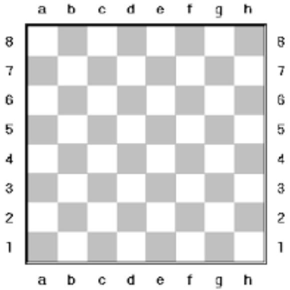
Backtracking

Class 35

Chessboard



- how many queens can you put on a chessboard so that no queen attacks another queen's position?
- obviously at least 1
- obviously fewer than 9 (why?)
- what is the maximum number?
- this is called the *n*-queens problem

n-Queens

- the n-queens problem can be phrased as a search problem
  - find how to place n queens on an $n \times n$ board
- or as a decision problem (yes or no answer)
  - can n queens be placed on an $n \times n$ board?
- or as an optimization problem
  - what is the largest number of queens that can be placed on an $n \times n$ board?

- there are no known greedy, divide-and-conquer, or dynamic programming algorithms that can solve this problem (actually that's not quite true, but pretend it's true)
- what do we do when all of our fancy algorithm techniques fail?

Strategy

- when all else fails, we normally resort to brute force
- brute force, or exhaustive search:
  try all possible combinations of potential solutions and check each one until a solution is found (or there's no solution)
- brute force has no intelligence — just try every combination of n queens on n^2 squares

$$\binom{64}{8} = 4,426,165,368$$

- try every one of the 4.4 billion arrangements and see if one of them works

- but often we can be smarter than that
- even when combinations are involved

Backtracking

- backtracking is an algorithm strategy that is a variation of brute force
- backtracking ranges over the entire all-possible-combinations search space, but does so with intelligence
- the all-possible-combinations search space is represented as a search tree

- the algorithm starts at the root of the search tree
- every leaf node is either a dead end or a solution
- backtracking is a depth-first traversal of the search tree
- but instead of doing a DFS of the entire tree, we prune as much of the tree as possible, shrinking the size of the search tree

Backtracking

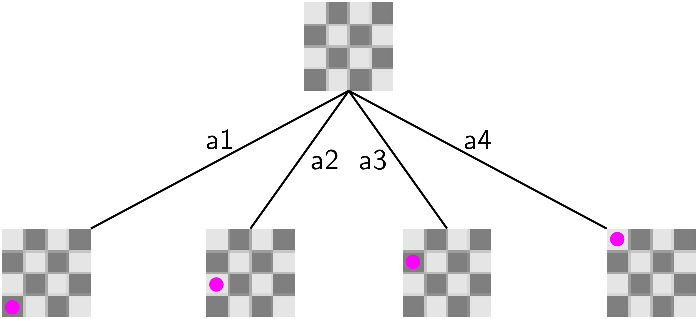
- we want the search tree to be logically organized
- to make sure we don't miss any possible cases
- backtracking always requires ordering the possible cases or choices

- for n-queens, there are various orderings we could use
- we will go in order
  1. column-by-column, a – h
  2. row-by-row, 1 – 8

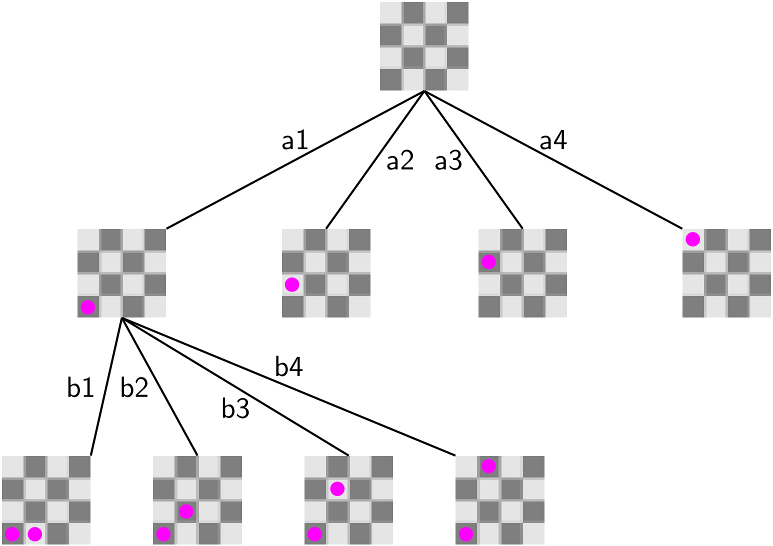
n-Queens Search Tree

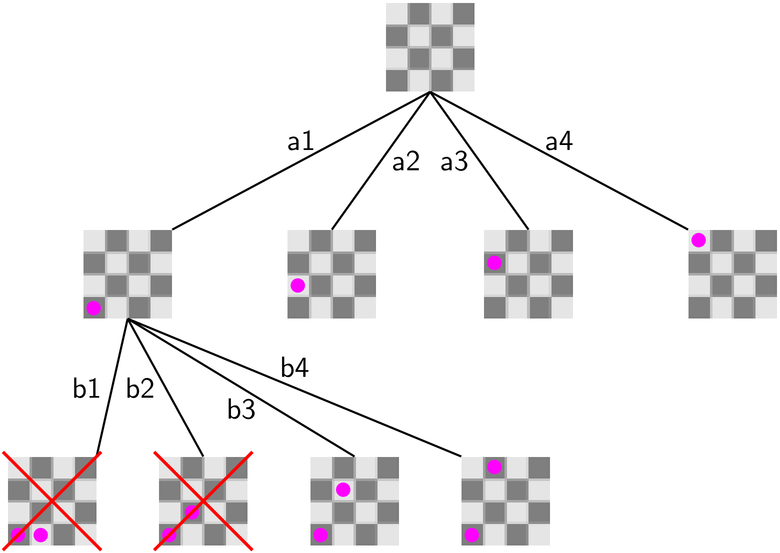
- at the root of the tree, no queens are on the board
- at one level down from the root, one queen is on the board
- at two levels down, two queens on the board
- if we reach level eight, there are eight queens on the board, and we have a solution

- there are 8 columns, and one queen must go in each column
- on level one we place a queen in column a
- on level two, we place a queen in column b
- if we reach column h with no conflicts, we have a solution



- once we try a placement, we test it
- if the test reveals a solution, we're done
- if the test reveals a conflict, we prune the search tree here and undo any actions at this level
- otherwise, we accept this partial solution and continue down the tree


a1
a2
a3
a4
b1
b2
b3
b4


a1
a2
a3
a4
b1
b2
b3
b4

Size of the Search Tree

- the search space for 4 queens has

row	nodes in row	running total
0	1	1
1	4	5
2	16	21
3	64	85
4	256	341

The total number of nodes in the tree is

$$\sum_{i=0}^{n} n^i = \frac{n^{n+1} - 1}{n - 1} \in \Theta(n^n)$$

Search Space

number of search tree nodes $\in \Theta(n^n)$

- this is the bad exponential
- worse than factorial
- true brute force would look at every node, clearly impossible
- the whole point of backtracking is to search as intelligently as possible

Backtracking General Structure

```
place the original problem onto stack s
while (!s.empty())
{
  problem p = s.pop()
  expand p into subproblems p1,...pk
  foreach pi
  {
    if (test(pi)) succeeds
    {
      announce success and halt
    }
    else if (test(pi) fails)
    {
      discard pi
    }
    else
    {
      s.push(pi)
    }
  }
}
announce failure
```

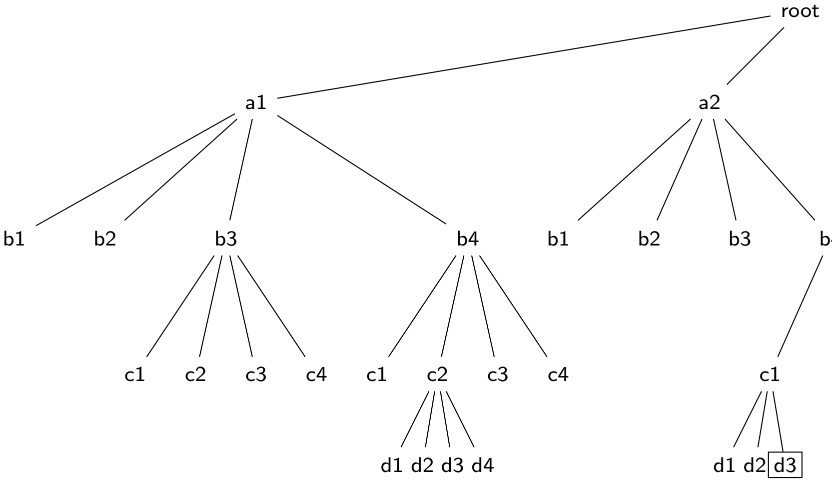
The Search Tree

- the previous slides were inaccurate in depicting the search tree
- DFS in a backtracking search tree is different from DFS in a normal graph
- a normal graph already exists, but a backtracking search tree is a concept
- a backtracking search tree does not exist in memory as adjacency lists

Node Elaboration

- the root node of the search tree is the starting point
- no move has yet been made
- for *n*-queens, the first move is a1
- in the diagrams above, I depicted a1, a2, a3, and a4 all on the first level
- but if a solution is found under a1, a2 is never reached
- a2 is only elaborated if a1 is a total dead end

Elaborated Nodes



Backtracking

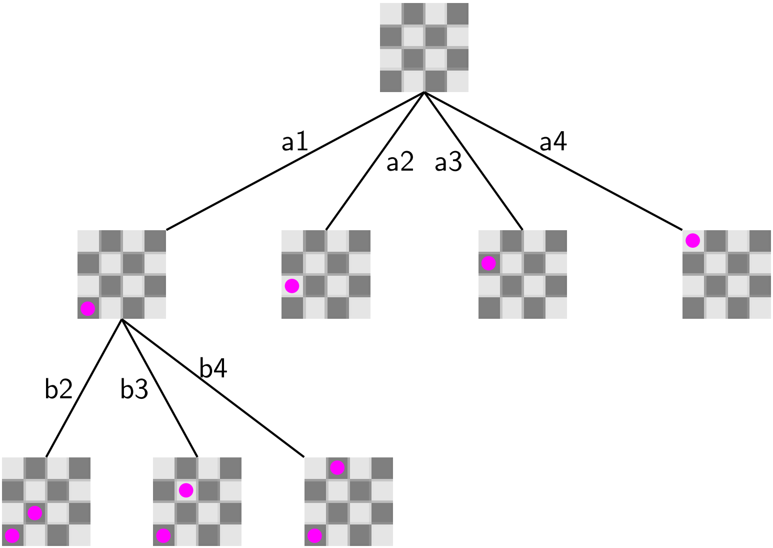
- there are four main things we can do to improve backtracking
  1. choose an ordering that places the solution as far left in the search tree space as possible
  2. choose an ordering that produces as small a search tree as possible
  3. choose a test function that prunes non-promising paths as high in the tree as possible
  4. choose a test function that is as efficient as possible
- each of these may or may not apply, given the specific problem

Ordering Solution at Left

- this typically only applies to problems whose search tree is asymmetrical
- does not apply to n-queens
- there are many solutions to n-queens due to rotation and symmetry
- they are evenly distributed left-to-right all across tree

Efficient Ordering

- once we have placed a queen in row 1 in a previous column, it's silly to place a queen in row 1 in a subsequent column
- instead of a fan-out of 4 in every row of the search tree, we could have a fan-out of 4 at level 1, 3 at level 2, etc.


a1
a2
a3
a4
b2
b3
b4

Smaller Search Tree

- this gives us a search tree of 65 nodes rather than 341
- number of search tree nodes $\in \Theta(n!)$
- a vast improvement over n^n
- still a huge number
- there's a trade-off in complexity vs improvement

Test Function Intelligence

- the smarter we are about realizing this path cannot lead to a solution, the sooner we can prune the path
- what is the algorithm to determine if any two queens on a board conflict?
- another trade-off in complexity vs improvement

Test Function Efficiency

- related to test function intelligence
- the test function gets called on every node visited
- that's a lot of function calls
- the efficiency of the function is crucial
- detecting conflicts among queens on a chessboard is an excellent example
- what's a good algorithm?