

# CS101 Algorithms and Data Structures

A\* search and Backtracking

# Outline of A\* Search Algorithm

For A\* search algorithm, we will cover:

- It solves the single-source shortest path problem
- Restricted to *physical* environments
- First described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael
- Similar to Dijkstra's algorithm
- Uses a hypothetical shortest distance to weight the paths

# Background

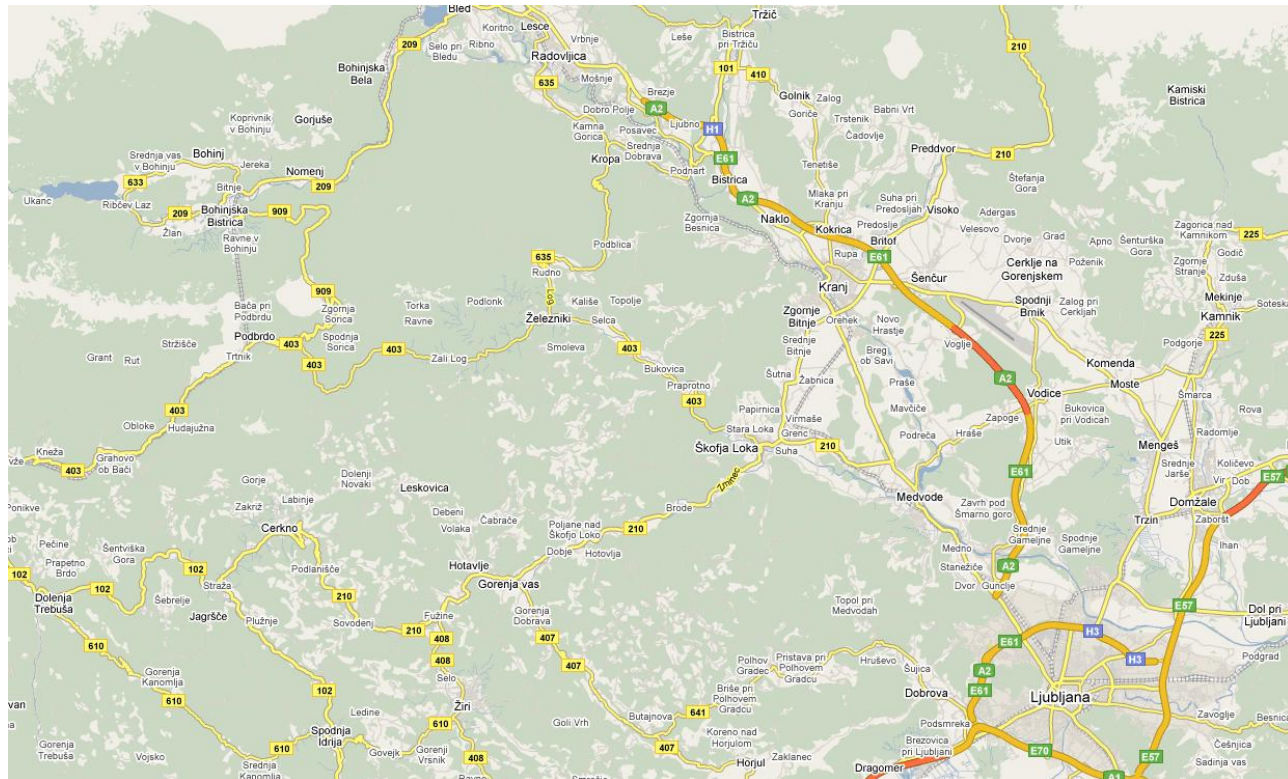
Assume we have a heuristic lower bound for the length of a path between any two vertices

*E.g.*, a graph embedded in a plane

- the shortest distance is the Euclidean distance  
“as the crow flies”
- use this to guide our search for a path  
“as the fox runs”

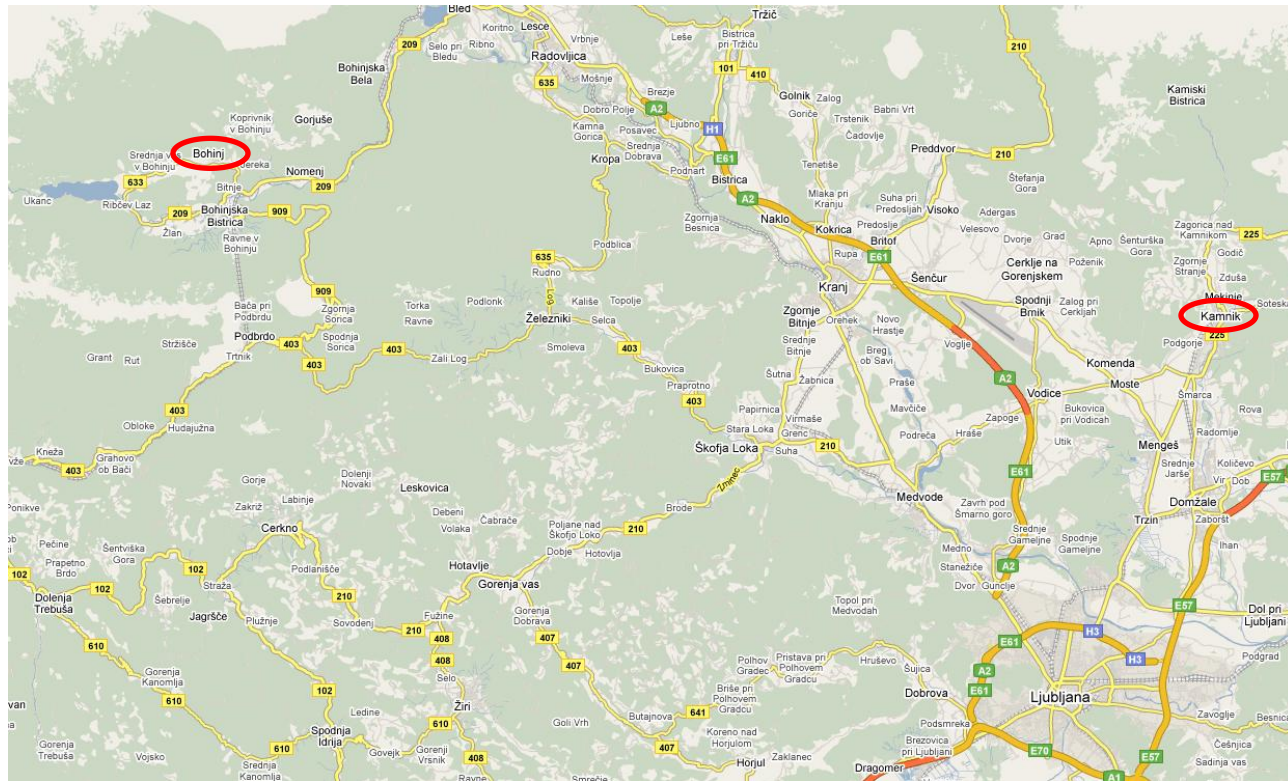
# Idea

Consider this map of Slovenia



# Idea

Suppose we want to go from Kamnik to Bohinj





# Idea

A lower bound for the length of the shortest path to Bohinj is  
 $h(\text{Kamnik, Bohinj}) = 53 \text{ km}$



# Idea

Any actual path must be at least as long as 53 km



# Idea

Suppose we have a 28 km shortest path from Kamnik to Kranj:

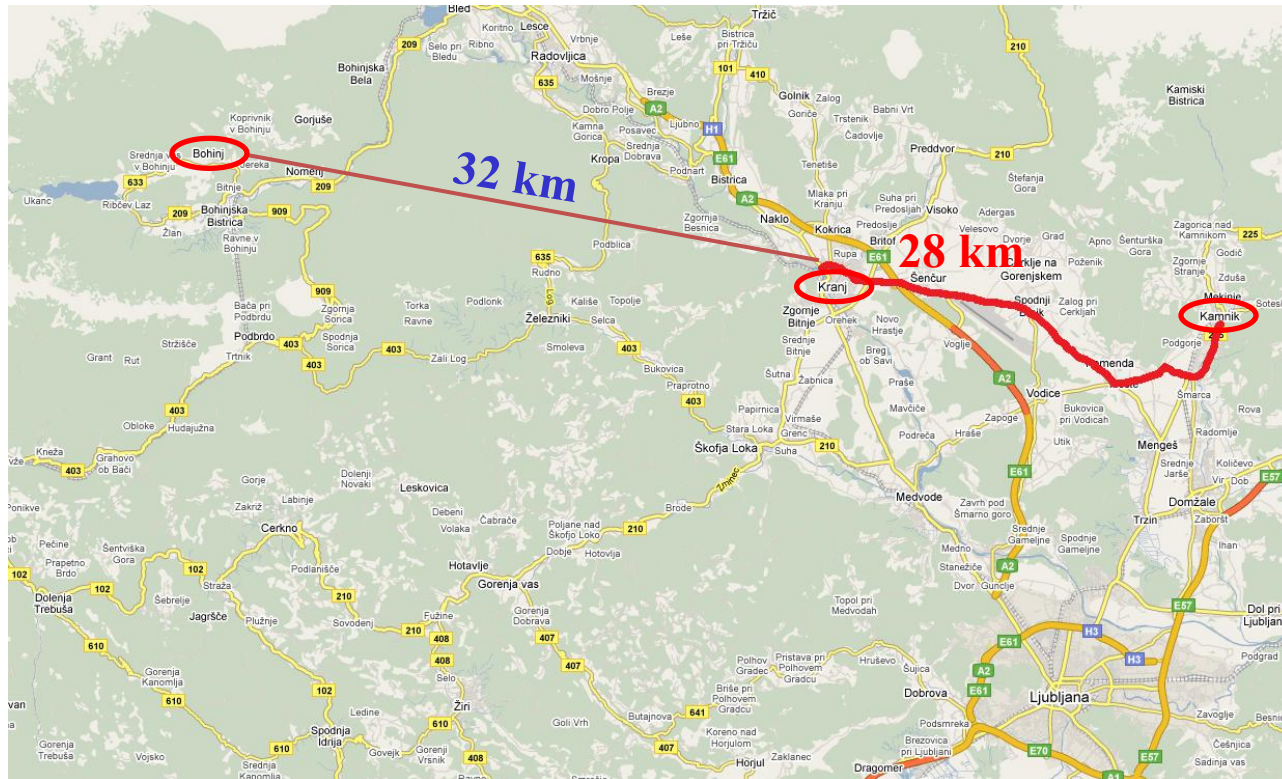
$$d(\text{Kamnik}, \text{Kranj}) = 28 \text{ km}$$





# Idea

A lower bound on the shortest distance from Kranj to the destination is now  $h(\text{Kranj}, \text{Bohinj}) = 32 \text{ km}$



# Idea

Thus, the weight of the path up to Kranj is

$$w(\text{Kranj}) = d(\text{Kamnik}, \text{Kranj}) + h(\text{Kranj}, \text{Bohinj}) = 60 \text{ km}$$





# Idea

Any path extending this given path to Bohinj must be at least 60 km

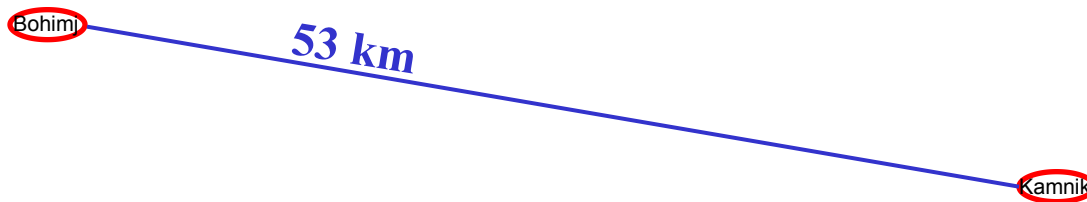


# Idea

The value  $w(\text{Kranj})$  represents the shortest possible distance from Kamnik to Bohinj given that we follow the path to Kranj

As with Dijkstra's algorithm, we must start with the null path starting at Kamnik:

$$\begin{aligned} w(\text{Kamnik}) &= d(\text{Kamnik}, \text{Kamnik}) + h(\text{Kamnik}, \text{Bohimj}) \\ &= 0 \text{ km} + 53 \text{ km} \end{aligned}$$





# Algorithm Description

Suppose we are finding the shortest path from vertex  $a$  to a vertex  $z$

The A\* search algorithm initially:

- Marks each vertex as unvisited
- Starts with a priority queue containing only the initial vertex  $a$ 
  - The priority of any vertex  $v$  in the queue is the weight  $w(v)$  which assumes we have found the shortest path to  $v$  (initialize it to be infinity except for the initial vertex  $a$ )
  - Shortest weights have highest priority
- For each vertex  $v$ ,  $d(a, v)$  is the shortest known distance from  $a$  to  $v$ ,  $d(a, a) = 0$  and  $d(a, v) = \text{infinity}$  for all  $v \neq a$
- For each vertex  $v$ ,  $h(v, z)$  is the heuristic distance from  $v$  to  $z$

# Algorithm Description I (*Tree Search*)

The algorithm then iterates:

- Pop the vertex  $u$  with highest priority
- For each adjacent vertex (neighbor)  $v$  of  $u$ :
  - If  $w(v) = d(a, u) + d(u, v) + h(v, z)$  is less than the current weight/priority of  $v$ , update the path leading to  $v$  and its priority
    - If  $v$  is not in the queue, push  $v$  into the queue

Continue iterating until the item popped from the priority queue is the destination vertex  $z$

# Algorithm Description II (*Graph Search*)

The algorithm then iterates:

- Pop the vertex  $u$  with highest priority
  - mark  $u$  as visited
- For each **unvisited** adjacent vertex (neighbor)  $v$  of  $u$ :
  - If  $w(v) = d(a, u) + d(u, v) + h(v, z)$  is less than the current weight/priority of  $v$ , update the path leading to  $v$  and its priority
    - If  $v$  is not in the queue, push  $v$  into the queue

Continue iterating until the item popped from the priority queue is the destination vertex  $z$

**Note:** Tree Search or Graph Search are just two different ways to search the solution.

# Comparison of Priorities

Suppose we have a path with

$$w(\text{Smarca}) = 4 \text{ km} + 52 \text{ km} = 56 \text{ km}$$





# Comparison of Priorities

We can extend this path to Moste:

$$w(\text{Moste}) = 4 \text{ km} + 5 \text{ km} + 48 \text{ km} = 57 \text{ km}$$



# Comparison of Priorities

We can also extend this path to Menges:

$$w(Menges) = 4 \text{ km} + 4 \text{ km} + 51 \text{ km} = 59 \text{ km}$$



# Comparison of Priorities

The smaller weight path to Moste has priority—extend it first



# Comparison with Dijkstra's Algorithm

This differs from Dijkstra's algorithm which gives weight only to the known path

- Dijkstra would chose Menges next:

$$d(\text{Kamnik}, \text{Moste}) = 9 \text{ km} > 8 \text{ km} = d(\text{Kamnik}, \text{Menges})$$

## Difference:

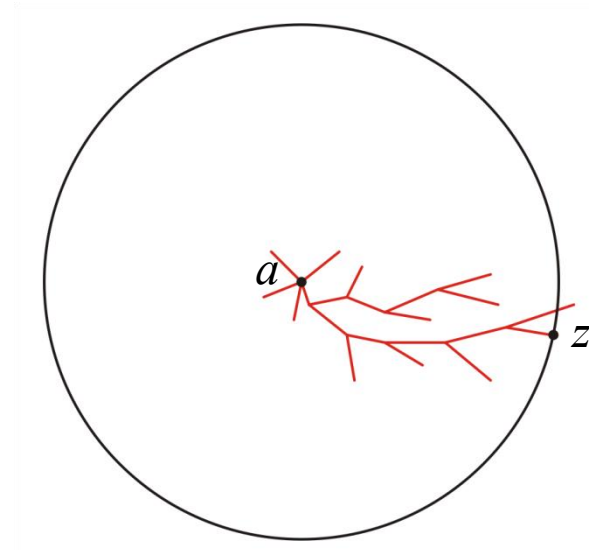
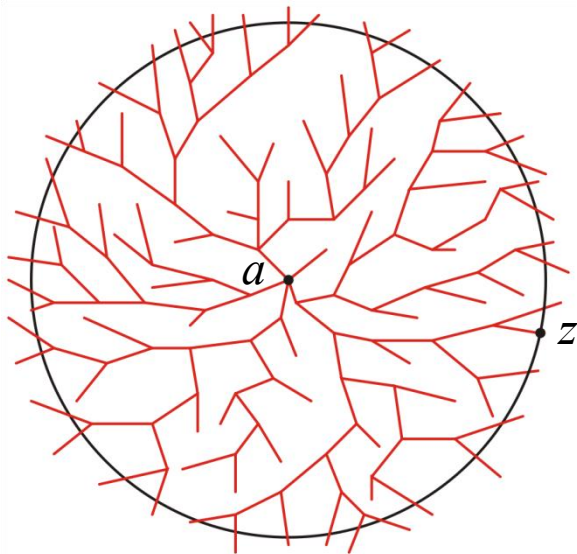
- Dijkstra's algorithm radiates out from the initial vertex
- The A\* search algorithm directs its search towards the destination



# Comparison with Dijkstra's Algorithm

Graphically, we can suggest the behaviour of the two algorithms as follows:

- Suppose we are moving from  $a$  to  $z$ :

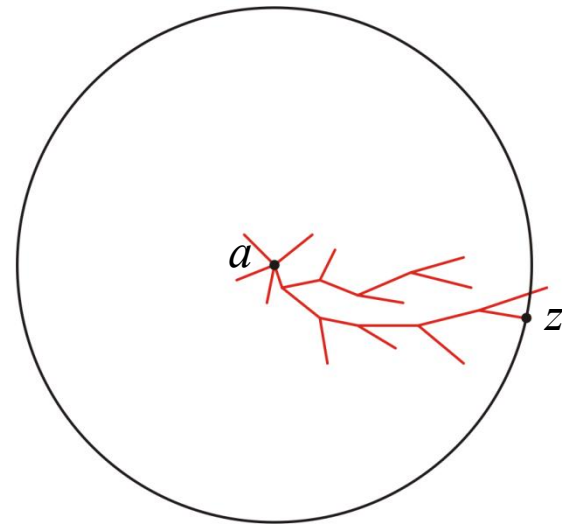
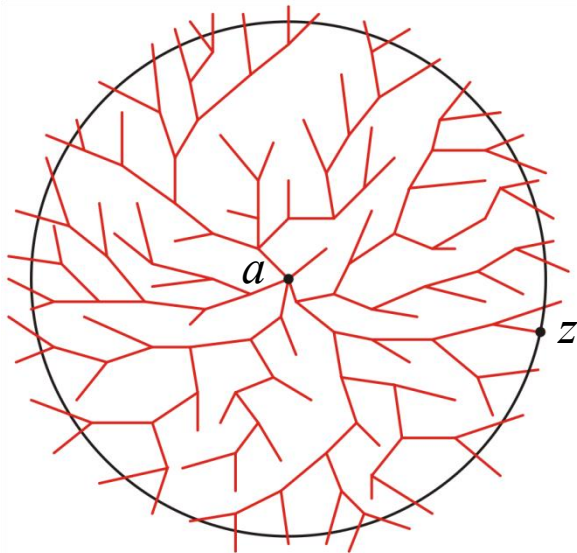


Representative search patterns for Dijkstra's and the A\* search algorithms

# Comparison with Dijkstra's Algorithm

Dijkstra's algorithm is the  $A^*$  search algorithm when  
the heuristic distance  $h(u, z) = 0$

- No vertex is better than any other vertex

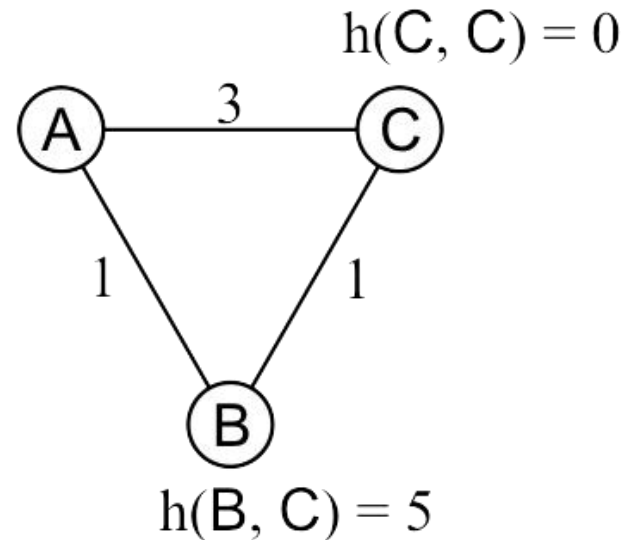


Representative search patterns for Dijkstra's and the  $A^*$  search algorithms

## Optimally Guarantees?

The A\* search algorithm will **not** always find the optimal path with a poor heuristic distance

- Find the shortest path from A to C:
  - B is enqueued with weight  $w(B) = 1 + 5 = 6$
  - C is enqueued with weight  $w(C) = 3 + 0 = 3$
- Therefore, C is dequeued next and as it is the destination, we are finished

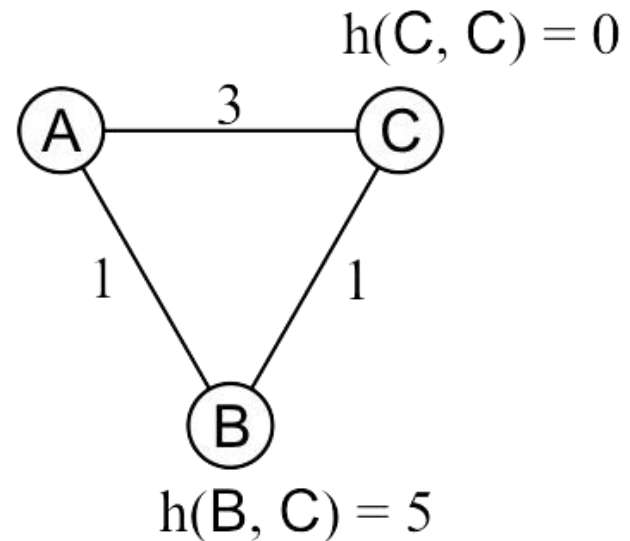


## Admissible Heuristics

This heuristic overestimates the actual distance from B to C

- The Euclidean distance doesn't suffer this problem:

*The path the crow flies is always shorter than the road the wolf runs*





## Admissible Heuristics

Admissible heuristics  $h$  must always be optimistic:

- Let  $d(u, v)$  represent the actual shortest distance from  $u$  to  $v$
- A heuristic  $h(u, v)$  is **admissible** if  $h(u, v) \leq d(u, v)$
- The heuristic is *optimistic* or a *lower bound* on the distance

Using the Euclidean distance between two points on a map is clearly an admissible heuristic

- The flight of the crow is shorter than the run of the wolf

A problem with fewer restrictions on the actions is called a relaxed problem

- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

**Theorem:**

If  $h(n)$  is admissible, A\* using TREE-SEARCH is optimal.

# Consistent Heuristics

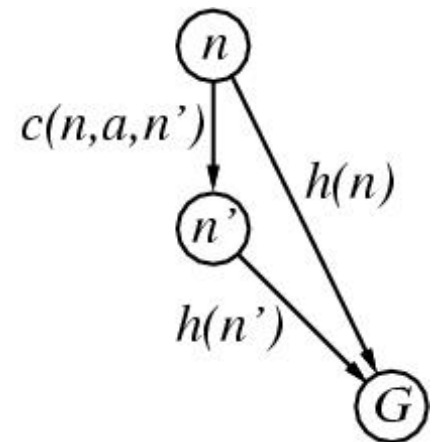
A heuristic is **consistent** if for every node  $n$ , every successor  $n'$  of  $n$  generated by any action  $a$ , we have

$$h(n) \leq c(n, a, n') + h(n')$$

If  $h$  is consistent, we have

$$\begin{aligned} w(n') &= d(n') + h(n') && \text{(by def.)} \\ &= d(n) + c(n, a, n') + h(n') && (d(n') = d(n) + c(n, a, n')) \\ &\geq d(n) + h(n) = w(n) && \text{(consistency)} \\ w(n') &\geq w(n) \end{aligned}$$

i.e.,  $w(n)$  is non-decreasing along any path.



**It's the triangle inequality !**

keeps all checked nodes  
in memory to avoid repeated states

**Theorem:**

If  $h(n)$  is consistent, A\* using GRAPH-SEARCH is optimal.

# Time Complexity

Exponential:  $O(b^d)$  where  $b$  is the branching factor (the average number of successors per state) and  $d$  is the depth of the solution

Can be shown to run in polynomial time if

$$|h(u, v) - d(u, v)| = O(\ln(d(u, v)))$$

where  $d(u, v)$  is the length of the actual shortest path<sup>1</sup>

- *i.e.*, doubling the length of the optimal solution only increases the error by a constant
- Not likely with a road map and the Euclidean distance
- E.g. when  $h(u, v) = d(u, v)$

<sup>1</sup>Pearl, Judea (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley.

## The $N$ Puzzle

Consider find the solution to the following puzzle

- Take a random permutation of 8 or 15 numbered tiles and a blank formed in a rigid square

	3	2
8	7	1
4	5	6

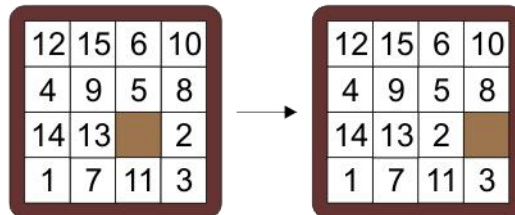
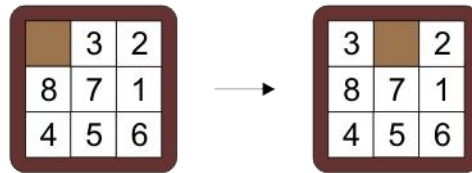
12	15	6	10
4	9	5	8
14	13		2
1	7	11	3



## A\* Search Algorithm

# The $N$ Puzzle

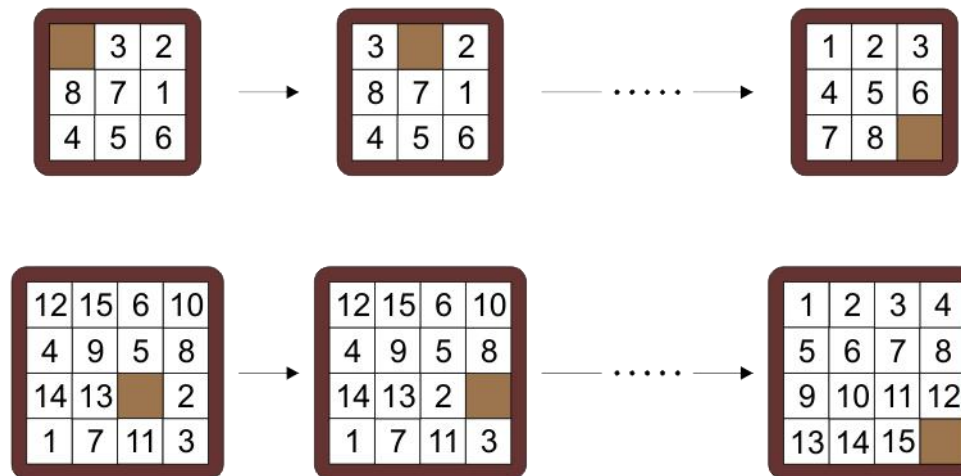
You are allowed to move a tile adjacent to the blank into the location of the blank



## A\* Search Algorithm

# The $N$ Puzzle

The objective is to find a minimum number of moves which will transform the tiles into a standard solution

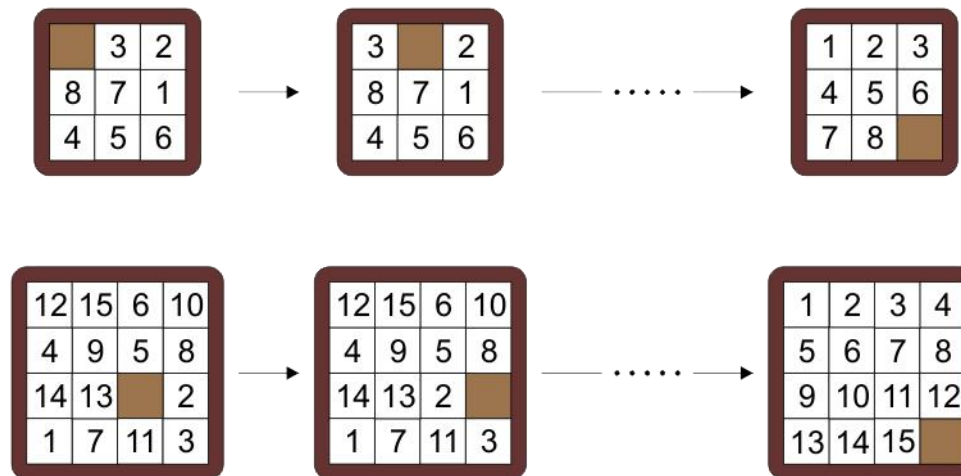


## A\* Search Algorithm

# The $N$ Puzzle

There are  $9!$  and  $16!$  initial permutations

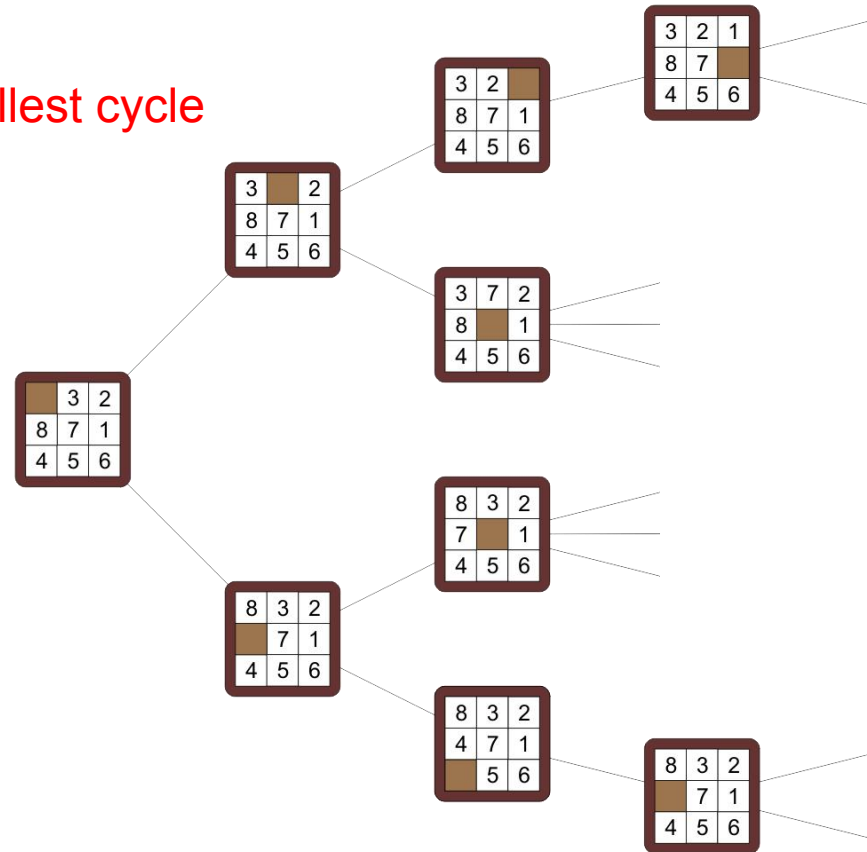
- Half of these, through a sequence of moves, can be transformed into the desired solutions



## The $N$ Puzzle

Each solution defines a vertex in a graph with edges denoting allowable moves

- It is not a tree, but the smallest cycle has a length of 12
  - E.g., cycle 8, 7, and 3



## The $N$ Puzzle

The graph of solvable eight puzzles has:

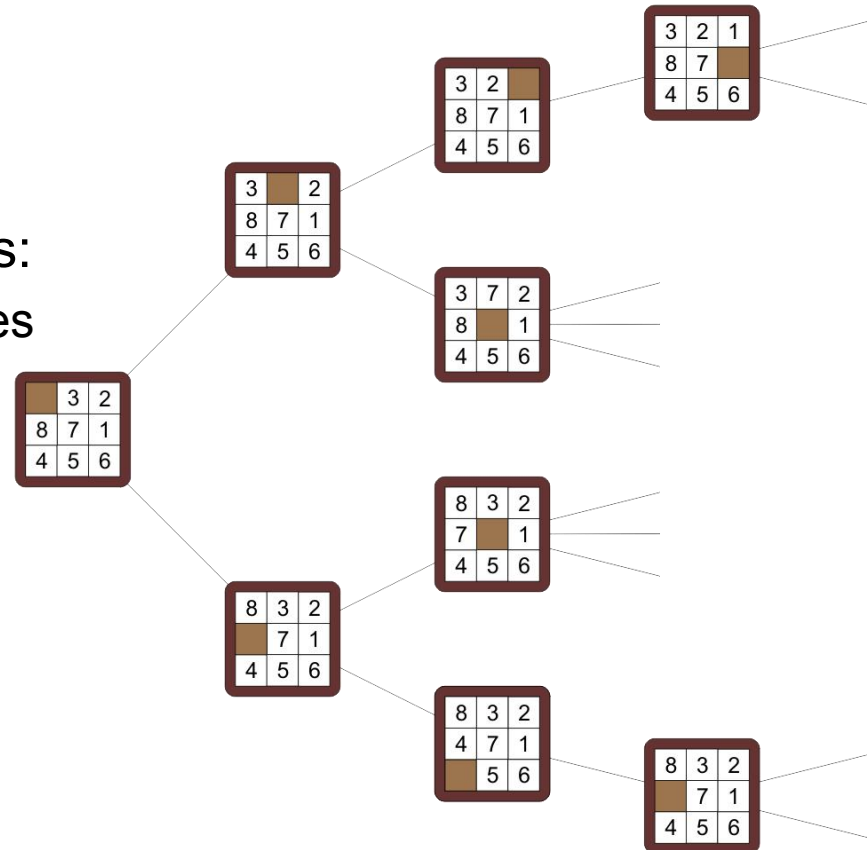
- 181 440 vertices
- 241 920 edges

The fifteen puzzle graph has:

- 10 461 394 944 000 vertices
- 15 692 092 416 000 edges

In general and  
in the limit:

- $(N + 1)!/2$  vertices
- $(N + 1)!$  edges





## A\* Search Algorithm


# The $N$ Puzzle

Finding a solution requires one to find the shortest path

- All edges have weight 1

Dijkstra's algorithm is painfully slow

- This puzzle requires that 179680 vertices be searched



5	1	4
2		7
3	6	8

To use the A\* search, we need a heuristic lower bound on the distance

# The $N$ Puzzle

We will consider three distances which we will use with the A\* search:

- The discrete distance
  - If two objects are equal, the distance is 0, otherwise it is 1
- The Hamming distance
  - The number of tiles (including the blank) which are in an incorrect location
- The Manhattan distance
  - The sum of the minimum number of moves required to put a tile in its correct location

## A\* Search Algorithm

# The $N$ Puzzle

For example, consider this permutation:

- It does not equal the solution, so the discrete distance is 1

5	2	7
8	4	
1	3	6

- 8 out of 9 tiles/blanks are in the incorrect location; therefore the Hamming distance is 8


5	2	7
8	4	
1	3	6

- It requires  $2 + 0 + 4 + 2 + 1 + 1 + 2 + 3 + 1 = 16$  moves to move each tile into the correct location, therefore the Manhattan distance is 16

5	2	7
8	4	
1	3	6

## The $N$ Puzzle

As discussed before, the discrete distance does not improve the situation: it is equivalent to Dijkstra's algorithm



5	1	4
2		7
3	6	8

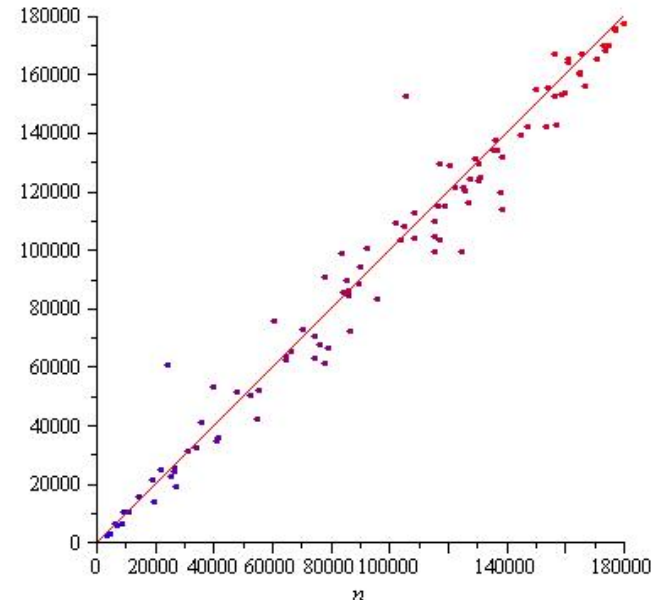
Using the same permutation as before:

- The Hamming distance isn't much better:
  - It reduces the vertices searched from 179 680 to 178 005
  - It is only useful when we are very close to the actual solution
- The Manhattan distance, however, allows the A\* search to find the minimal path with only 6453 vertices searched

# The $N$ Puzzle

How much better is it to use the Manhattan distance over the Hamming or discrete?

- With the Hamming distance, there is only a small change
- For 100 random puzzles, we have a plot the number of vertices visited using the discrete distance versus the number of vertices visited using the Hamming distance
- The line is the identity function
- The colour, blue to red, indicates the length of the solution; from 13 to 28

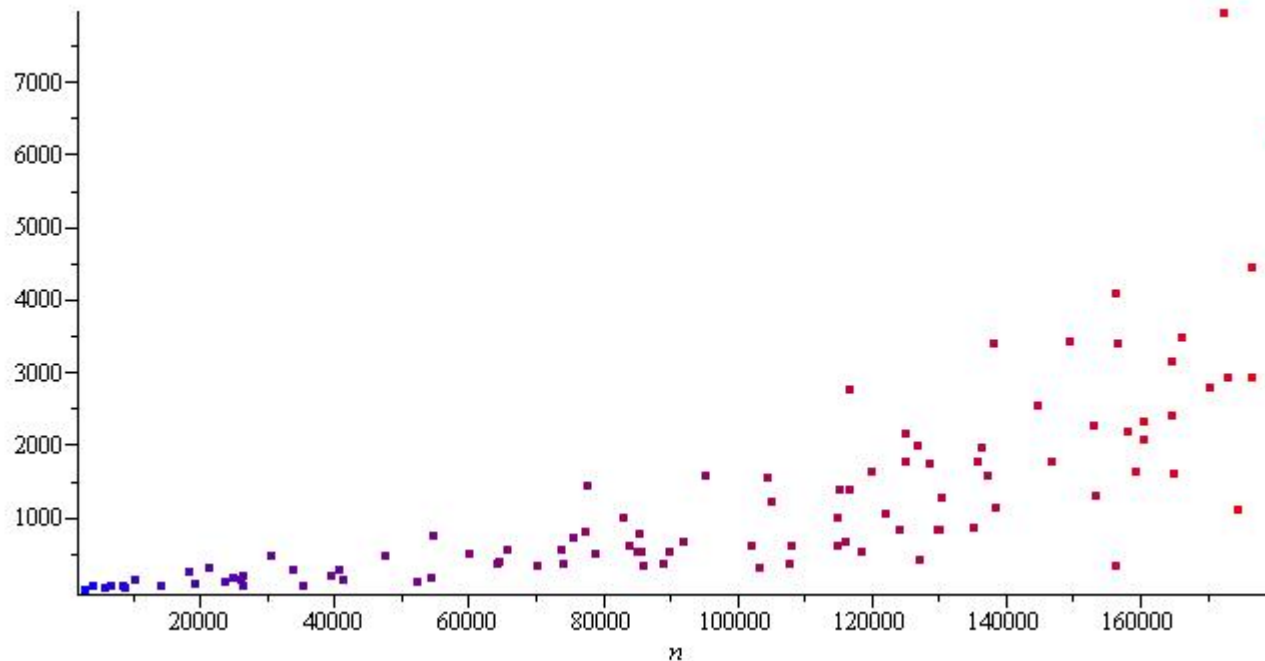




# The $N$ Puzzle

However, comparing the number of vertices visited when using the Manhattan distance, there is a significant reduction by a factor of almost 100

- A more significant improvement with shorter paths



# Summary

This topic has presented the A\* search algorithm

- Assumes a hypothetical lower bound on the length of the path to the destination
- Requires an appropriate heuristic
  - Useful for Euclidean spaces (vector spaces with a norm)
- Faster than Dijkstra's algorithm in many cases
  - Directs searches towards the solution

# Backtracking (Optional)

Suppose a solution can be made as a result of a series of choices

- Each choice forms a partial solution
- These choices may form either a tree or DAG
  - Separate branches may recombine or diverge

# Backtracking

With Dijkstra's algorithm, we keep track of all current best paths

- There are at most  $|V| - 1$  paths we could extend at any one time
- These can be tracked with a relatively small table

Suppose we cannot evaluate the relative fitness of solutions

- There may just be too many to record efficiently
- Are we left with a brute-force search?

Suppose there are just too many partial solutions at any one time to keep track of

- At any point in time in a game of chess or Go (围棋), there are a plethora of moves, each valid, but the usefulness of each will vary

# Sudoku

In the first case, consider the game Sudoku:

- The search space is  $9^{53}$

5		4					6	
	6		3			1		
	9	2	5	4				
6		8			1			
			8		4			
			2			6		1
				9	5	8	1	
		1			2		3	
	8					2		7



# Sudoku

At least for the first entry in the first square, only 1, 3, 7 fit

5		4						6
	6		3				1	
	9	2	5	4				
6		8			1			
			8		4			
			2			6		1
				9	5	8	1	
		1			2		3	
	8					2		7

5	1	4						6
	6		3				1	
	9	2	5	4				
6		8			1			
			8		4			
			2			6		1
				9	5	8	1	
		1			2		3	
	8					2		7

5	3	4						6
	6		3				1	
	9	2	5	4				
6		8			1			
			8		4			
			2			6		1
				9	5	8	1	
		1			2		3	
	8					2		7

5	7	4						6
	6		3				1	
	9	2	5	4				
6		8			1			
			8		4			
			2			6		1
				9	5	8	1	
		1			2		3	
	8					2		7

# Sudoku

If the first entry has a 1, the 2<sup>nd</sup> entry in that square could be 7 or 8

5	1	4					6
	6		3			1	
	9	2	5	4			
6		8			1		
			8		4		
			2			6	1
				9	5	8	1
		1			2		3
	8					2	7



5	1	4					6
7	6		3			1	
	9	2	5	4			
6		8			1		
			8		4		
			2			6	1
				9	5	8	1
		1			2		3
	8					2	7

5	1	4					6
8	6		3			1	
	9	2	5	4			
6		8			1		
			8		4		
			2			6	1
				9	5	8	1
		1			2		3
	8					2	7

# Sudoku

If the first entry has a 3, the 2<sup>nd</sup> entry in that square could be 7 or 8

5	3	4					6
	6		3			1	
	9	2	5	4			
6		8			1		
			8		4		
			2			6	1
				9	5	8	1
		1			2		3
	8					2	7



5	3	4					6
7	6		3			1	
	9	2	5	4			
6		8			1		
			8		4		
			2			6	1
				9	5	8	1
		1			2		3
	8					2	7

5	3	4					6
8	6		3			1	
	9	2	5	4			
6		8			1		
			8		4		
			2			6	1
				9	5	8	1
		1			2		3
	8					2	7

# Sudoku

If the first entry has a 7, the 2<sup>nd</sup> entry in that square could be 8

5	7	4					6
	6		3			1	
	9	2	5	4			
6		8			1		
			8		4		
			2			6	1
				9	5	8	1
		1			2		3
	8					2	7



5	7	4					6
8	6		3			1	
	9	2	5	4			
6		8			1		
			8		4		
			2			6	1
				9	5	8	1
		1			2		3
	8					2	7

# Sudoku

In the next child, there are no options available for the next entry

5	4					6		
6		3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	1	4				6		
6		3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	3	4				6		
6		3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	7	4				6		
6		3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	1	4				6		
7	6	3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	1	4				6		
8	6	3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	3	4				6		
7	6	3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	3	4				6		
8	6	3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	7	4				6		
1	6	3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	7	4				6		
8	6	3				1		
9	2	5	4					
6		8		1				
		8	4					
		2		6		1		
			9	5	8	1		
1			2		3			
8				2		7		

5	1	4				6		
7	6	×	3			1		
9	2	5	4					
6		8			1			
			8		4			
			2		6		1	
				9	5	8	1	
	1			2		3		
8					2		7	

Any candidate solution built from this partial is infeasible

– We can ignore this branch



# Sudoku

Three other branches lead to similar dead ends

5	4				6			
6		3			1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
1			2		3			
8				2	7			

5	1	4			6			
6		3			1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
1			2		3			
8				2	7			

5	3	4			6			
6		3			1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
1			2		3			
8				2	7			

5	7	4			6			
6		3			1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
1			2		3			
8				2	7			

5	1	4			6			
7	6	3			1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
1			2		3			
8				2	7			

5	1	4			6			
8	6	3			1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
1			2		3			
8				2	7			

5	3	4			6			
7	6	3			1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
1			2		3			
8				2	7			

5	3	4			6			
8	6	3			1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
1			2		3			
8				2	7			

5	7	4			6			
8	6	3			1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
1			2		3			
8				2	7			

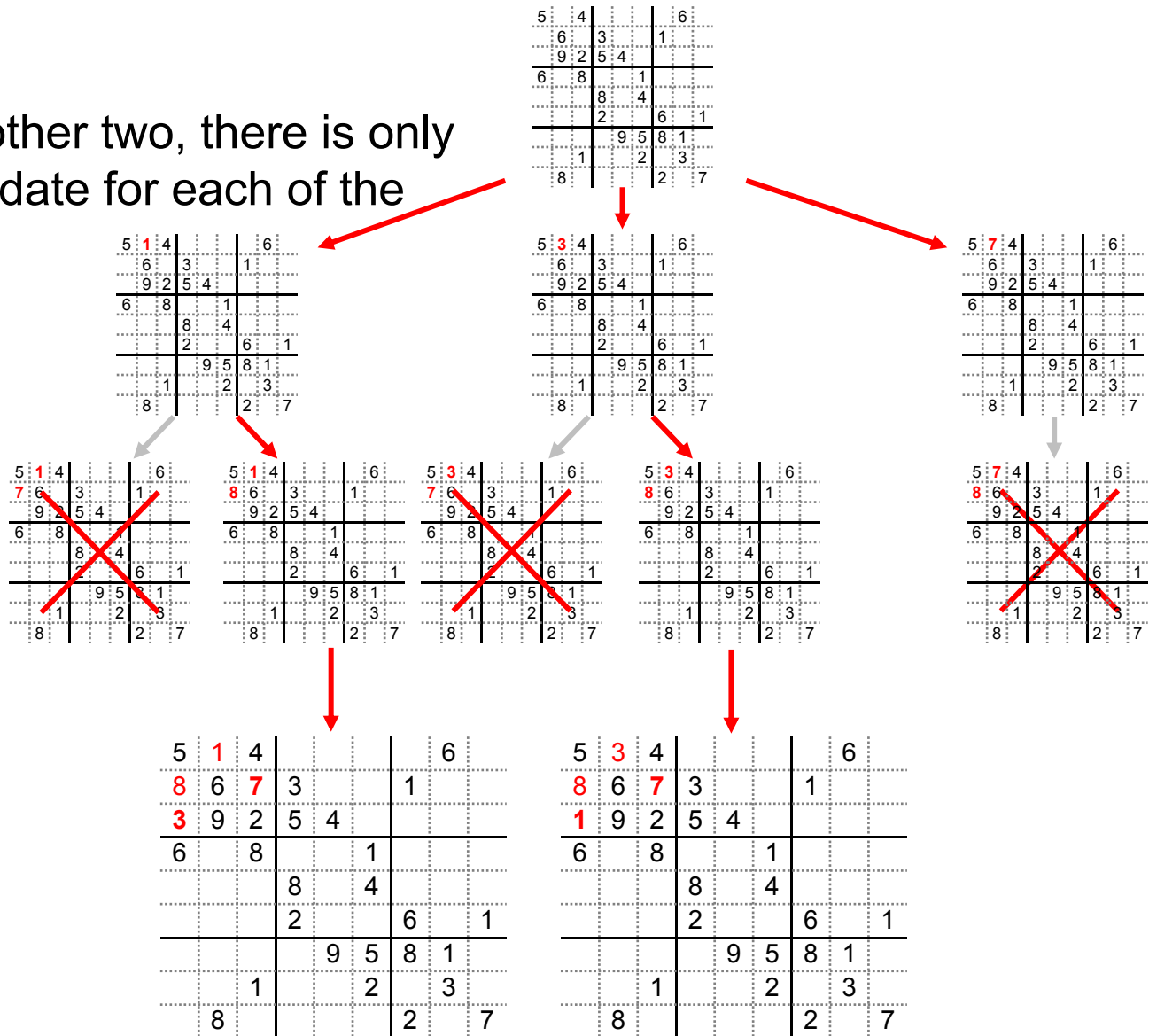
5	1	4			6			
7	6	×	3		1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
	1		2		3			
8				2	7			

5	3	4			6			
7	6	×	3		1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
	1		2		3			
8				2	7			

5	7	4			6			
8	6	×	3		1			
9	2	5	4					
6		8		1				
		8	4					
		2		6	1			
			9	5	8	1		
	1		2		3			
8				2	7			

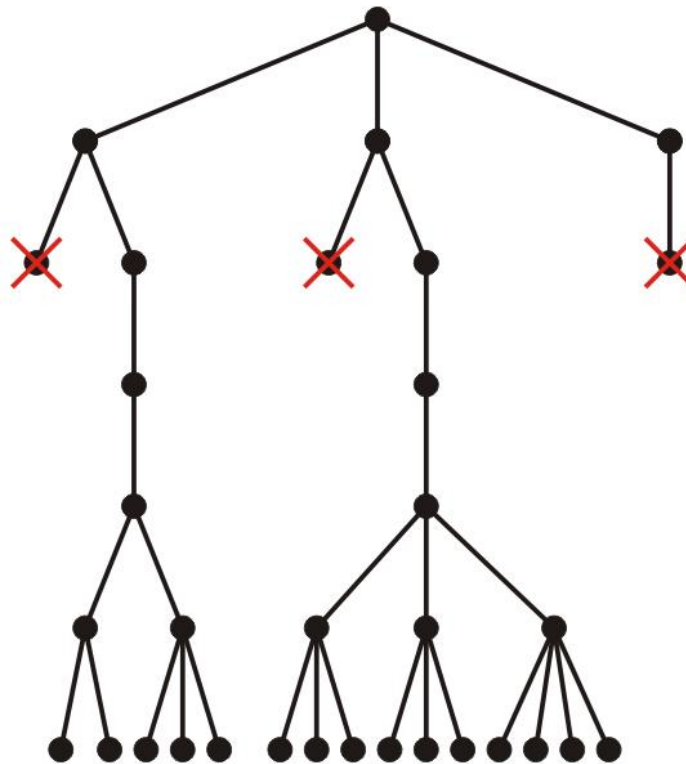
# Sudoku

With the other two, there is only one candidate for each of the last two entries



# Sudoku

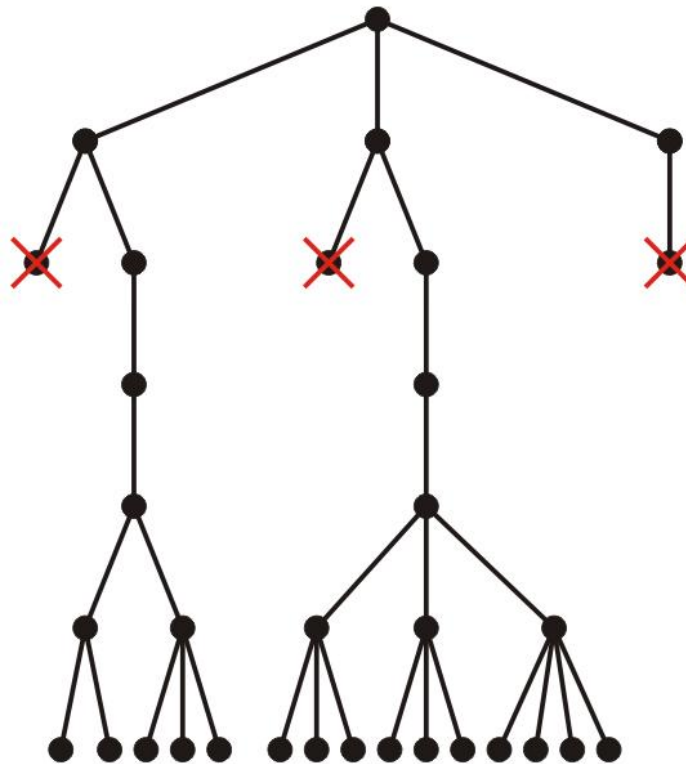
It may seem that this is a reasonably straight-forward method; however, the decision tree continues to branch quick once we start filling the second square



# Sudoku

A binary tree of this height would have around  $2^{54} - 1$  nodes

- Fortunately, as we get deeper into the tree, more get cut



# Implementation

Our straight-forward implementation takes a  $9 \times 9$  matrix

- Default entries are values from 1 to 9, empty cells are 0
- Two helper functions:
  - `bool next_location( int[9][9], int &i, int &j )`
    - Finds the next empty location returning false if none is found
  - `bool is_valid( int[9][9], int i, int j, int value )`
    - Checks if there are any conflicts created if `matrix[i][j]` is assigned value
- The backtracing function:
  - Finds the next unoccupied cell
  - For each value from 1 to 9, it checks if it is valid to insert it there
    - If so, backtracking is called recursively on the matrix with that entry set

The main function creates the initial matrix and calls backtrack

# Implementation

```
// Find the next empty location in 'matrix'
// If one is found, assign 'i' and 'j' the indexes of that entry
// Otherwise, return false
// - In this case, the values of 'i' and 'j' are undefined
bool next_location( int matrix[9][9], int &i, int &j ) {
    for ( int i1 = 0; i1 < 3; ++i1 ) {
        for ( int j1 = 0; j1 < 3; ++j1 ) {
            for ( int i2 = 0; i2 < 3; ++i2 ) {
                for ( int j2 = 0; j2 < 3; ++j2 ) {
                    i = 3*i1 + i2;
                    j = 3*j1 + j2;

                    // return 'true' if we find an
                    // unoccupied entry
                    if ( matrix[i][j] == 0 )
                        return true;
                }
            }
        }
    }

    return false; // all the entries are occupied
}
```

```
// If 'value' already appears in
// - the row 'm'
// - the column 'n'
// - the 3x3 square of entries it (m, n) appears in
// return false, otherwise return true
bool is_valid( int matrix[9][9], int m, int n, int value ) {
    // Check if 'value' already appears in either a row or column
    for ( int i = 0; i < 9; ++i ) {
        if ( (matrix[m][i] == value) || ( matrix[i][n] == value ) )
            return false;
    }

    // Check if 'value' already appears in either a row or column
    int ioff = 3*(m/3);
    int joff = 3*(n/3);

    for ( int i = 0; i < 3; ++i ) {
        for ( int j = 0; j < 3; ++j ) {
            if ( matrix[ioff + i][joff + j] == value )
                return false; // 'value' already in the 3x3 square
        }
    }

    return true; // 'value' could be added
}
```

# Implementation

```
bool backtrack( int matrix[9][9] ) {
    int i, j;

    // If the matrix is full, we are done
    if ( !next_location( matrix, i, j ) ) {
        return true;
    }

    for ( int value = 1; value <= 9; ++value ) {
        if ( is_valid( matrix, i, j, value ) ) {
            // Assume this entry is part of the
            // solution--recursively call backtrack
            matrix[i][j] = value;

            // If we found a solution, return
            // otherwise, reset the entry to 0
            if ( backtrack( matrix ) ) {
                return true;
            } else {
                matrix[i][j] = 0;
            }
        }
    }

    // No solution found--reset the entry to 0
    return false;
}
```

# Implementation

```
int main() {
    int matrix[9][9] = {
        {5, 0, 4, 0, 0, 0, 0, 6, 0},
        {0, 6, 0, 3, 0, 0, 1, 0, 0},
        {0, 9, 2, 5, 4, 0, 0, 0, 0},
        {6, 0, 8, 0, 0, 1, 0, 0, 0},
        {0, 0, 0, 8, 0, 4, 0, 0, 0},
        {0, 0, 0, 2, 0, 0, 6, 0, 1},
        {0, 0, 0, 0, 9, 5, 8, 1, 0},
        {0, 0, 1, 0, 0, 2, 0, 3, 0},
        {0, 8, 0, 0, 0, 0, 2, 0, 7}
    };

    // If found, print out the resulting matrix
    if ( backtrack( matrix ) ) {
        for ( int i = 0; i < 9; ++i ) {
            for ( int j = 0; j < 9; ++j ) {
                std::cout << matrix[i][j] << " ";
            }

            std::cout << std::endl;
        }
    }

    return 0;
}
```



# Implementation

In this case, the traversal:

- Recursively calls backtrack 874 times
  - The last one determines that there are no unoccupied entries
- Checks if a placement is valid 7658 times

5	3	4	1	7	8	9	6	2
8	6	7	3	2	9	1	4	5
1	9	2	5	4	6	3	7	8
6	7	8	9	3	1	5	2	4
2	1	5	8	6	4	7	9	3
3	4	9	2	5	7	6	8	1
4	2	3	7	9	5	8	1	6
7	5	1	6	8	2	4	3	9
9	8	6	4	1	3	2	5	7

# Backtracking

This should give us an idea, however:

- Perform a traversal
- Do not continue traversing if a current node indicates all descendants are infeasible solutions

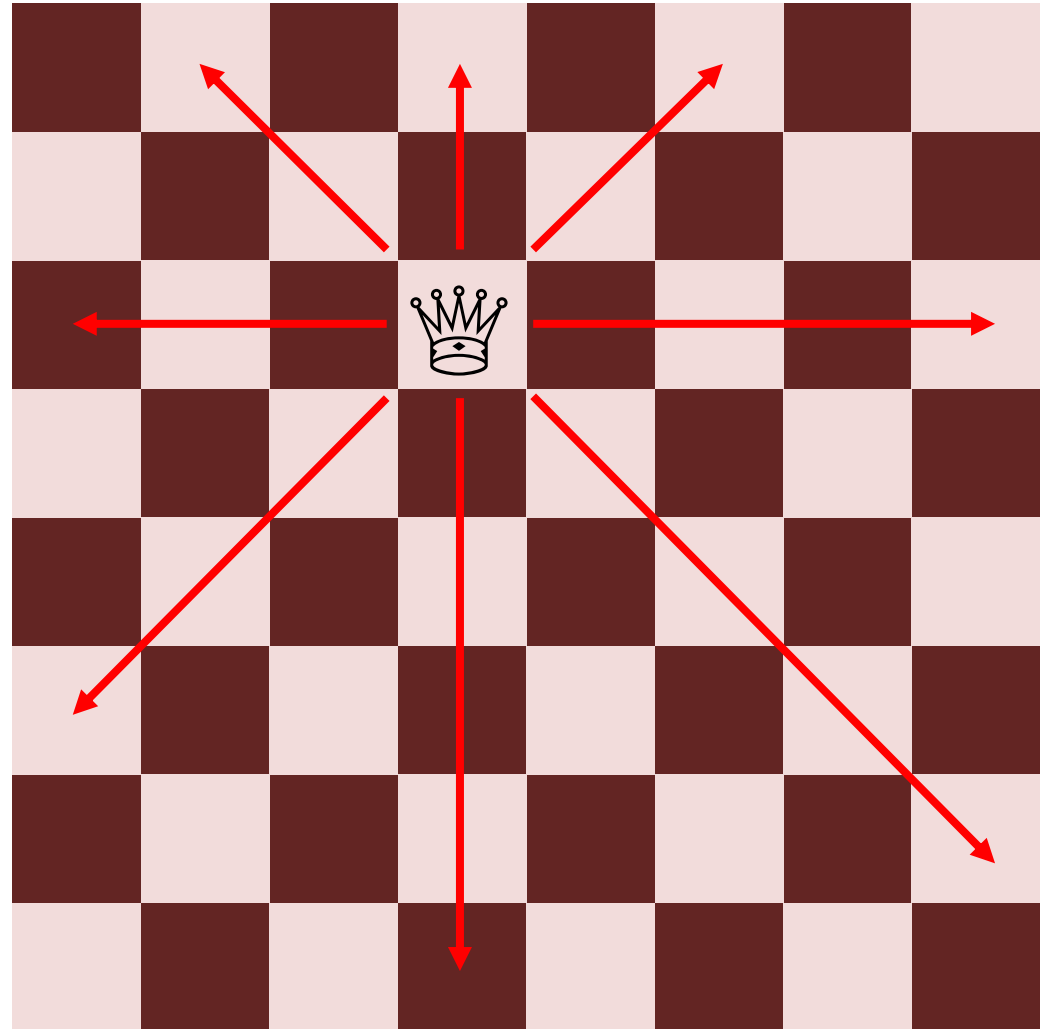
# Classical applications

Classic applications of this algorithm technique include:

- Eight queens puzzle
- Knight's tour
- Logic programming languages
- Crossword puzzles

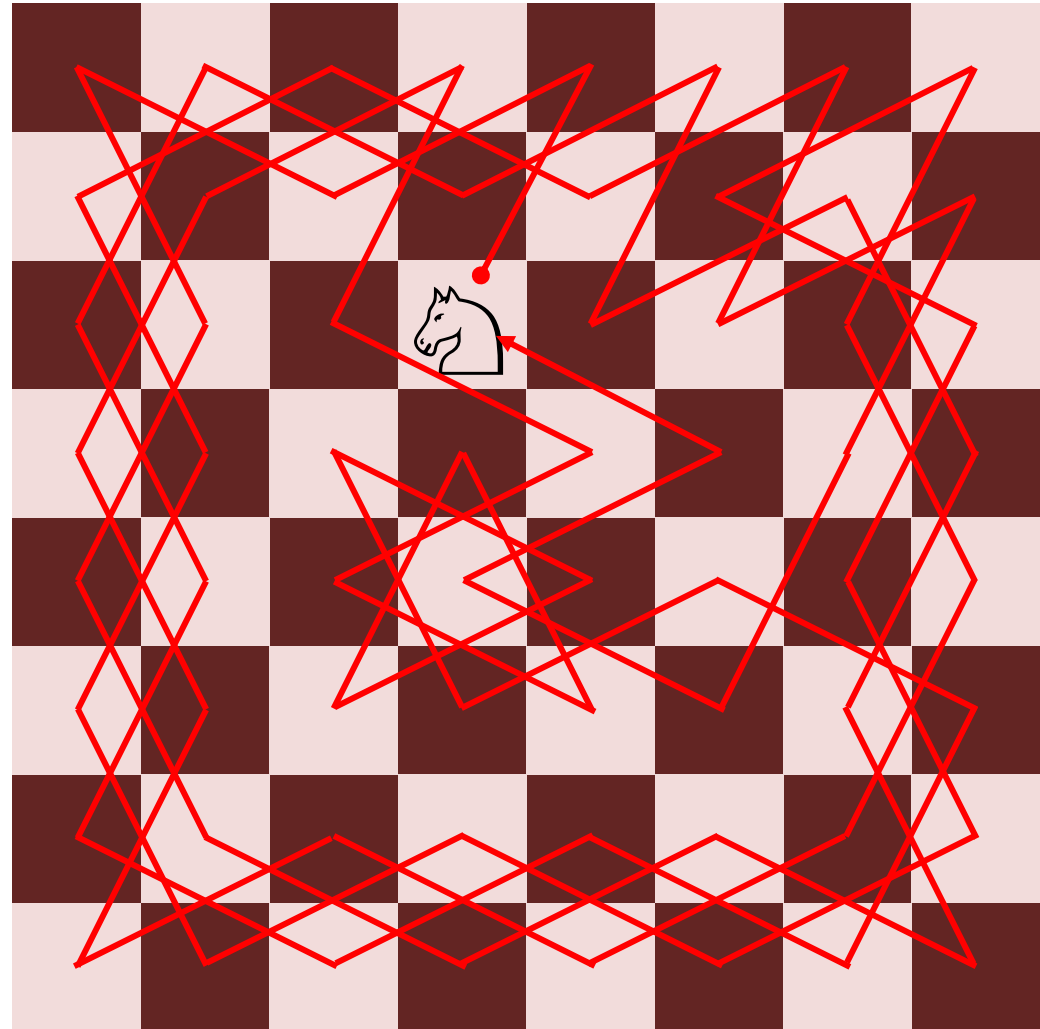
# Eight queens puzzle

Arrange eight queens on a chess board so that no queen can take another



# Knight's tour

Have a knight visit all the squares of a chess board either as a path or a cycle



# Logic programming languages

Consider the Prolog programming language where we state facts:

female(juliana).	parent(willem, catharina).	parent(beatrix, willem).
male(bernhard).	parent(maxima, catharina).	parent(claus, willem).
female(beatrix).	parent(willem, alexia).	parent(beatrix, friso).
male(claus).	parent(maxima, alexia).	parent(claus, friso).
male(firso).	parent(willem, ariane).	parent(beatrix, constantijn).
female(mabel).	parent(maxima, ariane).	parent(claus, constantijn).
male(constantijn).		
female(laurentien).	parent(firso, luana).	parent(juliana, beatrix).
	parent(mabel, luana).	parent(bernhard, beatrix).
female(catharina).	parent(firso, zaria).	parent(juliana, irene).
female(alexia).	parent(mabel, zaria).	parent(bernhard, irene).
female(ariane).		parent(juliana, margriet).
	parent(constantijn, eloise).	parent(bernhard, margriet).
female(luana).	parent(laurentien, eloise).	parent(juliana, christina).
female(zaria).	parent(constantijn, claus_ii).	parent(bernhard, christina).
	parent(laurentien, claus_ii).	
female(eloise).	parent(constantijn, leonore).	spouses(willem, maxima).
male(claus_ii).	parent(laurentien, leonore).	spouses(firso, mabel).
female(leonore).		spouses(constantijn, laurentien).
		spouses(beatrix, claus).
		spouses(juliana, bernhard).

# Logic programming languages

You can now define relationships between individuals

```
% Relationships
```

```
mother(M, X) :- parent(M, X), female(M).
```

```
father(F, X) :- parent(F, X), male(F).
```

```
sister(S, X) :- sibling(S, X), female(S), \+ (S = X).
```

```
brother(B, X) :- sibling(B, X), male(B), \+ (S = X).
```

```
grandparent(G, X) :- parent(G, P), parent(P, X).
```

```
% Symmetric
```

```
spouses(X, Y) :- spouses(Y, X);
```

```
sibling(X, Y) :- parent(P, X), parent(P, Y), \+ (X = Y).
```

```
cousin(X, Y) :- parent(A, X), parent(B, Y), sibling(A, B).
```

```
% Antisymmetric
```

```
uncle(U, X) :- male(U), sibling(U, Y), parent(Y, X).
```

```
uncle(U, X) :- male(U), spouse(U, Z), sibling(Z, Y), parent(Y, X).
```

```
aunt(A, X) :- female(A), parent(Y, X), sibling(A, Y).
```

```
aunt(A, X) :- female(A), spouse(A, Z), sibling(Z, Y), parent(Y, X).
```

# Logic programming languages

Given these relationships, you can now make queries:

cousin(zaria, alexia).

uncle(constantijn, alexia).

aunt(laurentien, alexia).

Backtracking can be used to determine whether the above relationships hold given the stated facts



# Parsing

Question: how do we define a programming language?

- Why are any of the following never valid?

`a + < b`

`c[3)`

`d?e;`

`54f`

`""";`

`g$ = "Hello world!";`

- Programming languages are defined by *grammars*
- The C++ programming language grammar is available here:  
<http://www.nongnu.org/hcb/>

# Parsing and grammars

Consider just the conditional statements from the pre-processor

- Square brackets is used to indicate something is optional

```
<group> ::=      <group-part>
               <group> <group-part>
<group-part> ::= <if-section>
                 <control-line>
<if-section> ::= <if-group> [<elif-groups>] [<else-group>] <endif-line>
<if-group> ::=    #if <constant-expression> <new-line> [<group>]
                 #ifdef <identifier> <new-line> [<group>]
                 #ifndef <identifier> <new-line> [<group>]
<elif-groups> ::= <elif-group>
                 <elif-groups> <elif-group>
<elif-group> ::= #elif <constant-expression> <new-line> [<group>]
<else-group> ::= #else <new-line> [<group>]
<endif-line> ::= #endif <new-line>
```

# Parsing and grammars

We cannot work with a full grammar for C++

- Instead, we will consider some vastly oversimplified versions

`<digit>` ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

`<pos>` ::= `<digit>` | `<digit><pos>`

`<int>` ::= `<pos>` | `+<pos>` | `-<pos>`

`<std>` ::= `<int>.<pos>` | `.<pos>` | `-.<pos>` | `+.<pos>`

`<sci>` ::= `<int>e<int>` | `<int>E<int>` | `<std>e<int>` | `<std>E<int>`

`<float>` ::= `<std>` | `<sci>`

`<nondigit>` ::= A | B | ... | Z | a | b | ... | z | \_

`<identifier>` ::= `<nondigit>` | `<identifier><nondigit>`  
| `<identifier><digit>`

`<declaration>` ::= `int <identifier> = <identifier>;`

| `int <identifier> = <int>;`

| `double <identifier> = <int>`

| `double <identifier> = <float>`

# Parsing and grammars

As you can see, each of these defines a tree

- Some of these trees are recursively defined

$\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\langle \text{pos} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{pos} \rangle$

$\langle \text{int} \rangle ::= \langle \text{pos} \rangle \mid +\langle \text{pos} \rangle \mid -\langle \text{pos} \rangle$

$\langle \text{std} \rangle ::= \langle \text{int} \rangle . \langle \text{pos} \rangle \mid . \langle \text{pos} \rangle \mid - . \langle \text{pos} \rangle \mid + . \langle \text{pos} \rangle$

$\langle \text{sci} \rangle ::= \langle \text{int} \rangle e \langle \text{int} \rangle \mid \langle \text{int} \rangle E \langle \text{int} \rangle \mid \langle \text{std} \rangle e \langle \text{int} \rangle \mid \langle \text{std} \rangle E \langle \text{int} \rangle$

$\langle \text{float} \rangle ::= \langle \text{std} \rangle \mid \langle \text{sci} \rangle$

$\langle \text{nondigit} \rangle ::= A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid \_$

$\langle \text{identifier} \rangle ::= \langle \text{nondigit} \rangle \mid \langle \text{identifier} \rangle \langle \text{nondigit} \rangle$   
 $\quad \mid \langle \text{identifier} \rangle \langle \text{digit} \rangle$

$\langle \text{declaration} \rangle ::= \text{int } \langle \text{identifier} \rangle = \langle \text{identifier} \rangle;$

$\quad \mid \text{int } \langle \text{identifier} \rangle = \langle \text{int} \rangle;$

$\quad \mid \text{double } \langle \text{identifier} \rangle = \langle \text{int} \rangle$

$\quad \mid \text{double } \langle \text{identifier} \rangle = \langle \text{float} \rangle$

# Parsing and grammars

Suppose we are trying to parse the string

```
int var0 = 3532700;  
double var1 = 3.5e-27;  
double var2 = -44.203;
```

What if we're parsing garbage?

```
double var0 = 3.5g-27;  
int 1var = 44203;  
double var2 = 0.0  
double var3 = 1.0;
```

# Backjumping

In some cases, the following may occur:

- Determining that one leaf does not constitute a solution may simultaneously determine that the corresponding sub-tree does not contain a solution, either
- In this case, return to the closest ancestor such that it has not yet been determined that all descendants have been ruled out
- This is described as *backjumping*

# Searching a maze

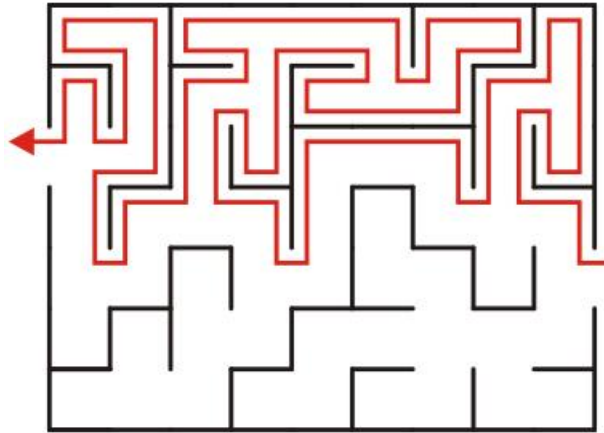
In trying to find your way through a maze, one simple rule works quite nicely:

- The right-hand rule:

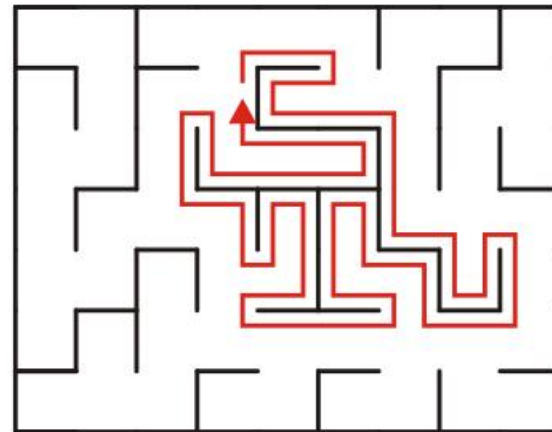
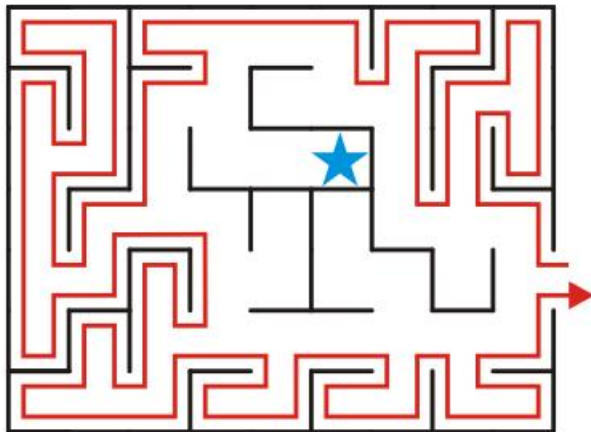
Touch a wall with your right hand, and continue forward always keeping your right hand touching a wall until you get out.

# Searching a maze

This works well in finding your way through a maze



It doesn't work if you're trying to get into the maze or out of a maze

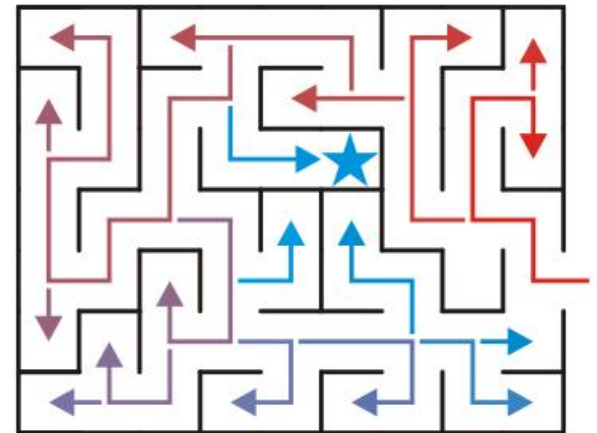




# Searching a maze

Consider the following algorithm:

- If the goal is reached, we are done
- If there is only one move into a previously unoccupied cell, move to it and flag it as occupied
- If there is more than one move into a previously unoccupied cell, push that position onto a stack, and take the right-most available path
- If there are no more moves, check the stack:
  - If the stack is empty, there is no path to the goal
  - If the stack is not empty, pop to top position and continue the algorithm from that point



# Searching a maze

In each example, the solution is always found

- In the normal maze, less work is required due to backjumping

