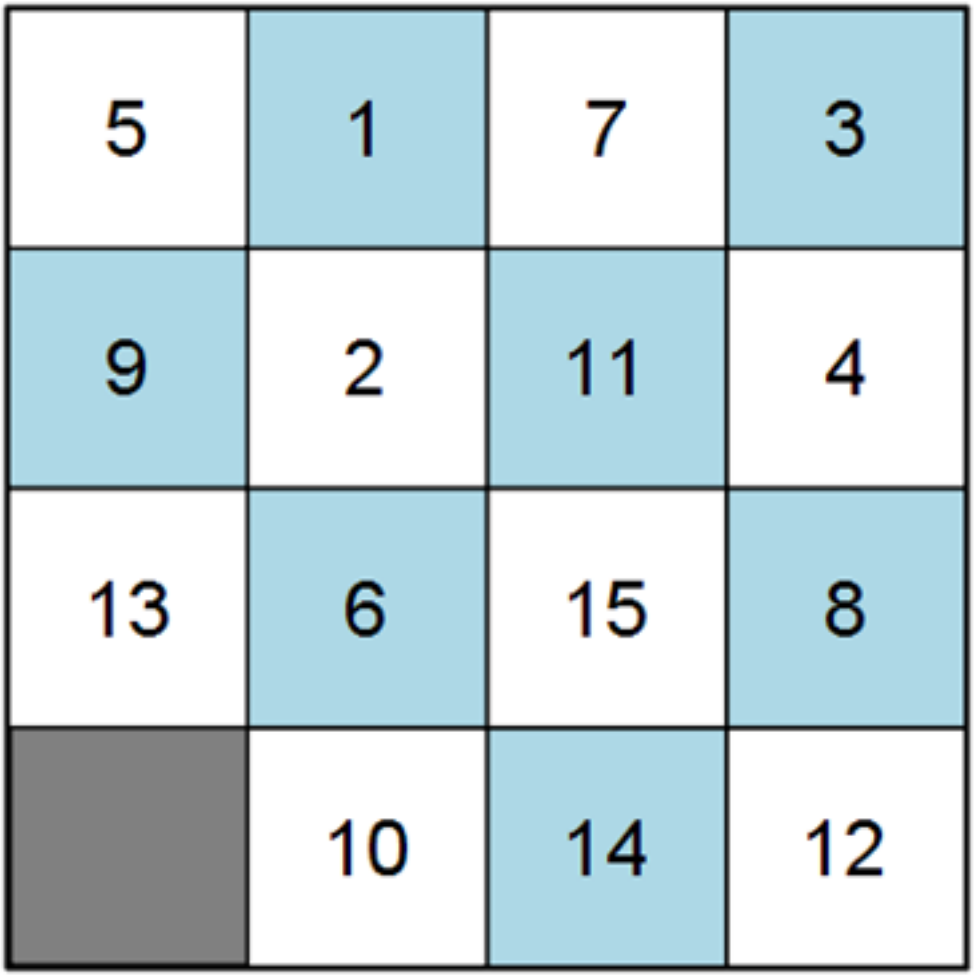
Solving the 15-puzzle

Curtis Bright, Maplesoft



- The 15-puzzle is a classic "sliding tile" puzzle. It consists of a 4 × 4 grid containing tiles numbered 1 through 15 along with one missing tile.
- The objective of the puzzle is to arrange the tiles so that they are in ascending order when read from left to right and top to bottom.
- The only moves allowed are those that slide a tile adjacent to the blank space into the blank space.
- It is known that every valid position can be solved in at most 80 moves.
- It's possible to solve this puzzle without any knowledge of search algorithms at all. To do this we encode the rules of the puzzle as constraints in mathematical logic and use Maple's built-in efficient SAT solver to find a solution. A SAT solver takes as input a formula in Boolean logic and returns an assignment to the variables which makes the formula true (if one exists). See the Satisfy command for more information.

Setting up the problem

- We'll use the Boolean variables $S[i, j, n, t]$ to denote that the square (i, j) contains tile n after t moves. Here $1 \leq i, j \leq 4$, $1 \leq n \leq 16$ (we let 16 denote the blank tile) and $0 \leq t \leq 80$ since each instance requires at most $M = 80$ moves to solve.
- We'll also use a function to compute squares which are adjacent to a given square (i, j).

```
Returns the set of squares adjacent to square (i, j)
```

Generating board constraints

- The board does not permit two tiles to occupy the same square at the same time. These constraints are of the form $\neg S[i, j, n, t] \vee \neg S[i, j, m, t]$ for each valid tile numbers $n \neq m$, valid indices i and j, and valid timestep t.
- We define the function *noSquareHasTwoTiles* to generate these constraints up to timestep M.

```
Returns a set of constraints specifying no two tiles occupy th
```

Generating transition constraints

- Next we need to generate constraints which tell the SAT solver how the state of the board can change from time t to time $t + 1$.
- There are two cases to consider, depending on if a square (or an adjacent square) contains the blank tile.

Static transition constraints

- The easier case is when a square (i, j) and none of the squares adjacent to that square are blank. In that case, the rules of the puzzle imply that the tile in square (i, j) does not change.
- We define the function *doesNotChange* that generates a constraint that says that the tile in square (i, j) does not change at time t. These constraints are of the form $(S[i, j, 1, t] \Leftrightarrow S[i, j, 1, t + 1]) \wedge \cdots \wedge (S[i, j, 16, t] \Leftrightarrow S[i, j, 16, t + 1])$

- For convenience, we also define the following functions:
  * *notEqualToBlank* that says square (i, j) does not contain the blank tile at time t
  * *notEqualOrAdjacentToBlank* that says square (i, j) (or any adjacent squares) do not contain the blank tile at time t
  * *swapSquares* that says the tile at square (i, j) swaps with the tile at square (k, l) at time t
- The static transition constraints are of the form $notEqualOrAdjacentToBlank(i, j, t) \Rightarrow doesNotChange(i, j, t)$ for all squares (i, j) and timesteps t.

```
Returns a constraint that states square (i, j) does not cont
```

Slide transition constraints

- The harder transition case is when a square (i, j) contains the blank tile. In that case, we need to encode the fact that the blank tile will switch positions with exactly one of the squares adjacent to square (i, j).
- If the blank tile on square (i, j) switches positions with the square $X = (x_1, x_2)$ at time t this can be encoded as the clause $swapSquares(x_1, x_2, i, j, t)$. We also need to enforce that all squares adjacent to (i, j) other than (x_1, x_2) do not change; these constraints are of the form $\bigwedge_Y doesNotChange(y_1, y_2, t)$ where $Y = (y_1, y_2) \neq (x_1, x_2)$ is adjacent to (i, j). We let *oneAdjacentTileMoved* denote the constraint that one tile adjacent to (i, j) moves to square (i, j) at time t.
- The slide transition constraints are of the form $equalToBlank(i, j, t) \Rightarrow oneAdjacentTileMoved(i, j, t)$ for all squares (i, j) and timesteps t.

```
Returns a constraint that states square (i, j) does not cont
```

Generating the starting and ending constraints

- Finally, we need to encode the starting position (the board state at time 0) and ending position of the puzzle.
- We let *startingConstraint* denote the constraint saying the board starts in its starting position and let *endingConstraints*(N, M) denote the constraint saying the board is solved sometime between timesteps N and M.

```
startingPosition := [[5, 1, 7, 3], [9, 2, 11, 4], [13, 6, 15, 8
```

Finding a solution

- We use the Satisfy command from the Logic package which finds a satisfying assignment of a logical formula if one exists.
- We use the time command to measure how quickly the solution is found.
- For efficiency reasons we only start looking for solutions with at most 5 moves; if no solution is found then we look for solutions using at most 10 moves and continue in this manner until a solution is found.

```
M := 5:
```

```
Generated 11682 constraints with 1536 variables and now searching
for a solution with at most 5 moves...
No solution found with at most 5 moves in 2.19 seconds.
Generated 21442 constraints with 2816 variables and now searching
for a solution with at most 10 moves...
No solution found with at most 10 moves in 3.77 seconds.
Generated 31202 constraints with 4096 variables and now searching
for a solution with at most 15 moves...
Solution found with at most 15 moves in 9.28 seconds.
```

Visualizing the solution

- The state of the board at each timestep t is determined by iterating over the set returned by Satisfy and checking which variables were assigned to true.

```
digits := Array(1..4, 1..4, 0..M):
```

- Once a solved state is reached any additional moves are irrelevant and we ignore them in the visualization. We let *numMoves* represent the length of the shortest found solution.

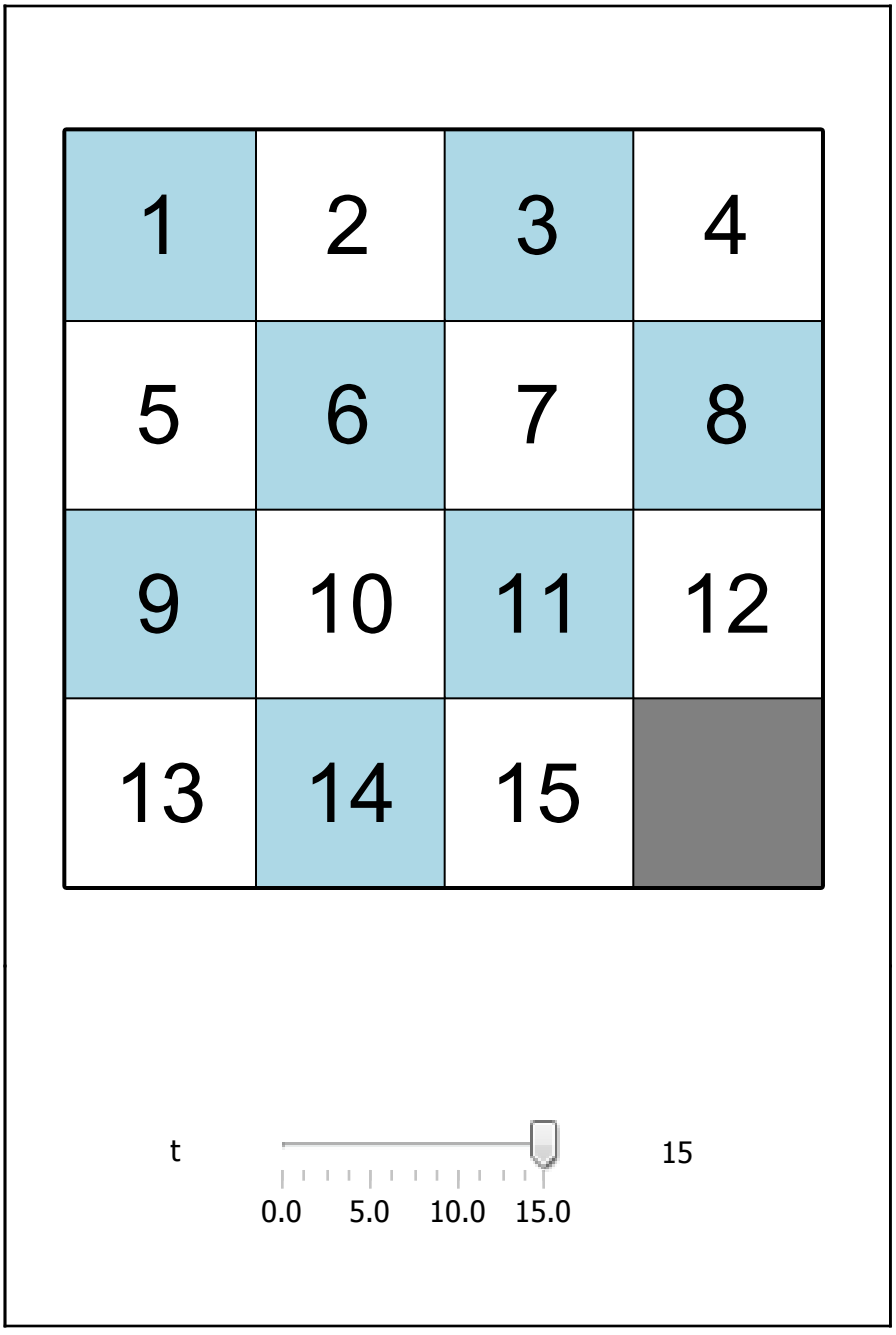
```
if satisfyingAssignment = NULL then
```

- We'll use commands from the plots and plottools packages to draw a visual representation of a 15-puzzle board. The commands to draw the tiles will be stored in an array *plotCommands*. To smoothen out the animation we use a parameter *res* to control the "resolution" of the animation, i.e., number of individual frames drawn in each timestep. The commands for the (i, j)th tile at real time r (where r will be a real number between 0 and *numMoves*) will be stored in $plotCommands[i, j, \text{floor}(r \cdot res)]$.

```
with(plots):
```

- We'll use the Explore command to allow us to see how the board state changes over time with a slider controlling what time t to view.

```
Returns the commands to draw the state of the game at real tim
```


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
t
15
0.0
5.0
10.0
15.0