data structures and recursion to real-world problems like solving mazes and finding words in Boggle. Inspired by these examples, I decided to use my newly-acquired knowledge of data structures and algorithms on a non-trivial problem incorporating my love of classical music and harmony. In fact, I have mulled over the idea of finding a shortest path of modulation when composing my own pieces, and now, equipped with the arsenal of CS106B, I can efficiently code a solution to this problem.

It was also enlightening to consider an assortment of search techniques. Breadth-first and depth-first search are the most common ways of traversing a decision tree, but in this particular problem, there was no limit to the levels of the tree, so I had to apply some ingenuity to make DFS workable.

I know that I can definitely use this function when writing my next music piece!

§4 Common Misconceptions

Misconceptions or bugs could come up in the following areas of this assignment:

- Generating the set of related keys
- Preventing an infinite loop or recursion in the BFS and DFS algorithms
- Extracting information from an input string and converting to enharmonic equivalents

A student may mistakenly try to use a for loop instead of a for-each loop over the set of allowed keys when constructing the neighbor set. It is important to keep in mind that a set is built upon a balance binary search tree, so a for loop is not defined. Also, the best way to find each related key is to store all possible keys in a Vector and apply simple arithmetic over the indices to jump the corresponding intervals. This exploits the predictable formulaic derivation of related keys. However, a student may forget that while the musical chromatic scale is cyclic and repeats in groups of 12, a Vector does not, so adding to or subtracting from an index may result in an out-of-bounds exception. To prevent this, the student needs to apply modular arithmetic.

Also, unlike the algorithmic problems we encountered in class (like perfect mazes and Boggle), this question does not have any bound to the levels of its decision tree. It is possible to get caught up in an infinite loop starting and ending at the same key. Students need to write a helper function to check if a key has already been explored in a path to ensure that no key is visited more than once. And for the depth-first search algorithm, iterative deepening is a requirement because the recursion will never terminate if there is no limit to the depth of levels searched.

Moreover, an important part of this assignment is string processing, and the `substr()` method is used extensively throughout the implementation to extract tonalities and modalities as well as to convert back and forth between enharmonic equivalents. There are multiple ways to do this, and I considered creating a map that stored all sharp keys and all flat keys to maintain traditional spellings, but I had to write too much text. Students can handle this task any way they want to, but they need to be careful about how they manipulate strings and Maps to cover all possible cases.

Finally, common errors that can show up in any problem of this nature are “attempting to dequeue an empty queue” exceptions, “index-out-of-bounds” errors on strings, Sets, and Vectors, and arms-length or incomplete recursion (forgetting to “unchoose” after exploring a particular path).