State space search
Blind search

Ernest Davis

January 24, 2022

First a couple of general comments, about this section and about the course in general

- State space search is not AI as such. It is a framework in which you can describe algorithms, and it is often useful for AI algorithms.
- This part of the course will seem a lot like the algorithms course; the rest of the course will be less so.
- But in general the course will be more math- and algorithm- oriented than you might expect.
- Partly that's me.
- But mostly that's the subject matter. Unless you are building a large, production system (e.g. Google) or a physical robot, AI doesn't tend to involve complex software engineering; except for particularly complex applications (e.g. playing poker) it often doesn't even involve complex code.
- The big difference from the algorithms class is that it's very rare that you can prove mathematically that an AI algorithm works correctly. You rely on empirical testing.

N-Queens problem

On an NxN chess board, place N queens s.t. (such that --- I will use this abbreviation throughout this course) no two queens attack.

That is, no two queens are in the same row, same column, or same diagonal.

(There is a simple known solution to this for all N ≥ 4, but we'll ignore that.)

The question is, how can we *systematically search* for a solution?

N-Queens problem

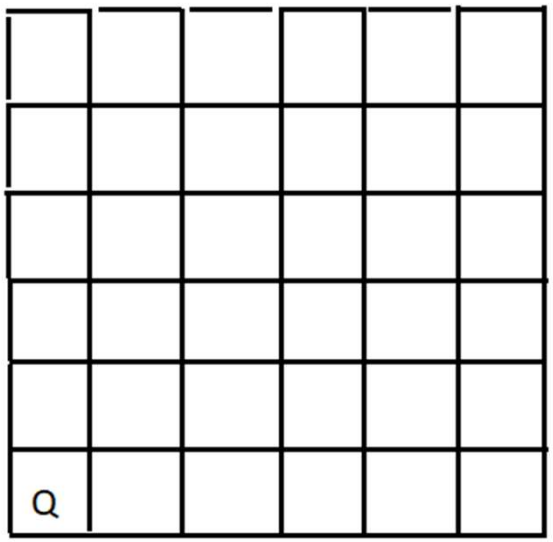
Since there are N columns, and N queens, and no two queens are in the same column, there must be exactly one queen in each column.

We will fill in queens left to right, putting each queen in the first row where she fits. If we get stuck, we will go back to the last point we had a choice, and try the next possible solution.

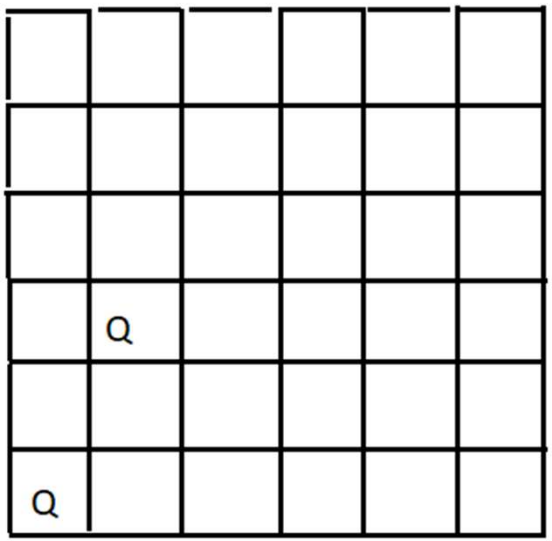
Backtracking a.k.a. depth-first search.

I'll take N=6. (No solution N=2 or 3; 4 is too easy; 5 involves no backtracking at all.)

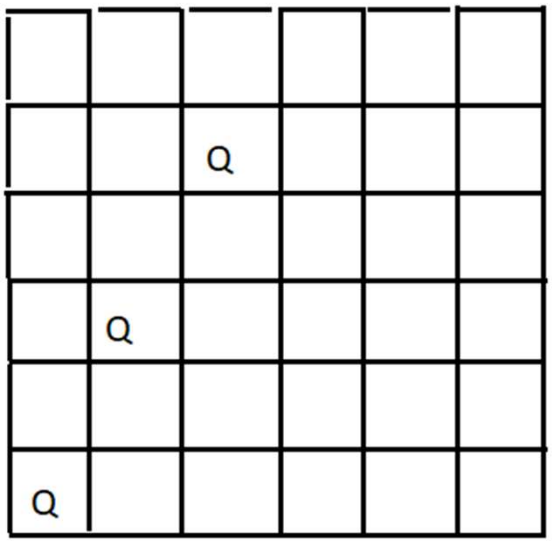
Start with 1st Queen in row 1



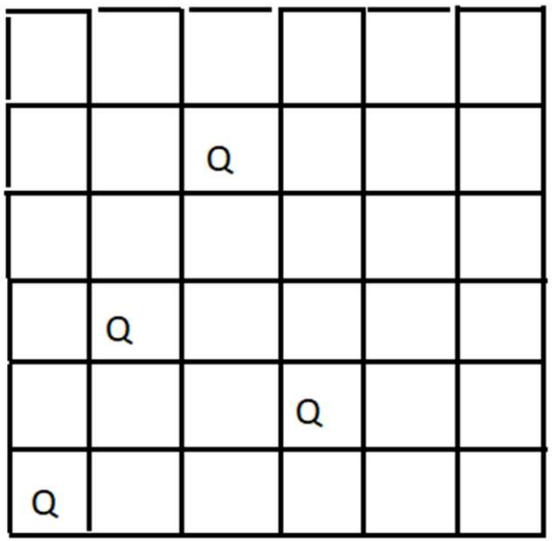
Add 2nd Queen in row 3



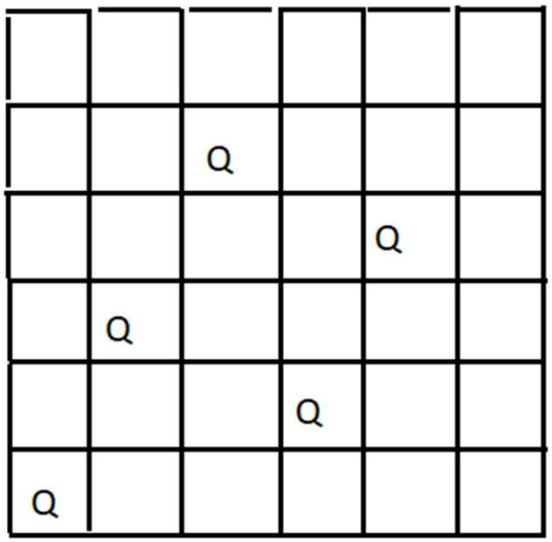
3rd Queen in row 5



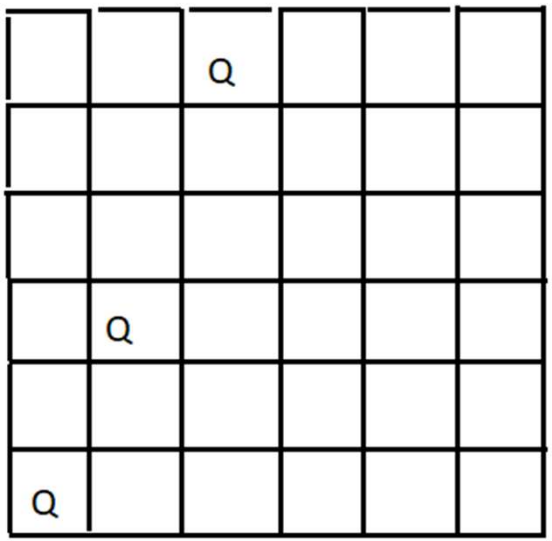
4th Queen in row 2



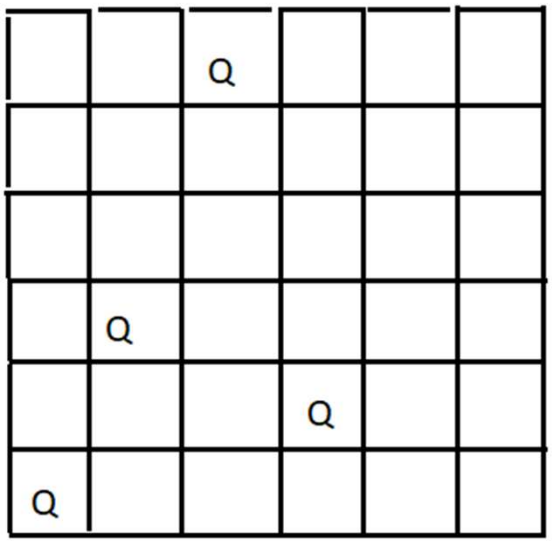
5th queen in row 4. But now we're stuck. Nowhere to put 6th Queen
Nowhere else to put 5th Queen. 4th Queen must have been wrong.
Nowhere else to put 4th Queen either. Go back to 3rd Queen



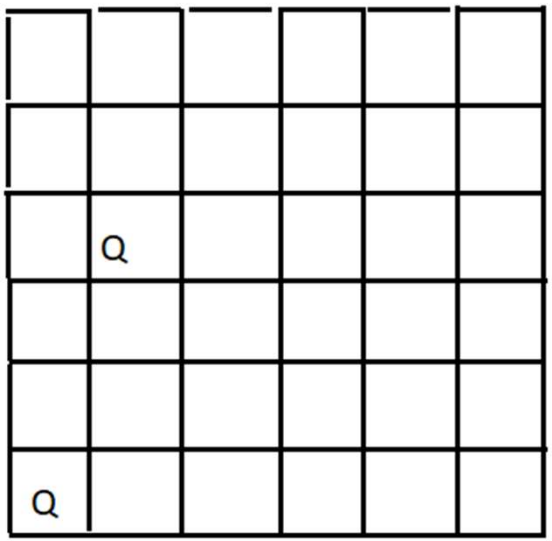
Try 3rd Queen in row 6



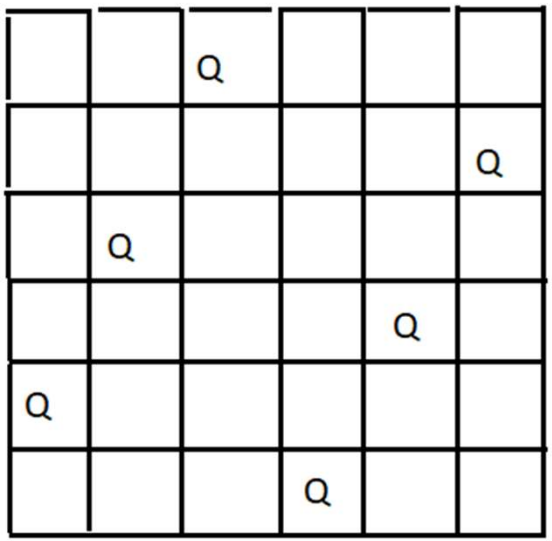
4th Queen in row 2. Stuck. Nowhere to put 5th Queen.
Nowhere else to put 4th Queen. Nowhere else to put 3rd Queen
2nd Queen must have been wrong.



Try 2nd Queen in row 4.



And so on. It turns out eventually that the first queen was wrong. About 20 boards later …



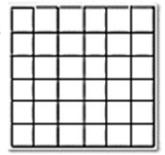
Abstract framework: State Space

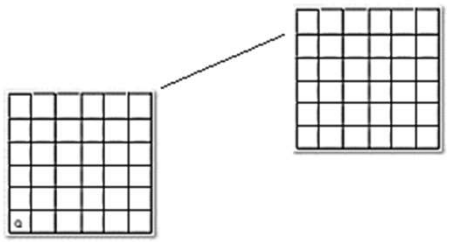
A *state space* consists of

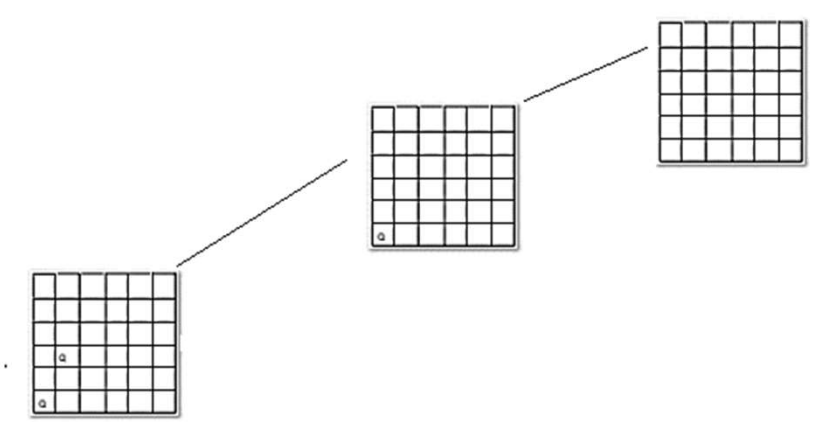
1. A definition of a *state.* For N-Queens, a state is a valid placement of the first k Queens in the first k columns, for some k between 0 and N.
2. A way to compute the *successors* of a state (or a collection of *operators* on a state). For N-Queens, if S has k queens then the successors of S add the k+1st queen in column k+1
3. A way to recognize the *goal* state. For N-Queens, a state where k=N.
4. A start state. For N-Queens, the empty board.

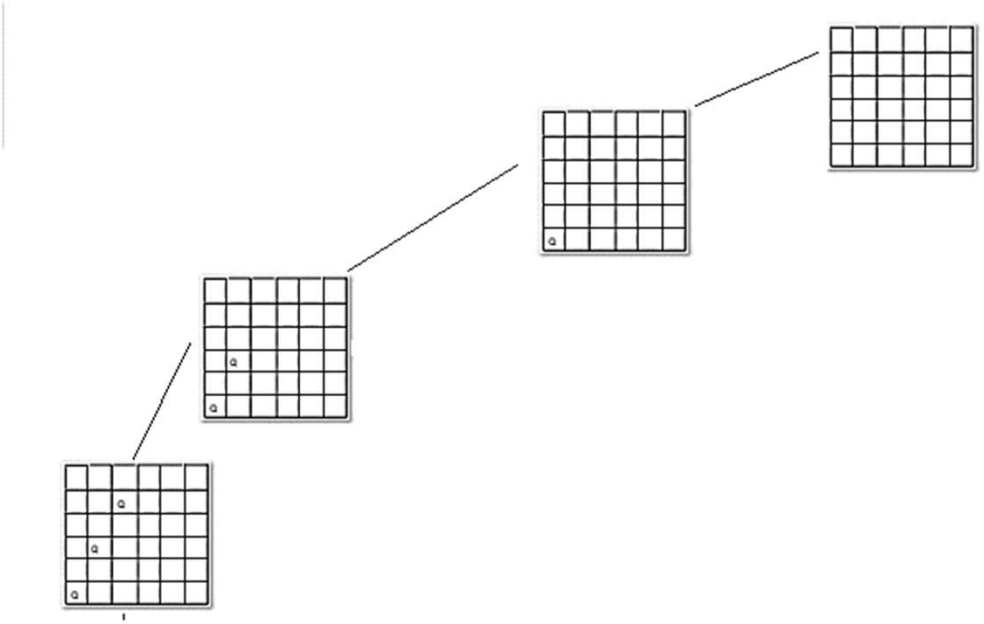
State space search

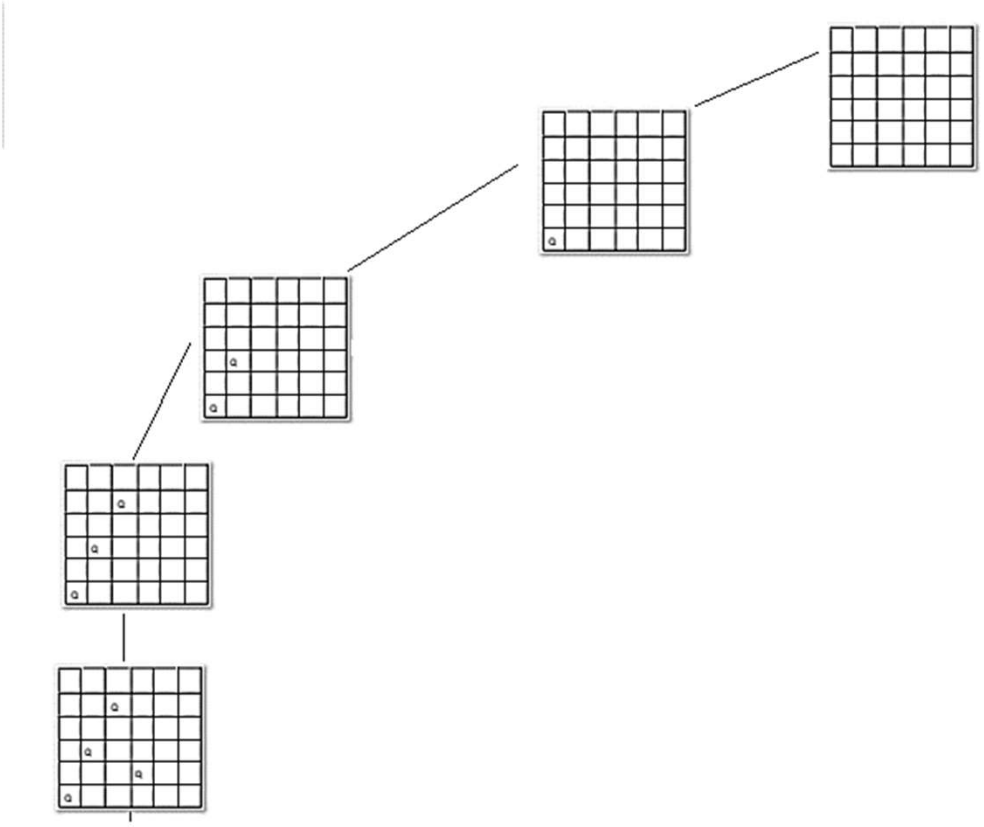
In state space search, the program starts at the start state, and then generates successors, and successors to the successors, and so on until it reaches a goal state.

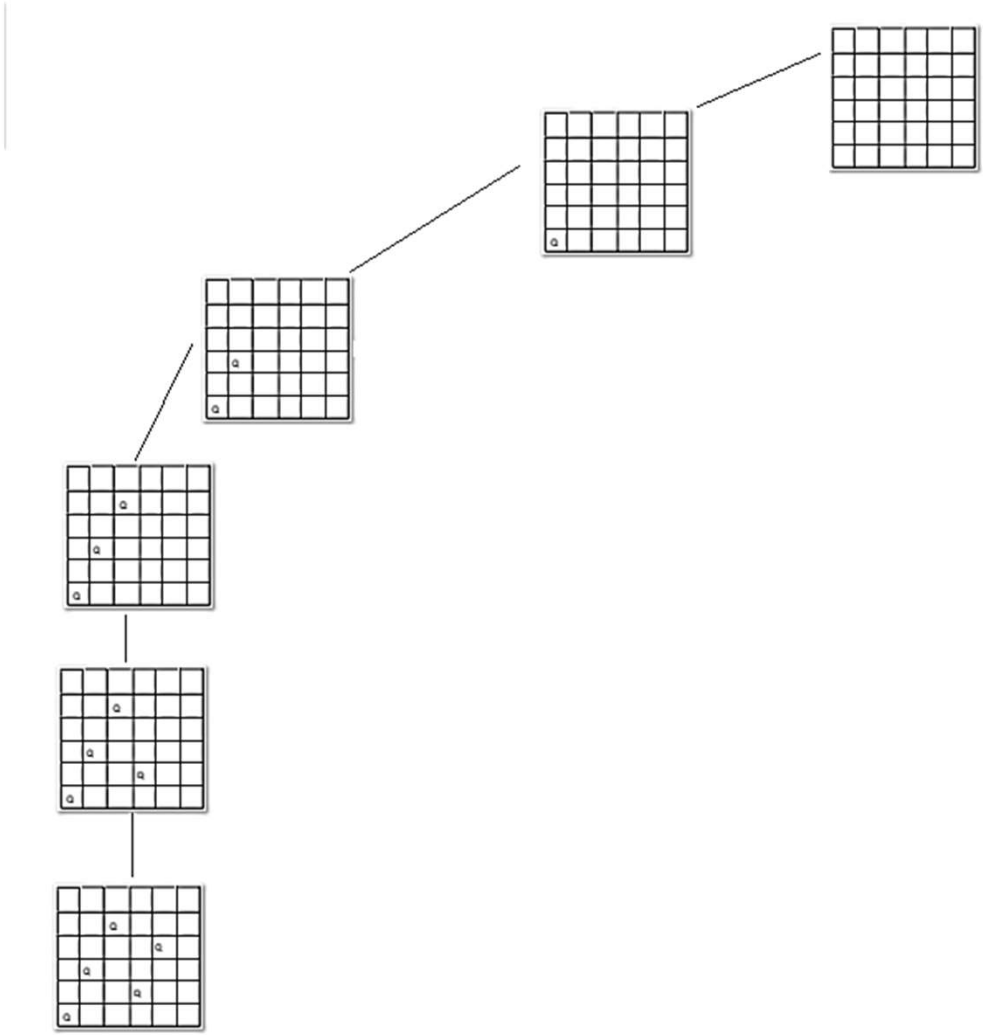


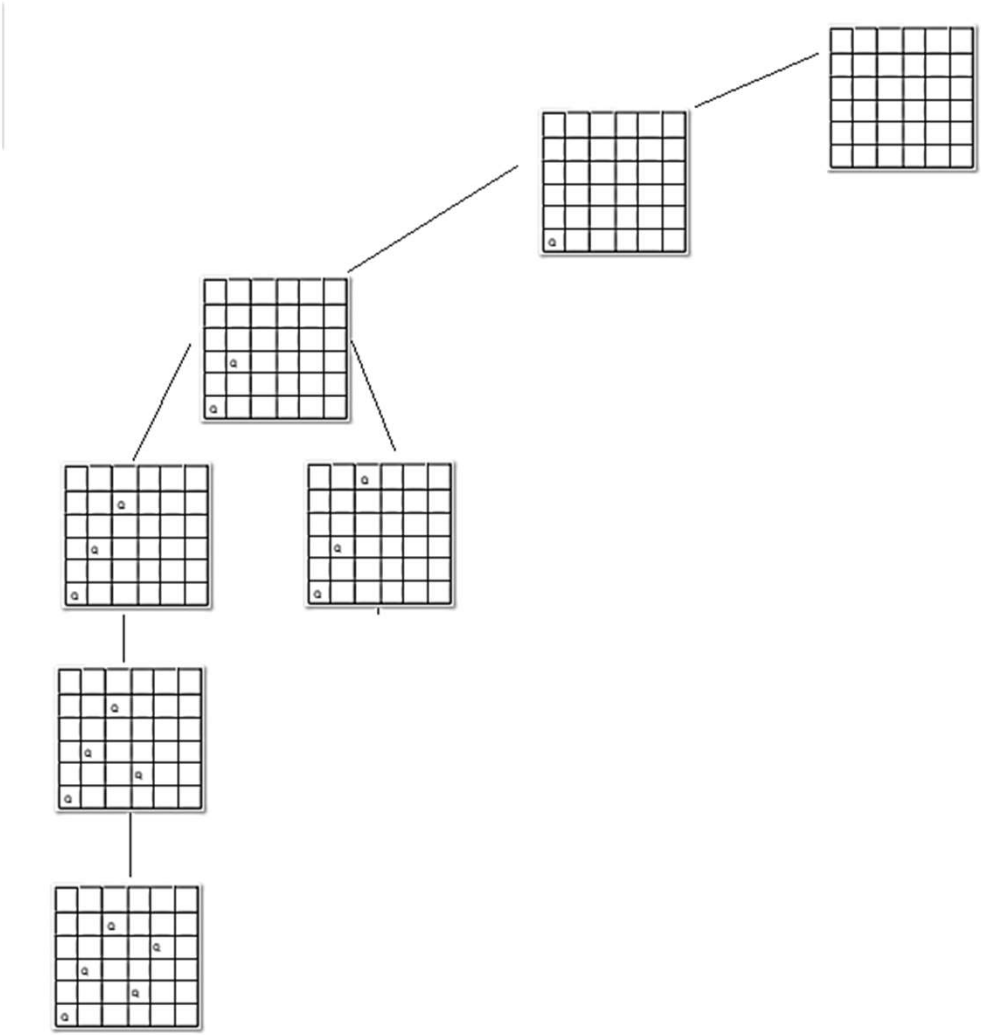


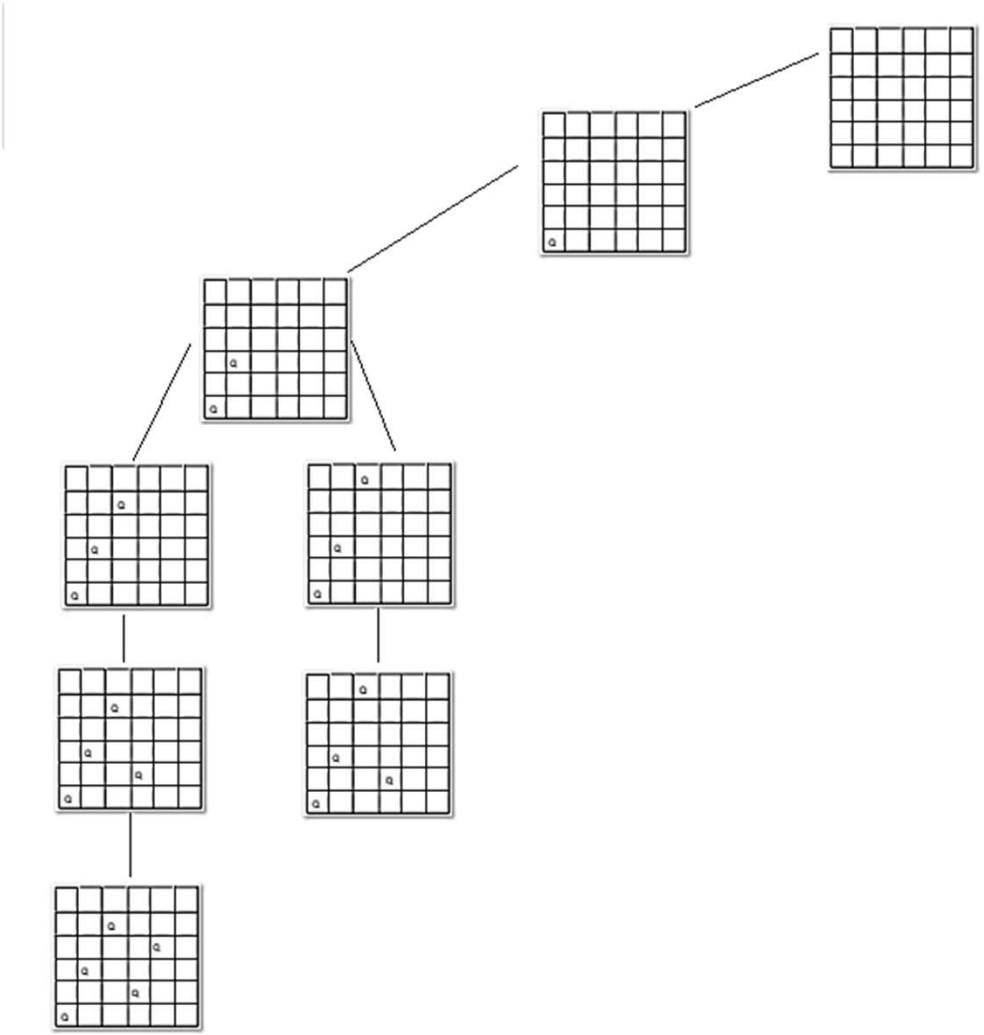


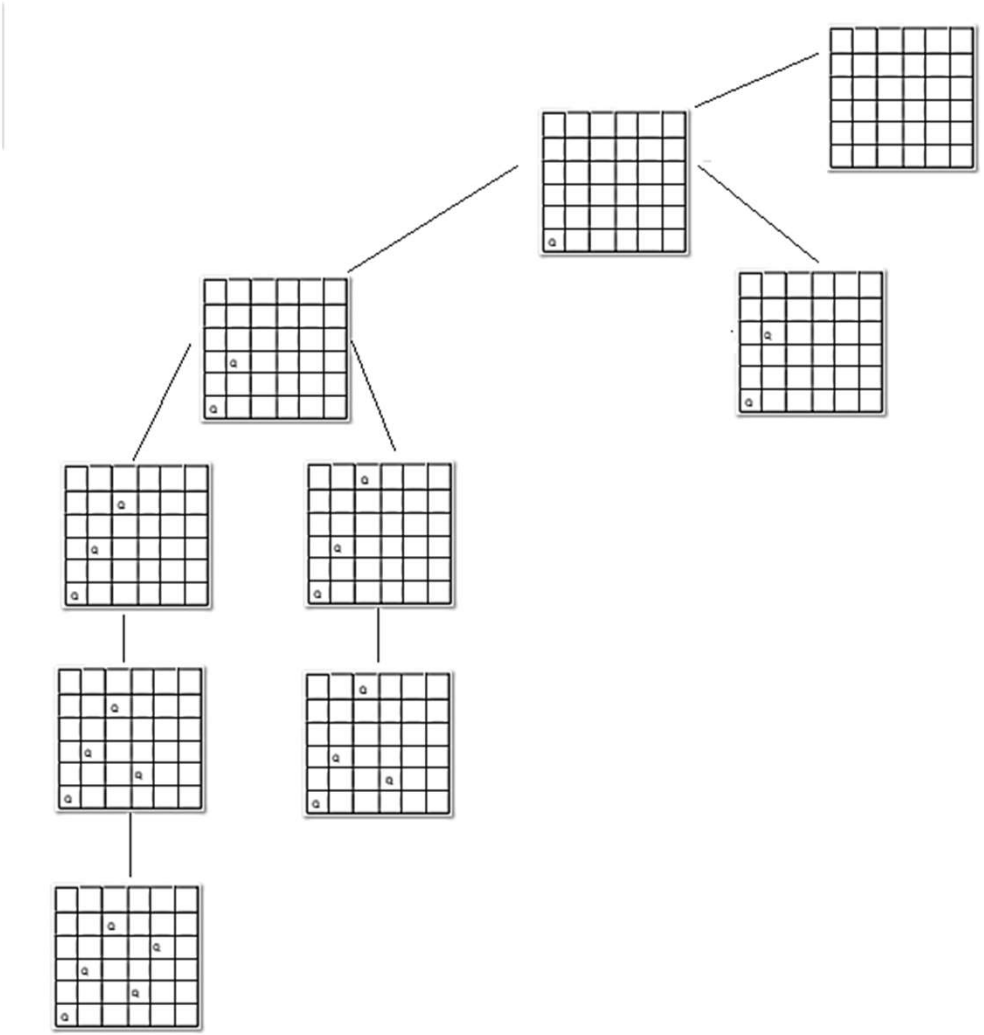


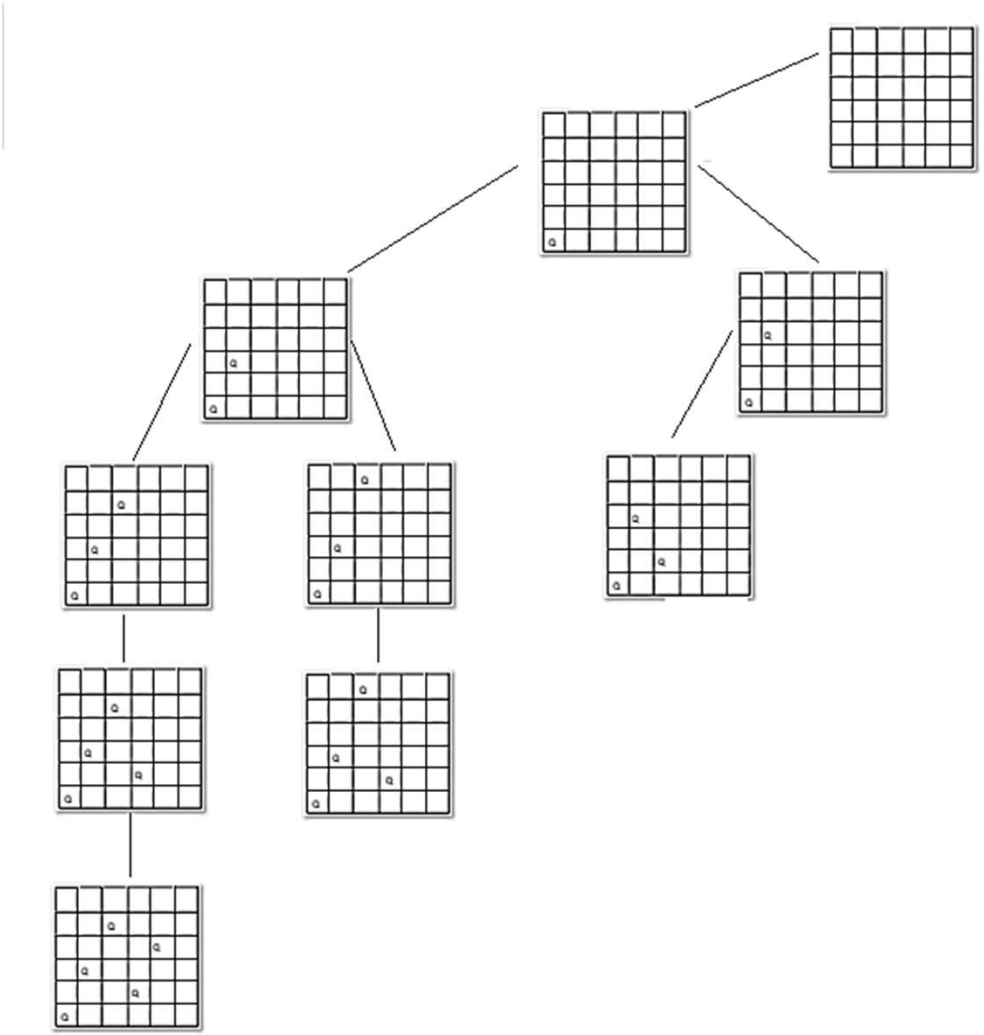












Very important!

A state space can be thought of as a graph: vertices are states, arcs point from a state to its successors.

But it is not like the graphs studied in algorithms class.

The state space is a *virtual* graph, **NOT A DATA STRUCTURE.**

Your program does not construct the tree in the previous slides.

All it constructs are the states.

The state space is a framework for you to use in thinking about the problem, designing algorithms for the problem, describing the problem.

Depth-first search through a state space

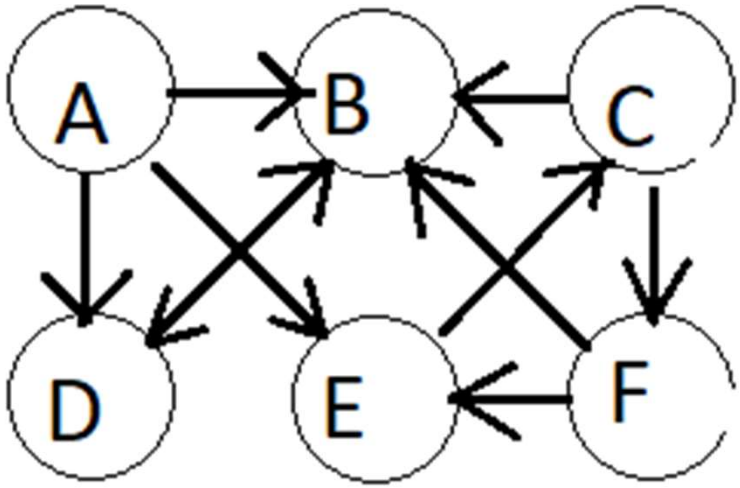
```
dfs() { return dfs1(start); } % returns a goal state, or “FAIL”
dfs1(s) {                     % returns a goal in the subtree rooted as s
    if (goal(s)) return s;
    for (c in successors(s)) {
        ans = dfs1(c);
        if (ans != “FAIL”) return ans;
       }
    return “FAIL”;  % if none of the successors of s work,
  }                 % then there is no goal in the subtree of s
% Two base cases for recursion:
% A goal is found, or state s has no successors.
```

More examples of state space search
Hamiltonian path

Given a directed graph G and a starting vertex V, find a path through G starting at V that included every vertex of G exactly once.

E.g. In this graph, a Hamiltonian path starting at A is

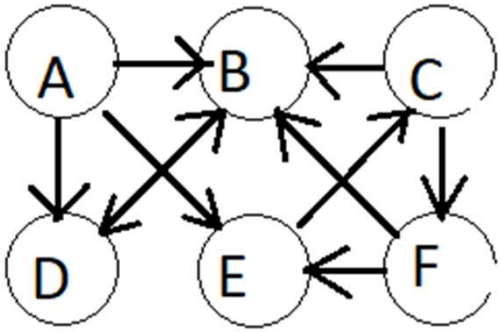
A→E→C→F→B→D



State space for Hamiltonian path

- State: A path through G with k non-repeated vertices starting at the start vertex.
- Successor. Given a path P ending with vertex U, for some arc U->V where V is not already in P, add V at the end of P.
- Goal: k=n (the number of vertices)
- Start: Path consisting of start vertex.

State space for example. (Note this handy trick for drawing trees in typewriter mode.)




```
[A]--->[A,B]--->[A,B,D]
    |
    |->[A,D]--->[A,D,B]
    |
    |->[A,E]--->[A,E,C]--->[A,E,C,--->[A,E,C,
                          |   B]         B,D]
                          |
                          |->[A,E,C,--->[A,E,C--->[A,E,C
                             F]         F,B]      F,B,D]
```

Example: Exact set cover

Given: A set Ω and a collection C of subsets of Ω.

Find: A subcollection D of C s.t. every element of Ω appears exactly once in D.

Example: Ω = {a,b,c,d,e,f,g,h}

C1={a,b,g}. C2={a,c,d}. C3={b,e}. C4={c,h}. C5={b,f,h}. C6={c,f,h}. C7={e,g}

Solution: C2, C5, C7.

Exact set cover: State space

State: A subcollection of C with no repeated elements.

Successor to state S: Add a set in C that does not overlap with S and that comes later in C than any of the sets in S.*

Start state: The empty collection.

Goal: A collection that covers all of Ω.

* The reason for this condition is to avoid repeating states. You do not want to first generate the state {C1, C4}, and then to generate the state {C4, C1}. In general, when you are using states that are sets, you want to add elements in some kind of fixed sequence, or find some other way of avoiding repeating states.

State space: Exact set cover

Example:
Ω = {a,b,c,d,e,f,g,h}

C1={a,b,g}.

C2={a,c,d}.

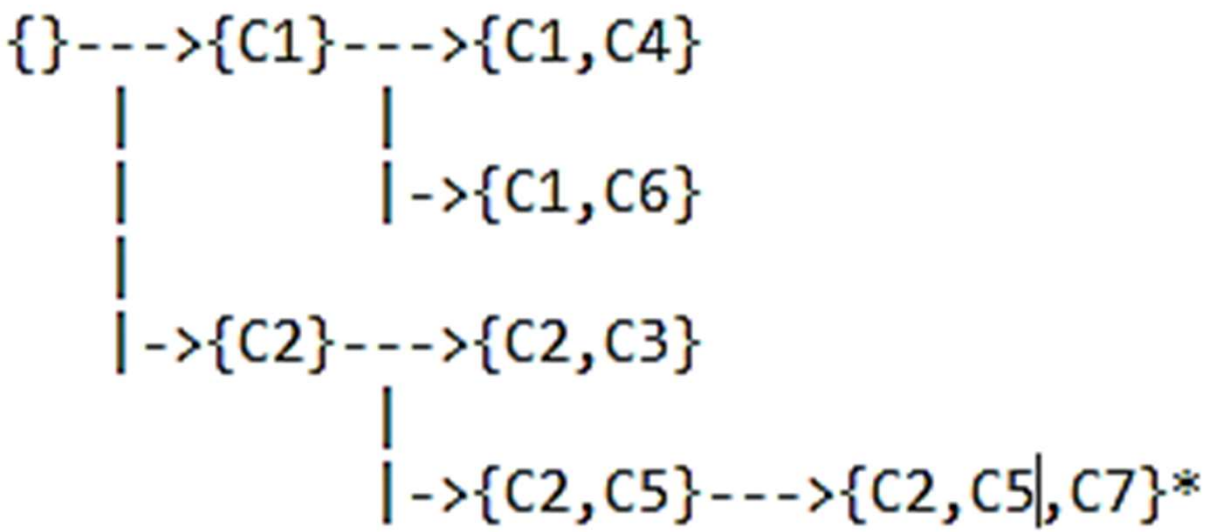
C3={b,e}.

C4={c,h}.

C5={b,f,h}.

C6={c,f,h}.

C7={e,g}



Alternative state spaces for a problem

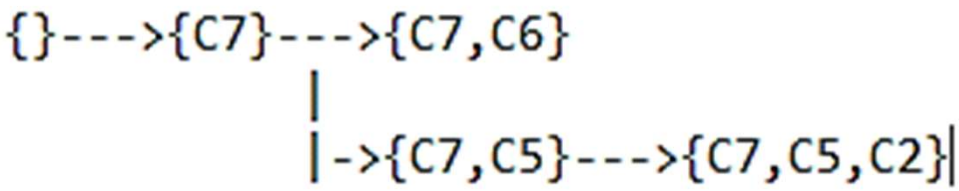
There can be more than one state space for a problem.

For instance, in exact set cover, you could work in backward order. That gives different states.

C1={a,b,g}. C2={a,c,d}. C3={b,e}.

C4={c,h}. C5={b,f,h}. C6={c,f,h}.

C7={e,g}.



Alternative state spaces for a problem

What is critical is this:

Every solution to the problem has to be a goal state that can be constructed by starting at the start state and applying successor operations.

Characteristics of a problem

- Branching factor (B). The maximum number of successors of any state.
- Depth (D). The greatest depth of any node in the state space.
- Size of the state space (S). Certainly not more than $1+B+B^2+B^3+ \ldots +B^D$ is $O(B^D)$, but generally much less.
- Is the state space a tree? A state space is a tree if, for any state S, there is only one way to construct S from the start state. Also known as *systematic* search.
- Is the depth of the shallowest goal known in advance?

Characteristics of our examples

N-Queens: Branching factor: N. Depth: N. Size < N! (a lot less). Tree? Yes. Depth of goal known? Yes, N.

Hamiltonian Path: Branching factor: Maximal outdegree of any vertex. Depth: N. Size: B^N**.** Tree? Yes. Depth of goal known? Yes, N.

Exact Set Cover: Branching factor: Number of sets in C. Depth: Number of sets in C. Size: B^N**.** Tree? Yes.

Depth of goal known? No. Suppose that $|\Omega|=100$ and that C contains sets that range from size 50 to size 4. Then there might be a solution might have only two sets of size 50, or 25 sets of size 4, or something in between.

Breadth-first search

Breadth-first search: First explore all the states at depth 1, then all the states at depth 2, then at depth 3, and so on until a goal is found.

```
bfs() {
    if (goal(START) ) return START;
    FIFO queue Q = [START];
    while (!Q.empty())  {
        S  = Q.pop()
        for (each successor C of S) {
            if (goal(C)) return C;
            Q.push(C);
           }
        } return FAIL;
}
```

Breadth-first search

Breadth-first search only searches until the first goal state is found.

Therefore, if the tree expands exponentially and there is a goal state that is much shallower than the depth of the tree, BFS may run much more quickly than DFS.

Suppose that the tree has depth D, there is one goal state at depth G on the far right side of the tree and G < D.

Then the number of states constructed in DFS is $1+B+B^2+\ldots+B^D = O(B^D)$.

The number of states constructed in BFS is $1+B+B^2+\ldots+B^G = O(B^G)$.

But the memory required in a DFS is only D, the maximum length of a path from the start state to the current state.

The memory required in a BFS is the maximum length of the queue, which is the width of the tree: $O(B^G)$.

In particular, some state spaces have infinite depth.

If there is a goal state, BFS will eventually find it.
DFS may go off onto some infinite wrong path.

Iterative deepening: The best of both worlds

```
ids() {
    for (i=0 to d) {
        ans = do a DFS to depth i;
        if goal(ans) return ans;
      }
    return FAIL;
}
```

First, construct the whole tree to depth 0. Then construct the whole tree to depth 1. Then to depth 2, then to depth 3, etc. When you get to depth G, you will find the shallowest goal.

Iterative Deepening

Time requirement: The time to do a DFS to depth i is B^i. So the time for the IDS is $1+B^1+B^2+\ldots B^G$, which is $O(B^G)$.

Memory requirement: The depth-first search to depth G requires only memory O(G).

It's quite counter-intuitive, which is why it was two decades before anyone (Richard Korf, 1985) discovered it, because at each iteration you're simply throwing away everything you've done already, and redoing it from scratch.

The intuition is as follows: In an exponentially growing tree with B significantly larger than 1, almost all the nodes are leaves. There are many more nodes at level G than in all the shallower levels combined (almost B times as many). So repeating the shallower levels multiple times has negligible cost; almost all the work is done at level G.

Comparison of search methods

B = branching factor.
D = depth of state space.
G = depth of shallowest goal.

	DFS	BFS	IDS
Running Time	$O(B^D)$	$O(B^G)$	$O(B^G)$
Memory	$O(D)$	$O(B^G)$	$O(G)$

State spaces that are not trees

Many state spaces are not trees. E.g. Rubik's cube. 15-puzzle. Moving blocks around on a table. A state is a configuration, and there are multiple ways to get from one configuration to another. There are also cycles. (In this kind of state space, you generally speak of "neighbors" rather than "successors".)

If the state space is not a tree, and you're doing blind search, then you *have* to keep track of all the states that you've generated; otherwise, you end up repeating the same state, generally exponentially many times, or even infinitely many times. So you're best off using a modified BFS.

BFS for state spaces that are not trees

```
bfs()  {
   if goal(START) return START;
   FIFO queue Q = [START]
   Hashtable H = {START};
   while (!Q.empty()) {
         S = Q.pop();
         for (each neighbor N of S) {
             if (goal(N)) return N;
             if (!H.get(N)) {
                 H.add(N);
                 Q.push(N);
               }
          }
       } return FAIL;
 }
```