Programming in Logic: Prolog

Solving the 8-Puzzle

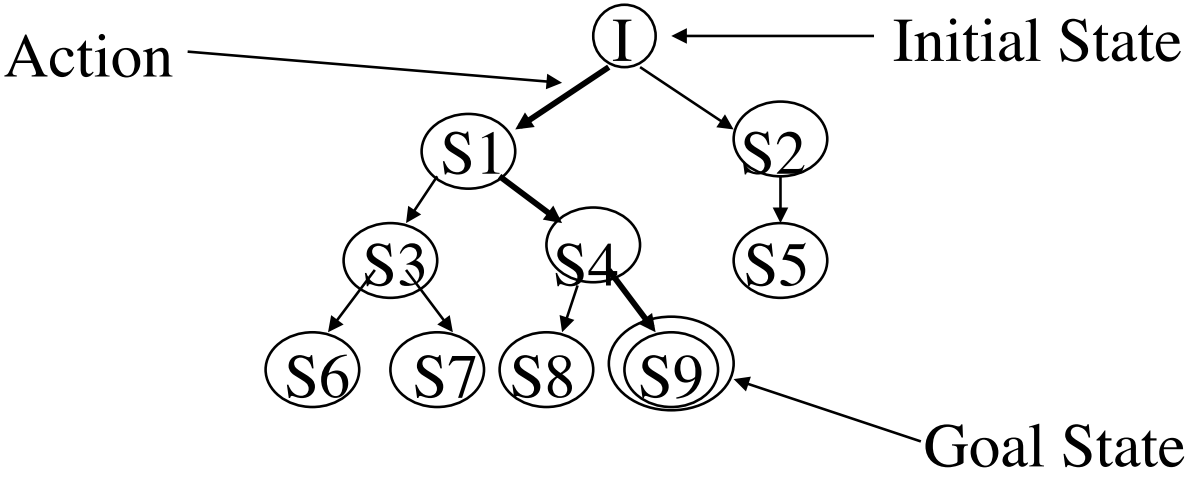
Readings: 11.1-2

Types of Puzzles

- The puzzles we've seen so far have involved inferring new facts about a state that are consistent with the given facts and some "rules" about the domain.

- Going to look at a puzzle, where the question is to find some sequence of actions/states which lead from an initial state to a goal state.

State Space

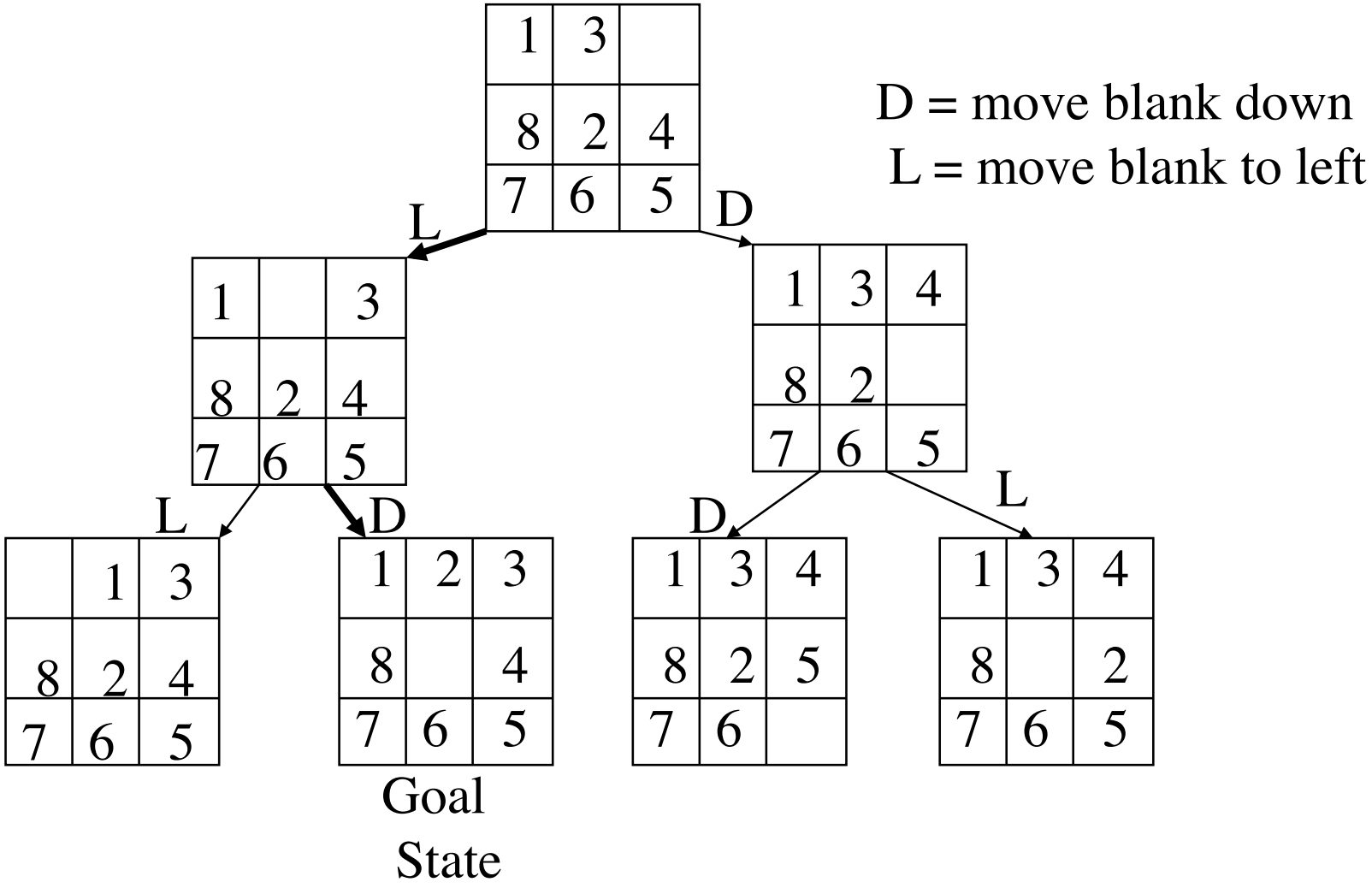


Solution = [I, S1, S4, S9]

8 Puzzle

Find a sequence of actions/states that transform the initial state into the goal state. Actions involve sliding a tile into the empty/blank slot. Easiest to imagine moving the blank slot.

8 Puzzle State Space



How to Approach

- First, what does a solution structure look like.
- Second, how to generate all possible candidates.
- Third, how to identify actual solutions.

Solution Structure

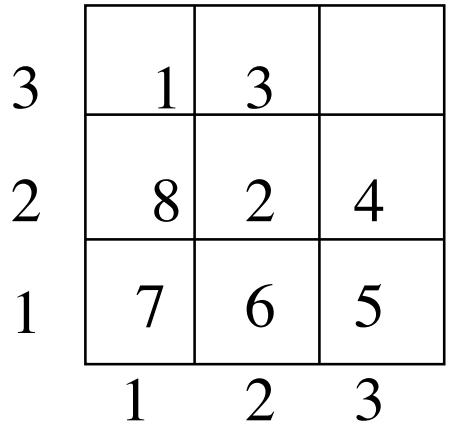
- Our solution structure will be a list of states, *[S0, S1, S2, …, Sn]*, with *S0* the initial state, *Sn* the goal state, and each *Si*, *Si+1* being connected by a legal move.

What's in a State?

- Just as in the 8 queens puzzle, there's different ways of representing a state:
  - List the positions of the tiles.
  - List the contents of the positions.
  - List the positions of the blank and the tiles.
- A state will be a list of the positions of blank and tiles (in ascending order). E.g.,
  - *[BlankPos, T1Pos, T2Pos, …, T8Pos]*

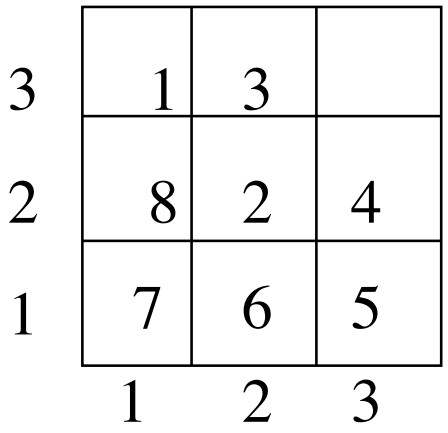
Representing Position

- How should position of a tile be represented?
- Number of different ways, we choose cartesian coordinates.



Tile 8's position is coordinates: 1, 2. What data structure should we use to represent coordinates?
Same as for 8 queens: *X*/*Y*

State Representation cont'd



This state is represented as:

[3/3,1/3,2/2,2/3,3/2,3/1,2/1,1/1,1/2]

b 1 2 3 4 5 6 7 8

Naïve Approach to Candidate Generation

- Generate sequences of valid states & test each:
  - is first state the initial state?
  - valid sequence of legal transitions between states?
  - is last state the goal state?
- How many states are there? ~50,000 states
- How many sequences of length **n**? 50,000 ** **n**
- Naïve approach too computationally expensive.

Test Incorporation

- How to narrow down number of candidates?
- We have 3 testers:
  - is first state the initial state?
  - valid sequence of legal transitions between states?
  - is last state the goal state?
- Try incorporating some of them into the generator.

Incorporating Testers

- Is first state the initial state?
  - Generate only those sequences that begin with *S0*.
- Valid sequence of legal transitions between states?
  - Only add legal transition states to end of sequences.
- Is last state the goal state?
  - Can't incorporate this on top of preceding two, it will remain a tester.

Our "*solve/3*" Relation

- **Signature:** *ids(+StartState, +GoalState, ?RevSol)*

- *ids(StartState, GoalState, RevSol) :*
  path(StartState, EndState, RevSol), **generator**
  EndState = GoalState. **tester**

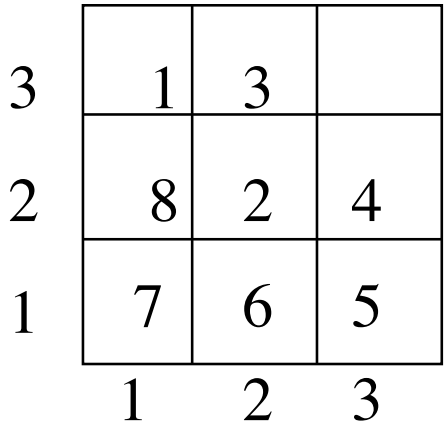
Generating Solution Candidates

- Forward chaining:
  - List containing just *S0* is candidate.
  - Given candidate, adding legal transition state to end is candidate.
- Forward chaining defines the search space topology, i.e., the nodes & edges.

Computing Legal Transitions

- Given *[S0, S1, …, Si]*, compute the legal transition states that can be added to end.
- Legal transitions only depend on last state.
- Given last state *Si*, what is easiest way to compute legal transitions?
- Could look at each tile & see which ones move.
- Could look at blank & see which way can move.

Computing Legal Transitions Based on Blank Position



Depending on where blank is, it can move in as many as 4 different directions: up, down, left, right. In example, blank is in upper right hand corner & can only move down or left.

/* newState(+OldState, ?NewState) */

newState([OldBX/OldBY|OldRest], NewState) :-

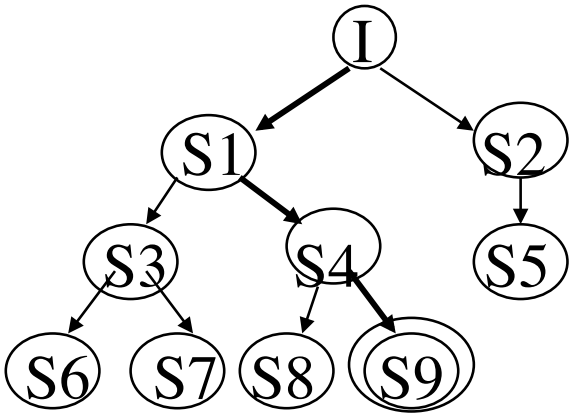
x/y coords of blank

```
        OldBX > 1,
        moveLeft([OldBX/OldBY|OldRest], NewState).

newState([OldBX/OldBY|OldRest], NewState) :-
        OldBX < 3,
        moveRight([OldBX/OldBY|OldRest], NewState).

newState([OldBX/OldBY|OldRest], NewState) :-
        OldBY > 1,
        moveDown([OldBX/OldBY|OldRest], NewState).

newState([OldBX/OldBY|OldRest], NewState) :-
        OldBY < 3,
        moveUp([OldBX/OldBY|OldRest], NewState).
```

Search Space Traversal

- Given search space topology, & initial node, there are number of different ways to traverse that space: depth-first, breadth-first, etc.
- Each has advantages & disadvantages.
- Iterative deepening combines advantages of both depth-first & breadth -first.
- Iterative deepening does succession of bounded depth-first searches.
- If iteration does not find solution, bound is increased.

Iterative Deepening Traversal



Iterative deepening would do the following traversals:

I *(fail)*

I - S1 - S2

(fail)

- S2 -S5 *(fail)*
- S7 - S4 - S8 - S9 *(success)*

I - S1 - S3 - S4

I - S1 - S3 - S6

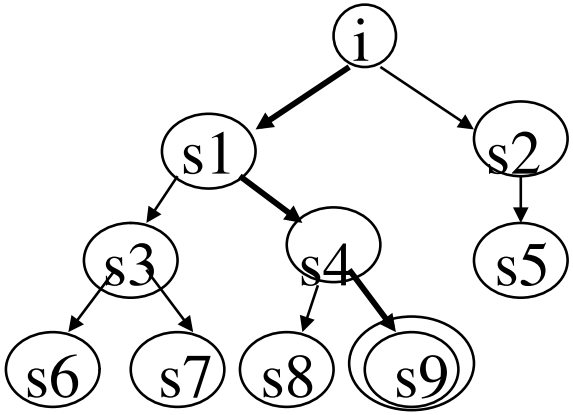
Solution Candidate Generator

- Our solution candidate generator will generate the candidates using iterative deepening.
- Our generator's signature is:

  path(+StartState, +EndState, ?RevSol)
- *RevSol* will start out as a list containing just *S0*.
- Next, sequences of legal state transitions of length 2 are generated, etc.

Our Generator in Prolog

```
/* path(+StartNode, ?EndNode, ?Path) */
path(StartNode, StartNode, [StartNode]).

path(StartNode, EndNode, [EndNode|Path]) :-
        path(StartNode, PenUltimateNode, Path),
        newState(PenUltimateNode, EndNode),
        not(member(EndNode,Path)).
```



[*path(i,s9,RS)*]

2. RS = [s9|P1]

[*path(i, Pen, P1)*
newState(Pen, End)
…]

1. *Pen = i*
P1 = [i]

[

/* path(+StartNode, ?EndNode, ?Path) */

1. path(StartNode, StartNode, [StartNode]).

2. path(StartNode, EndNode, [EndNode|Path]) :-
path(StartNode, PenUltimateNode, Path),
newState(PenUltimateNode, EndNode),
not(member(EndNode,Path)).