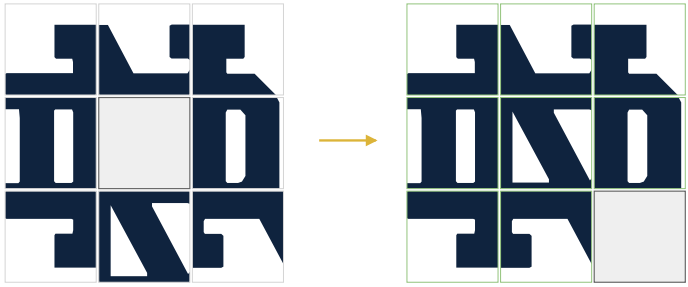
Unit 01 · Chapter 02

Uninformed Search

Before asking how to solve a puzzle, ask how we can systematically explore its possibilities.

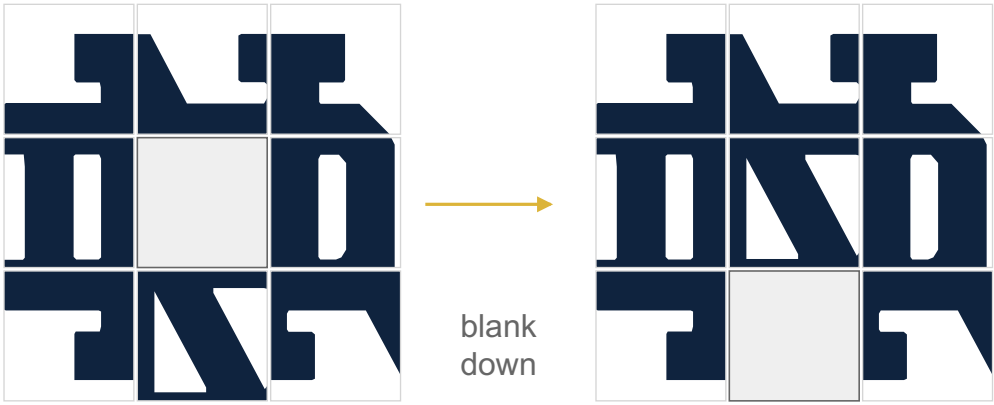


START TARGET

We will begin with **traversal**: organizing an exploration. Then we will add a goal, recover a solution, and see why search is a basic tool of artificial intelligence.

A Puzzle Made of States

Eight picture tiles and one empty square form a 3 × 3 board. A legal move slides a neighboring tile into the blank.



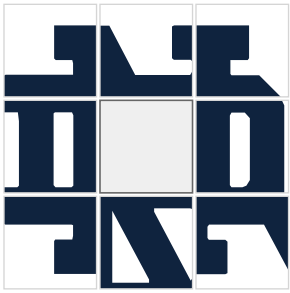
A: BEFORE **D: AFTER**

Only horizontal and vertical neighbors can move. Tiles do not rotate, jump over one another, or move diagonally.

A **state** is the arrangement of the entire board. One move produces a **successor state**. Here, the blank moves down while the picture tile below it moves up.

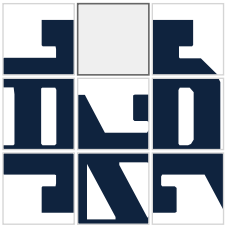
What Can Happen Next?

First, set aside the task of solving the picture. We want to explore the configurations reachable from A. Which states can one move produce?



A

UP



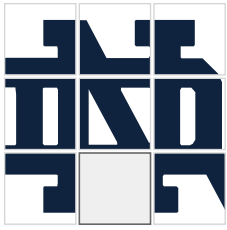
B

LEFT



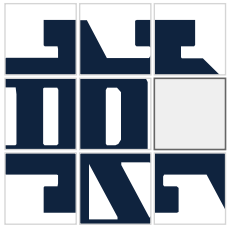
C

DOWN



D

RIGHT



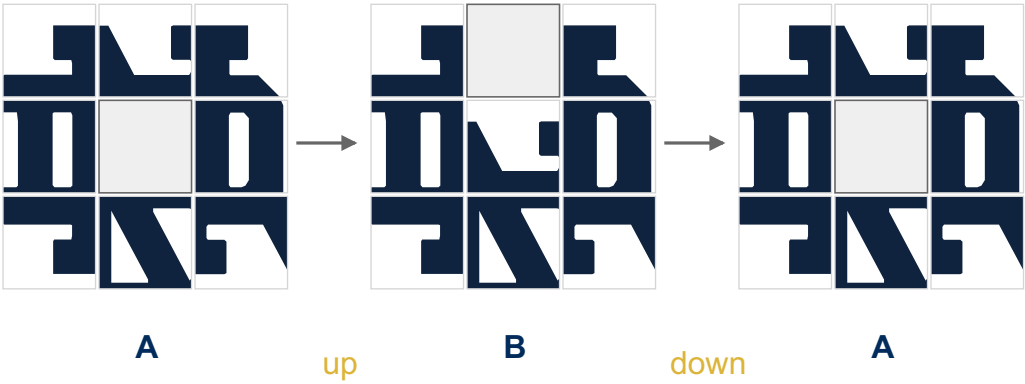
E

We will always try blank moves in the order **up, left, down, right**, skipping illegal moves. From A, all four moves are legal.

Each letter names a whole board configuration, not an individual picture tile.

A Move Can Undo a Move

From B, moving the blank down returns to A. We have followed two edges but discovered no new configuration.



The puzzle is a **graph**, not a tree: configurations can be revisited. Without a rule for repeated states, exploration can keep undoing and redoing the same move.

For traversal, record each state when it is first discovered. A later encounter does not add it to the frontier again. This lets us explore each reachable state once.

Our First Thirteen States

This visual key names A and the new states reached within two moves. Keep it beside the traversal traces. The gray square is the blank.



A



B



C



D



E



F



G



H



I



J



K



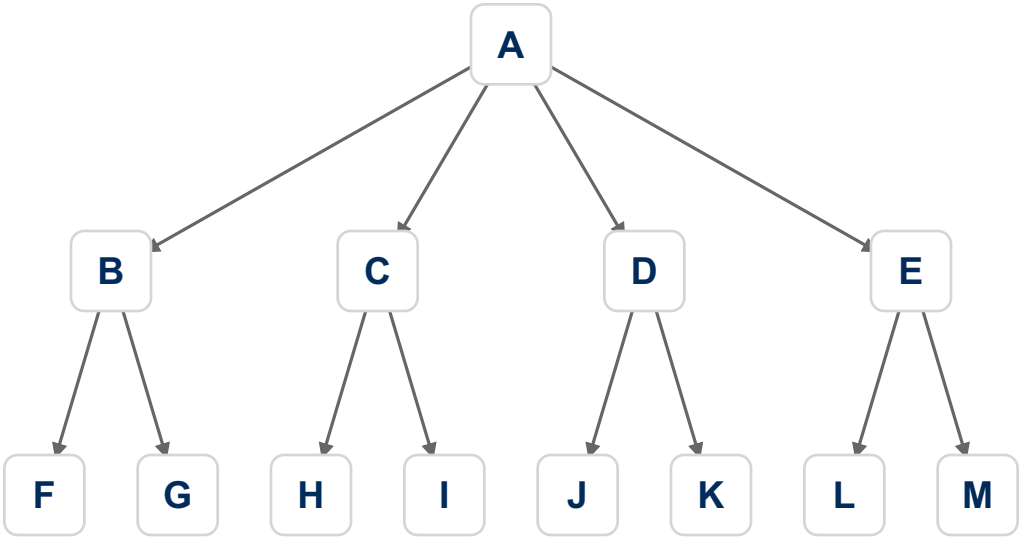
L



M

Draw the Exploration

We can unfold the graph into a tree of paths from A. For now, draw through depth 2 and omit moves back to an already discovered state.



Depth counts moves from A. B–E are at depth 1; F–M are at depth 2. The next traces traverse this finite drawing so we can see each step.

The drawing ends; the puzzle does not. F–M still have legal moves outside this excerpt. Later, we will search the actual puzzle without treating this page boundary as a dead end.

Traversal Needs a Frontier

Traversal systematically visits every reachable node in the structure being explored. To do that, it must remember discovered nodes that still need attention.

After visiting A



A has revealed four alternatives. This waiting collection is the **frontier**. Visiting B must not make us forget C, D, and E.

B reveals F and G



For breadth-first traversal, F and G join the back. C has waited longer, so it is next. A **queue** gives this first-in, first-out behavior.

Front is on the left. “Visit” means select for processing.

Breadth-First Traversal

A queue visits every node at one depth before moving to the next. On our finite excerpt, watch the frontier grow and then drain.

VISIT	FRONTIER AFTER PROCESSING
start	A
A	B C D E
B	C D E F G
C	D E F G H I
D	E F G H I J K
E	F G H I J K L M

The remaining nodes have no children in this drawing. Remove F, then G, …, then M; nothing new is added.

A B C D E F G H I J K L M

Depth-First Traversal

Depth-first traversal follows one branch before returning to an unfinished alternative. On the same excerpt, B's branch goes before C.

Visit A; choose B next



B is next; C, D, and E remain waiting.

Visit B; choose F next



F and G go ahead of the older alternatives.

Finish F; resume with G



F has no children in this finite drawing.

A B F G C H I D J K E L M

The stack's next item is shown at the left.

The Same Loop, Two Orders

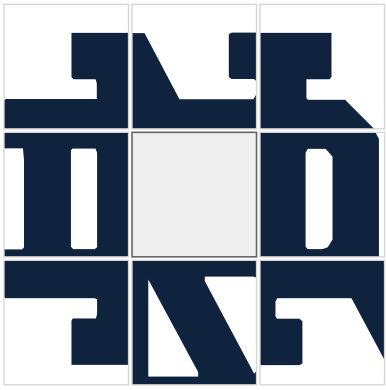
The frontier policy changes the order, not the set of nodes visited. This function traverses a supplied graph; children returns its ordered neighbors.

```
from collections import deque
def traverse(start, children, depth_first=False):
    frontier = deque([start])
    seen = {start}
    while frontier:
        if depth_first:
            state = frontier.pop()
        else:
            state = frontier.popleft()
        yield state
        next_states = list(children(state))
        if depth_first:
            next_states.reverse()
        for child in next_states:
            if child not in seen:
                seen.add(child)
                frontier.append(child)
```

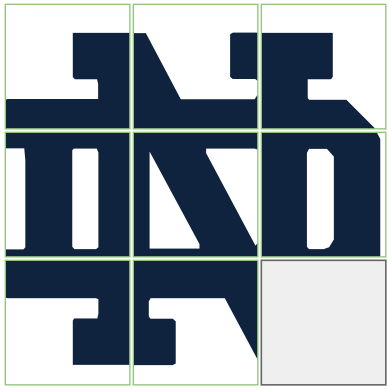
seen prevents rediscovery. Reverse children for DFS so that the first legal move is selected first.

Now Ask for a Solution

Traversal asks, “What can we reach?” Search asks, “Can we reach a state with the property we want?” For our puzzle, that property is matching the target.



START



TARGET

Add a **goal test**: compare the selected board with the target board. Once it matches, stop. There is no need to traverse the rest of the state space.

But returning “found it” is not enough. A puzzle solution must tell us **which moves to make**. The sequence of examined boards is usually not a legal sequence of moves.

Why Search Belongs in AI

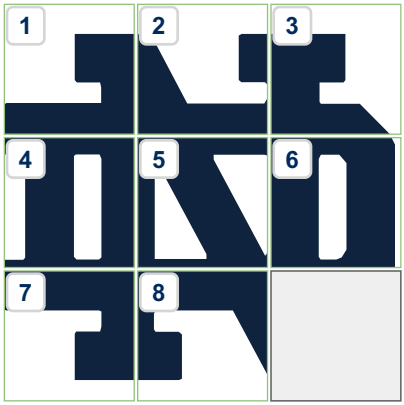
A solver can reason about actions by exploring their consequences before taking them. The puzzle gives us a concrete model of that process.

State	One complete arrangement of the tiles.
Action	One legal slide into the blank.
Goal	The required picture arrangement.
Cost	One per move; minimize the number of moves.

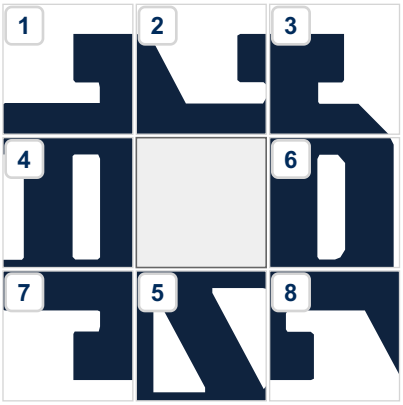
In classical AI, symbolic states and explicit rules let a program find an action sequence without being given the answer. Planning uses this same idea: states record what is true; actions change it.

Represent the Picture

We see an ND monogram. The program needs stable identities for the pieces. Number each tile by its position in the target; use 0 for the blank.



TARGET TILE IDS



START TILE IDS

```
GOAL  = (1, 2, 3, 4, 5, 6, 7, 8, 0)
START = (1, 2, 3, 4, 0, 6, 7, 5, 8)
```

Read the board row by row. A tuple is immutable and can be used in a set or dictionary, so equality and repeated-state checks are straightforward.

These IDs do not measure closeness to the goal. **Uninformed** search uses legal moves and a goal test, but no estimate of the remaining distance.

Generate the Legal Moves

successors returns (move, child) pairs. A move names the blank's direction. Swapping with a neighboring piece produces a fresh board.

```
def successors(state):
    blank = state.index(0)
    row, col = divmod(blank, 3)
    steps = [("U", -1, 0), ("L", 0, -1),
             ("D", 1, 0), ("R", 0, 1)]
    for move, dr, dc in steps:
        r, c = row + dr, col + dc
        if 0 <= r < 3 and 0 <= c < 3:
            other = 3*r + c
            child = list(state)
            child[blank] = state[other]
            child[other] = 0
            yield move, tuple(child)
```

For START, blank = 4 and (row, col) = (1, 1). A down move swaps positions 4 and 7, producing board D. Boundary checks prevent crossing a board edge.

BFS Searches the Puzzle

Keep the queue, but test each selected board for the goal. The first few selections match our traversal excerpt; reaching K changes what happens next.

Boards examined

A B C D E

F G H I J K

K matches the target: stop here.

Moves in the returned solution



A down D right K

BFS makes **11 goal tests** and returns **2 moves**. B and C are not steps in the solution.

Remember How We Got Here

When a board is first discovered, record its parent and the move that produced it. Those records survive even after the parent leaves the queue.

parent[D] = A

action[D] = down

parent[K] = D

action[K] = right

Trace backward from the goal

K → **D** → **A**

K's parent is D; D's parent is A. A is the start, marked by a parent value of None. Collect right, then down while following these links.

Reverse to get execution order

down → right

BFS: Queue and Parents

parent doubles as the discovered-state record. Once a board has a parent, it never enters the queue again. recover follows the records on the next page.

```
def bfs(start, goal):
    frontier = deque([start])
    parent = {start: None}
    action = {}
    while frontier:
        state = frontier.popleft()
        if state == goal:
            return recover(state, parent, action)
        for move, child in successors(state):
            if child not in parent:
                parent[child] = state
                action[child] = move
                frontier.append(child)
    return None
```

If the queue empties, every reachable board has been examined without finding the goal. None means failure, not an empty solution.

Return Moves, Not Visits

recover walks from the goal to the start. Each recorded action points forward from a parent to its child, even though reconstruction follows parents backward.

```
def recover(state, parent, action):
    moves = []
    while parent[state] is not None:
        moves.append(action[state])
        state = parent[state]
    return moves[::-1]
```

Two different successful answers

From A, BFS returns ["D", "R"].

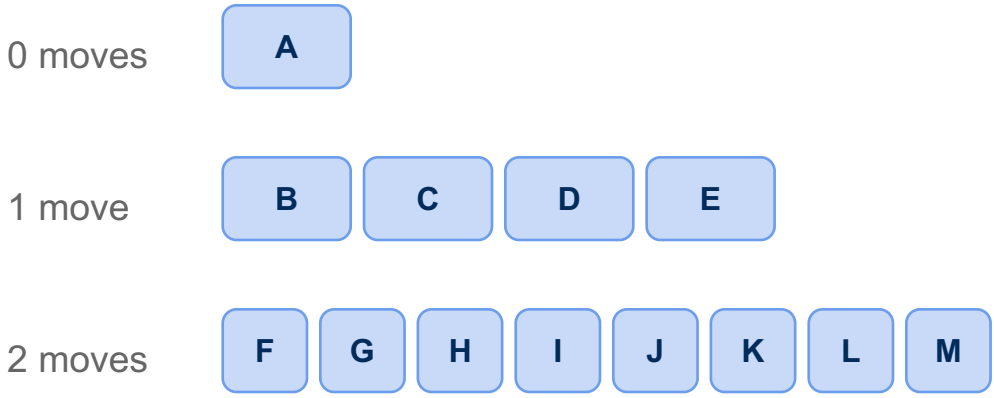
From the target itself, BFS returns [].

The empty list is success with zero moves. It is different from None, which means no solution was found.

All code uses hashable board tuples; the blank is 0.

Why BFS Finds Fewest Moves

A queue keeps shallower boards ahead of deeper boards. Expanding a depth-d board appends only depth-(d + 1) successors at the back.



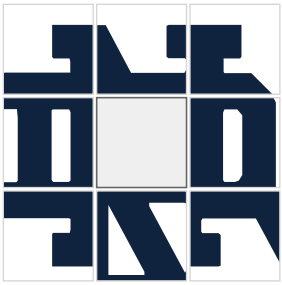
When K is selected at depth 2, every depth-0 and depth-1 board has already failed the goal test. A one-move solution cannot have been missed.

The first discovery of a board is also by a shortest route to that board. That is why keeping its first parent is safe for BFS.

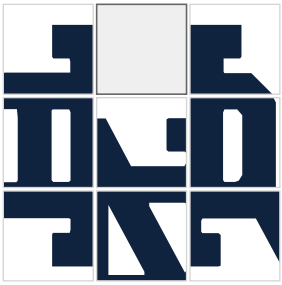
Each move costs 1, so fewest moves means least cost. Unequal costs need a different selection rule.

DFS Can Go the Wrong Way

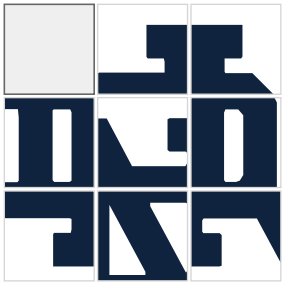
BFS must retain its broad frontier. DFS can instead pursue one branch while keeping the choices needed to return. But its move order does not know which way helps.



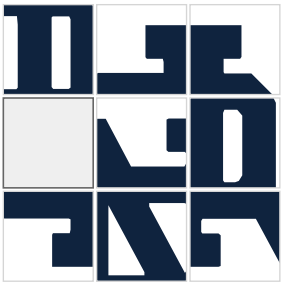
A: depth 0



B: depth 1



F: depth 2



depth 3



depth 4



depth 5

DFS goes up, then left, then continues below F. The two-move solution through D is still waiting. F was only a leaf of our drawing.

Finding any solution and finding a shortest solution are different requirements.

DFS: Calls Keep the Choices

Each call follows one successor and waits for its answer. The caller's loop remembers the remaining alternatives. path blocks moves back to an ancestor.

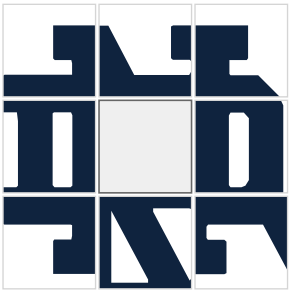
```
def dfs(path, goal):
    state = path[-1]
    if state == goal:
        return []
    for move, child in successors(state):
        if child in path:
            continue
        rest = dfs(path + [child], goal)
        if rest is not None:
            return [move] + rest
    return None
```

A failed child returns None; the parent resumes its loop. A successful child returns a move list; the parent prepends its own move and returns immediately.

This recursive version explains control flow. Deep searches can exceed Python's recursion limit; the bounded version below controls depth.

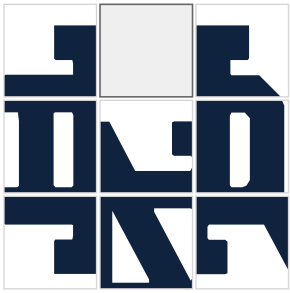
Why Put a Limit on Depth?

A path check prevents cycles, but it does not force DFS to return quickly. Let a pass examine at most L moves from the starting board.



A

moves left = 2



B

moves left = 1



F

moves left = 0

With L = 2, the call at F cannot continue to depth 3. It returns control to B, which can now try G. Once B's permitted choices finish, A can try C.

A limit is not evidence that no solution exists. It only says this pass is not allowed to look deeper. We must preserve that distinction in the return value.

Found, Exhausted, or Cut Off

A depth-limited call needs three outcomes. “I did not find the goal” alone does not tell the caller whether a deeper pass might help.

A move list

Success. At K, return [] even if no moves remain. The caller adds the move that reached K.

None

Failure. All allowed continuations are exhausted, with no success and no cutoff.

CUTOFF

The boundary blocked further exploration. At F with zero moves left, another legal board exists but cannot be examined.

Test the goal before the boundary: a goal exactly at L succeeds.

Depth-Limited DFS: Boundary

First recognize success. Then identify children that do not repeat an ancestor. Distinguish an exhausted branch from one stopped by the limit.

```
CUTOFF = object()
def limited(path, left, goal):
    state = path[-1]
    if state == goal:
        return []
    children = [(m, s) for m, s in successors(state)
                if s not in path]
    if not children:
        return None
    if left == 0:
        return CUTOFF
```

CUTOFF is a unique sentinel, distinct from both None and a list of moves. The rest of this same function continues on the next page.

At a non-goal boundary, we inspect legal moves without recursively exploring the resulting boards.

Depth-Limited DFS: Returns

Continue inside limited. A child receives one fewer move. Success returns immediately; a cutoff is remembered while the other children are still tried.

```
    cut = False
    for move, child in children:
        rest = limited(path + [child],
                       left - 1, goal)
        if rest is CUTOFF:
            cut = True
        elif rest is not None:
            return [move] + rest
    return CUTOFF if cut else None
```

At limit 2, B’s descendants are cut off, but that does not end the whole pass. A still tries C and then D. D eventually reaches K and returns ["R"]. A returns ["D", "R"].

If no child succeeds: any cutoff means CUTOFF; otherwise every child failed, so return None.

Let the Limit Grow

We do not know the required depth in advance. Iterative deepening restarts depth-limited DFS at limits 0, 1, 2, … until a pass succeeds or exhausts the search.

Limit 0



1 goal test; CUTOFF

Limit 1



5 goal tests; CUTOFF

Limit 2



10 goal tests; return ["D", "R"]

16 tests across all passes; 2 solution moves. The last pass uses DFS order. IDS finds the same minimum depth as BFS, not the same visitation order.

IDS: Restart Only on Cutoff

Each pass begins at the original start. A move list or None is a final answer; only CUTOFF asks us to try a larger limit.

```
def iddfs(start, goal):
    limit = 0
    while True:
        result = limited([start], limit, goal)
        if result is not CUTOFF:
            return result
        limit += 1
```

Why does this eventually work?

With finitely many legal moves per state, every bounded pass can finish. If the shallowest goal is at depth d, earlier passes rule out shallower solutions and the depth-d pass finds one.

Thus IDS is complete for goals at finite depth and returns a fewest-move solution. Equal move costs make that solution least-cost too.

Why Repeat Earlier Work?

IDS trades repeated computation for avoiding BFS's broad frontier. The overhead depends on how the search tree branches.

Depth	Nodes	Passes	Tests
0	1	5	5
1	10	4	40
2	100	3	300
3	1,000	2	2,000
4	10,000	1	10,000

In this illustrative 10-way tree, one full depth-4 pass tests 11,111 nodes; limits 0–4 test 12,345. That is about 11% extra work because most nodes are deepest.

This is not the puzzle's branching pattern. On a single chain to depth d, IDS instead repeats about $d^2/2$ visits versus about d for one traversal.

What Are We Keeping?

Memory depends on both the exploration order and the repeated-state policy. A stack alone does not guarantee a small total memory footprint.

BFS: retain the discovered region

Our BFS keeps a queue plus a parent entry for every discovered board. That can be expensive as the reachable region grows.

DFS and IDS: retain the active branch

Our recursive searches check only the current path, so another branch may revisit the same board. They retain active calls and unfinished choices, not a global discovered set.

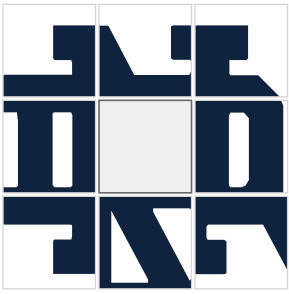
The earlier traversal code deliberately kept a global seen set to visit each state once. That set still costs memory even when the frontier is a stack.

A shared path with append/pop avoids copying each path. Recursive call frames still consume memory.

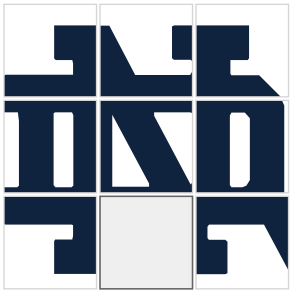
Run the Same Puzzle

Use the functions in this chapter together. Both solvers return directions for the blank, which you can replay one move at a time.

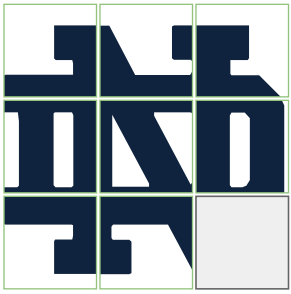
```
START = (1, 2, 3, 4, 0, 6, 7, 5, 8)
GOAL  = (1, 2, 3, 4, 5, 6, 7, 8, 0)
print(bfs(START, GOAL))
# ['D', 'R']
print(iddfs(START, GOAL))
# ['D', 'R']
print(bfs(GOAL, GOAL))
# []
```



A



D



K

BFS makes 11 goal tests. IDS makes 16 across three passes. Both return exactly two moves. These counts use goal testing when a state is selected.

From Exploring to Solving

We began with traversal: systematically visit the states. Adding a goal test and solution reconstruction turned that machinery into search.

BFS explores by increasing depth

With finite branching, BFS reaches any finite-depth goal by a fewest-move path. It stores both the frontier and the discovered states.

DFS pursues one branch

On a finite graph, path checks let DFS eventually exhaust the simple paths. It can postpone a nearby goal and does not guarantee a shortest solution.

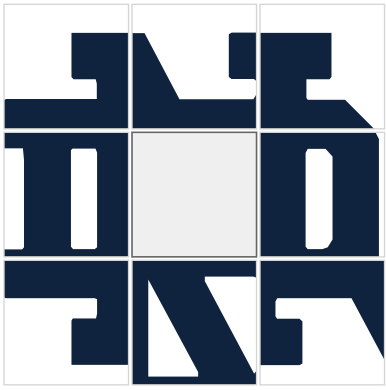
IDS tries successively deeper passes

It finds a fewest-move solution without retaining a broad frontier, but repeats work and may revisit states along different paths.

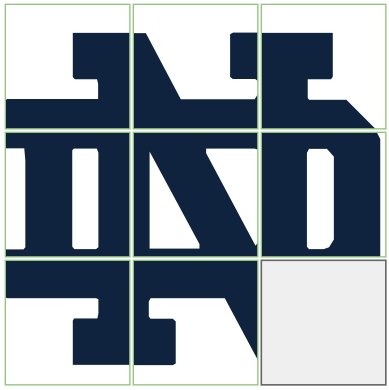
The puzzle is finite. On infinite spaces, DFS can follow an endless branch.

What Uninformed Search Ignores

You can look at A and notice that much of the picture is already in place. Our algorithms never use that information to select a board.



START



TARGET

They know how to move and how to recognize success. They do not estimate which unfinished possibility is closer to the goal.

Next: informed search. Keep this same puzzle, but use estimates such as misplaced tiles or Manhattan distance to guide exploration.

Later, Sudoku will motivate a different formulation: variables, possible values, and constraints that an assignment must satisfy.

Puzzle artwork: University of Notre Dame monogram