OPTIMAL INSTRUMENT FINGERING

Andrew Chellis & Nik Vanderhoof

Is there an efficient way to minimize finger movement when playing a song?

Wind Instrument:

- A number of valves or keys
- A number of possible notes
- A number of key combinations for each note

Brass Instrument:

- 3 valves
- 8 valve configurations
- 34 possible notes

Input:

- A sequence of notes
- A table of possible fingerings for each note

Output:

- An optimal sequence of fingerings
- The total number of finger movements

Note Fingerings on Baritone

000	010	100	110	001	011	101	111
46	45	44	43	43	42	41	40
53	52	51	50	50	49	48	47
58	57	56	55	55	54	53	52
62	61	60	59	59	58	57	56
65	64	63	62	62	61	60	59
68	67	66	65	65	64	63	62
70	69	68	67	67	66	65	64
72	71	70	69	69	68	67	66
74	73	72	71	71	70	69	68

- Three Algorithms
  - Brute Force Unpruned Tree
  - Pruned Tree
  - Bottom-up dynamic programming

Class Node:

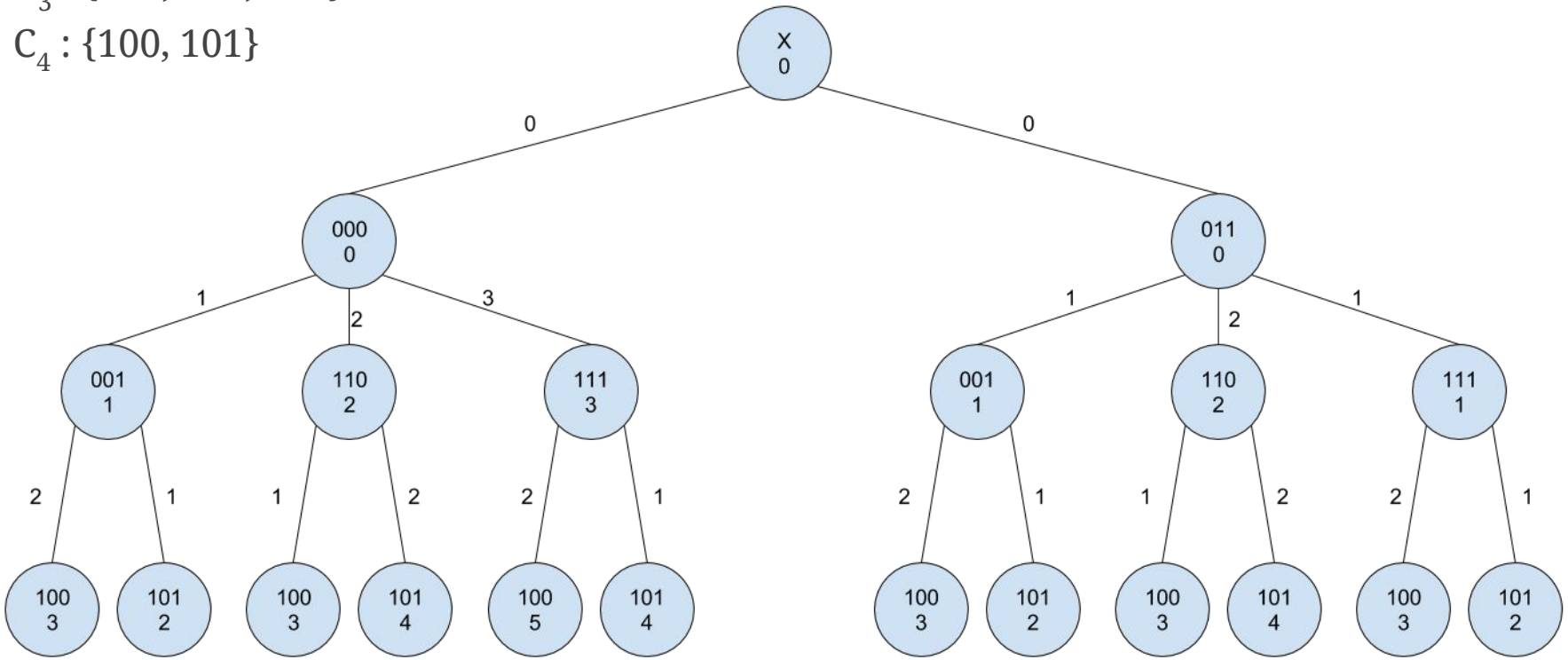
```
1.  Node(comb, parent=Nil):
2.      this.comb = comb
3.      this.parent = parent
4.      if parent is Nil:
5.          this.index = 0
6.          this.cost = 0
7.      else:
8.          this.index = 1 + parent.index
9.          this.cost = parent.cost + Dist(comb, parent.comb)
10.     this.choices = N2F[notes[this.index]
```

UNPRUNED TREE

Bb_3 : {000, 011}

B_3 : {110, 001, 111}

C_4 : {100, 101}



Brute-Force(*n*, *notes*, *N2F*):

```
1.  items = Queue()
2.  for comb in N2F[notes[0]]                    // up to h times, negligible
3.      Enqueue(Node(comb), items)
4.  while items                                  // run for every node
5.      cur =  Dequeue(items)
6.      if cur.index == n:                       // at last level
7.          Enqueue(cur, items)
8.          break
9.      for fingering in cur.choices:            // expand each level * h nodes
10.         ins = Node(fingering, parent=cur)
11.         Enqueue(ins, items)
12. return Min(items)                            // O(h^n)
```

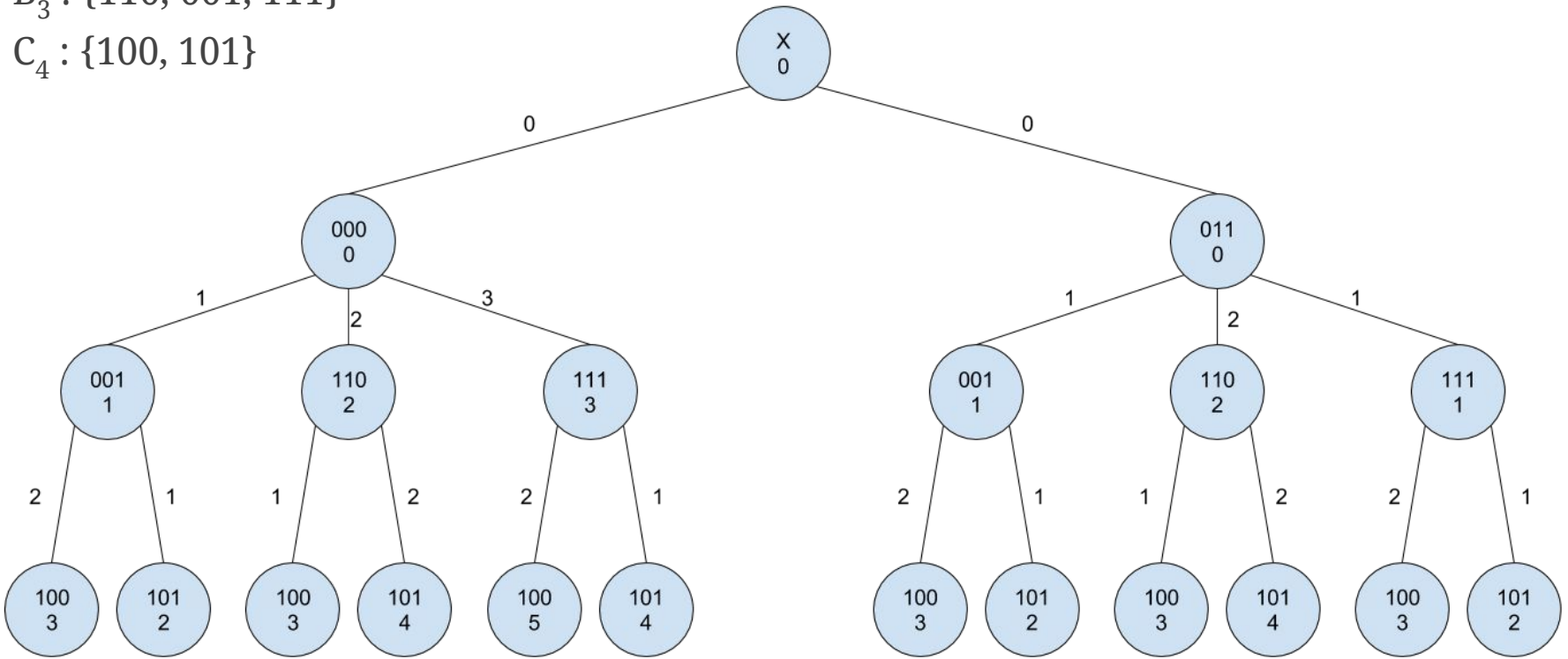
UNPRUNED COMPLEXITY

- Expands all possible solutions
- Chooses solution with lowest weight at the end
- Worst case: $O(h^n)$
  - h is max number of fingerings per note
  - n is number of notes
  - On average, expansions are lower than h

Bb_3 : {000, 011}

B_3 : {110, 001, 111}

C_4 : {100, 101}



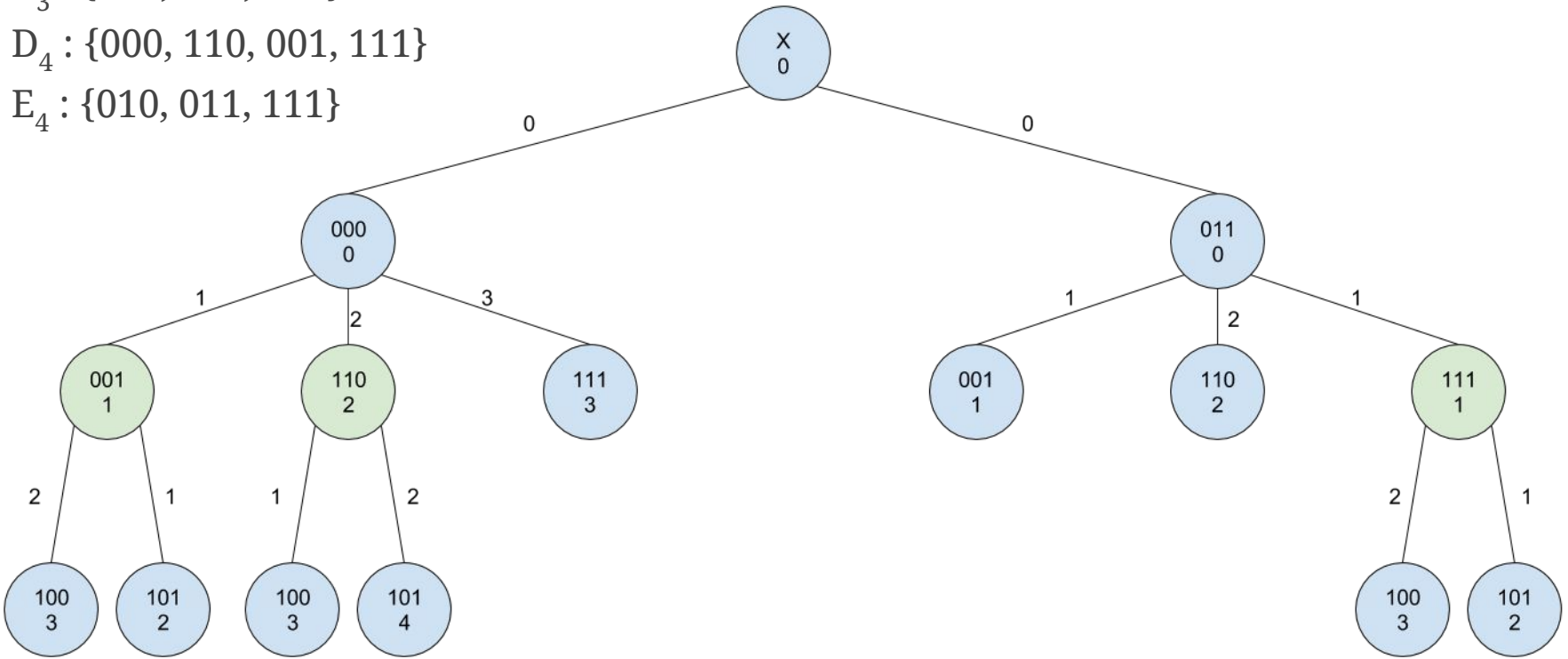
PRUNED TREE

Bb_3 : {000, 011}

B_3 : {110, 001, 111}

D_4 : {000, 110, 001, 111}

E_4 : {010, 011, 111}



```
Tree-Pruning(n, notes, N2F):
1.      items = Queue()
2.      for comb in N2F[notes[0]]:                  // up to h times
3.          Enqueue(Node(comb), items)
4.      index = 0
5.      while index < n:                            // n times
6.          choices = Map(list)
7.          while items:                            // up to h times
8.              cur = Dequeue(items)
9.              for comb in cur.choices:            // up to h times
10.                 ins = Node(comb, parent=cur)
11.                 choices[comb].append(ins)
12.         for key in choices:                     // up to h times
13.             Enqueue(Min(choices[key], items)    // up to h times
14.         index += 1
15.     return Min(items)                           // O(nh^2))
```

- Expand every possibility for a given level
- Choose the lowest cost way to reach each resultant fingering and add back to processing
- Repeat until at lowest level, minimum of that level is result
- Time is $2(h^2)n$ for $O(h^2n)$
  - On average, expansions are lower than h
  - For us, n growth is more important

Idea:

Bottom-up with a table for each sub-problem

$$T_n[f_{prev}][f_{next}] = \begin{cases} \text{dist}(f_{prev}, f_{next}), & n = 0 \\ \text{dist}(f_{prev}, f_{next}) + \min(T_{n-1}[f_{prev}][\cdots]), & n > 0 \end{cases}$$

Bottom-Up(*n*, *notes*, *N2F*)

```
1.  tables[0…n-1][][]                          // n h*h tables
2.  tables[0][…][…] = 0                        // fill in first table
3.  prev_choices = N2F[notes[0]]
4.  for i=1 to n-1:                            // n-1 times
5.      cur_choices = N2F[notes[i]]
6.      for prev in prev_choices:              // up to h times
7.          for cur in cur_choices:            // up to h times
8.              tables[i][prev][cur] = Dist(prev, cur) +
    Min(tables[i-1][prev][…])                  // h items to check
9.  return Min(tables[n-1])                    O(nh^3)
```

Bb_3 : {000, 011}

B_3 : {110, 001, 111}

C_4 : {100, 101}

T[0]	Nil
000	**0**
011	**0**

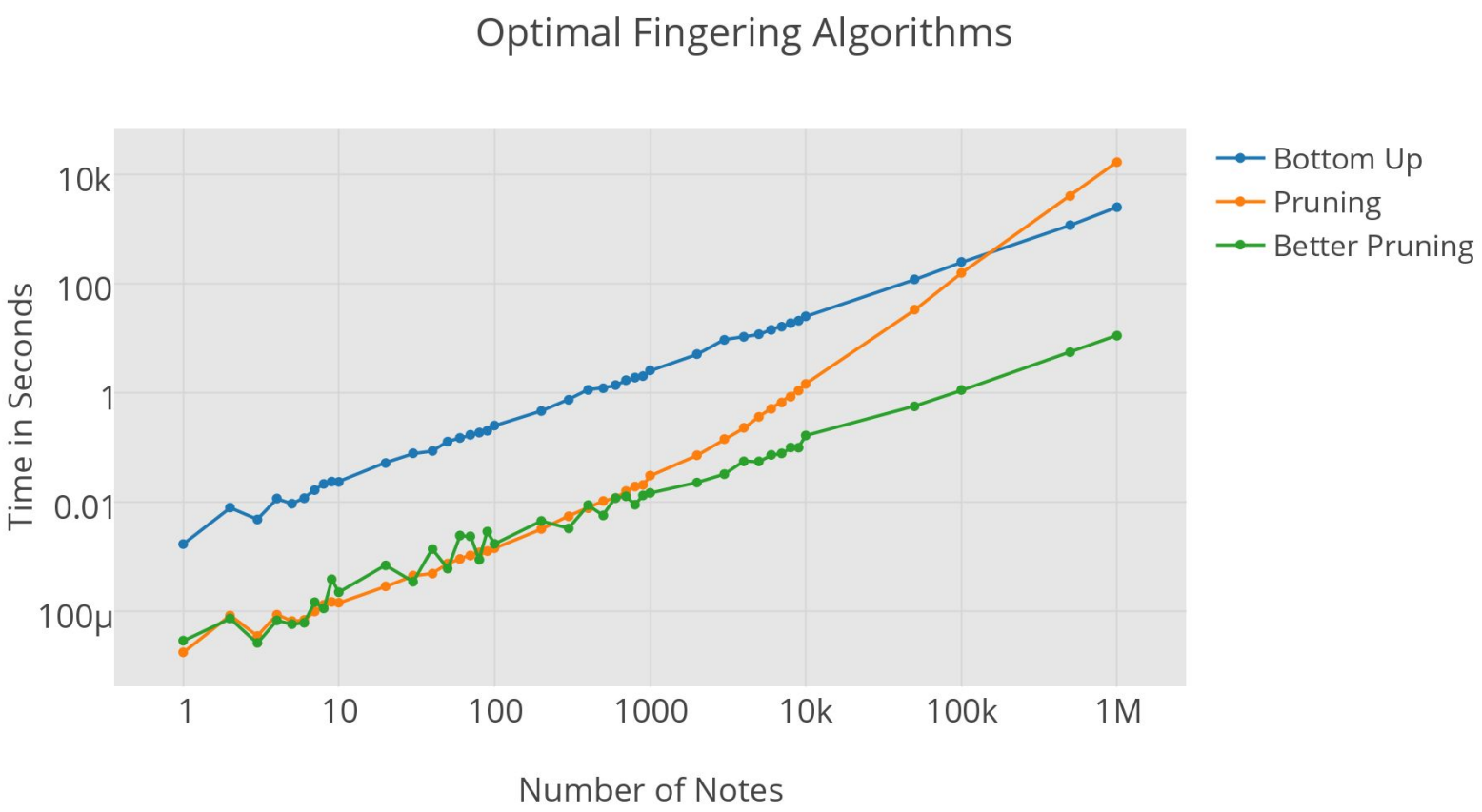
T[1]	000	011
110	**2**	**2**
001	**1**	**1**
111	**3**	**1**

T[2]	110	001	111
100	**3**	**3**	**3**
101	**4**	**2**	**2**

- Python 3
- Pruning and Brute-Force
  - Node class described earlier
  - collections.deque for Queue
  - collections.defaultdict for Map
- Bottom-Up
  - Tables are Pandas DataFrames

EXPERIMENTAL PLAN

- Generate tests with random notes within the instrument’s range
- Input sizes < 10000, average time for 10 trials
- Larger input sizes, run once


Optimal Fingering Algorithms
Time in Seconds
10k
100
1
0.01
100µ
1
10
100
1000
10k
100k
1M
Number of Notes
Bottom Up
Pruning
Better Pruning

RESULTS

Size	Bottom Up (s)	Pruning (s)	Better Pruning (s)
1	1.72E-03	1.79E-05	2.93E-05
2	7.99E-03	8.45E-05	7.46E-05
3	4.85E-03	3.57E-05	2.67E-05
4	1.17E-02	8.73E-05	6.88E-05
5	9.42E-03	6.69E-05	5.88E-05
6	1.19E-02	7.03E-05	6.22E-05
7	1.67E-02	1.00E-04	1.48E-04
8	2.17E-02	1.33E-04	1.14E-04
9	2.39E-02	1.50E-04	3.90E-04
10	2.36E-02	1.45E-04	2.26E-04
20	5.23E-02	2.89E-04	7.00E-04
30	7.90E-02	4.54E-04	3.51E-04
40	8.72E-02	4.95E-04	1.39E-03
50	1.29E-01	7.55E-04	6.12E-04
60	1.51E-01	9.16E-04	2.46E-03
70	1.72E-01	1.06E-03	2.38E-03
80	1.89E-01	1.21E-03	8.91E-04
90	2.04E-01	1.28E-03	2.90E-03
100	2.54E-01	1.44E-03	1.74E-03
200	4.69E-01	3.25E-03	4.55E-03
300	7.53E-01	5.54E-03	3.33E-03
400	1.14E+00	7.79E-03	8.89E-03
500	1.22E+00	1.06E-02	5.76E-03
600	1.40E+00	1.20E-02	1.19E-02
700	1.72E+00	1.60E-02	1.29E-02
800	1.92E+00	1.94E-02	9.04E-03
900	2.04E+00	2.08E-02	1.34E-02
1000	2.60E+00	3.09E-02	1.48E-02
2000	5.16E+00	7.28E-02	2.30E-02
3000	9.40E+00	1.43E-01	3.28E-02
4000	1.07E+01	2.31E-01	5.61E-02
5000	1.19E+01	3.67E-01	5.60E-02
6000	1.43E+01	5.14E-01	7.37E-02
7000	1.64E+01	6.70E-01	7.86E-02
8000	1.90E+01	8.58E-01	1.01E-01
9000	2.11E+01	1.11E+00	1.00E-01
10000	2.54E+01	1.47E+00	1.67E-01
50000	1.20E+02	3.36E+01	5.70E-01
100000	2.51E+02	1.58E+02	1.14E+00
500000	1.18E+03	4.04E+03	5.60E+00
1000000	2.52E+03	1.68E+04	1.12E+01

LIMITATIONS

- Currently, we can only process baritone due to hard-coded finger to note mappings
- Our method of distance calculation may not work on string instruments

- Represent problem as a DAG
  - Compare shortest-path algorithms with ours
- Compute less entries in Bottom-Up tables
- A generalized version of the problem
  - h states
  - A sequence $a_1 \ldots a_n$ of actions to perform
  - Actions can be in multiple states
  - State transitions have different costs

CONCLUSION

- We developed multiple algorithms with varying time complexities to effectively solve our problem of effectively minimizing finger movement
- Roles:
  - Andrew Chellis: Problem analysis, test gen
  - Nik Vanderhoof: Testing, musical data, code quality
  - Both: Algorithm Design and Implementation

THANKS!

Any questions?

You can find our code at

http://github.com/nvander1/optimal-fingering