Intelligent Agents

Curtis Larsen

Utah Tech University—Computing

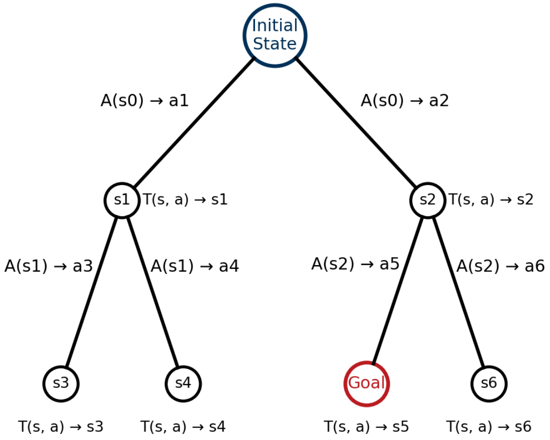
Fall 2025

Problem Formulation Recap

A search problem is defined by 5 components:

1. **Initial state:** s_0
   (the starting point of the search)
2. **Actions:** $A(s) \rightarrow \{a_1, a_2, \ldots\}$
   Returns the set of possible actions in state s
3. **Transition model:** $T(s, a) \rightarrow s'$
   Returns the resulting state when action a is applied in state s
4. **Goal test:** $G(s) \rightarrow \{\text{true}, \text{false}\}$
   Checks whether state s is a goal state
5. **Path cost:** $C(s, a, s') \rightarrow \mathbb{R}_{\geq 0}$
   Assigns a numeric cost to the step from s to s' via a

Search Tree: Actions and Transitions

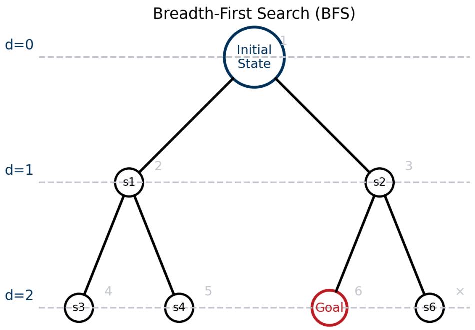


From Formulation to Algorithms

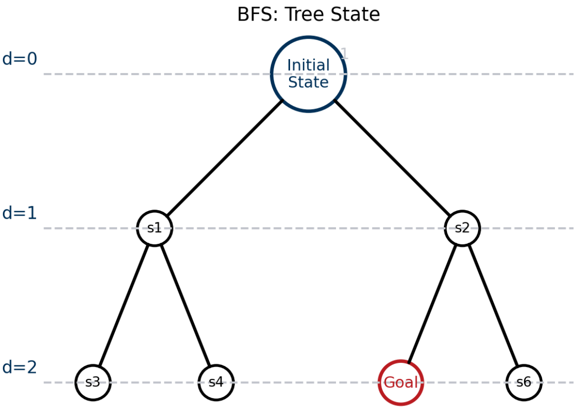
- Now that we know how to define a search problem...
- Let's look at systematic strategies for exploring the state space.
  - Breadth First Search (BFS)
  - Uniform Cost Search (UCS)
  - Depth First Search (DFS)
  - Depth Limited Search (DLS)
  - Iterative Deepening Search (IDS)

Breadth-First Search (BFS): Intuition

- Expand shallowest nodes first.
- Explore all nodes at depth d before $d + 1$.

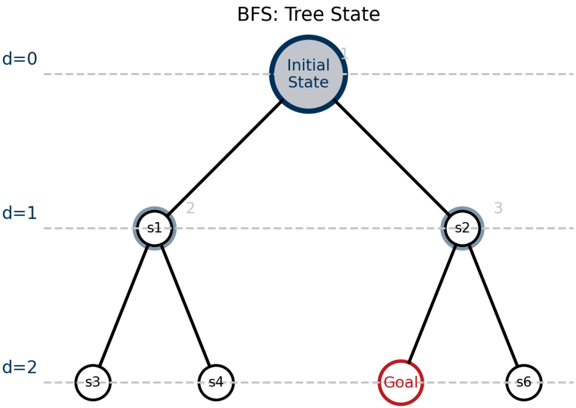


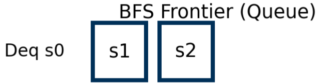
BFS: Step 0


BFS: Tree State
d=0
d=1
d=2
Initial State
s1
s2
s3
s4
Goal
s6

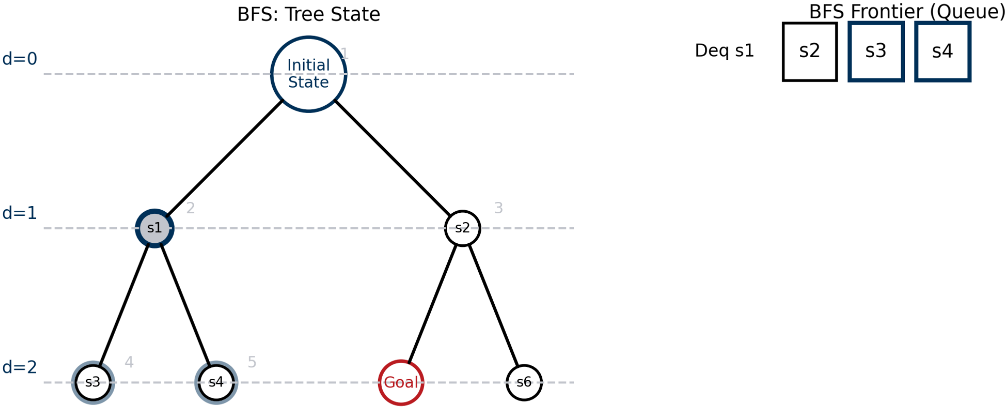

BFS Frontier (Queue)
Init
s0

BFS: Step 1

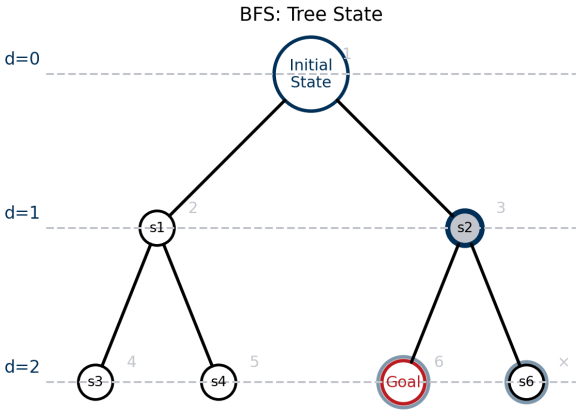

BFS: Tree State
d=0
d=1
d=2
Initial State
1
s1
2
s2
3
s3
s4
Goal
s6

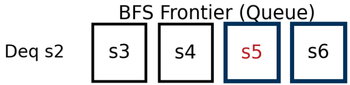

BFS Frontier (Queue)
Deq s0
s1
s2

BFS: Step 2

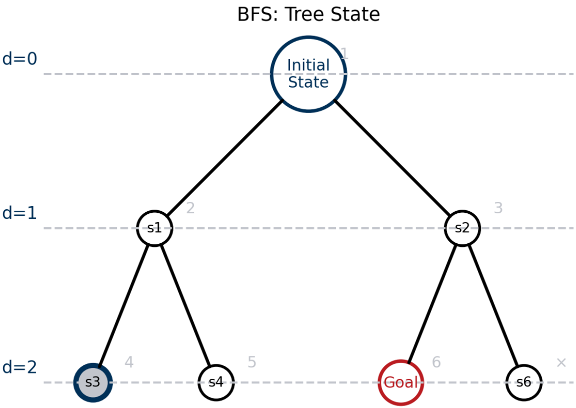

BFS: Tree State
d=0
d=1
d=2
Initial State
1
s1
2
s2
3
s3
4
s4
5
Goal
s6
BFS Frontier (Queue)
Deq s1
s2
s3
s4

BFS: Step 3


BFS: Tree State
d=0
d=1
d=2
Initial State
1
s1
2
s2
3
s3
4
s4
5
Goal
6
s6
×

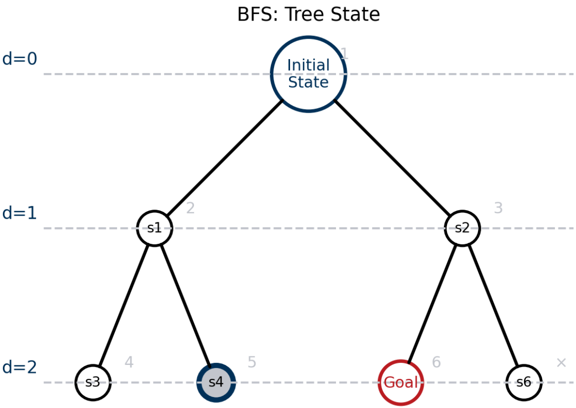

BFS Frontier (Queue)
Deq s2
s3
s4
s5
s6

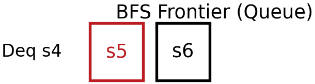
BFS: Step 4


BFS: Tree State
d=0
Initial State
1
d=1
s1
2
s2
3
d=2
s3
4
s4
5
Goal
6
s6
×


BFS Frontier (Queue)
Deq s3
s4
s5
s6

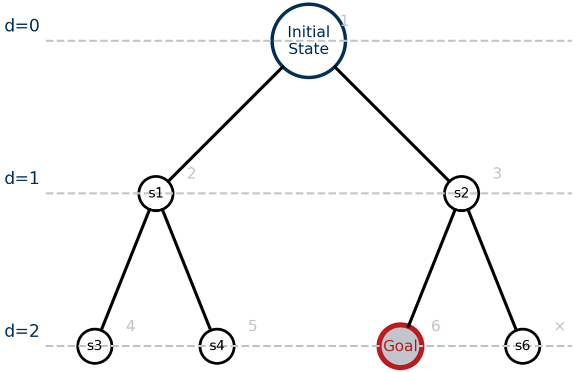
BFS: Step 5


BFS: Tree State
d=0
d=1
d=2
Initial State
1
s1
2
s2
3
s3
4
s4
5
Goal
6
s6
×


BFS Frontier (Queue)
Deq s4
s5
s6

BFS: Step 6

BFS: Tree State



BFS Frontier (Queue)

Deq s5 = GOAL

Stop: goal found (frontier not exhausted)

Breadth-First Tree Search (BFS) Algorithm

Algorithm 1 *

Breadth-First Tree Search (BFS)

Initialize the frontier as an empty FIFO queue
ENQUEUE(frontier, s_0)
while frontier is not empty **do**
    $n \leftarrow$ DEQUEUE(frontier)
    if GOAL-TEST(n) **then**
        return solution path from s_0 to n
    end if
    for each $a \in Actions(n)$ **do**
        $s' \leftarrow Transition(n, a)$
        ENQUEUE(frontier, s')
    end for
end while
return failure

Breadth-First Graph Search (BFS) Algorithm

Algorithm 2 *

Breadth-First Graph Search (BFS)

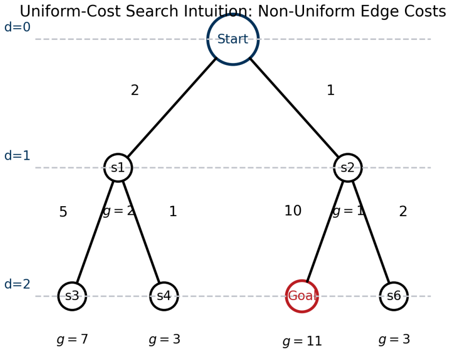
Initialize the frontier as an empty FIFO queue
ENQUEUE(frontier, s_0)
Initialize the explored set as empty
while frontier is not empty **do**
    $n \leftarrow$ DEQUEUE(frontier)
    if GOAL-TEST(n) **then**
        return solution path from s_0 to n
    end if
    Add n to explored
    for each $a \in Actions(n)$ **do**
        $s' \leftarrow Transition(n, a)$
        if $s' \notin$ frontier and $s' \notin$ explored **then**
            ENQUEUE(frontier, s')
        end if
    end for
end while
return failure

BFS Properties

- Complete (if branching factor finite).
- Optimal for uniform step costs.
- Time/space complexity: $O(b^d)$.

Uniform-Cost Search (UCS): Intuition

- Like BFS, but expands the *cheapest* path so far, not the shallowest.
- Appropriate when step costs are **non-uniform** and **strictly positive**.
- Frontier is a **min-priority queue** keyed by path cost $g(n)$.
- Goal is tested when a node is **popped** (removed as lowest-cost), ensuring optimality.



Frontier & Explored Sets

- **Frontier:** Min-heap / priority queue ordered by $g(n)$.
- **Explored/Visited:** Track the best known cost to each state.
- **Duplicate handling:**
  - If we discover a cheaper path to a state already in frontier/explored, **update** (decrease-key or reinsert) and keep the cheaper one.
  - Discard dominated (more expensive) paths to the same state.

UCS (Graph Search) — Pseudocode

Algorithm 3 *

Uniform-Cost Graph Search (UCS)

Require: initial state s_0; Actions$(\cdot)$; Transition$(\cdot, \cdot)$; GOAL-TEST$(\cdot)$; step cost $c(s, a) > 0$

Initialize the frontier as an empty **min-priority queue** (keyed by g)
$g(s_0) \leftarrow 0$; PUSH(frontier, s_0, key $= g(s_0)$)
best_g $\leftarrow$ empty map from state $\rightarrow$ best known path cost; best_g[s_0] $\leftarrow 0$
while frontier is not empty **do**
    $n \leftarrow$ POP-MIN(frontier) ▷ state with lowest $g(n)$
    if GOAL-TEST(n) **then**
        return solution path from s_0 to n
    end if
    for each $a \in$ Actions(n) **do**
        $s' \leftarrow$ Transition(n, a)
        $g' \leftarrow g(n) + c(n, a)$
        if $s' \notin$ best_g **or** $g' <$ best_g[s'] **then**
            best_g[s'] $\leftarrow g'$
            PUSH(frontier, s', key $= g'$)
        end if
    end for
end while
return failure

Why Goal-Test on Pop?

- When a node is popped, it has the *minimum* g among all frontier nodes.
- With strictly positive step costs, any other path to the goal would be $\geq$ its current g.
- Therefore, the first time a goal state is popped, its path is **optimal**.
- Testing at *generation* can break optimality (a cheaper path may appear later).

Properties of UCS

- **Completeness:** Yes, if all step costs $c > 0$ and minimum step cost $\epsilon > 0$.
- **Optimality:** Yes, returns a least-cost solution under $c > 0$.
- **Time:** Expands all nodes with $g(n) < C^*$; often expressed as $O\left(b^{1+\left\lfloor \frac{C^*}{\epsilon} \right\rfloor}\right)$ in the worst case.
- **Space:** Same order as time (frontier can be large).

Implementation Gotchas

- **Decrease-key** support: if unavailable, insert a new entry and let the stale one be ignored on pop.
- **Visited vs. best_g:** In weighted graphs, a simple "visited set" is insufficient—track best known g.
- **Zero/Negative costs:** Zero-cost cycles can cause huge frontiers; negative costs *break* UCS assumptions.
- **Tie-breaking:** Define stable policy (e.g., FIFO by insertion time) for deterministic debugging/diagrams.

When to Prefer UCS

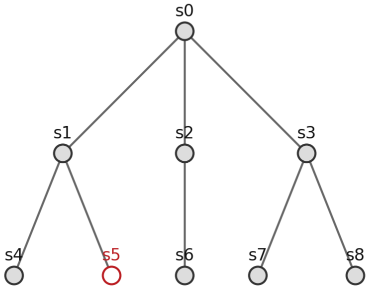
- Costs vary and you require **optimal** solutions.
- No trustworthy heuristic is available (otherwise consider A*).
- Step costs are strictly positive and not dominated by zero-cost cycles.

UCS Summary

- UCS systematically explores cheapest paths first using a min-priority queue on g.
- Test the goal only when popped to preserve optimality.
- Equivalent to Dijkstra for shortest paths; reduces to BFS when costs are uniform.

Depth-First Search (DFS): Intuition

- Dive down a path as far as possible before backtracking.
- Uses a **stack** (explicit or recursion) for the frontier.
- Great when solutions are deep and branching factor is manageable.
- **Risks:** can get stuck in deep/loopy parts without care.
- Tree-DFS vs. Graph-DFS (with **explored set** to avoid repeats).



DFS (Tree Search)

Algorithm 4 DFS-Tree (iterative, stack-based; explicit *Actions* and *Transition*)

```
frontier ← stack containing Make-Node(problem.initial)
while frontier ≠ ∅ do
    node ← POP(frontier)                                  ▷ LIFO
    if GOAL-TEST(node.state) then
        return SOLUTION(node)
    end if
    A ← ACTIONS(problem, node.state)
    for each a ∈ REVERSE(A) do                            ▷ reverse so leftmost expands first
        s′ ← TRANSITION(node.state, a)
        child ← Make-Node(s′, node, a)
        PUSH(frontier, child)
    end for
end while
return FAILURE
```

Notes: This is *tree search* (no explored set). For graphs or repeated states, use the next variant.

DFS (Graph Search) — Stack-Based

Algorithm 5 DFS-Graph (iterative; explicit *Actions* and *Transition*)

```
frontier ← stack containing Make-Node(problem.initial)
explored ← ∅
while frontier ≠ ∅ do
    node ← POP(frontier)
    if GOAL-TEST(node.state) then
        return SOLUTION(node)
    end if
    if node.state ∉ explored then
        add node.state to explored
        A ← ACTIONS(problem, node.state)
        for each a ∈ REVERSE(A) do
            s′ ← TRANSITION(node.state, a)
            child ← Make-Node(s′, node, a)
            if s′ ∉ explored and no node in frontier has state s′ then
                PUSH(frontier, child)
            end if
        end for
    end if
end while
return FAILURE
```

Key: LIFO frontier implements depth-first behavior; the explored set prevents cycles and re-expansion.

DFS: Properties and Trade-offs

- **Completeness:**
  - Tree-DFS: *No* (can go down infinite branch).
  - Graph-DFS: *No* in infinite-depth graphs; *Yes* if finite and cycles blocked.
- **Optimality:** *No* (does not expand by path cost or shallowest depth).
- **Time:** $O(b^m)$ where b branching factor, m max depth.
- **Space:** $O(bm)$ (linear in depth; much better than BFS).

When is DFS attractive?

- Memory constraints are tight.
- Solutions are *deep* and the graph isn't too loopy.
- Need a quick, low-overhead probe of the search space.

Gotchas

- Infinite paths or very deep trees.
- Heavily order-dependent behavior.

Depth-Limited Search (DLS): Idea

- DFS with a hard **depth cutoff** L.
- Explore along a path but **do not expand** nodes deeper than L.
- Returns one of three outcomes:
  - a **solution** (goal found),
  - CUTOFF (depth limit prevented full search),
  - FAILURE (no solution in the explored portion).
- Useful when you have a **reasonable bound** on solution depth, or as the inner loop of **Iterative Deepening**.

DLS (Tree Search): Iterative, Stack-Based

Algorithm 6 DLS-Tree(*problem*, *L*)   (explicit *Actions* and *Transition*)

```
frontier ← stack containing Make-Node(problem.initial, depth = 0)
cutoff ← false
while frontier ≠ ∅ do
    node ← POP(frontier)                                        ▷ LIFO
    if GOAL-TEST(node.state) then
        return SOLUTION(node)
    end if
    if node.depth = L then
        cutoff ← true                                           ▷ hit the limit; do not expand
        continue
    end if
    A ← ACTIONS(problem, node.state)
    for each a ∈ REVERSE(A) do                                  ▷ reverse so leftmost is popped next
        s′ ← TRANSITION(node.state, a)
        child ← Make-Node(s′, node, a, depth = node.depth + 1)
        PUSH(frontier, child)
    end for
end while
return CUTOFF if cutoff else FAILURE
```

Tree-search version (no explored set). For graphs, add an *explored* set and skip repeated states.

Depth-Limited Search: Properties

Guarantees

- **Completeness:**
  - If a solution exists at depth $\leq L$ and branching is finite: **Yes**.
  - Otherwise: **No** (may return CUTOFF).
- **Optimality: No** in general (not by shallowest or least-cost).

Complexity

- **Time:** $O(b^L)$
- **Space:** $O(bL)$ (like DFS, linear in depth)

When to use

- You have a **good bound** on solution depth.
- Memory is tight but pure DFS risks going too deep.
- As the inner loop of **Iterative Deepening** $(L = 0, 1, 2, \ldots)$.

Iterative Deepening DFS (IDS): A DFS/BFS Hybrid

- Performs DFS to depth limit L, then increases $L = 0, 1, 2, \ldots$
- **Completeness:** Yes (like BFS) if step costs uniform and branching finite.
- **Optimality:** Yes for unit step costs (finds shallowest goal).
- **Time:** $O(b^d)$; **Space:** $O(bd)$ (like DFS).
- **Why use it?** BFS-like guarantees with DFS-like space.

Uninformed Search: Summary Table

Algorithm	Frontier (Data Structure)	Complete?	Optimal?	Time	Space
BFS	FIFO queue	Yes[a]	Yes[a]	$O(b^{d+1})$	$O(b^{d+1})$
UCS	Min-priority queue by $g(n)$	Yes[b]	Yes	$O\left(b^{1+\lfloor C^*/\varepsilon \rfloor}\right)$	same
DFS	LIFO stack	No[c]	No	$O(b^m)$	$O(b\,m)$
DLS (ℓ)	Stack + depth limit ℓ	No/Yes[d]	No	$O(b^{\min(\ell,m)})$	$O(b\,\min(\ell, m))$
IDS	Repeated DLS for limits $0..d$	Yes	Yes[a]	$O(b^d)$	$O(b\,d)$

Symbols: b = branching factor, d = depth of shallowest goal, m = max depth, C^* = optimal solution cost, ε = minimum step cost > 0.

[a] Assuming unit step costs. [b] Assuming all step costs $\geq \varepsilon > 0$. [c] May fail on infinite-depth trees or cycles without limits/explored set. [d] Complete if $\ell \geq d$ (finite b).

Day 2 Wrap-Up

- Uninformed algorithms: BFS, UCS, DFS, DLS, IDS.
- Tradeoffs in completeness, optimality, efficiency.
- Motivation: we need **heuristics** to go further.