CS101 Algorithms and Data Structures

A* search and Backtracking

Outline

A星

In this topic, we will look at the A* search algorithm:

- 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

假设！

Background

探索性的

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

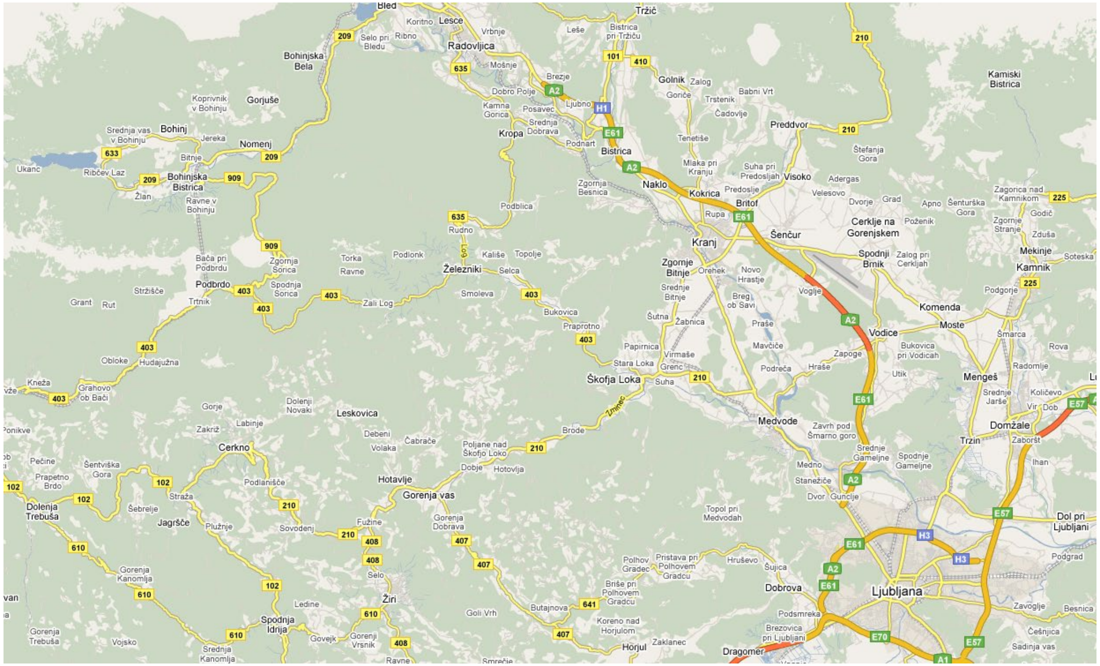
  “as the crow flies”

- use this to guide our search for a path

  “as the fox runs”

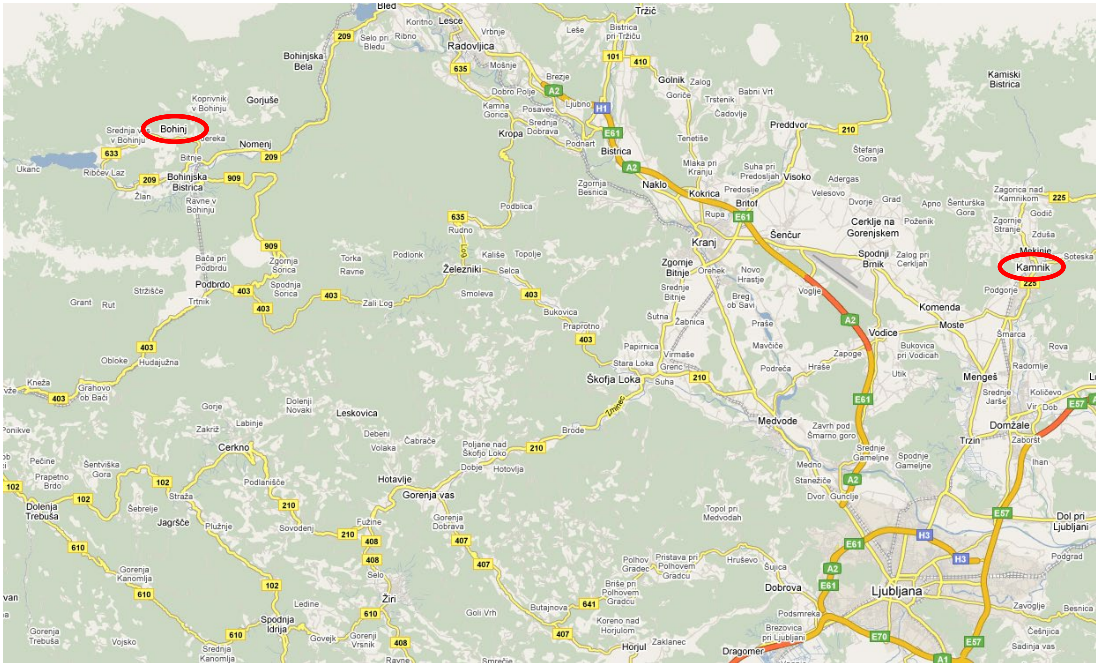
Idea

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(Kamnik, Bohinj) = 53 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(Kamnik, Kranj) = 28 km

Idea

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

Idea

Thus, the weight of the path up to Kranj is

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

Idea

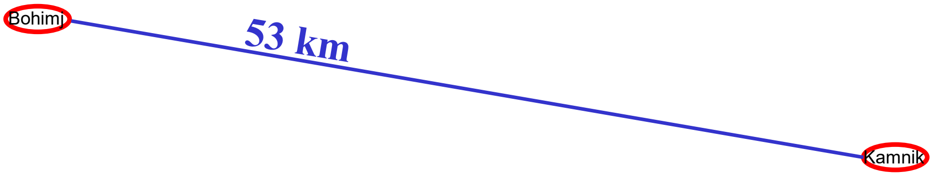
Any path extending this given path to Bohimj must be at least $60\ \mathrm{km}$

Idea

The value w(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:

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



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 $\mathrm{w}(v)$ 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)= infinity* for all $v \neq a$
- For each vertex *v*, *h(v, z)* is the heuristic distance from *v* to *z*

估值 起到当前 当前点到目标估值

$$f(n) = g(n) + h(n)$$

(各种距离)

$|x_1-x_2|+|y_1-y_2|$

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 $\mathrm{w}(v) = \mathrm{d}(a, u) + \mathrm{d}(u, v) + \mathrm{h}(v, z)$ 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

正比

weight prority

越大 越大

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

pop u

neighbors v of u

visited / unvisited

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

Comparison of Priorities

Suppose we have a path with

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

Comparison of Priorities

We can extend this path to Moste:

$$w(\mathit{Moste}) \quad = \color{red}{4\text{ km}} + \color{red}{5\text{ km}} + \color{blue}{48\text{ km}} = 57\text{ km}$$

Comparison of Priorities

We can also extend this path to Menges:

$$w(\mathit{Menges}) \quad = \color{red}{4\text{ km}} + \color{red}{4\text{ km}} + \color{blue}{51\text{ km}} = 59\text{ km}$$

Comparison of Priorities

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

A*

graph

1. 首先需要创建两个集合，一个存储 待访问 的节点（nodeLists），一个存储 已经访问过 的节点(visitedNodeLists)

2. 添加 `起点（2,1）` 到 `nodeLists` 列表，并且计算该点的 `预估值`。

Weight 距起点. 估值

这里就需要用到上面讲到的预估函数了，
$f(n) = g(n) + h(n)$
我们可以很清楚的看到，起点到当前节点的距离为0，即：$g(n) = 0$。
而当前顶点（2,1）到目标顶点（2,5）的距离为4，即：$h(n) = 4$。

这里的h值可以采用 **曼哈顿距离公式** 计算得来：
$h = |x_1 - x_2| + |y_1 - y_2|$

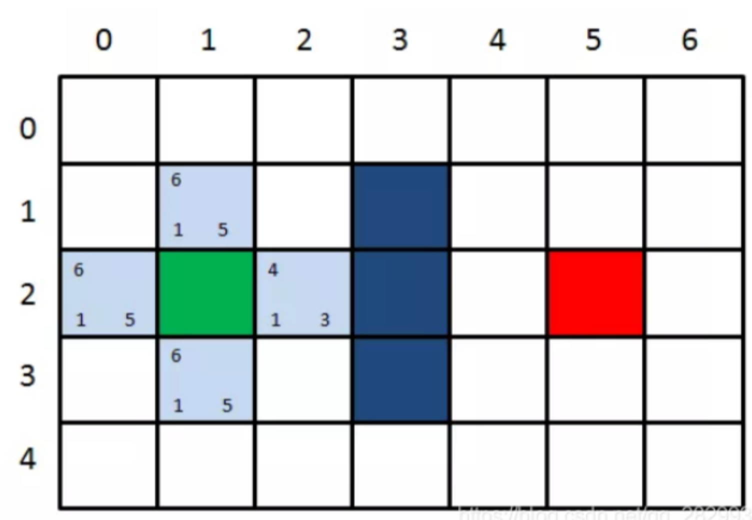
这里需要注意的是：h 值是在 `不考虑障碍` 的情况下计算的

towards 接近

3. 查找nodeLists 里预估值最小的节点，作为当前访问的节点，并且从nodeLists 删除 该节点。（放进 visited list）

4. 获取 当前节点 的 邻居节点，计算出他们的预估值 并且添加到nodeLists列表中。

如图：




```
function startSearch(){
    //把起点添加到nodeLists
    nodeLists.push(startNode);
    //循环
    while (nodeLists.length > 0) {
        //查找预估值最小的节点
        let currentNode = findMinNode();
        //获取并且添加邻居节点到待访问列表
        findAndAddNeighborNode(currentNode);
        //把当前节点添加到已访问列表
        visitedNodeLists.push(currentNode);
        //判断是否是目标节点
        if (this.isTarget(currentNode)) {
            return currentNode
        }
    }
    return null
}
```

这里只考虑上下左右4个方位，并且分别计算了他们的各个预估值，
右下角：$h(n)$值
左下角：$g(n)$值
左上角：$f(n)$值

除了计算预估值，我们还需要把**当前节点**作为**每个邻居节点**的父节点，以便为了后面确定最终的路线。

这里需要注意的是：我们在添加到 **待访问列表**之前需要处理一下边界问题，并且判断一下是否是障碍物，如果是就不需要添加到列表中。

5. 把当前节点添加到visitedNodeLists中，代表已经访问过了。

6. 重复以上步骤 3 - 5 ，直到找到目标节点位置为止。

7. 循环输出 最终节点的父节点，就是我们需要的路径了。

Comparison with Dijkstra's Algorithm

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

- Dijkstra would chose Menges next:

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

Difference:

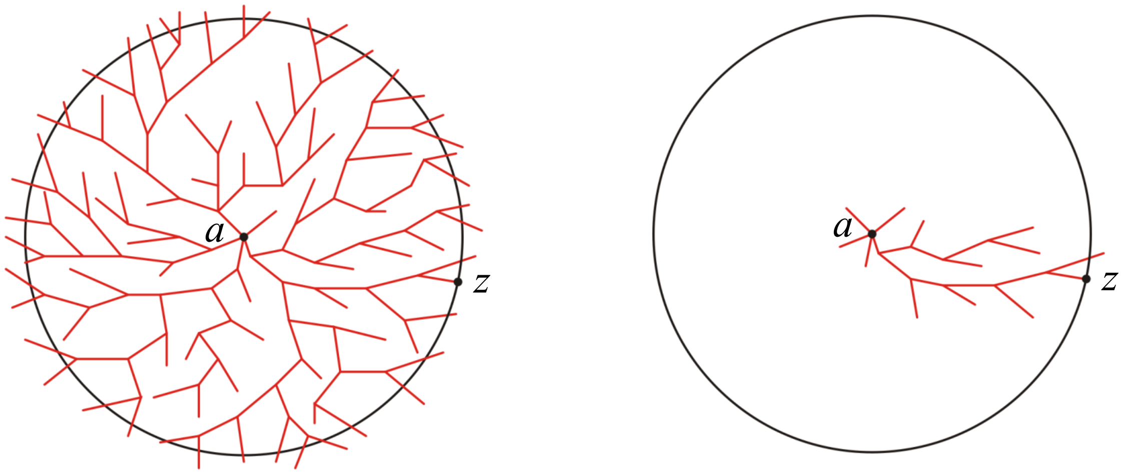
辐射

- 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:

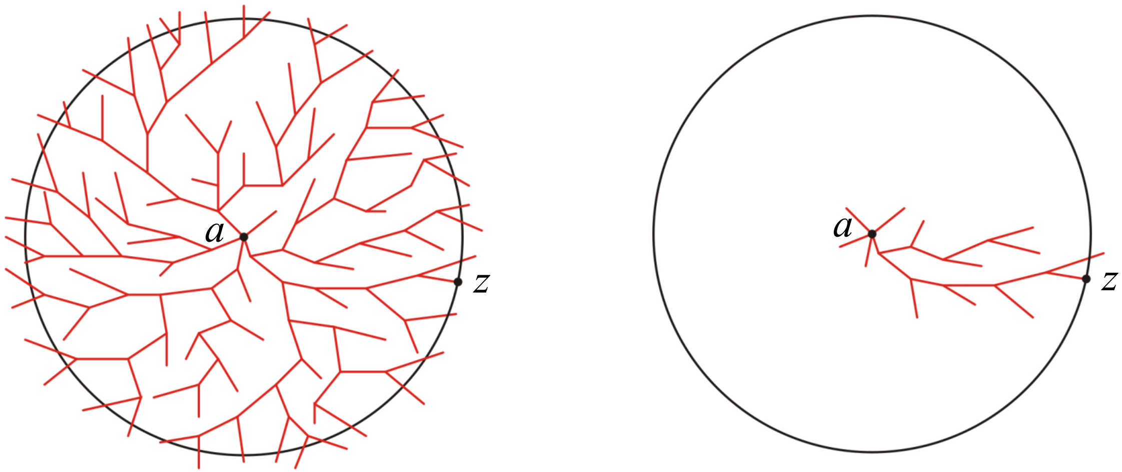


Representative search patterns for Dijkstra's and the A* search algorithms

Comparison with Dijkstra's Algorithm

Dijkstra's algorithm is the A* search algorithm when using the discrete distance $\mathrm{h}(u,v) = \begin{cases} 0 & u = v \\ 1 & u \neq v \end{cases}$

– No vertex is better than any other vertex



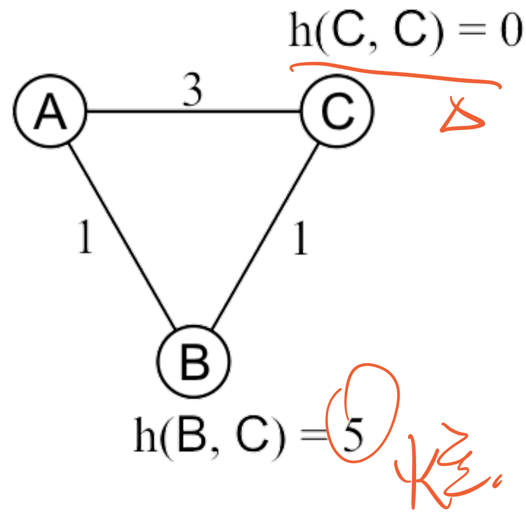
Representative search patterns for Dijkstra's and the A* search algorithms

Optimally Guarantees?

烂估值 Not

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



Reference: Andrew Moore, Carnegie Mellon University

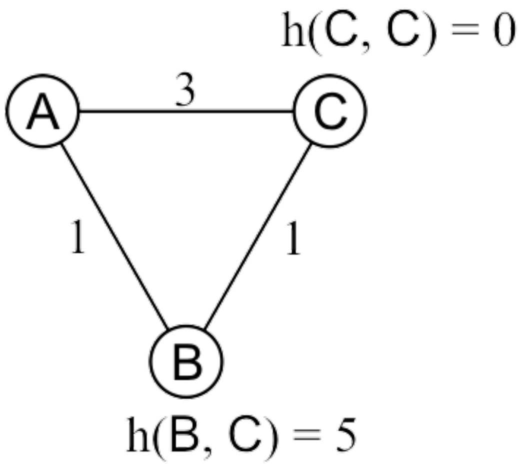
Admissible Heuristics

可容许，可采纳

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



Reference: Andrew Moore, Carnegie Mellon University

Admissible Heuristics

Admissible heuristics h must always be optimistic:

- Let $\mathrm{d}(u, v)$ represent the actual shortest distance from u to v
- A heuristic $\mathrm{h}(u, v)$ is *admissible* if $\mathrm{h}(u, v) \leq \mathrm{d}(u, v)$
- 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 → non-decreasing

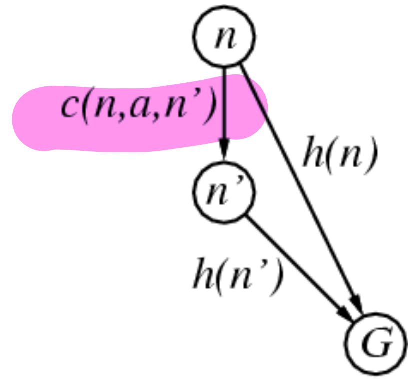
Consistent Heuristics

→ Graph

继承者 / 后

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

w(n')	= d(n') + h(n')	(by def.)
	= d(n) + c(n,a,n') + h(n')	(d(n')=d(n)+c(n.a.n'))
	≥ d(n) + h(n) = w(n)	(consistency)
w(n')	≥ w(n)	

It's the triangle inequality !

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

keeps all checked nodes in memory to avoid repeated states

tree-search

Theorem:

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

$h(n) \le \underbrace{c(n, a, n') + h(n')}$

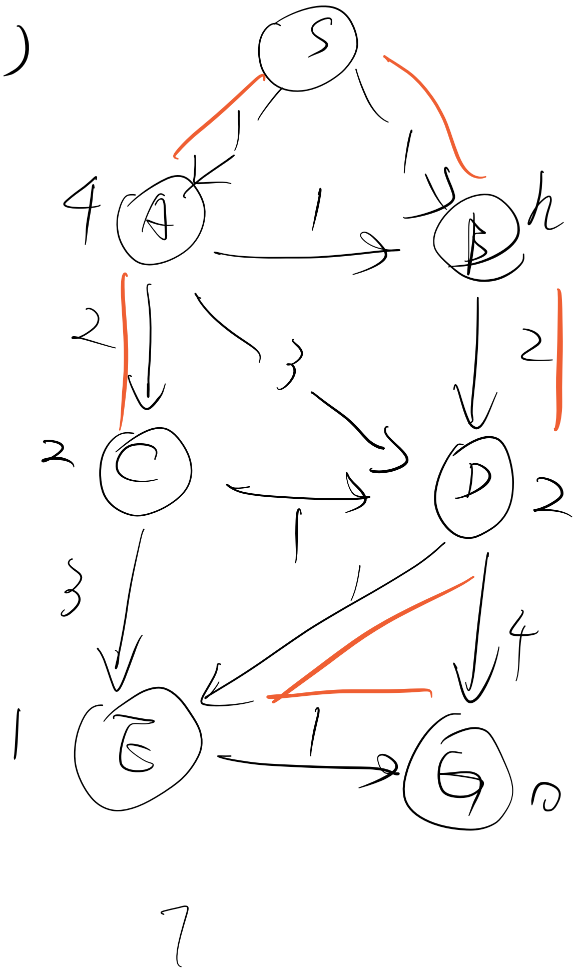
$h \le 2+2$.

$4 \le 1+h$

$\underbrace{5 \le 1+h}_{\text{start}}$

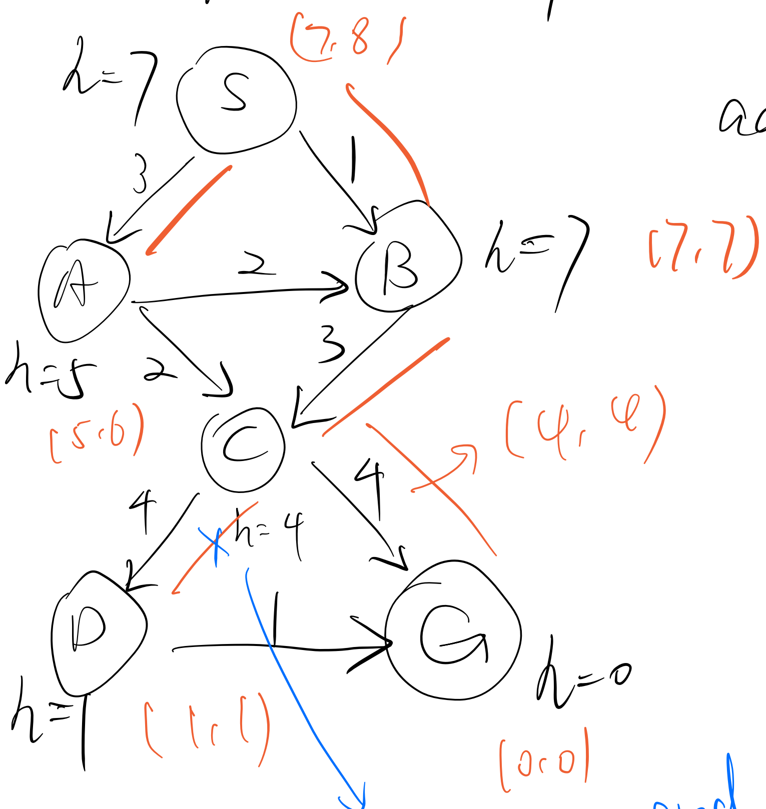
$h = 4$

$S \to A \to B \to \cancel{C}$
$\to D \to E \to G$



ad: $h \le d$. ✓

consis: 强了, ok



G 未 expand, D 不要了

Time

Time Complexity

$O(b^d)$

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

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

where $d(u, v)$ is the length of the actual shortest path[1]

- *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 $h(u, v) = d(u, v)$

[1] Pearl, Judea (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley.

ad > consist admi consist

(c) Prove that if a heuristic is consistent, it must be admissible.

We can prove that consistency implies admissibility through induction.

Recall that consistency is defined such that $h(n) \leq c(n, n+1) + h(n+1)$.
Base Case: We begin by considering the $n-1$th node in any path where n denotes the goal state.

$$h(n-1) \leq c(n-1, n) + h(n) \tag{1}$$

Because n is the goal state, by definition, $h(n) = h^*(n)$. Therefore, we can rewrite the above as

$$h(n-1) \leq c(n-1, n) + h^*(n)$$

and given that $c(n-1, n) + h^*(n) = h^*(n-1)$, we can see:

$$h(n-1) \leq h^*(n-1)$$

which is the definition of admissibility!

Inductive Step: To see if this is always the case, we consider the $n-2$nd node in any of the paths we considered above (e.g. where there is precisely one node between it and the goal state). The cost to get from this node to the goal state can be written as

$$h(n-2) \leq c(n-2, n-1) + h(n-1)$$

From our base case above, we know that

$$h(n-2) \leq c(n-2, n-1) + h(n-1) \leq c(n-2, n-1) + h^*(n-1)$$

$$h(n-2) \leq c(n-2, n-1) + h^*(n-1)$$

And again, we know that $c(n-2, n-1) + h^*(n-1) = h^*(n-2)$, so we can see:

$$h(n-2) \leq h^*(n-2)$$

By the inductive hypothesis, this holds for all nodes, proving that consistency does imply admissibility!

The N Puzzle

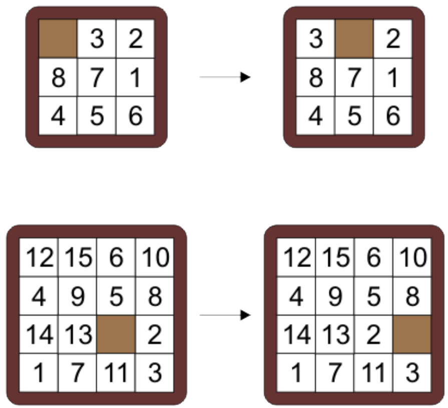
Consider find the solution to the following puzzle

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

Reference: Andrew Moore, Carnegie Mellon University

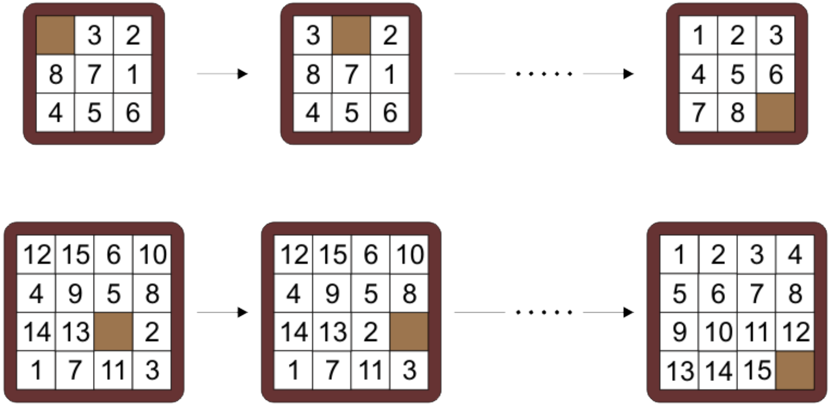
The N Puzzle

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



The N Puzzle

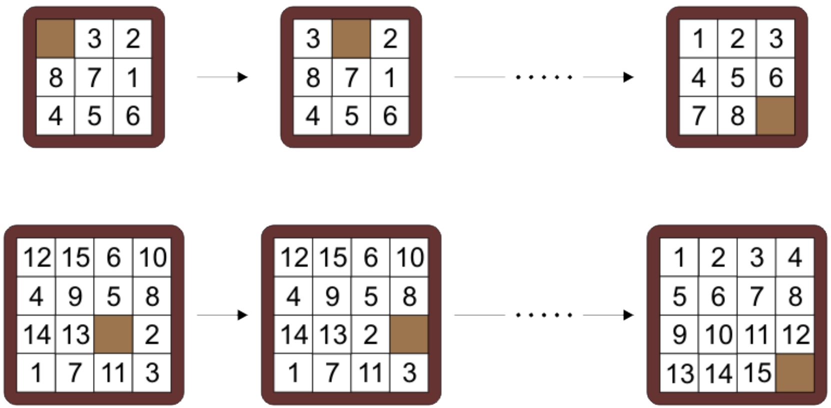
The objective is to find a minimum number of moves which will transform the tiles into a standard solution



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

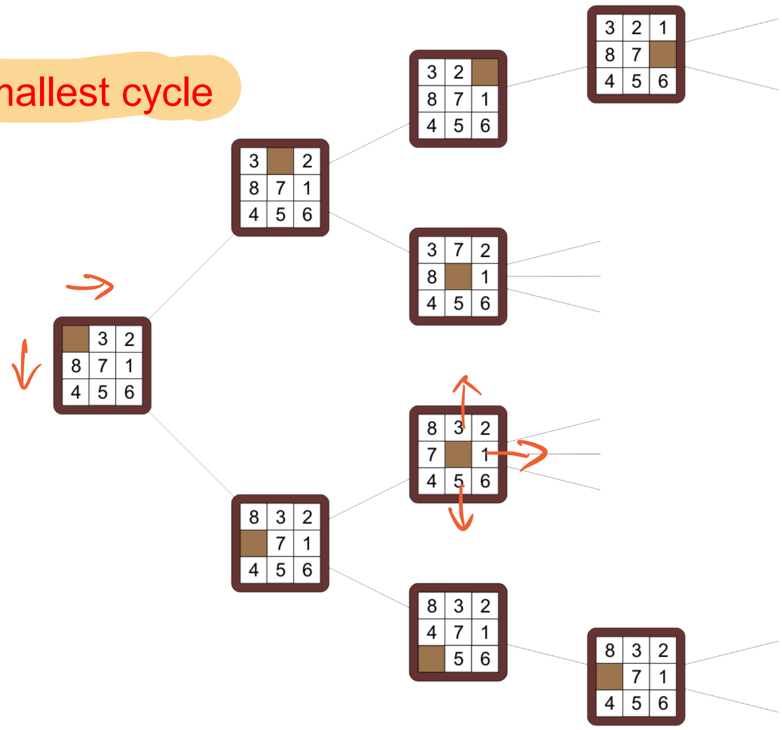


The N Puzzle

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

not tree



The N Puzzle

The graph of solvable eight puzzles has:

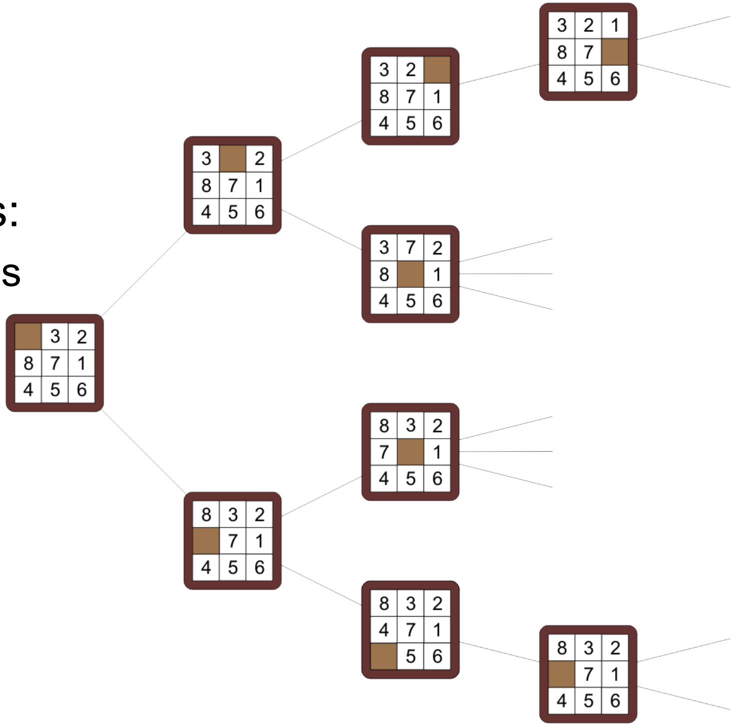
- 181 440 vertices
- 241 920 edges

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



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

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

found

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

0, 1

现在与目标的差距

moves consumption

eg.

1011101

1001001

ham(=2) 2个错误

The N Puzzle

123
406
78

For example, consider this permutation:

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

1

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

8

- 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

16

(去视障碍)

The N Puzzle

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

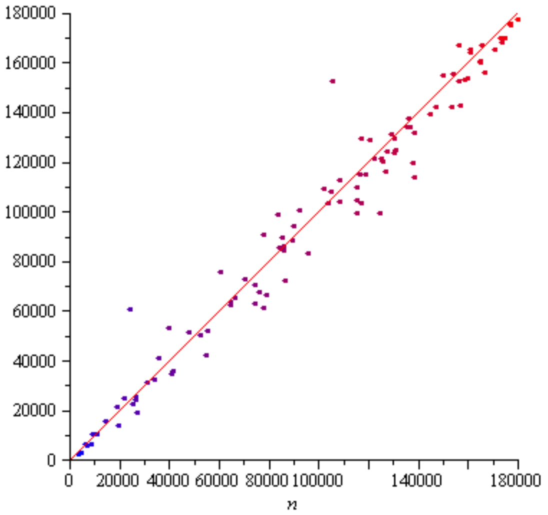
Using the same permutation as before:

- 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?

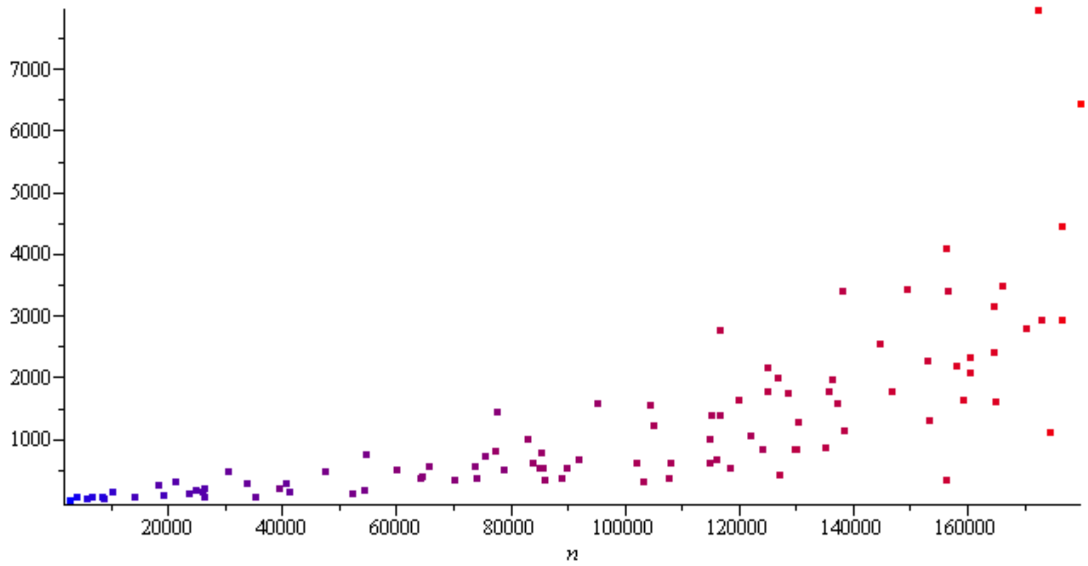
– With the Hamming distance, there is only a small change
– For 100 random puzzles, we have a plot the number of vertices visited using the discrete distance versus the number of vertices visited using the Hamming distance
– The line is the identity function
– The colour, blue to red, indicates the length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

http://www.sudokuoftheday.com/

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 2nd 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 2nd 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

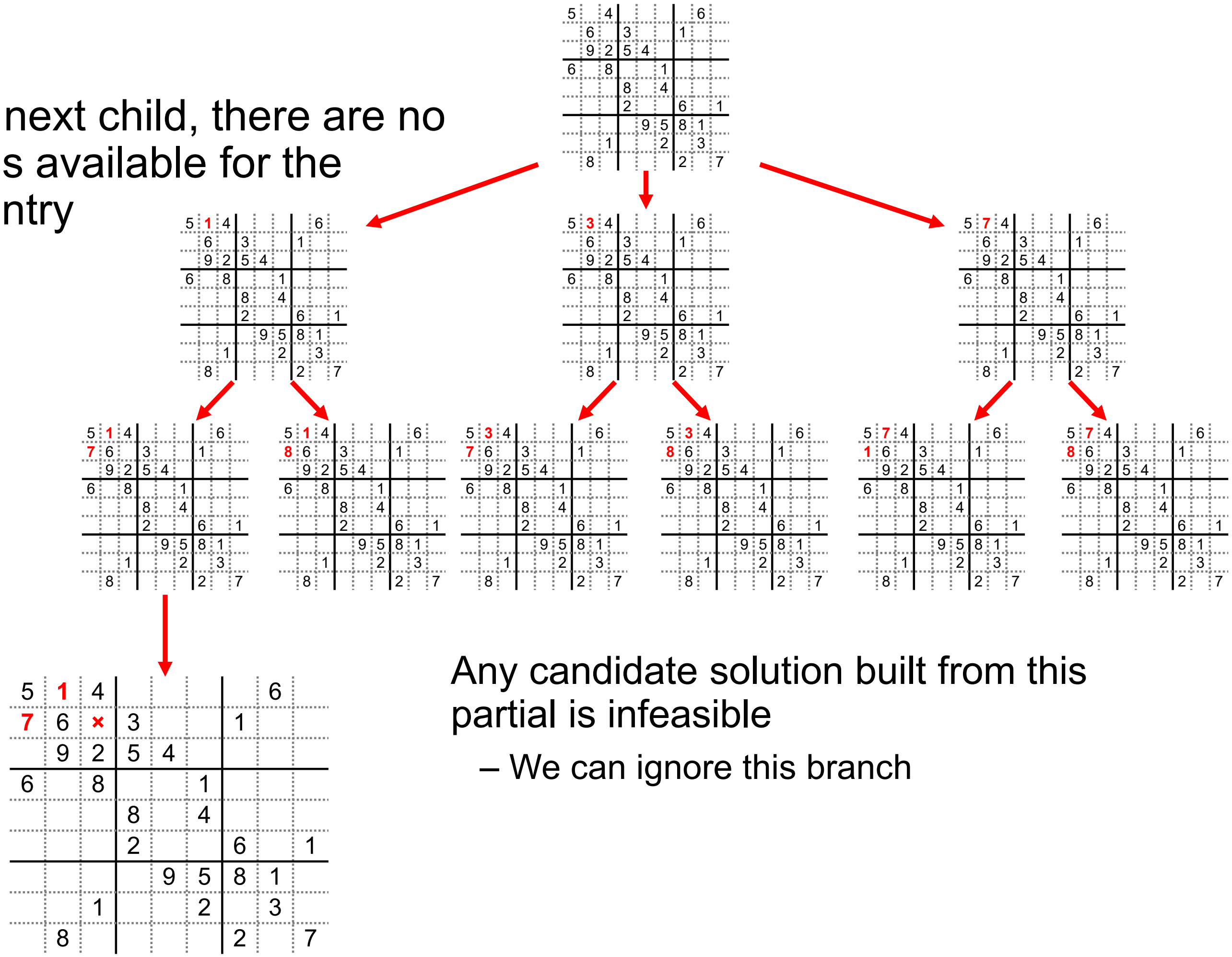
If the first entry has a 7, the 2nd entry in that square could be 8

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

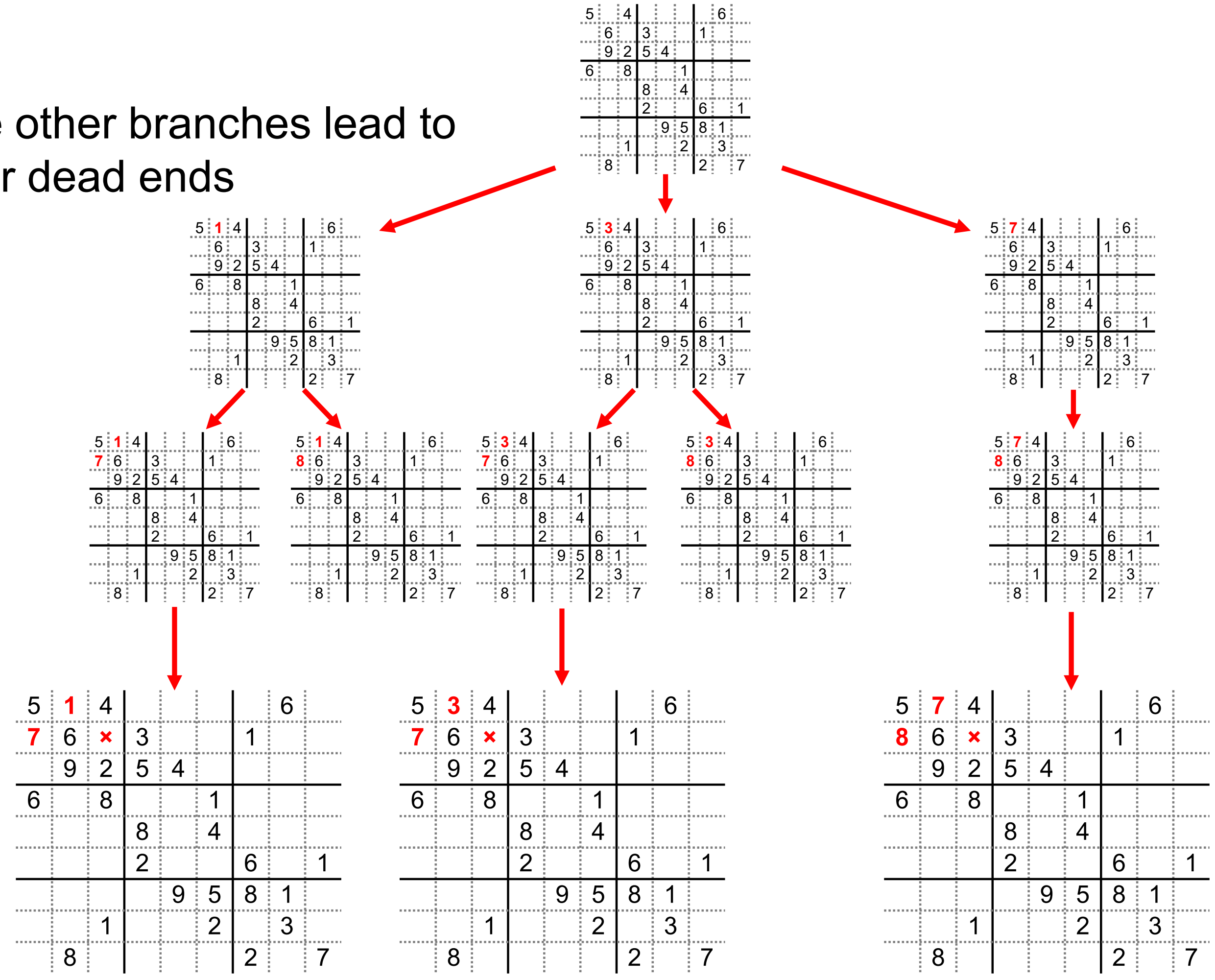


Any candidate solution built from this partial is infeasible

- We can ignore this branch

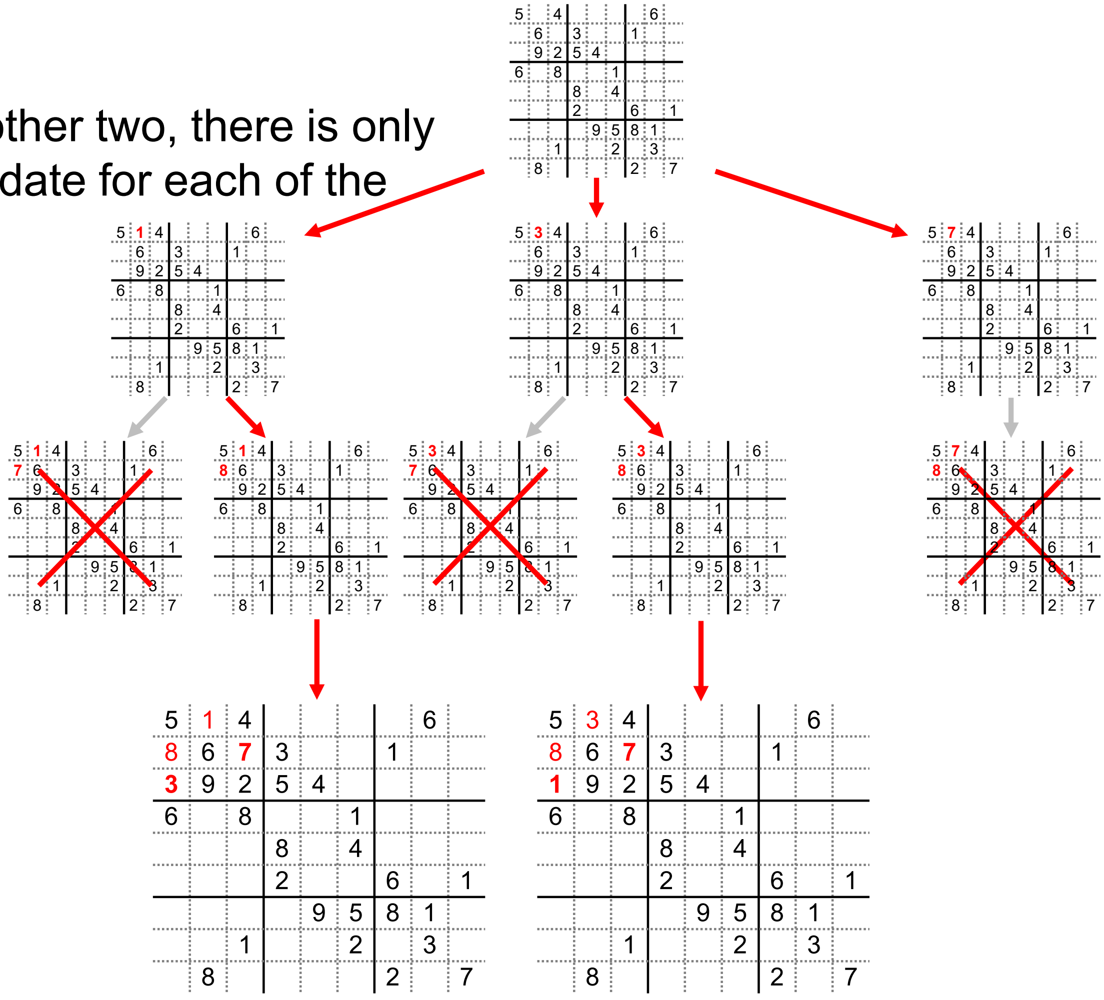
Sudoku

Three other branches lead to similar dead ends



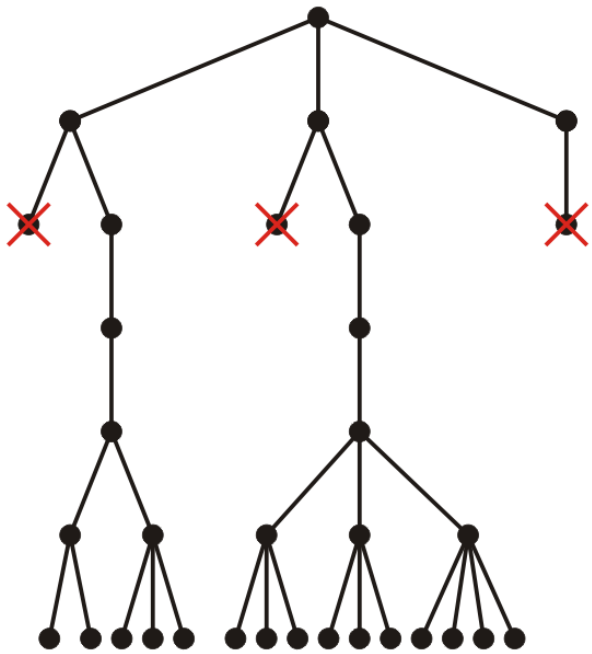
Sudoku

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



Sudoku

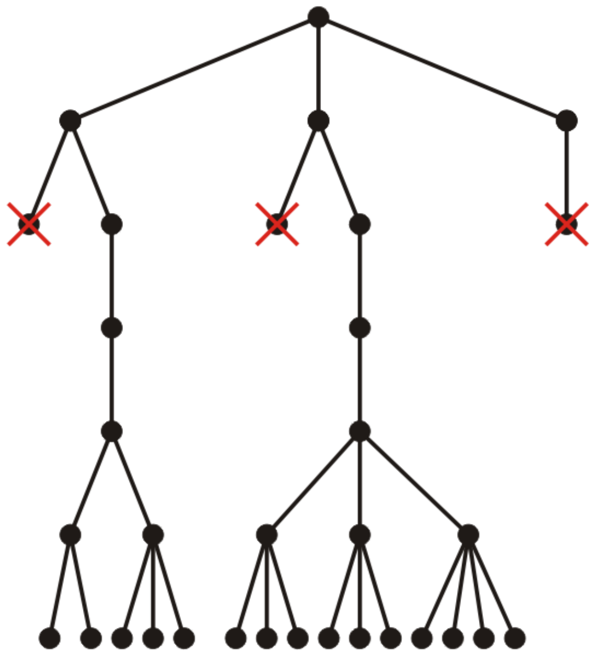
It may seem that this is a reasonably straight-forward method; however, the decision tree continues to branch quick once we start filling the second square



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 × 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

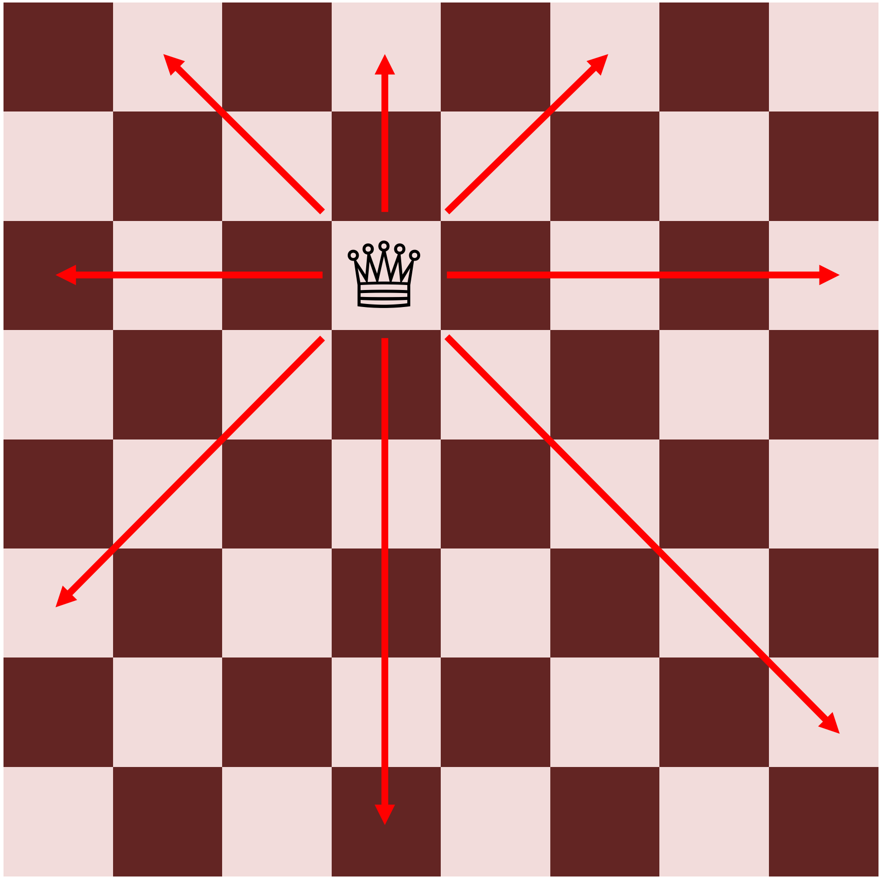
Classical applications

Classic applications of this algorithm technique include:

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

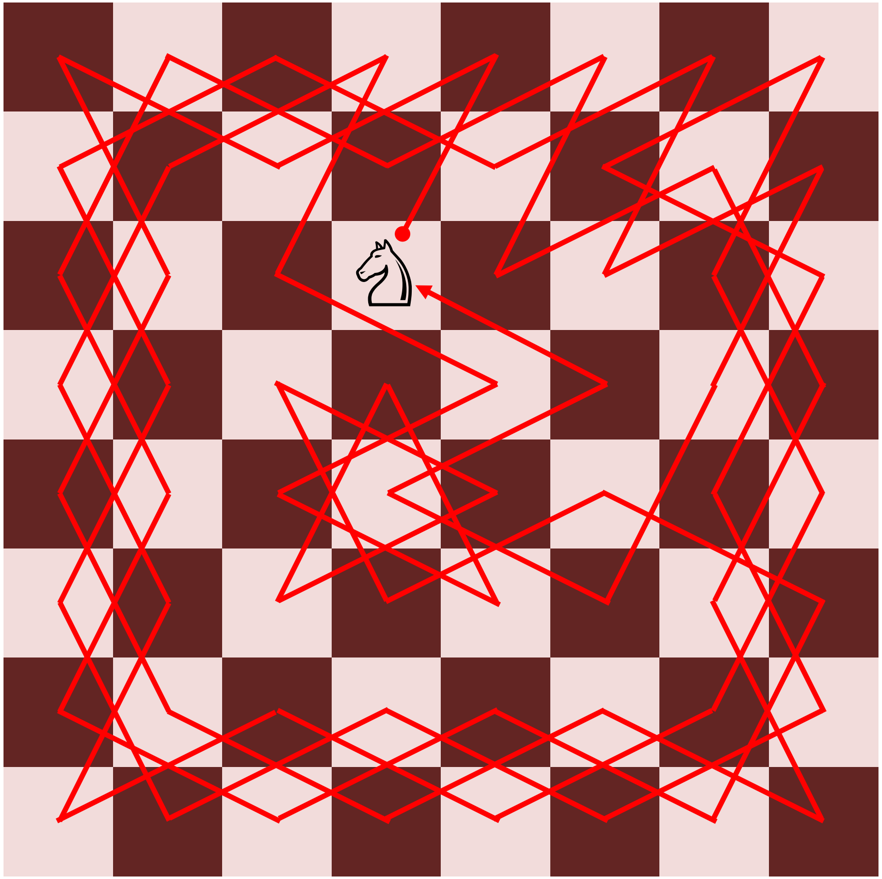
Eight queens puzzle

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).
male(bernhard).
female(beatrix).
male(claus).
male(firso).
female(mabel).
male(constantijn).
female(laurentien).

female(catharina).
female(alexia).
female(ariane).

female(luana).
female(zaria).

female(eloise).
male(claus_ii).
female(leonore).
```

```
parent(willem, catharina).
parent(maxima, catharina).
parent(willem, alexia).
parent(maxima, alexia).
parent(willem, ariane).
parent(maxima, ariane).

parent(firso, luana).
parent(mabel, luana).
parent(firso, zaria).
parent(mabel, zaria).

parent(constantijn, eloise).
parent(laurentien, eloise).
parent(constantijn, claus_ii).
parent(laurentien, claus_ii).
parent(constantijn, leonore).
parent(laurentien, leonore).
```

```
parent(beatrix, willem).
parent(claus, willem).
parent(beatrix, friso).
parent(claus, friso).
parent(beatrix, constantijn).
parent(claus, constantijn).

parent(juliana, beatrix).
parent(bernhard, beatrix).
parent(juliana, irene).
parent(bernhard, irene).
parent(juliana, margriet).
parent(bernhard, margriet).
parent(juliana, christina).
parent(bernhard, christina).

spouses(willem, maxima).
spouses(firso, mabel).
spouses(constantinjn, 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

```
<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

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

回跳

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

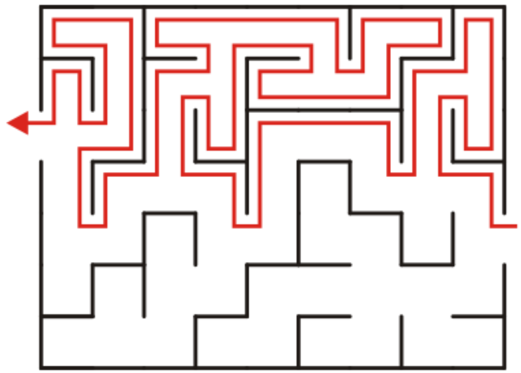
In trying to find your way through a maze, one simple rule works quite nicely:

- The right-hand rule:

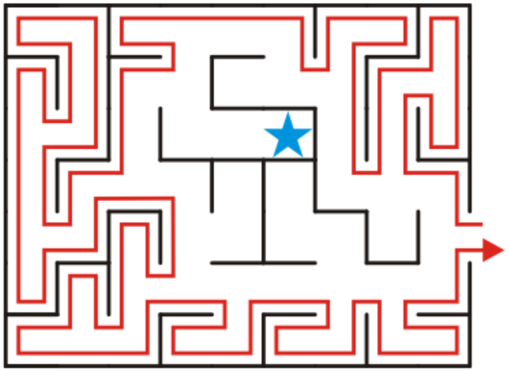
  Touch a wall with your right hand, and continue forward always keeping your right hand touching a wall until you get out.

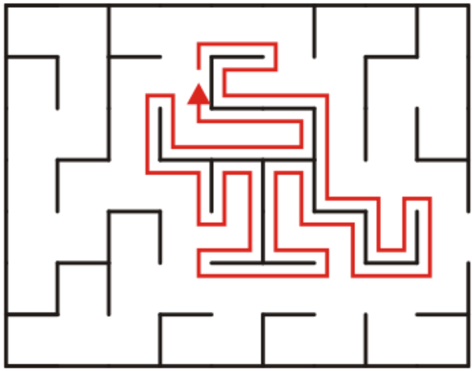
Searching a maze

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

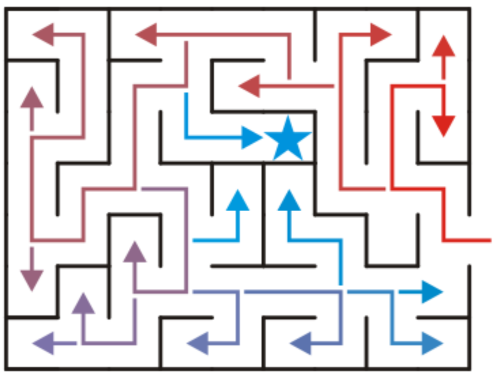




Searching a maze

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