Lecture 3 CS6800 Artificial Intelligence:

- Search Strategies for Production Systems
- Backtracking Strategies
- Cycle Avoiding Backtracking Strategies
- Graph-Search Strategies

Search Strategies

Tentative control strategies are also called search strategies because at any instant we do not know precisely which rule we should apply to the database. For the next few days we will look at different varieties of search strategies, and their properties.

Backtracking Strategies

We start with an empirical observation:

For problems requiring only a small amount of search, backtracking control strategies are often perfectly adequate and efficient.

We will now consider a recursive procedure that captures the essence of the operation of a production system under backtracking control.

A Backtracking Procedure

The procedure BACKTRACK takes one argument, DATA, the initial global database.

Upon successful termination, the procedure returns a list of rules that if applied in sequence to the initial database, produces a database satisfying the termination condition.

Recursive procedure BACKTRACK(*DATA*)

1. **If TERM**(*DATA*), **return** *NIL*;
2. **If DEADEND**(*DATA*), **return** *FAIL*;
3. *RULES* ← **APPRULES**(*DATA*);
4. LOOP: **if NULL**(*RULES*), **return** *FAIL*;
5. *R* ← **FIRST**(*RULES*);
6. *RULES* ← **TAIL**(*RULES*);
7. *RDATA* ← **R**(*DATA*);
8. *PATH* ← **BACKTRACK**(*RDATA*);
9. **if** *PATH* = *FAIL*, **go** LOOP;
10. **return CONS**(*R*, *PATH*);

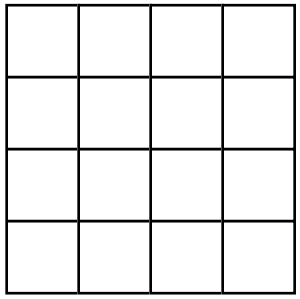
Backtracking and the 4 Queens!

We will now apply this procedure to the 4-queens problem. We must therefore specify the **TERM** and **DEADEND** predicates, and the **APPRULES** function.

The procedure BACKTRACK specifies the control strategy, we must also specify the production rules and termination condition.

The 4-queens problem is akin to the 8-queens problem. It consists of placing a queen in each row of a 4x4 chessboard so that none can capture any other.

The 4-Queens Problem



Consider the following board:

The initial database will consist of an empty board.

To satisfy the termination condition, we must have modified the database with four queens such that none can capture any other.

The production rules:

For $1 \leq i, j \leq 4$:

$R_{i,j}$:

Precondition:

$i = 1$: There are no queens in the array

$1 < i \leq 4$: There is a queen in row $i - 1$

Effect:

Puts a queen in row *i*, column *j*.

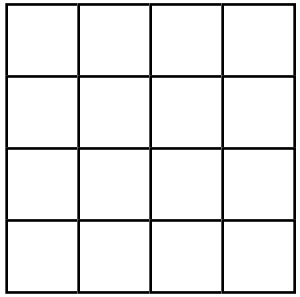
DEADEND, APPRULES

We will define the predicate **DEADEND** so that it is satisfied for databases with queen marks in mutually capturing positions.

We will at this time, specify the **APPRULES** by saying that the $R_{i,j}$ comes before $R_{i,k}$ if $j < k$

To start we must begin with the top row. Why?

So we order the rules for the first row. $R_{1,1}$, $R_{1,2}$, $R_{1,3}$, $R_{1,4}$



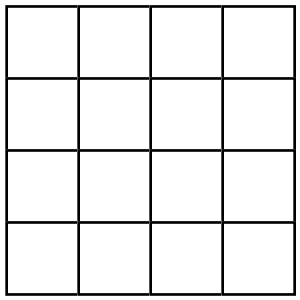
So we choose $R_{1,1}$ to get:

This brings us to step 8 in our algorithm.

Now what rules are applicable?

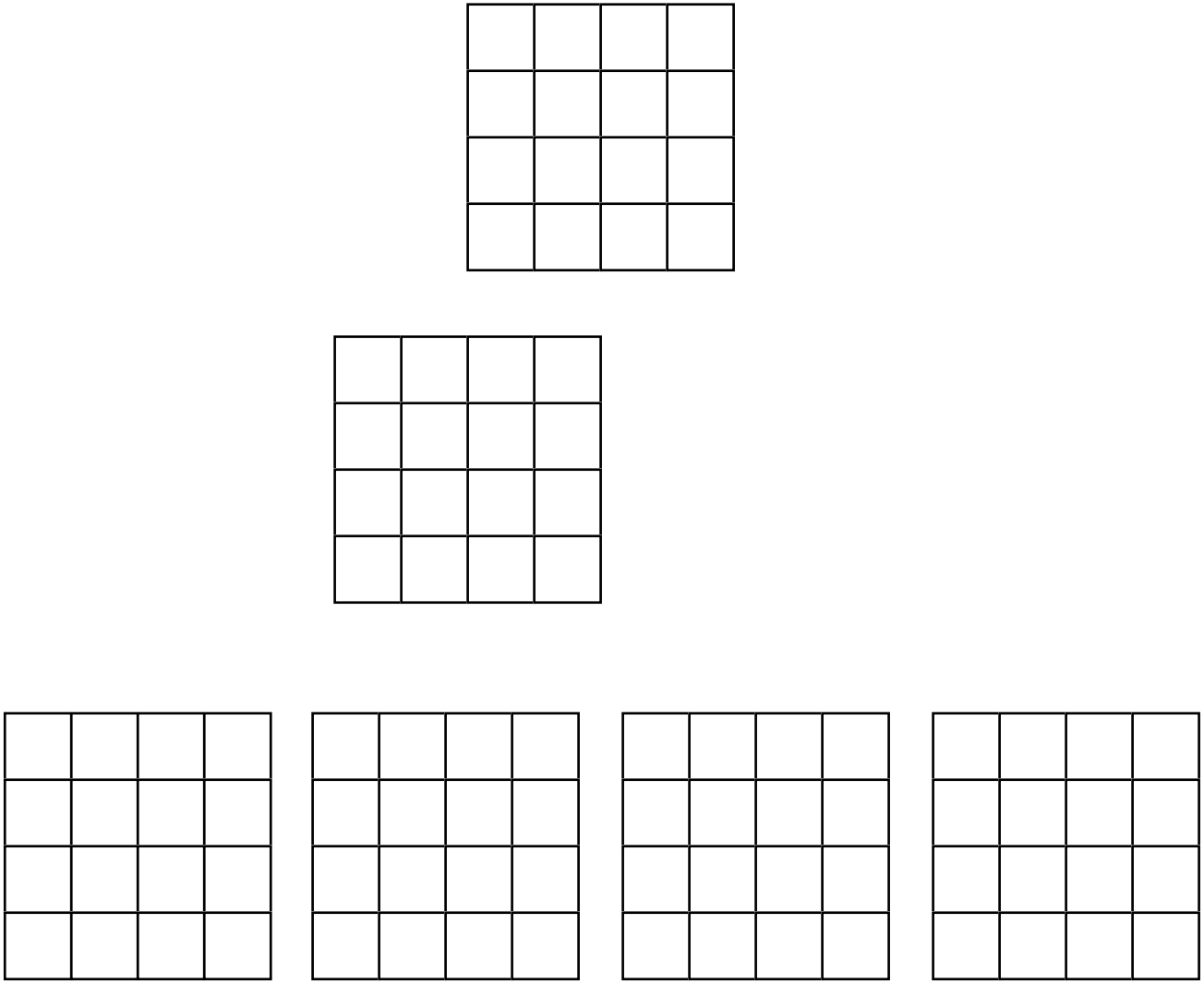
Which rule will be chosen?

What is the resultant state?



What will happen next?

Graphically:



Efficiency of Uninformed Backtracking

How efficient do you think this method is?

How often will backtracking occur?

It occurs 22 times before a solution is found.

Backtracking must go all the way back to the start node, and change the first rule to $R_{1,2}$.

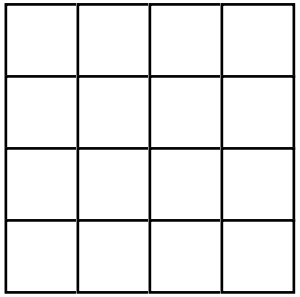
Why does this backtracking occur?

Heuristics

Is there a better way of ordering the rules to be considered?

What might make a good heuristic?

Well, first lets consider what caused the problems in the first case.



A good heuristic

We can associate with any rule $R_{i,j}$ a constant $d_{i,j}$ that is the length of the longest diagonal passing through the cell (i, j).

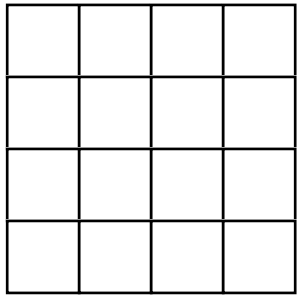
What does this quantity measure?

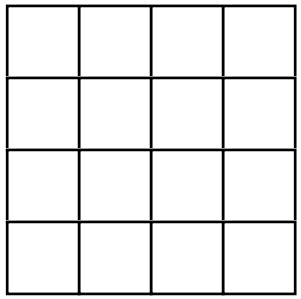
We will order the rules according to:

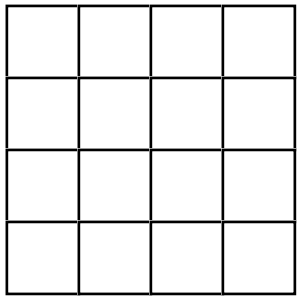
A rule $R_{i,j}$ will be applied before another rule $R_{i,k}$ if:

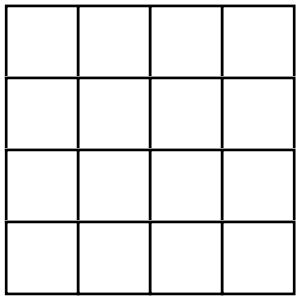
$d_{i,j} < d_{i,k}$

Lets now look at the power of this heuristic.









Heuristic Search

Cycle-avoiding backtracking algorithm

Remember the 8-puzzle. One condition for backtracking was our creating a state that was previously occurring on the solution path.

The procedure BACKTRACK is not capable of detecting the occurrence of such cycles in the database. To avoid cycles, a backtracking procedure must check the current database with all databases on the solution path. To accomplish this, the entire chain of databases must be an argument to the procedure. The following procedure also incorporates a ***depth bound***.

Procedure BACKTRACK1(*DATALIST*)

1. *DATA* ← **FIRST**(*DATALIST*);
2. **if MEMBER**(*DATA*, **TAIL**(*DATALIST*)), **return** *FAIL*;
3. **If TERM**(*DATA*), **return** *NIL*;
4. **If DEADEND**(*DATA*), **return** *FAIL*;
5. **if LENGTH**(*DATALIST*) > *BOUND*, **return** *FAIL*;
6. *RULES* ← **APPRULES**(*DATA*);
7. LOOP: **if NULL**(*RULES*), **return** *FAIL*;
8. *R* ← **FIRST**(*RULES*);
9. *RULES* ← **TAIL**(*RULES*);
10. *RDATA* ← **R**(*DATA*);
11. *RDATALIST* ← **CONS**(*RDATA, DATALIST*);
12. *PATH* ← **BACKTRACK1**(*RDATALIST*);
13. **if** *PATH* = *FAIL*, **go** LOOP;
14. **return CONS**(*R, PATH*);

Graph-Search Strategies

In backtracking control, the system effectively forgets any trial paths that end in failures. Only the path currently being extended is stored explicitly.

A more flexible procedure would involve the explicit storage of all trial paths so that any of them could be candidates for further extension.

Consider the following case:

Graph-Search

In order to achieve this kind of flexibility, a control system must keep explicit track of a graph of databases linked by rule applications. Control schemes that use this approach are called ***graph-search*** strategies.

We can think of a graph-search control strategy as a means of finding a path in a graph from a node representing the initial database to one representing a database that satisfies the termination condition of the production system.

Graph Notation

- A graph consists of a set of ***nodes***.
- Certain pairs of nodes are connected by ***arcs***.
- In a ***directed graph,*** these arcs are ***directed*** from one member of a pair to another.

Terminology cont.

- If an arc directed from node n_i to node n_j, then node n_i is said to be a ***parent*** of node n_j and node n_j is said to be a ***successor*** of n_i.
- A ***tree*** is a special case of a graph in which each node has at most one parent.
- A node in the tree having no parent is the ***root node***.
- A node in the tree having no successors is called a ***tip node*** or ***terminal*** node.
- The root node is defined to be at ***depth*** zero.
- The depth of any other node in the tree is defined to be the depth of its parent plus 1.

Graph node terminology

- A sequence of nodes $(n_{i1}, n_{i2}, \ldots, n_{ik})$, with n_{ij} a successor of $n_{i,j-1}$ for $j = 2, \ldots, k$, is called a ***path of length*** $k-1$ from node n_{i1} to node n_{ik}.

- If a path exists from node n_i to node n_j, then node n_j is said to be ***accessible*** from node n_i.

Graph terminology cont.

- If n_j is accessible from node n_i, then node n_j is a ***descendent*** of node n_i, and node n_i is an ***ancestor*** of node n_j.
- For our purposes, nodes correspond to databases, and arcs to rules. The problem of finding a sequence of rules transforming one database into another is equivalent to finding a path in the graph.

