Ss
State-space Search

Basic Search Algorithms

CSE 415: Introduction to Artificial Intelligence
University of Washington
Winter, 2020

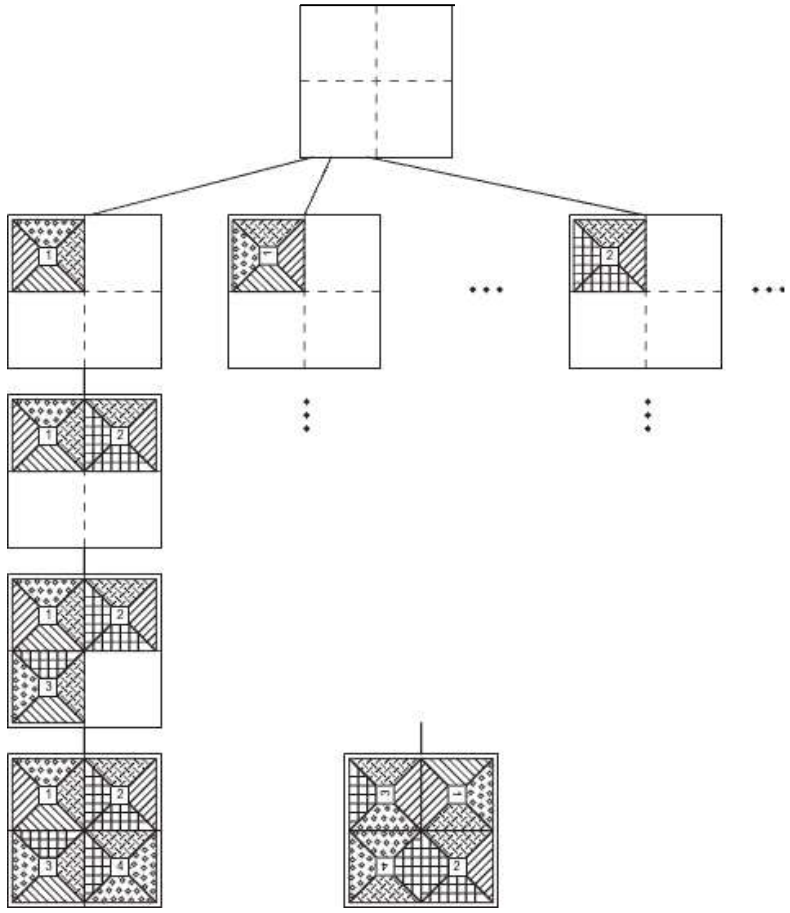
Outline

- 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

Ss
State-space Search

Painted-Squares Puzzle

In the previous lecture, we introduced a simple puzzle to illustrate tree search. Now let's formulate the search algorithm.



Tree of states for a 2x2 Painted Squares puzzle

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}).
    }
  }
```

*Sometimes covered in CSE 143 or CSE 373. Also known as Recursive Backtracking Search.

Ss
State-space Search

Graph Search

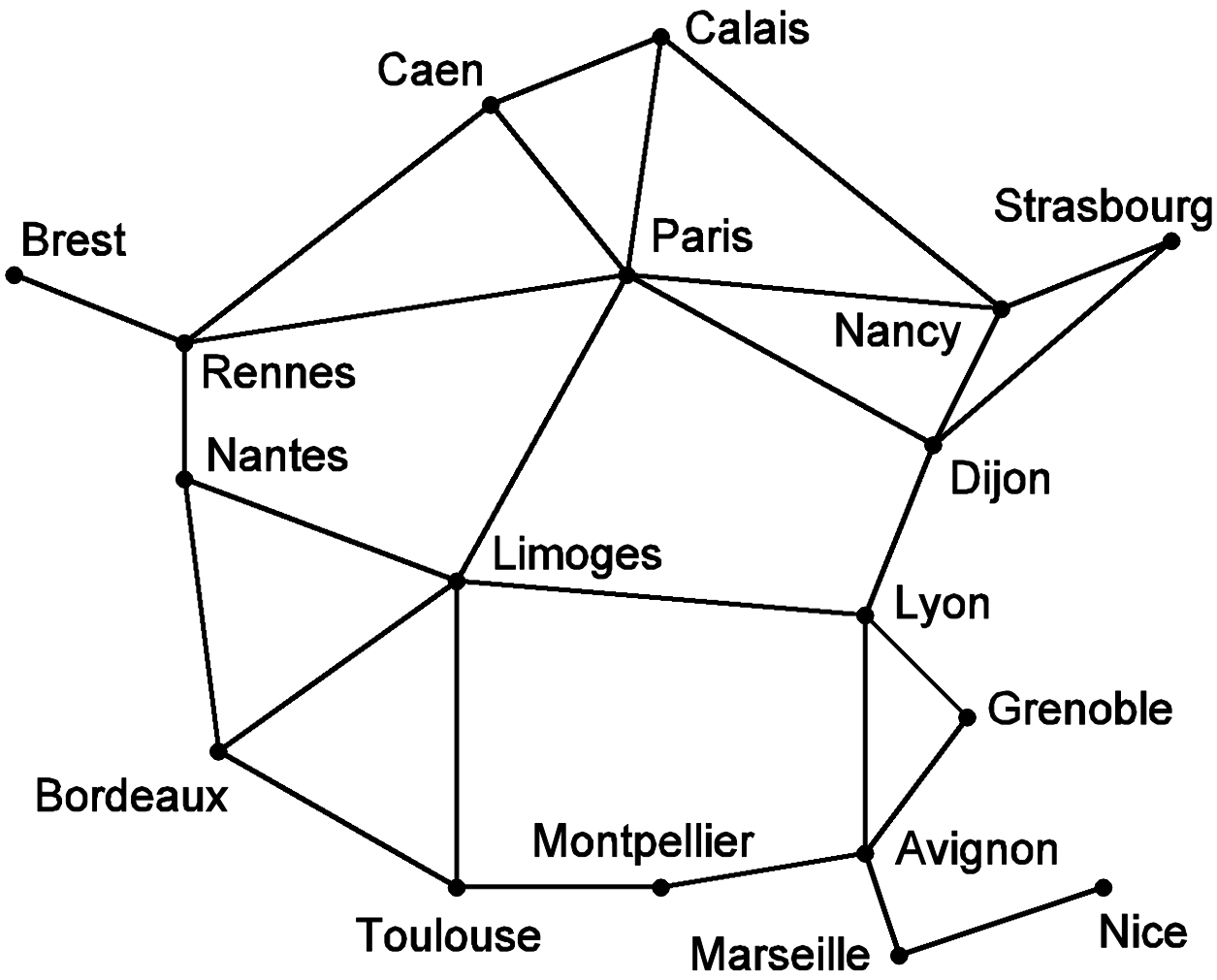
When descendant nodes can be reached with moves via two or more paths, we are really searching a more general graph than a tree.

Whether we are in a tree or a more general graph, the basic idea of depth-first search is the same... do not backtrack until all descendants have been tried.

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 from OPEN any members of OPEN that occur on L.

   Insert all members of L at the *front* of OPEN.

6. Go to Step 2.

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 from L 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

Graph Search vs. Tree Search

Note that these 2 formulations are for searching general graphs.

If we know we are doing tree search, then we don't need to worry about deleting anything in steps 4 and 5, and we don't need a CLOSED list.

Ss
State-space Search

Comparing DFS and BreadthFS

Optimality of the solution path:

When BreadthFS arrives at a goal node, a shortest (i.e., optimal) path is easily extracted by "backtracing" (not backtracking).

However, DFS may arrive at a goal node via a very non-optimal path, and the shortest path is *not* readily available at that point.

Memory Utilization:

BreadthFS usually requires a large amount of memory. For example, in a tree with branching factor b, the OPEN list will require storage $O(b^d)$, where d is the depth of the closest goal node.

On the other hand, DFS requires only O(bd) storage for its OPEN list.

(No CLOSED lists are needed if we know that the problem-space graph is acyclic.)

Ss
State-space Search

Iterative Deepening DFS

We can combine the benefits of DFS and BreadthFS to get optimal paths without huge memory requirements.
Instead of regular BreadthFS, we do a sequence of DFS executions, but with a depth limit for each execution. We make the depth limit increase by 1 in each execution, starting from 0.

```
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

Is Iterative Deepening DFS Efficient?

IDDFS saves memory, but at the expense of extra time, due to repeated searching of the upper tree levels.

Assuming we are searching a tree with an average branching factor b > 1, then an analysis of the number of repeat visits to nodes leads us to conclude "Yes, it's efficient."

The repeated work is bounded by a small factor (effectively constant, because it's independent of the size of Σ).

Russell & Norvig

Ss
State-space Search

Iterative Deepening DFS Example ∠ = 0

Limit = 0

Russell & Norvig

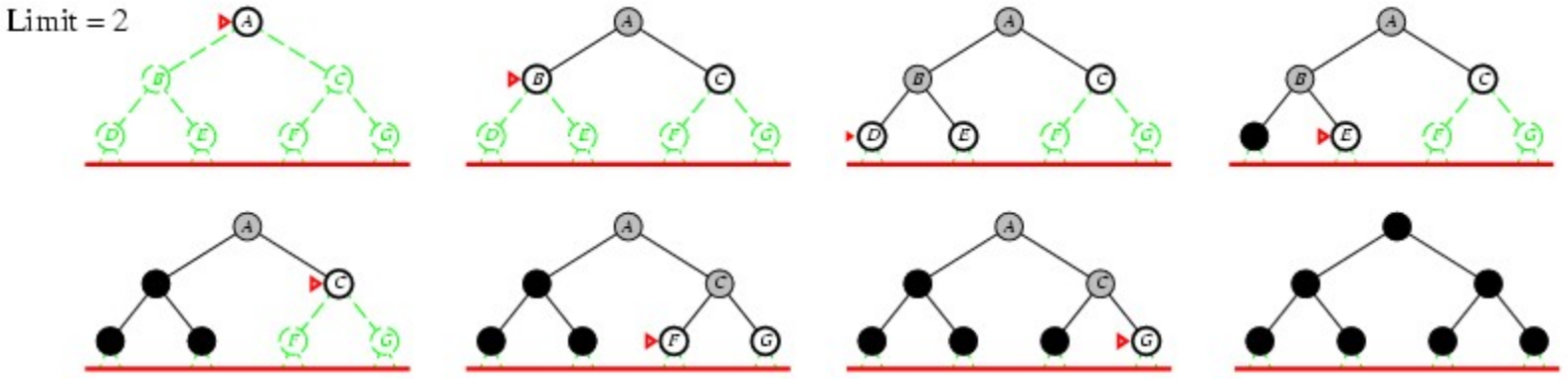
Ss
State-space Search

IDDFS Example (cont.)
∠ = 1

Russell & Norvig

Ss
State-space Search

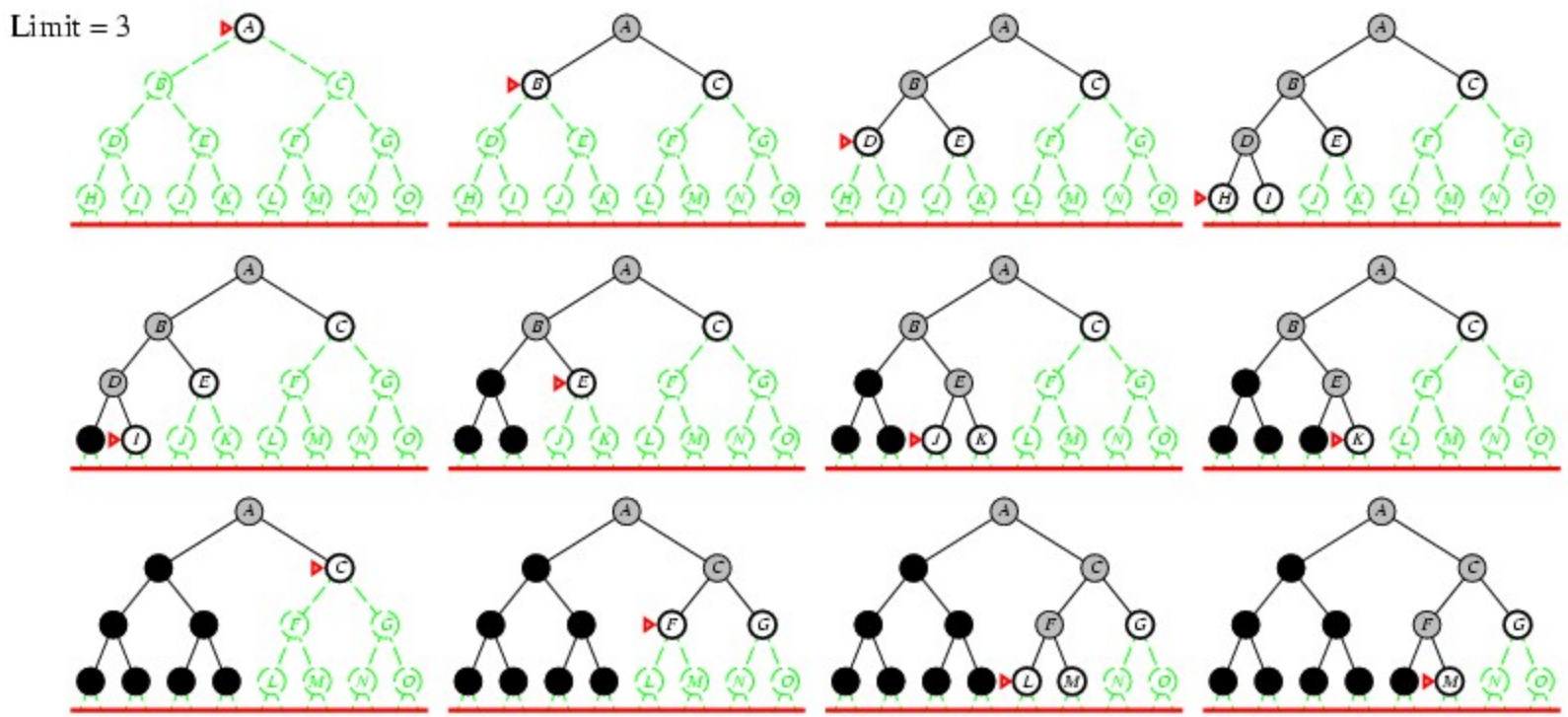
IDDFS Example (cont.)
$\angle = 2$



Russell & Norvig

Ss
State-space Search

IDDFS Example (cont.)
∠ = 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

When to Use Iterative Deepening

a. The memory-saving case for IDDFS is often clearer for *tree search* than graph search, if cycles in the graph are possible. (The amount of memory used to save information about previously visited states may approach that needed for Breadth-First Search.)

b. Higher branching factors tend to favor IDDFS. (Even 2 is good, and higher factors mean relatively less repeated work. But 1 is bad, leading to worst case $\theta(n^2)$ vs $\theta(n)$ time performance vs ordinary DFS or BFS.)

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

Problem-Space Graphs with Weighted Edges

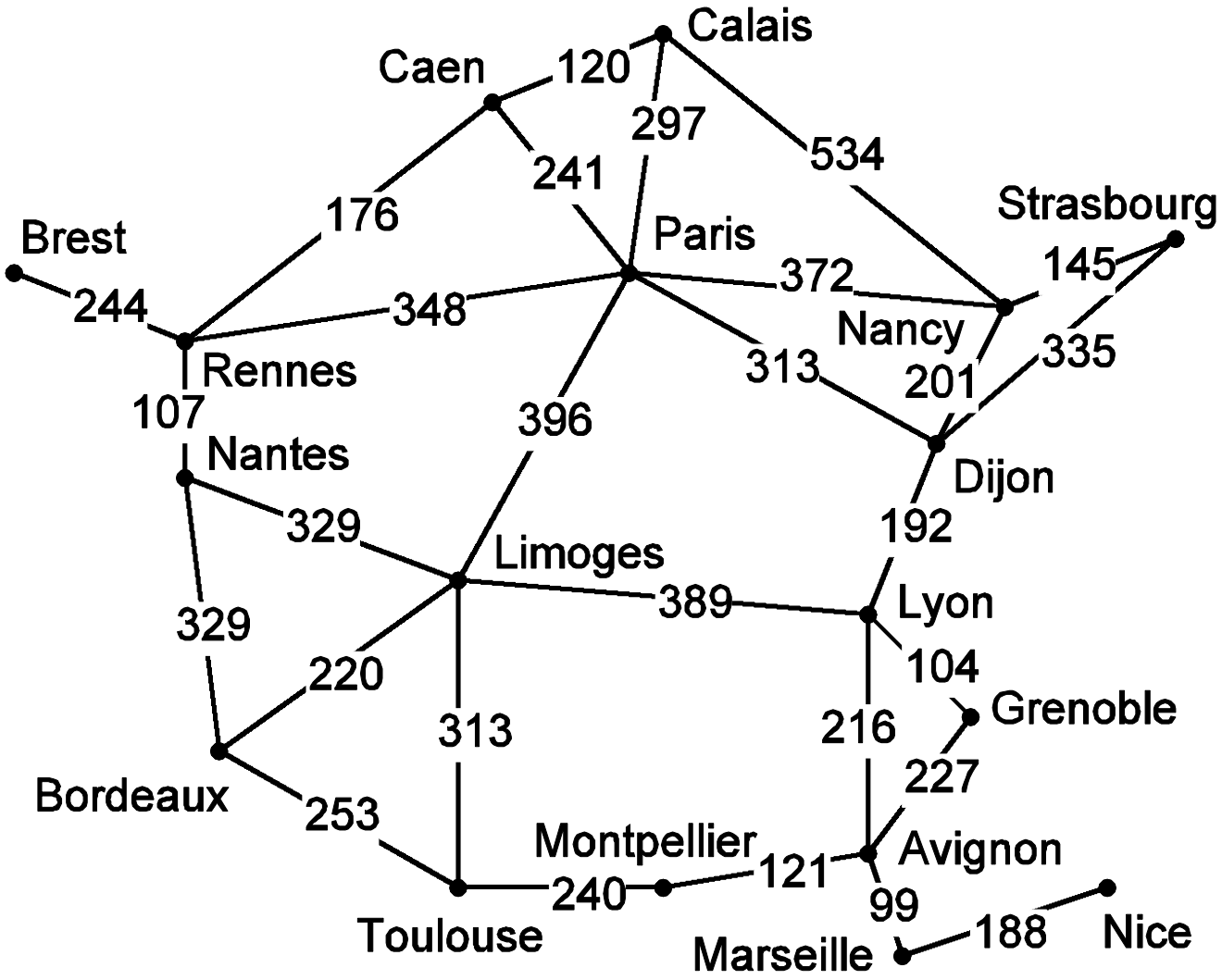
Let Σ be space of possible states, and S be the corresponding set of nodes.
Let (s_i, s_j) be an edge representing a move from σ_i to σ_j.
$w(s_i, s_j)$ is the weight or *cost* associated with moving from σ_i to σ_j.

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, $\text{cost}(P) \leq \text{cost}(P')$.

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 node.

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

As a node s_i from OPEN is processed, its successors are generated. The tentative distance for a successor s_j of node 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: $\Sigma \rightarrow \Re^+$ 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.

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 node, goes through s, and ends at a goal node.

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

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

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

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

Ss
State-space Search

Estimated Distances in A* Search

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

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

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

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

Ss
State-space Search

Admissibility of A* Search

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

The conditions are:

h′(s) must not exceed h(s) for any s. *"The heuristic must be admissible."*

$w(s_i, s_j) > 0$ for all s_i and s_j. *"Costs must be positive."*

a goal state must be reachable from the starting state. *"Reachability"*

The heuristic h' is admissible if and only if it satisfies $0 \leq h'(s) \leq h(s)$, for all nodes s.

Sometimes we say that a particular A* algorithm is admissible. We can say this when its h' function satisfies the admissibility condition.

Ss
State-space Search

Consistency of A* Search

Under an even stronger condition on the heuristic than admissibility, A* search doesn't need to use a CLOSED list.

The condition has two parts:
For every edge (s_i, s_j) in G, $h'(s_i) - h'(s_j) < w(s_i, s_j)$,
and
$h'(s) = 0$, if s is a goal node.

The heuristic h' is *consistent* if and only if it satisfies these constraints.

(A consistent heuristic is sometimes also called a "monotone" heuristic.)

Search Algorithm Summary

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