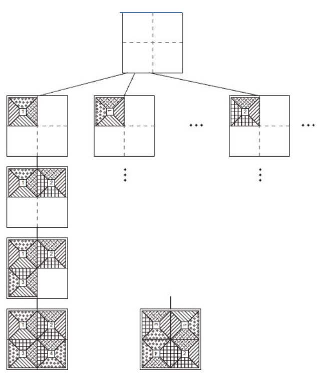
Search: Basic Algorithms

CSE 415: Introduction to Artificial Intelligence
University of Washington
Spring, 2017

Outline

- Combinatorics of the Painted Squares Puzzle
- Recursive Depth-First Search
- Graph Search
- Iterative Depth-First Search
- Breadth-First Search
- Iterative Deepening
- Graphs with Edge Costs
- Uniform-Cost Search
- Heuristics and Best-First Search
- A* Search

Tree of States for a 2x2 Painted Squares Puzzle



Combinatorics of the Painted Squares Puzzle

Consider placements to be unconstrained.

Branching factor:

$$b = n\_pieces\_left \cdot n\_places\ left \cdot n\_orientations$$

At the root: $b = 4 \cdot 4 \cdot 4 = 64$
At ply 1: $b = 3 \cdot 3 \cdot 4 = 36$
At ply 2: $b = 2 \cdot 2 \cdot 4 = 16$
At ply 3: $b = 1 \cdot 1 \cdot 4 = 4$

Total leaf nodes (including repetitions): 64 · 36 · 16 · 4 = 147,456.
Total nodes: 1 + 64 + 2304 + 36864 + 147456 = 186,689.

Combinatorics of the Painted Squares Puzzle

Number of filled boards using the 4 pieces, allowing violations of the side-matching constraints:

$$n\_permutations \cdot n\_orientations\text{\textasciicircum}n\_pieces$$

$$4! \cdot 4^4 = 24 \cdot 256 = 6144$$

If we constrain piece placements to go to the next available space on the board, then this is the number of leaf nodes.

Note that dividing 147,456 by 4! gives 6144, too.

The Combinatorial Explosion

Assume the branching factor is constant.
Suppose a search process begins with the initial state.

Then it considers each of b possible moves. Each of those may have b possible subsequent moves.

In order to thoroughly look n steps ahead, the number of states that must be considered is
$1 + b + b^2 + \ldots + b^n$.

For $b > 1$, the value of this expression grows exponentially as n increases. This is known as the *combinatorial explosion*.

Ss State-space Search

Recursive Depth-First Method

Current board B ← empty board.
Remaining pieces Q ← all pieces.
Call Solve(B, Q).

```
Procedure Solve(board B, set of pieces Q)
 For each piece P in Q, {
  For each orientation A {
    Place P in the first available
      position of B in orientation A, obtaining B'.
    If B' is full and meets all constraints, output B'.
    If B' is full and does not meet all constraints, return.
    Call Solve(B', Q - {P}).
  }
 }
```


Ss State-space Search

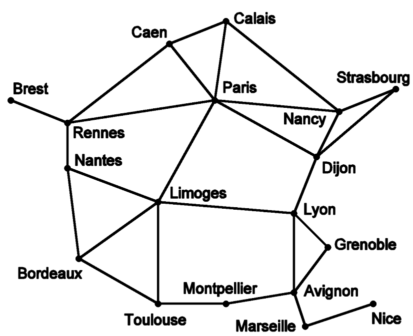
Graph Search

When descendant nodes can be reached with moves via two or more paths, we are really searching a more general graph than a tree.

Depth-First Search: Examine the nodes of the graph by fully exploring the "descendants" of a node before trying any "siblings" of a node.

Ss State-space Search

Sample Graph



Ss State-space Search

Depth-First Search: Iterative Formulation

1. Put the start state on a list OPEN
2. If OPEN is empty, output "DONE" and stop.
3. Select the first state on OPEN and call it S.
   Delete S from OPEN.
   Put S on CLOSED.
   If S is a goal state, output its description
4. Generate the list L of successors of S and delete
   from L those states already appearing on CLOSED.
5. Delete any members of OPEN that occur on L.
   Insert all members of L at the *front* of OPEN.
6. Go to Step 2.

Ss State-space Search

Breadth-First Search: Iterative Formulation

1. Put the start state on a list OPEN
2. If OPEN is empty, output "DONE" and stop.
3. Select the first state on OPEN and call it S.
   Delete S from OPEN.
   Put S on CLOSED.
   If S is a goal state, output its description
4. Generate the list L of successors of S and delete
   from L those states already appearing on CLOSED.
5. Delete any members of OPEN that occur on L.
   Insert all members of L at the *end* of OPEN.
6. Go to Step 2.

Ss State-space Search

Iterative Deepening

We can combine the benefits of Depth FS and Breadth FS.
Instead of BFS, do a sequence of DFS steps, but with a depth limit. Let the depth limit increase by 1 in each step.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   inputs: problem, a problem
   for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

Russell & Norvig

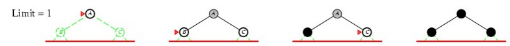
Ss State-space Search

Iterative Deepening
∠ = 0

Russell & Norvig

Ss State-space Search

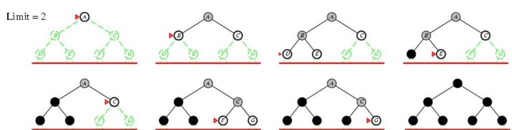
Iterative Deepening
∠ = 1



Russell & Norvig

Ss State-space Search

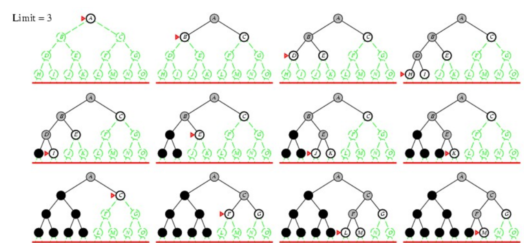
Iterative Deepening
∠ = 2



Russell & Norvig

Ss State-space Search

Iterative Deepening
∠ = 3



Russell & Norvig

Ss State-space Search

Overhead for Iterative Deepening

Repeated work takes place mainly near the root, where there are relatively few nodes.

With b = 2, the overhead is less than a factor of 2. (e.g., 57/31)

Depth	N in level	N in tree	IDDFS
0	1	1	1
1	2	3	4
2	4	7	11
3	8	15	26
4	16	31	57

Ss State-space Search

Search Algorithms

Alternative objectives:
Reach any goal state
Find a short or shortest path to a goal state

Alternative properties of the state space and moves:
Tree structured vs graph structured, cyclic/acyclic
Weighted/unweighted edges

Alternative programming paradigms:
Recursive
Iterative
Iterative deepening
Genetic algorithms

Ss State-space Search

State-Space Graphs with Weighted Edges

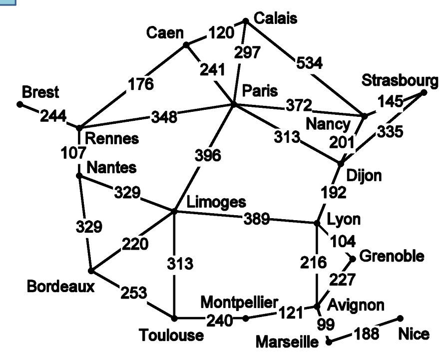
Let S be space of possible states.
Let (s_i, s_j) be an edge representing a move from s_i to s_j.
$w(s_i, s_j)$ is the weight or *cost* associated with moving from s_i to s_j.

The *cost of a path* [$(s_1, s_2), (s_2, s_3), \ldots, (s_{n-1}, s_n)$] is the sum of the weights of its edges.

A *minimum-cost path* P from s_1 to s_n has the property that for any other path P' from s_1 to s_n, cost(P) $\leq$ cost(P').

Ss State-space Search

Graphs with Weighted Edges



Ss State-space Search

Uniform-Cost Search

A more general version of breadth-first search.

Processes states in order of increasing path cost from the start state.

The list OPEN is maintained as a priority queue. Associated with each state is its current best estimate of its distance from the start state.

As a state s_i from OPEN is processed, its successors are generated. The tentative distance for a successor s_j of state s_i is computed by adding $w(s_i, s_j)$ to the distance for s_i.

If s_j occurs on OPEN, the smaller of its old and new distances is retained. If s_j occurs on CLOSED, and its new distance is smaller than its old distance, then it is taken off of CLOSED, put back on OPEN, but with the new, smaller distance.

Ss State-space Search

Heuristics

A heuristic is a “rule of thumb” for operating in unknown, uncertain, or complex environments or problem-solving contexts.

A heuristic evaluation function, in state-space search, is a function h: $S \to \Re^+$ that can be used as an estimate of how close a state is to a goal or simply to prioritize states for attention.

Examples:

Euclidean distance between a city and the goal. (in the routing problem)
Number of pieces not yet placed in a puzzle. (painted squares).
Average distance a puzzle piece (in the 8-puzzle) has to move on the board to get to its destination.
Hot-cold (in a game of Find-the-hidden-object). Hot: close to 0. Cold: much greater than 0.

Ss State-space Search

Best-First Search

Provided we have a heuristic evaluation function, we can prioritize states for expansion using the function.

By changing our iterative formulation of Depth-First Search to use a PRIORITY QUEUE to implement the OPEN list, we get Best-First Search.

Ss State-space Search

Ideal Distances in A* Search

Let f(s) represent the cost (distance) of a shortest path that starts at the start state, goes through s, and ends at a goal state.

Let g(s) represent the cost of a shortest path from the start state to s.

Let h(s) represent the cost of a shortest path from s to a goal state.

Then f(s) = g(s) + h(s)

During the search, the algorithm generally does not know the true values of these functions.

Estimated Distances in A* Search

Let g'(s) be an estimate of g(s) based on the *currently known shortest distance* from the start state to s.

Let the h'(s) be a *heuristic* evaluation function that estimates the distance (path length) from s to the nearest goal state.

Let f'(s) = g'(s) + h'(s)

Best-first search using f'(s) as the evaluation function is called *A* search.*

Admissibility of A* Search

Under certain conditions, A* search will always reach a goal state and be able to identify a shortest path to that state as soon as it arrives there.

The conditions are:
h'(s) must not exceed h(s) for any s.
$w(s_i, s_j) > 0$ for all s_i and s_j.

This property of being able to find a shortest path to a goal state is known as the admissibility property of A* search.

Sometimes we say that a particular A* algorithm is admissible. We can say this when its h' function satisfies the underestimation condition and the underlying search problem involves positive weights.

Search Algorithm Summary

	Unweighted graphs	*Weighted graphs*
blind search	Depth-first Breadth-first	Depth-first Uniform-cost
uses heuristics	Best-first	A*