Sliding Tile Puzzle

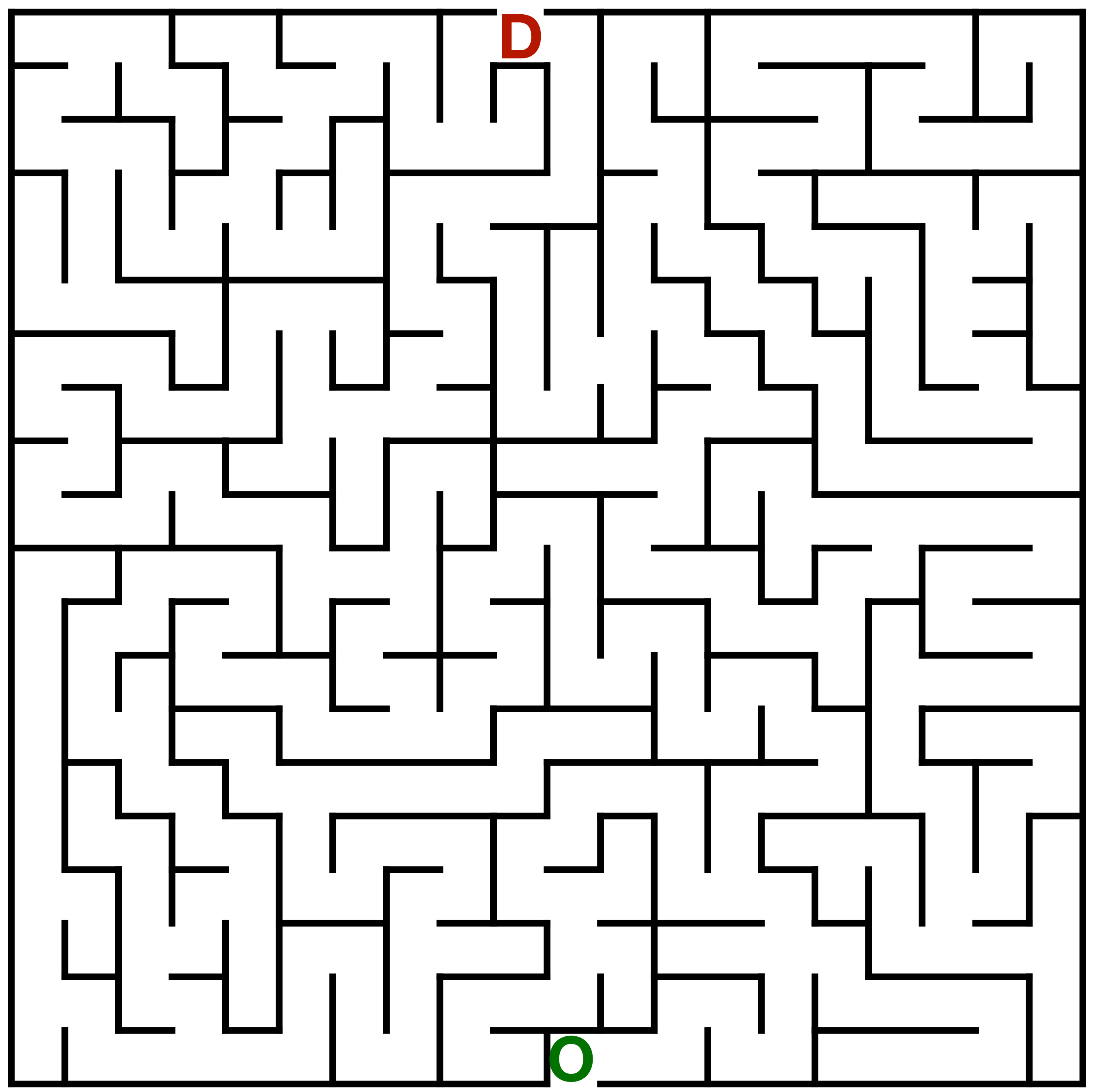
CMPUT 355: Games, Puzzles, and Algorithms

Lecture Outline

1. Logistics & Recap
2. Sliding Tiles Puzzle
3. Exhaustive Search
4. Solvability & Inversions

Recap: Maze Puzzles & Exhaustive Search

- **Maze puzzles:** Find a path from origin cell to a destination cell
- Completely random exploration is guaranteed to find it eventually
  - ...but can be arbitrarily slow
- Can straightforwardly represent as a graph
- **Depth-first search** and **breadth-first search** systematically search the graph
  - guaranteed to find the destination
  - might have to search **entire graph**
  - will never have to search **more** than the entire graph



Logistics

- **Practice quiz questions:** Posted last Friday
    - Answers released yesterday
- **Help with class material:**
    - TA office hours tomorrow
    - Canvas discussion forum
- **Quiz 1:** This Friday, **Jan 23**
    - In-class, full 50 minutes
    - No need to email if you have to miss it; up to 3 replaced by final exam automatically
    - Questions will be very similar to practice questions

Sliding Tile Puzzle

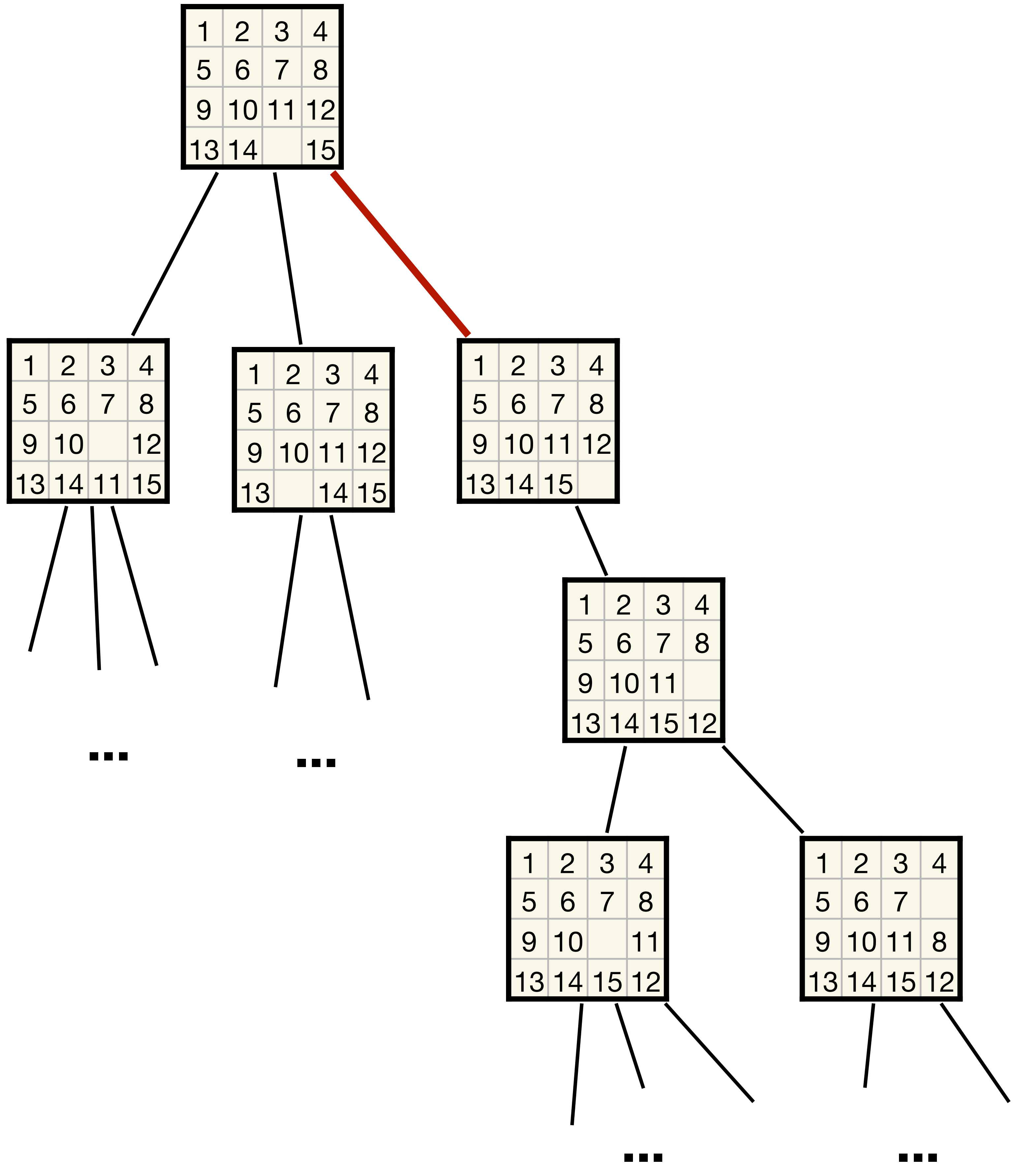
- A **sliding tile puzzle** is a $k \times k$ grid
  - One grid cell is "blank"
  - Every other cell contains a unique number from 1 to $k^2 - 1$ inclusive
- A puzzle is **solved** if the numbers are in order, with the blank in the last cell
- A puzzle is **solvable** if it can be transformed to solved by a series of blank moves
- A **blank move** exchanges the blank cell with the cell immediately above, below, left, or right of it

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Sliding Tile as Graph Search

Representing a sliding tiles puzzle as a graph search is easy:

- Each **position** is a **node**
- Two positions are **neighbours** if one can be transformed into the other with a single **blank move**
  - Draw an **edge** between each pair of **neighbours**
- A **solution** is a **path** from the starting position to the solved position



Implementation: **stile/stp_search2.py**

Breadth-first search

```
# use a parent dictionary to
#   - track seen states (all are in dictionary)
#   - record parents, to recover solution transition sequence
Parent = { start : start}
Fringe = deque() # the sliding tile states (strings) we encounter
Fringe.append(start)
print('  0 iterations, level 0 has 1 node')
while len(Fringe) > 0:
  stst = Fringe.popleft() # popleft() and append() give FIFO
  if stst == target:
    print('found target')
    while True:
      print(pretty(stst, self.cols, True))
      p = Parent[stst]
      if p == stst:
        return
      stst = p
  ndx0 = stst.index('0')
  for shift in self.legal_shifts(ndx0):
    nbr = str_swap(stst,ndx0,shift)
    if nbr not in Parent:
      Parent[nbr] = stst
      Fringe.append(nbr)
print('\nno solution found')
print('here is the last position encountered:')
print(pretty(stst, self.cols, True))
```

Compute shifts for each iteration

```
def legal_shifts(self,psn): # list of legal shifts
  S = []
  c,r = psn % self.cols, psn // self.cols # column number, row number
  if c > 0:           S.append(self.LF)
  if c < self.cols-1: S.append(self.RT)
  if r > 0:           S.append(self.UP)
  if r < self.rows-1: S.append(self.DN)
  return S
```

Manage the representation

```
def str_swap(s,lcn,shift): # swap chars at s[lcn], s[lcn+shift]
  a , b = min(lcn,lcn+shift), max(lcn,lcn+shift)
  return s[:a] + s[b] + s[a+1:b] + s[a] + s[b+1:]
```

Efficiency of Exhaustive Search

Questions:

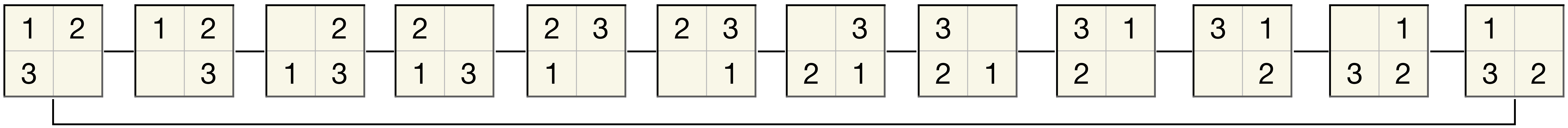
1. How many **possible positions** for a $k \times k$ puzzle?
2. How many positions need to be explored in the **worst case**?
3. Is breadth-first search **guaranteed** to find a solution if it exists? (**why?**)
4. Is unmodified breadth-first search practical for the standard 4×4 puzzle?

Solvability

Question: Is every possible starting position solvable?

- Any position that you can create from the solved position is solvable (**why?**)
- So an unsolvable position must not be reachable from the solved position

- Consider a 2×2 puzzle
- From every position, only one horizontal slide and one vertical slide available
  - Two horizontal slides in a row "cancel"
  - So the only way to get beyond neighbours is to alternate vertical and horizontal slides
- Reachable positions from a given position are a **cycle** (**why?**)
- All **solvable** positions are part of the **same** cycle (**why?**)

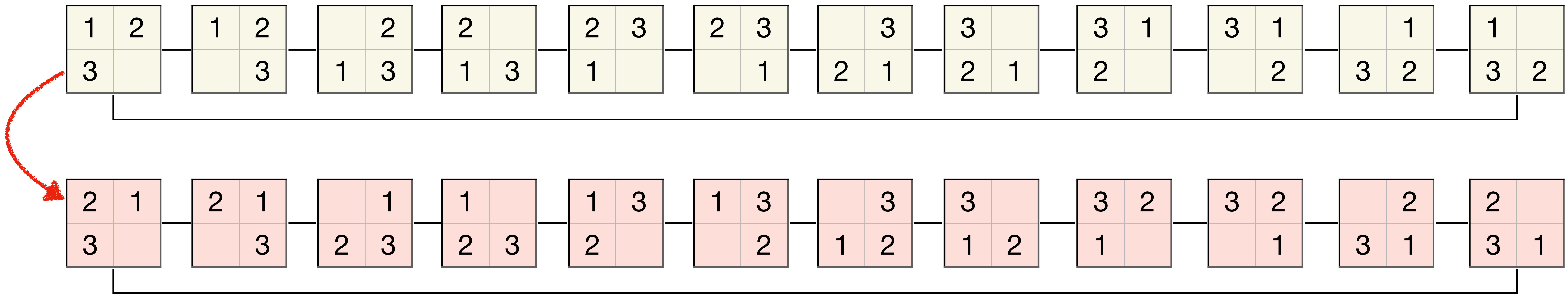


Questions:

1. How many **possible** positions in a 2×2 puzzle?
2. How many **solvable** positions in a 2×2 puzzle?

Unsolvable Positions

- **Question:** How can we create an unsolvable position?
  - Perform a transformation that cannot be implemented by a blank move
- Any position that can be reached by a blank move from an unsolvable position is also unsolvable (**why?**)
- We'll see in a moment that the search graph has exactly **two connected components**: one for the **solvable** positions, and one for the **unsolvable** positions



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

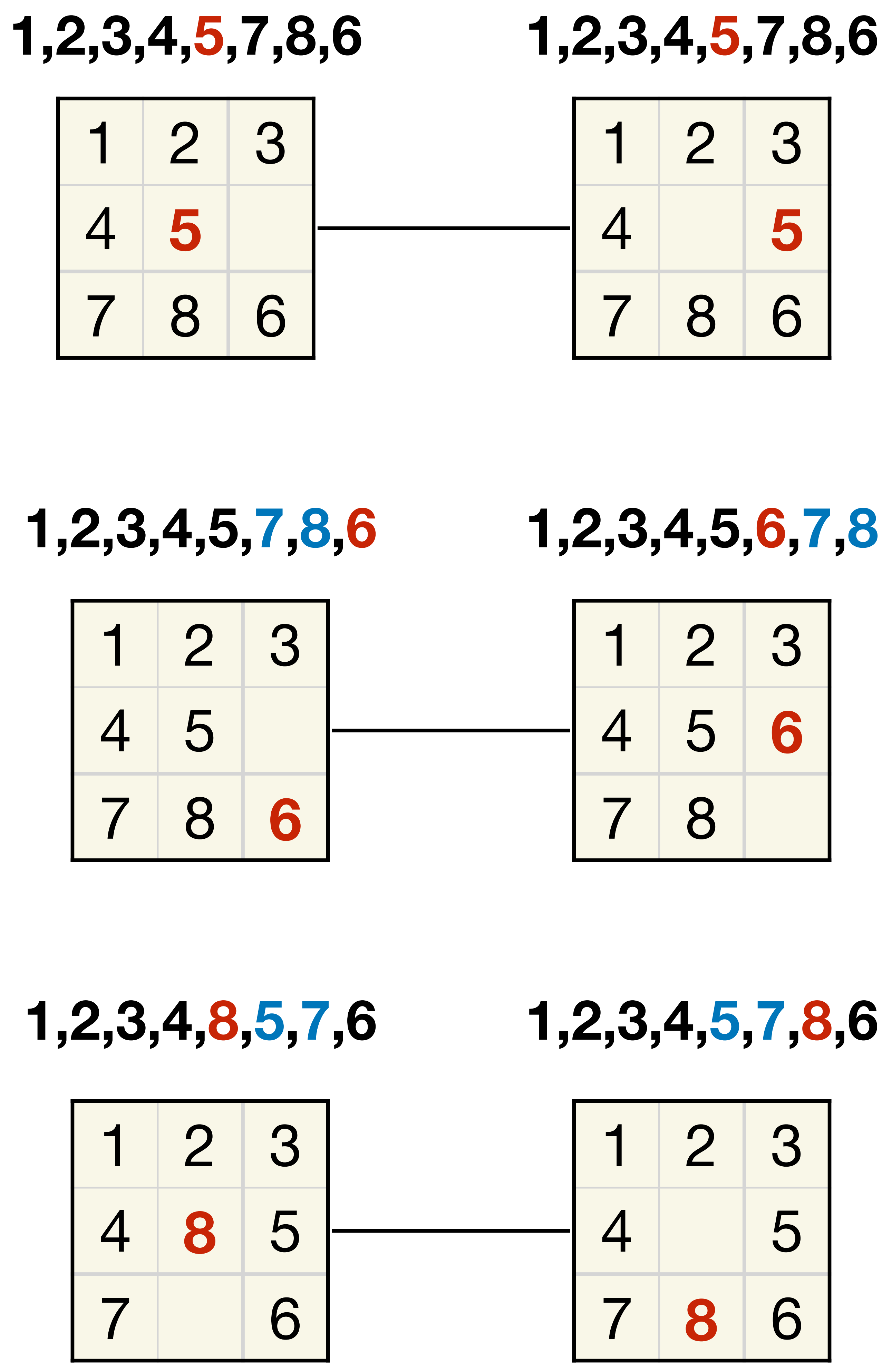
3, 2, 1

{1,2}: 2 before 1 👎
{1,3}: 3 before 1 👎
{2,3}: 3 before 2 👎

3 inversions

Inversions and Solvability

- 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
- For odd k: (rule for even k is slightly more complicated)
  - Solved position has 0 inversions (even)
  - Flipping even number of inversions means number is still even
  - **Every solvable position must have an even number of inversions**
  - This explains why all unsolvable positions are reachable from each other (**why?**)



Inversions as a Solvability Bound

- Suppose a **solvable** $k \times k$ sliding tile position has m **inversions**
- **Question:** what is the **minimum** number of moves required to solve it?
  - Need to get to 0 inversions
  - Each move reduces inversions by at most $k - 1$
  - So **no fewer** than $\left\lceil \frac{m}{k-1} \right\rceil$ moves
- Number of inversions gives a **lower bound** on how bad your position is
  - Even though it doesn't tell you **exactly** how bad it is
- We'll see in the next lecture that this is a very useful measurement to have

Summary

- **Sliding tile puzzle:**
  - Find a sequence of **blank moves** to transform a position into the solved state
  - **Solved state**: All numbers in order, blank at bottom right
  - All **solvable** positions are **reachable** from each other
    - (All non-solvable positions are also reachable from each other)
- **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
  - For all k, number of inversions induces a **lower bound** on the number of moves needed for a solution