Basic state-space search methods

P. Chardaire

University of East Anglia
School of Computing Sciences

Autumn semester 2018-19

Introduction

Often a solution to a problem is built step by step from an initial configuration.

- The intermediate configurations generated in the construction are called states.
- The initial configuration is called initial state.
- At each step of the construction you perform some operation to transform the current state into another.

Introduction

- Transforming a state using an operation is called a move.
- You stop when you have found a state that provides a solution with a desired property. Such a state is called a goal state.

The difficulty is that there can be many alternative choices of operation at each step, and therefore a considerable number of paths to explore of which many may lead to dead ends.

Our objective is to study methods to search such spaces.

State space representation

Example 1 (*n*-sliding-tile puzzles)

A typical class of problems are puzzles such as the Rubic's cube or *n*-sliding-tile puzzles.

Tile configurations and move in the 8-puzzle:

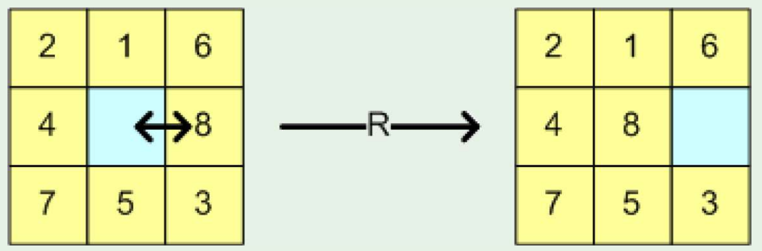


Figure 1: “Right” operation for the 8-puzzle

State space representation

- In the 8-puzzle the initial state and goal state are particular configurations of tiles.
- Operations consist of swapping the blank tile with one of its vertical or horizontal neighbours (in fact sliding a tile into the space) and may be represented as R(ight), L(eft), U(p), D(own) according to the direction of movement of the blank tile.

State space representation

- NOT all operations may be applicable to all configurations, e.g. If the blank tile is at the top left corner operations L and U cannot be applied.
- The objective is to find a sequence of operations that transforms the initial state into the goal state.
- A secondary objective may be to find the shortest sequence of such operations.

State space representation

It is useful to think of the state space as a graph where the states are the nodes of the graph (graph theoreticians use the word vertex for node) and the operations are represented by directed arcs.

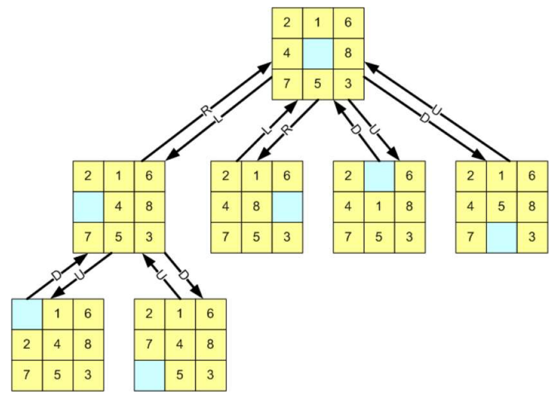


Figure 2: Portion of the state-space graph for the 8-puzzle

State space representation

Formal definition of a simple directed graph:

A simple directed graph $G(V,E)$ is specified by a set of vertices V and a set or directed arcs $E \subseteq V^2$.

For example $V = \{1,2,3\}$ and $E = \{(1,2),(1,3)\}$ is the simple directed graph that can be depicted as follows.

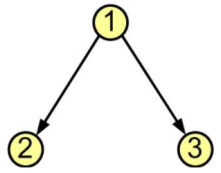


Figure 3: a simple directed graph

State space representation

Formal definition of a simple undirected graph:

A simple undirected graph $G(V,E)$ is specified by a set of vertices V and a set or undirected arcs (edges) $E \subseteq 2^V$ where each element in E is a set of 2 vertices.

For example $V = \{1,2,3\}$ and $E = \{\{1,2\},\{1,3\}\}$ is the simple undirected graph that can be depicted as follows.

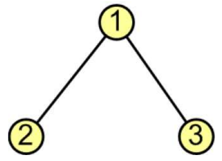


Figure 4: a simple undirected graph

State space representation

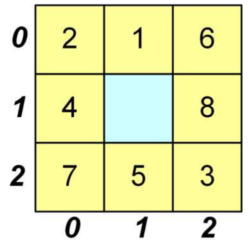
In most problems the arcs of the underlying graph are implicitly defined by moves.

Let us fix a representation for the *n*-puzzle. A tile configuration will be represented by a list containing

- the row index `i` and column index `j` of the blank tile.
- the configuration of the tiles as a list of lists where each of the latter represents a row. The blank tile is represented by a zero.

State space representation

For example



0	2	1	6
1	4		8
2	7	5	3
	0	***1***	***2***

is represented as `[1,1, [[2,1,6],[4,0,8],[7,5,3]] ]`

Python implementation

The blank tile position is changed by this generator:

```
def move_blank(i,j,n):
    if i+1 < n:
        yield (i+1,j)
    if i-1 >= 0:
        yield (i-1,j)
    if j+1 < n:
        yield (i,j+1)
    if j-1 >= 0:
        yield (i,j-1)
```

Generators are an interesting feature for search problems as state are generated just on time in search algorithm.

```
def move(state):
    [i,j,grid]=state
    n = len(grid)
    for pos in move_blank(i,j,n):
        i1,j1 = pos
        grid[i][j], grid[i1][j1] = grid[i1][j1], grid[i][j]
        yield [i1,j1,grid]
        grid[i][j], grid[i1][j1] = grid[i1][j1], grid[i][j]

# a test
state = [1,1,[ [2,1,6],[4,0,8],[7,5,3]]]
for nextState in move(state):
    print(nextState)
```

Notice how the grid is restored when you go back to the function after a `yield`. This avoids a bit of copying.

Example 2 (*n*-queen problem)

In the 8-queen problem the objective is to find a goal state that represents a solution to the problem

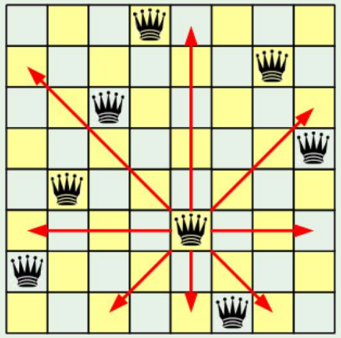


Figure 5: A solution to the 8-queen problem

The state space for this problem may be defined as follows

- A state represents an $n \times n$ board on which $0 \leq p \leq n$ queens are placed in columns $0, 1, 2, \ldots, p-1$ so that they cannot attack each other.
- The initial state is an empty board ($p = 0$).
- An operation consists of placing a queen on the first unoccupied column of a board so that it does not attack the queens which are already on the board.
- Goals are states with n queens.

Note that we are not interested in the path to the solution.

State space representation

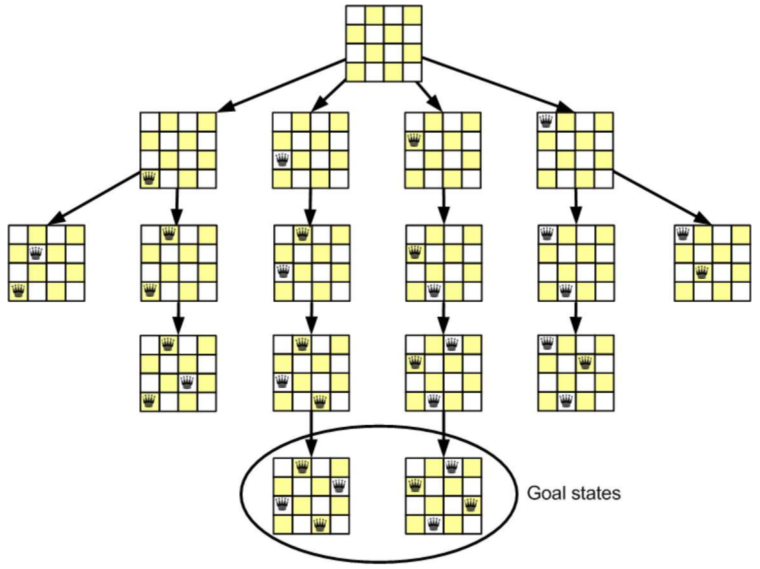


Figure 6: The 4-queen search space

State space representation

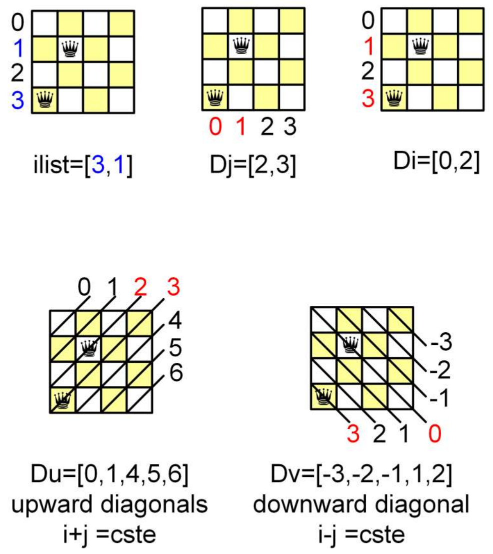


Figure 7: State representation for Python implementation

State space representation

```
def move(state):
    [ilist,dj,di,du,dv] = state
    j = dj[0]
    for i in di:
        u = i+j
        v = i-j
        if u in du and v in dv:
            ilist1 = ilist + [i]
            dj1 = dj[1:]
            di1 = di.copy() ; di1.remove(i)
            du1 = du.copy() ; du1.remove(u)
            dv1 = dv.copy() ; dv1.remove(v)
            yield [ilist1,dj1,di1,du1,dv1]

state = [[3],[1,2,3],[0,1,2],[0,1,2,4,5,6],[-3,-2,-1,0,1,2]]
for nextState in move(state):
    print(nextState)
```

State space representation

Example 3 (The monkey-and-bananas problem)

- A monkey is in a room containing a box and a bunch of bananas. The bananas are hanging from the ceiling out of reach of the monkey.
- What actions will allow the monkey to get the bananas? (The monkey is supposed to go to the box, push it under the bananas, climb on top of it, and grasp the bananas.)
- This problem illustrates the importance of the choice of a good state-space representation for a problem.

State space representation

A state will be represented by a quadruple (w, x, y, z) where

- w is the horizontal position of the monkey (a two-dimensional vector)
- x is 1 if the monkey is on top of the box and is 0 otherwise
- y is the horizontal position of the box (a two-dimensional vector)
- z is 1 if the monkey has the bananas and is 0 otherwise

Problem: If you think of each different value of (w, x, y, z) as describing a unique state you have an infinite description of states as there is an infinite number of locations in the room.

State space representation

Instead you use quadruples that represent schemas, i.e. combination of constants and variables (symbols).

The possible operators are:

- $\text{goto}(u) : (w, 0, y, z) \to (u, 0, y, z)$ (u is a variable.)
- $\text{pushbox}(v) : (w, 0, w, z) \to (v, 0, v, z)$ (v is a variable.)
- $\text{climbbox} : (w, 0, w, z) \to (w, 1, w, z)$
- $\text{climbdown} : (w, 1, w, z) \to (w, 0, w, z)$
- $\text{grasp} : (c, 1, c, 0) \to (c, 1, c, 1)$

where c is the location on the floor directly below the bananas.

State space representation

The monkey and banana problem is most often illustrated with a Prolog implementation as Prolog has a concept of symbols (variable).

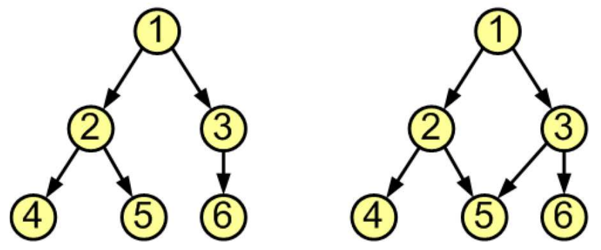
However, depending or the order of the moves tested in the program (clauses in Prolog) the monkey may end up walking aimlessly or pushing the box endlessly.

Therefore we need to design search methods that alleviate this problem of state move ordering.

Basic tree-search techniques

First, we will examine search methods in rooted trees.

In a rooted tree every node has exactly one immediate ancestor, except the root of the tree that has none.



$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[dr] & \\
2 \arrow[d] \arrow[dr] & & 3 \arrow[d] \\
4 & 5 & 6
\end{tikzcd}
\qquad
\begin{tikzcd}
 & 1 \arrow[dl] \arrow[dr] & \\
2 \arrow[d] \arrow[dr] & & 3 \arrow[dl] \arrow[d] \\
4 & 5 & 6
\end{tikzcd}$$

Figure 8: Rooted tree and DAG (directed acyclic graph)

Rooted trees and DAGs are search spaces that typically underline construction problems where a move consists of adding an element to a set.

Construction problems that have an acyclic search-space representation include:

- The 8-queen problem.
- Soduku.
- The game number in the TV show, countdown.

The main data structure used to search a rooted tree is a list:

- OPEN: List of states that have been generated but not yet expanded.

The algorithms you shall see are:

- Breadth-First-Search (BFS): In this case OPEN is a queue, i.e. a FIFO structure
- Depth-First-Search (DFS): OPEN is a stack, i.e. a LIFO structure.

In the remaining of this lecture when you hear of tree you should understand rooted tree.

- In BFS the nodes are explored in non decreasing distance (in the number of arcs) from the root node.
- BFS guarantees that the goal node found is the closest goal node to the root.

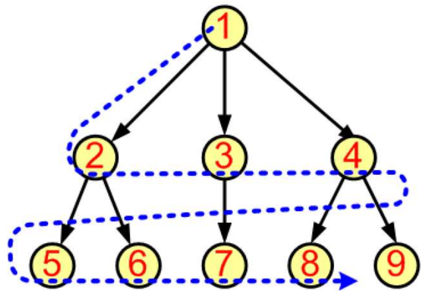


Figure 9: Order of exploration of nodes in BFS

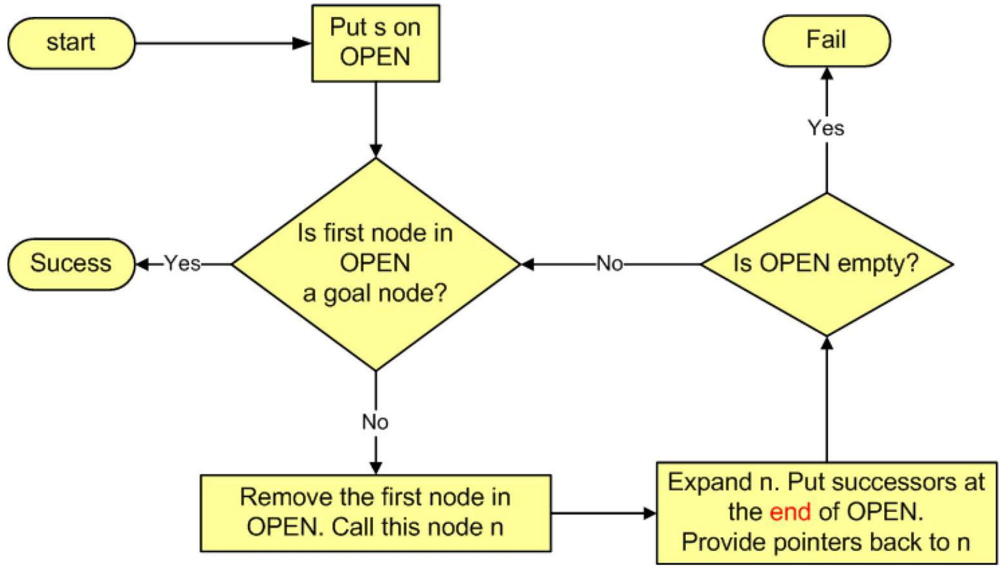

start
Put s on OPEN
Fail
Yes
Sucess
Yes
Is first node in OPEN a goal node?
No
Is OPEN empty?
No
Remove the first node in OPEN. Call this node n
Expand n. Put successors at the end of OPEN. Provide pointers back to n

Figure 10: Flow chart of BFS for trees

DFS expands the deepest node first (deepest in terms of distance in number of arcs from the root node).

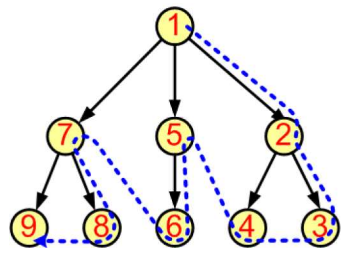


Figure 11: Order of exploration of nodes in DFS

Note this is valid for a stack of nodes. For a recursive program (stack of calls) the tree is explored from left to right.

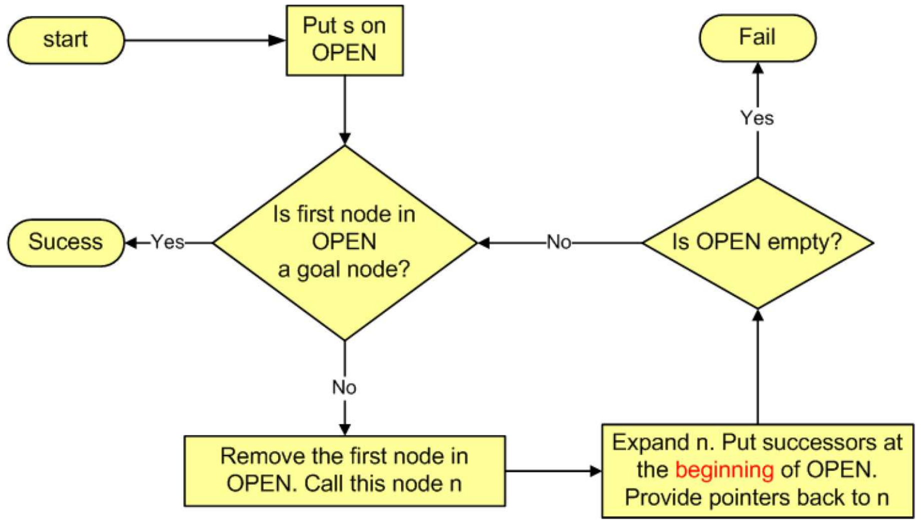
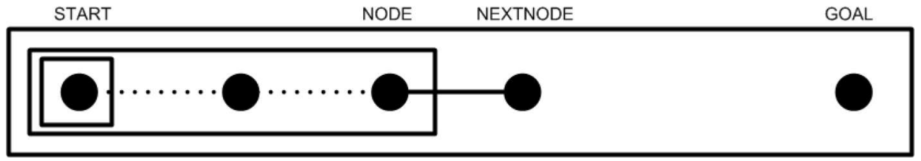

start
Put s on OPEN
Fail
Yes
Is first node in OPEN a goal node?
Sucess
Yes
No
Is OPEN empty?
No
Remove the first node in OPEN. Call this node n
Expand n. Put successors at the beginning of OPEN. Provide pointers back to n

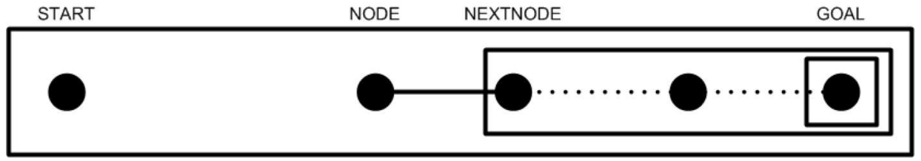
Figure 12: Flow chart of DFS for trees

Python implementation

There are two inductive representation of a path:



In DFS for DAGs the simplest approach is to use this inductive representation:



Python code

```
#
# How sweet
#
def dfs(state):
    if is_goal(state):
        return True,[state]
    else:
        for nextState in move(state):
            found, path = dfs(nextState)
            if found:
                return found,[state]+path
    return False,[]
```

BFS

BFS guarantees that a closest goal (in the number of moves) is found.

BFS

In a tree with branching factor b (number of children of each node) the size of OPEN grows exponentially with the depth of exploration. OPEN contains b^d vertices when expanding the first node at depth d.

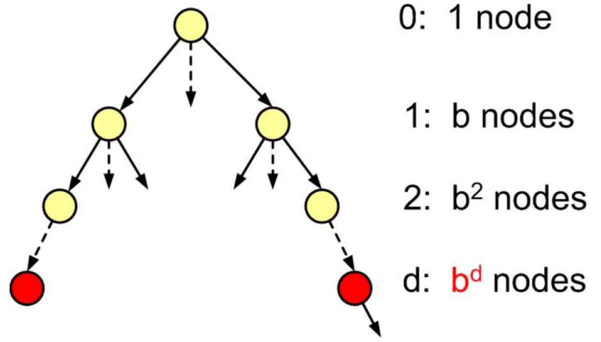


Figure 13: Size of OPEN in BFS

BFS

Note that our BFS implementations work for state spaces that are general graphs, albeit with unnecessary re-examination of states.

DFS

DFS uses moderate memory as it only stores an exploration path with its branching neighbours at any point in time. So when it reaches level d the size of OPEN is $d(b-1)+1$. (When using a Python recursive implementation you still have a space complexity proportional to d.)

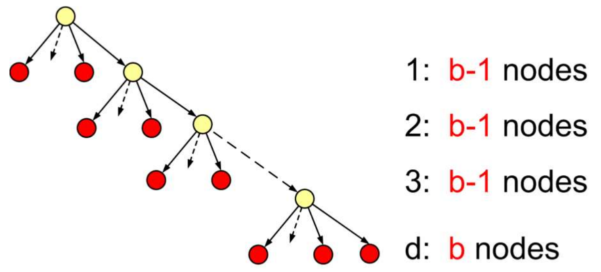


Figure 14: Size of OPEN in DFS

DFS

However in large trees, DFS could get stuck into exploring a subtree that does not contain a goal node.

DFS

Our DFS implementation does not work for general graphs as DFS could get stuck in an infinite cycling path.

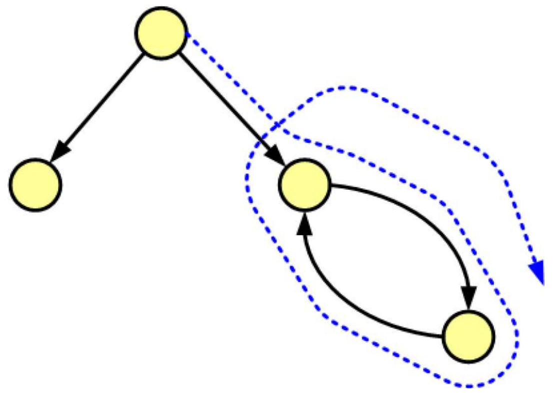


Figure 15: DFS stuck in cycle

Now you have an idea!

If you knew in advance the exact level of the goal state closest to the initial state, you could simply apply DFS with limited depth to generate all paths to that level.

This would also avoid the problem of infinite cycling in DFS applied to general graphs (more on that in a mo)

You would obtain the same result as BFS but at a fraction of the memory cost.

What a good idea you had!

But how do you set `MaxMove`?

- If you set it too low you may not reach a goal.
- If you set it too high you may waste a lot of time exploring an unprofitable region of the search space.

The solution is to use DFS with iterative deepening (DFID)

This method tries consecutive values $0, 1, 2, 3 \dots$ for `MaxMove` until a goal node is found.

At first, this seems very wasteful as each try has to re-examine all states examined in the preceding try.

Assume that the tree has a branching factor b. The performance of DFS with limited depth d is proportional to the number of nodes, N_d, at depth less than or equal to d.

$$N_d = 1 + b + b^2 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

If we iterate on depth $0, 1, \ldots, d$, instead, the number of nodes examined is

$$\sum_{i=0}^{d} N_i = \sum_{i=0}^{d} \frac{b^{i+1} - 1}{b - 1} = \frac{1}{b-1}\left(b\frac{b^{d+1} - 1}{b - 1} - (d+1)\right) \approx \frac{b}{b-1} N_d$$

So we deduce that the increase in computing time compared with BFS is not by more than $\frac{100}{b-1}$ percent.

For a branching factor of 2 the computing time could double, and for a branching factor of 5 it would increase by 25%.

This is an upper bound on the actual increase as the cost per state of BFS is greater than for DFID in particular if the OPEN list is large.

Also memory is a scarcer resource than time. A computer memory can be very rapidly flooded by the search.

Execution results for an 8-tile problem instance requiring 10 moves (programmed in Prolog):

For DFID:

```
% 653,952 inferences, 0.234 CPU in 0.312 seconds (75% CPU,...
```

For BFS:

```
% 1,882,875 inferences, 3.510 CPU in 3.619 seconds (97% CPU,..
true.
```

Execution results for an 8-tile problem instance requiring 16 moves:

For DFID:

```
% 332,607,510 inferences, 105.285 CPU in 105.519 seconds...
```

For BFS:

```
% 3,436,365 inferences, 9.142 CPU in 9.329 seconds...
ERROR: Out of global stack
```

These results were obtained using the 32-bit version that has a 128 MB global stack limit.

General-graph search

When searching general graphs a possible modification to the DFS and BFS search algorithms is as follows.

If a node has already been expanded then do not expand it again, i.e. do not put it in OPEN again. This enables to avoid generating cycles and paths with common endings.

The main data structures used to search a general graph are two lists:

- OPEN: List of states that have been generated but not yet expanded.
- CLOSED: List of states that have been expanded.

General-graph search

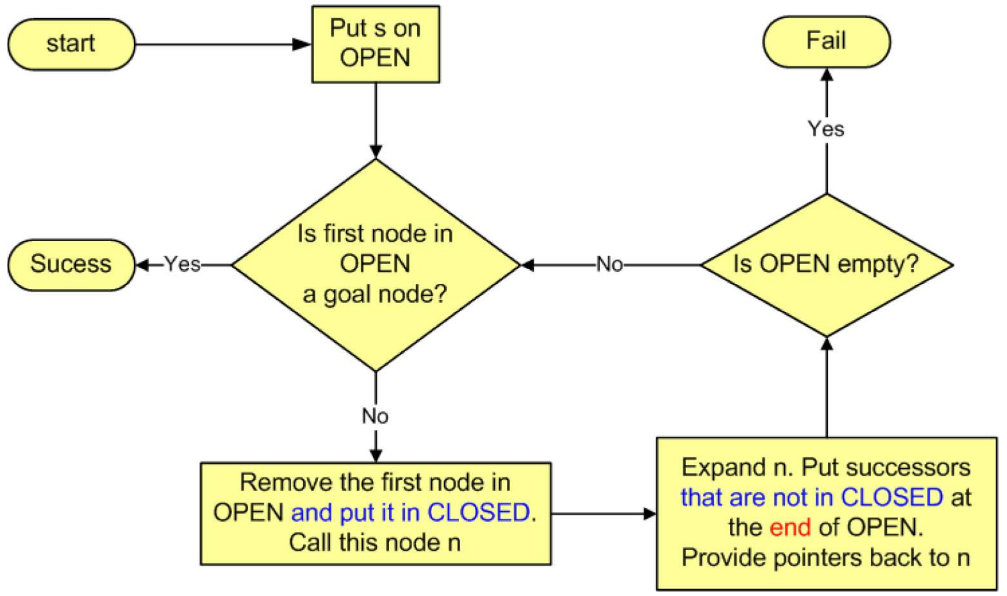

start
Put s on OPEN
Fail
Yes
Sucess
Yes
Is first node in OPEN a goal node?
No
Is OPEN empty?
No
Remove the first node in OPEN and put it in CLOSED. Call this node n
Expand n. Put successors that are not in CLOSED at the end of OPEN. Provide pointers back to n

Figure 16: Flow Chart of BFS for general graph

General-graph search

BFS still returns a closest goal node to the starting state.

The flow chart of DFS for tree search is similarly modified for general search.

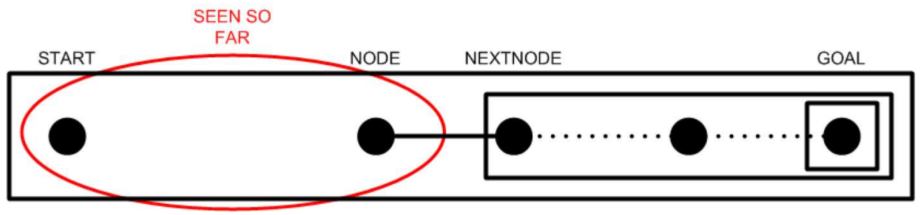
Memorizing all visited nodes is however memory-expensive and only applicable to small state spaces.

A more practical idea is to prevent the generation of cycling paths.

We will focuss on DFID.

General-graph search

To prevent cycling paths we need to memorize the nodes seen so far along the current exploration path:



General-graph search

For the 16-move 8-tile problem instance:

For DFID without cycle detection:

```
% 332,607,510 inferences, 105.285 CPU in 105.519 seconds...
```

For DFID with cycle detection using membership test:

```
% 1,882,192 inferences, 0.718 CPU in 0.842 seconds...
```

What do you conclude?

Conclusion

- All the methods you have seen are general.
- They are however limited because they are brute-force techniques.
- They search all paths without discrimination.
- In order to solve difficult problems you need to limit the number of states examined.
- In the second part of the lectures on state space search you will see how you can create intelligent methods based on the ideas developed so far for the exploration of a search space.
- The price to pay will be in the design of specific metrics associated to each problem at hand.