Predicting BFS Runtime

CMPUT 355: Games, Puzzles, and Algorithms

Lecture Outline

1. Inversions and solvability
2. Estimates of runtime in sliding tile problem

Recap: Inversions

Definition: A sliding tile puzzle has m **inversions** if there are m distinct unordered pairs of numbers $\{x, y\}$ such that $x < y$ but x appears later than y when the numbers of the puzzle are written row-by-row.

2	3
1	

2, 3, 1

{1,2}: 2 before 1 ❌
{1,3}: 3 before 1 ❌
{2,3}: 2 before 3 ✅
2 inversions

1	3
2	

1, 3, 2

{1,2}: 1 before 2 ✅
{1,3}: 1 before 3 ✅
{2,3}: 3 before 2 ❌
1 inversion

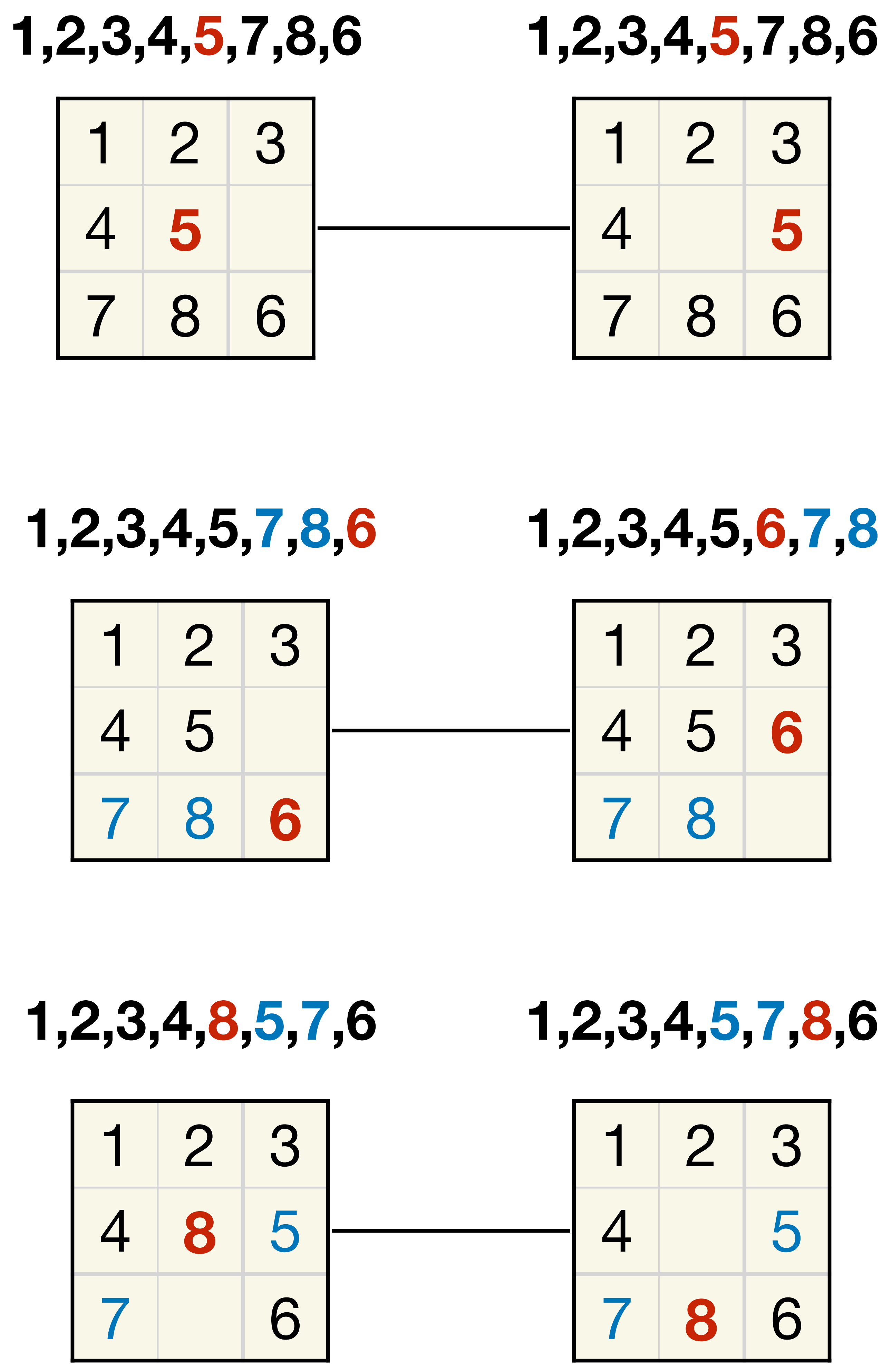
3	
2	1

3, 2, 1

{1,2}: 2 before 1 ❌
{1,3}: 3 before 1 ❌
{2,3}: 3 before 2 ❌
3 inversions

Inversions and Solvability: Odd k

- Horizontal slides don't change the number of inversions at all (**why?**)
- Vertical slides "jump" a number n over $k-1$ **skipped** numbers
  - All pairs that do not contain n have same inversion value (inverted or not) after slide
  - All pairs that include n and a **skipped** number have their inversion value flipped
- **If k is odd:**
  - Solved position has 0 inversions (even)
  - Starting from solved position, every move **flips** an even number of inversions
  - Flipping even number of inversions means number is still even
  - **Every solvable position must have an even number of inversions when k is odd**



Inversions and Solvability: Even k

If k is even:

- Vertical slides "jump" a number n over $k - 1$ **skipped** numbers
  - So each vertical slide flips an **odd** number of inversions
- **Solved** position has an even number of inversions
- After **odd** number of vertical slides, position has **opposite** inversion parity (**why?**)
- After **even** number of vertical slides, position has same inversion parity (**why?**)
- If the blank is an **odd** number of rows away from bottom (call that "**blank height**") in a solvable position, inversion parity must be **odd** (**why?**)
  - Similarly, **even** number of rows away means **even** inversion parity
- **In every solvable position, inversion parity must equal parity of blank height**
  - Equivalently: parity of **(# inversions) + (blank height)** must be **even**

2	3
1	

2, 3, 1

{1,2}: 2 before 1 👎
{1,3}: 3 before 1 👎
{2,3}: 2 before 3 ✅
2 inversions
blank height = 0

3	
2	1

3, 2, 1

{1,2}: 2 before 1 👎
{1,3}: 3 before 1 👎
{2,3}: 3 before 2 👎
3 inversions
blank height = 1

Necessary vs. Sufficient

Proposition: In any $k \times k$ sliding tile puzzle:

1. If k is odd, then in any solvable position the **number of inversions** must be even
2. If k is even, then in any solvable position the **number of inversions plus the blank height** must be even.

- We have proven that **if** the position is **solvable**,
  then parity (of either inversions or inversions plus blank height) must be **even**
- We have therefore proven that
  if the parity (of either inversions or inversion plus blank height) is **odd**,
  then the position is **not solvable** (**why?**)
- We have **not** proven that **if** the parity is **even**, **then** the position is **solvable**
  - Equivalently: have not proven that even-parity positions are a **connected component** of the search space
  - This is **actually true**, but the proof is a bit involved

Estimating Runtime

- Last time we used number of positions to guess about feasibility:
  - 3×3 puzzle has $9! = 362{,}880$ unique positions
    - seems OK
  - 4×4 puzzle has $16! \approx 2.092 \times 10^{13} \approx 20$ **trillion** unique positions
    - seems bad?
- **Demo:** time stile/stp_search2.py on 3×3 input and 4×4
- **Question:** Can we estimate how much runtime 4×4 would require?
  1. Estimate ratio of runtimes for unknown runtime and known runtime (e.g., 4×4 input vs. 3×3 input)
  2. Multiply ratio by known runtime

Runtime of BFS

- BFS will have to explore every position in the worst case scenario
- So ratio of **number of positions** seems like a good start

$$\frac{16!}{9!} = 16 \times 15 \times 14 \times 13 \times 12 \times 11 \times 10 = 57{,}657{,}600$$

  - (57 million seconds is about **1 year and ten months**)
- Worst-case runtime of breadth-first search is roughly proportional to number of **edges** in the search graph
  - So ratio of **number of edges** is actually a **better estimate**
- **Question:** Is this approach likely to give us a usable estimate?

Validating the Ratio of Edges Approach

Plan:

1. Compute **number of edges** for 3×3 and 2×5 inputs
2. Multiply **ratio** $\frac{\text{\# edges in } 2 \times 5}{\text{\# edges in } 3 \times 3}$ by the **runtime** for 3×3 input
3. Compare the prediction to the **actual** runtime for 2×5 input
4. If good approximation:
    1. compute number of edges for 4×4 input
    2. Multiply 2×5 runtime by ratio of edges $\frac{\text{\# edges in } 4 \times 4}{\text{\# edges in } 2 \times 5}$

Question: Why use 2×5 instead of 4×4?

Number of Neighbours for 3×3

1	2	3
4	5	6
7	8	

Position A

1	2	3
4	5	6
7		8

Position B

1	2	3
4		6
7	5	8

Position C

- How many **neighbours** does Position A have?
  - How many positions with the **same number ordering** have **2 neighbours**?
- How many neighbours does Position B have?
  - How many positions with the same number ordering have **3 neighbours**?
- How many neighbours does Position C have?
  - How many positions with the same number ordering have **4 neighbours**?

Number of Edges for 3×3

1	2	3
4	5	6
7	8	

1	2	3
4	5	6
7		8

1	2	3
4		6
7	5	8

- Each number orderings corresponds to 9 different positions
  - 1/9 positions have **4 neighbours**
  - 4/9 positions have **3 neighbours**
  - 4/9 positions have **2 neighbours**
- Total number of **neighbours**:

$$(9!\text{ positions}) \times \frac{1}{9}(4\text{ neighbours}) + (9!\text{ positions}) \times \frac{4}{9}(3\text{ neighbours}) + (9!\text{ positions}) \times \frac{4}{9}(2\text{ neighbours})$$

$$= (9!\text{ positions}) \times \left(\frac{1}{9}(4\text{ neighbours}) + \frac{4}{9}(3\text{ neighbours}) + \frac{4}{9}(4\text{ neighbours}) \right)$$

$= 967{,}680$ **neighbours**

$967{,}680/2 =$ **$483,840$ edges**

Number of Neighbours for 2×5

1	2	3	4	5
6	7	8	9	

Position A

1	2	3	4	5
6	7	8		9

Position B

- How many **neighbours** does Position A have?
    - How many positions with the **same number ordering** have **2 neighbours**?
- How many neighbours does Position B have?
    - How many positions with the same number ordering have **3 neighbours**?

Number of Edges for 2×5

1	2	3	4	5
6	7	8	9	

1	2	3	4	5
6	7	8		9

- Each number orderings corresponds to 10 different positions
    - 4/10 positions have **2 neighbours**
    - 6/10 positions have **3 neighbours**
- Total number of **neighbours**:

$$(10!\ \text{positions}) \times \frac{4}{10}(2\ \text{neighbours}) + (10!\ \text{positions}) \times \frac{6}{10}(3\ \text{neighbours})$$

$$= (10!\ \text{positions}) \times \underbrace{\left(\frac{4}{10}(2\ \text{neighbours}) + \frac{6}{10}(3\ \text{neighbours})\right)}_{=2.6}$$

$$= 9{,}434{,}880\ \textbf{neighbours}$$

$$9{,}434{,}880/2 = \mathbf{4,717,440}\ \textbf{edges}$$

Validate Prediction

- Ratio: $\dfrac{4{,}717{,}440 \text{ edges in } 2 \times 5}{483{,}840 \text{ edges in } 3 \times 3} = 9.75$
- Actual 3×3 runtime: $0.468s$
- Estimated 2×5 runtime: $9.75 \times 0.468s = 4.563s$
- Actual 2×5 runtime: $4.327s$

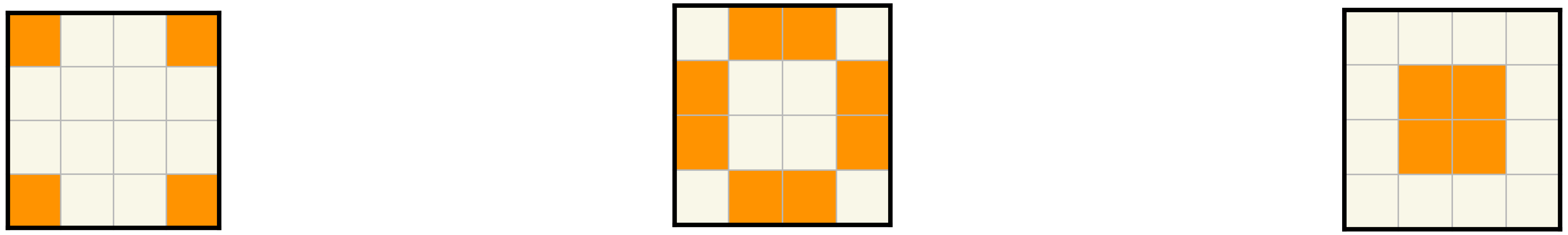
```
% time python3 stp_search2.py < in/33no > /dev/null

real    0m0.468s
user    0m0.385s
sys     0m0.016s
```

```
% time python3 stp_search2.py < in/25.0 > /dev/null

real    0m4.327s
user    0m4.151s
sys     0m0.075s
```

Number of Edges in 4×4



2 neighbours 3 neighbours 4 neighbours

$$(16!\text{ positions}) \times \underbrace{\left(\frac{4}{16}(2\text{ neighbours}) + \frac{8}{16}(3\text{ neighbours}) + \frac{4}{16}(4\text{ neighbours})\right)}_{=3}$$

$$= \frac{16! \times 3}{2}\text{ edges}$$

Ratio:

$$\frac{16! \times 3\text{ neighbours for } 4 \times 4}{10! \times 2.6\text{ neighbours for } 2 \times 5} = 16 \times 15 \times 14 \times 13 \times 12 \times 11 \times \frac{3}{2.6} = 652{,}800$$

Prediction: $652{,}800 \times 4.327s = 28{,}786{,}665s \approx$ (**11 months**)

Summary

- **Inversions:**
  - Number of pairs of **numbers** that are **out of order** (ignoring blank)
  - For **odd** k, a $k \times k$ position is **solvable** only if it has an **even number** of inversions
  - For **even** k: (steps from bottom row + inversions) must be even
- **Runtime:**
  - BFS runtime is roughly proportional to **number of edges** in search graph
  - Can estimate runtime for large instances as follows:
    1. Compute **ratio** $R = \dfrac{\text{\# edges in large instance}}{\text{\# edges in smaller instance}}$
    2. Run smaller instance to get **runtime** T
    3. Estimate that larger instance will take RT
       (i.e., R times longer than smaller instance)