Search: The Core of Planning

Dr. Neil T. Dantam

CSCI-498/598 RPM, Colorado School of Mines

Spring 2018

Outline

Planning and Search Problems

Basic Search
- Depth-First Search
- Breadth-First Search

Properties of Search and Planning

More Search Variations
- Iterative-Deepening Search
- Backward Search
- Bidirectional Search

Outline

Planning and Search Problems

Basic Search
- Depth-First Search
- Breadth-First Search

Properties of Search and Planning

More Search Variations
- Iterative-Deepening Search
- Backward Search
- Bidirectional Search

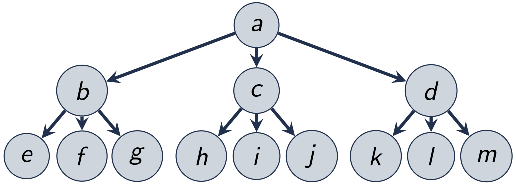
The Planning / Search Problem

Given:
1. State space: $\mathcal{Q}$
2. Transition function $\delta : \mathcal{Q} \mapsto \mathcal{P}(\mathcal{Q})$
3. Start state: $q_0 \in \mathcal{Q}$
4. Goal set: $A \subseteq \mathcal{Q}$

Find: Path $p = (p_0, \ldots, p_n)$ from start to goal such that:

- $p_0 = q_0$ is the start state
- $p_n \in A$ is a goal state
- Subsequent states are valid transitions: $p_{k+1} \in \delta(p_k)$

Search Trees



State Space: $Q = \{a,\ b, c, d,\ e, f, g,\ h, i, j,\ k, l, m\}$

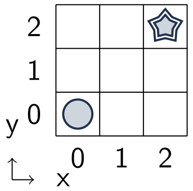
Transition Function:

- $\delta(a) = \{b, c, d\}$
- $\delta(b) = \{e, f, g\}$
- $\delta(c) = \{h, i, j\}$
- $\delta(d) = \{k, l, m\}$

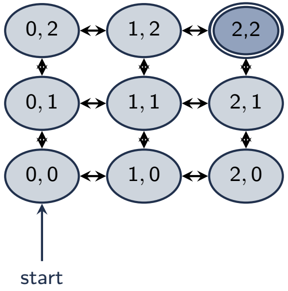
Explore / construct search tree until finding the goal

Example Domain 0: Gridworld

Cartoon



Graph



Symbols

State Space: $Q = \{(0,0), (0,1), (0,2)$
$(1,0), (1,1), (1,2)$
$(2,0), (2,1), (2,2)\}$

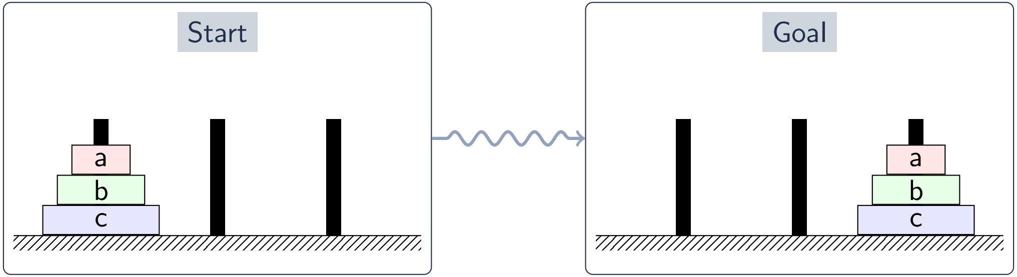
Transitions:

- $\delta((0,0)) = \{(0,1), (1,0)\}$
- $\delta((1,0)) = \{(0,0), (1,1), (2,0)\}$
- $\delta((2,0)) = \{(1,0), (2,1)\}$
- $\ldots$

Start: $q_0 = (0,0)$

Goal: $A = \{(2,2)\}$

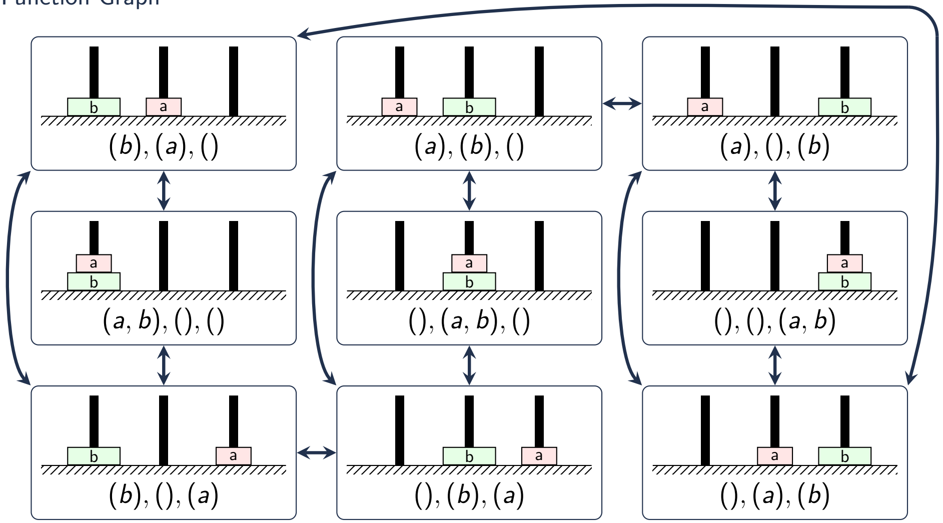
Example Domain 1: Towers of Hanoi



1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack.
3. No disk may be placed on top of a smaller disk.

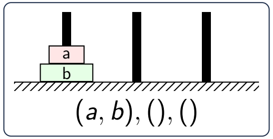
Example Domain 1: Towers of Hanoi

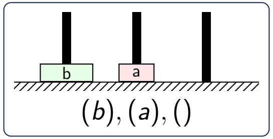
Transition Function Graph

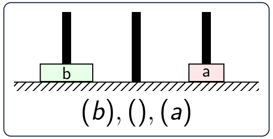


Example Domain 1: Towers of Hanoi

Symbolic Transition Function







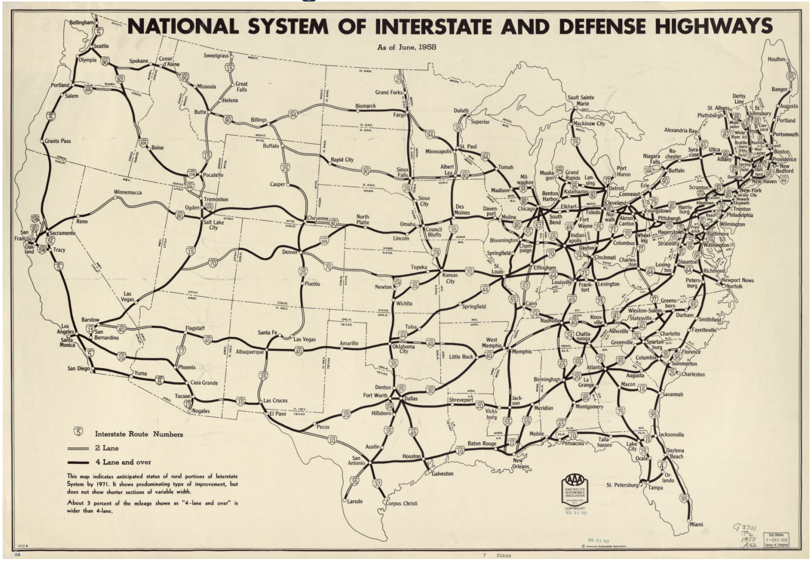
Transition Function:

- $\delta\left(\Big((a, b), (), ()\Big)\right) = \left\{ \Big((b), (a), ()\Big), \Big((b), (), (a)\Big) \right\}$
- $\delta\left(\Big((b), (a), ()\Big)\right) = \left\{ \Big((ab), (), ()\Big), \Big((), (a), (b)\Big), \Big((b), (), (a)\Big) \right\}$
- $\delta\left(\Big((b), (), (a)\Big)\right) = \left\{ \Big((ab), (), ()\Big), \Big((b), (a), ()\Big), \Big((), (b), (a)\Big) \right\}$
- ...

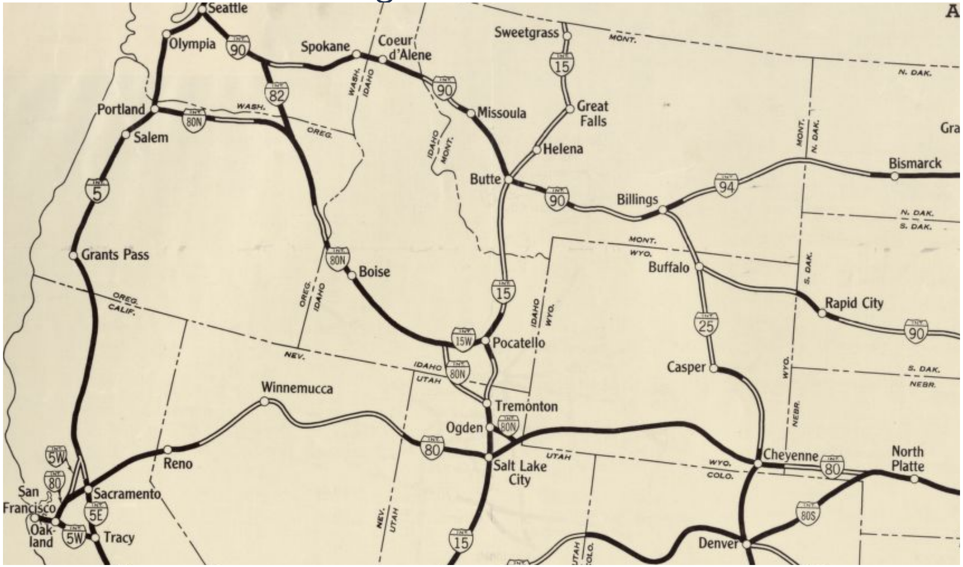
Start: $q_0 = \Big((a, b), (), ()\Big)$

Goal: $A = \left\{ \Big((), (), (a, b)\Big) \right\}$

Exercise Domain 2: Auto Navigation


NATIONAL SYSTEM OF INTERSTATE AND DEFENSE HIGHWAYS
As of June, 1958
Interstate Route Numbers
2 Lane
4 Lane and over
This map indicates anticipated status of rural portions of Interstate System by 1971. It shows predominating type of improvement, but does not show shorter sections of variable width.
About 3 percent of the mileage shown as "4-lane and over" is wider than 4-lane.

Exercise Domain 2: Auto Navigation



Exercise Domain 2: Auto Navigation

Outline

Planning and Search Problems

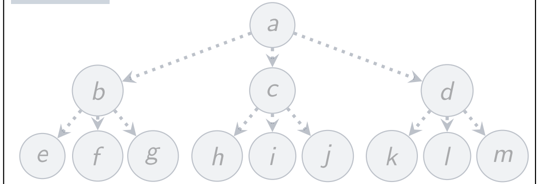
Basic Search
- Depth-First Search
- Breadth-First Search

Properties of Search and Planning

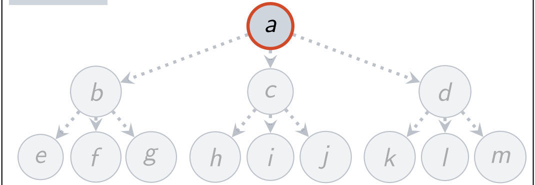
More Search Variations
- Iterative-Deepening Search
- Backward Search
- Bidirectional Search

Depth-First Search

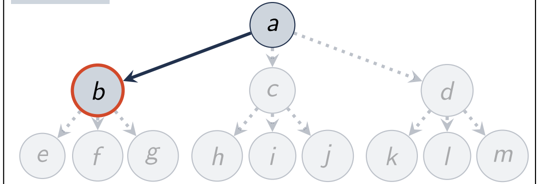
Step 0 $W = (a)$



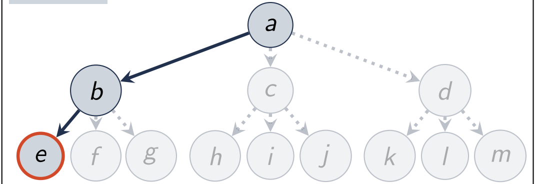
Step 1 $W = (b, c, d)$



Step 2 $W = (e, f, g, c, d)$



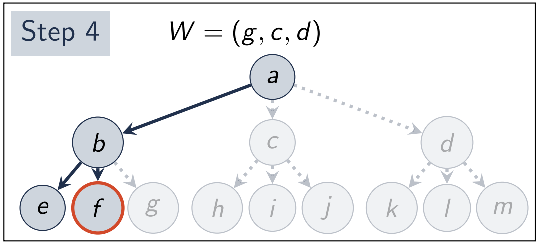
Step 3 $W = (f, g, c, d)$



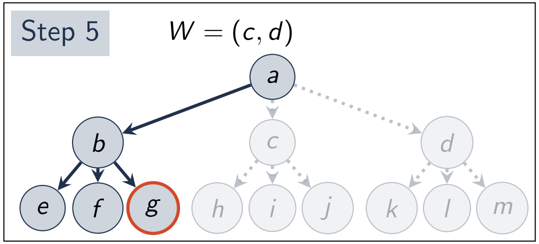
Depth-First Search

continued – 1

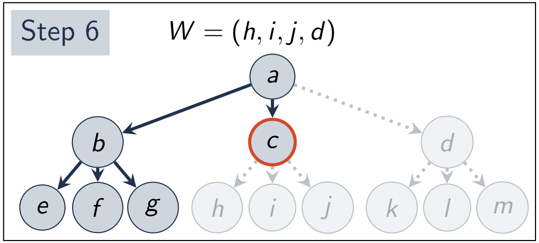
Step 4 $W = (g, c, d)$



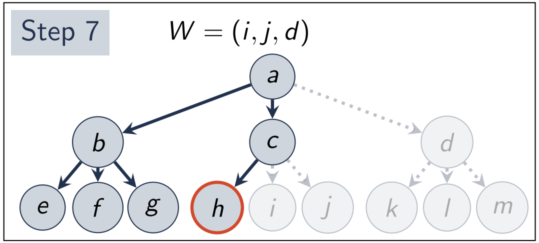
Step 5 $W = (c, d)$



Step 6 $W = (h, i, j, d)$

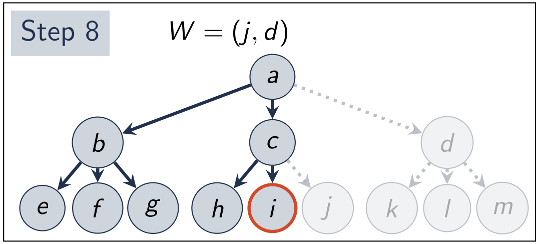


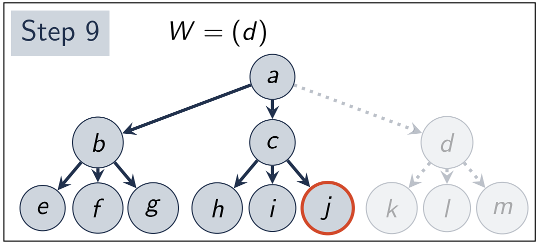
Step 7 $W = (i, j, d)$

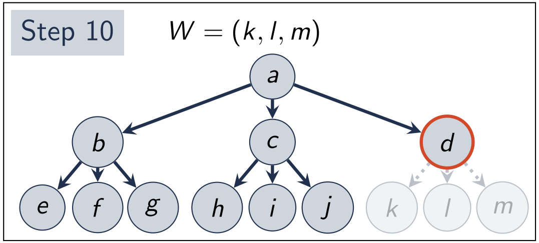


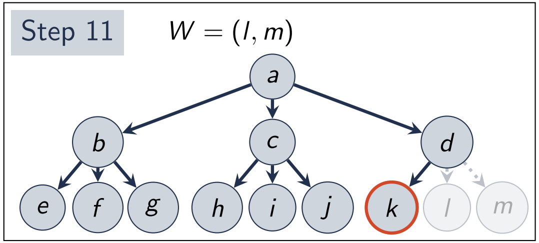
Depth-First Search

continued – 2

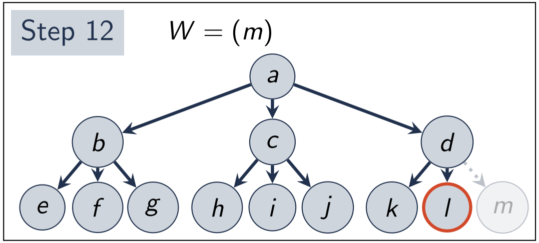


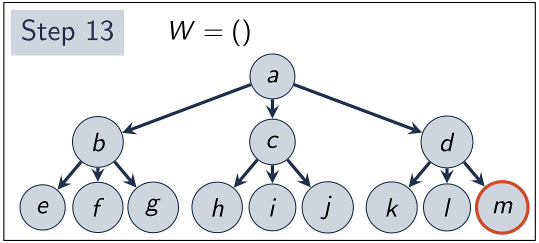






Depth-First Search
continued – 3





DFS Outline

Recursive

Track search progress via function call stack:

1. Visit node x
   1.1 Visit each neighbor of node x
   1.2 Return

Procedural

Track search progress via stack data structure:

1. Pop node x from stack
2. Push all neighbors of x onto stack
3. Repeat

DFS Algorithm

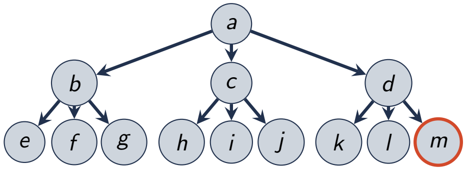
Recursive

Procedure dfs(q_0, δ, A)

```
T[q_0] ← nil ; // Search Tree
function visit(q) is
    if $q \in A$ then
        return tree-path(T, q);
    else
        foreach $q' \in \delta(q)$ do
            if ¬contains(T,q') then
                $T[q'] \leftarrow q$;
                let $p =$ visit($q'$) in
                    if p then return p ;
        return nil;
return visit($q_0$);
```

Tree-Path

```
Procedure tree-path($T, q$)
1 function $rec(q,path)$ is
2     if $q$ then // Recursive Case
3         return rec $\left( \overbrace{T[q]}^{\text{parent of } q} , \textbf{cons}(q, path) \right)$;
4     else // Base Case: at the root
5         return path;
6 return rec $(q, \textbf{nil})$;
```



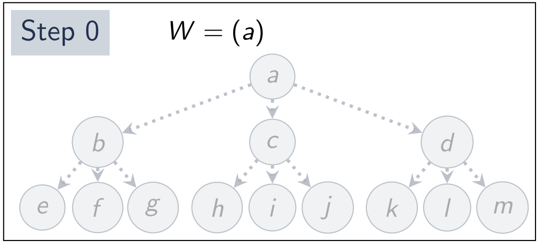
1. `rec(m, ())`
2. `rec(d, (m))`
3. `rec(a, (d, m))`
4. `rec(nil, (a, d, m))`
5. `(a, d, m)`

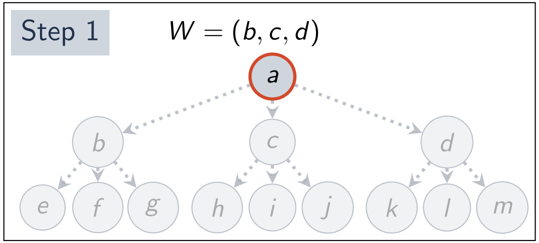
DFS Algorithm

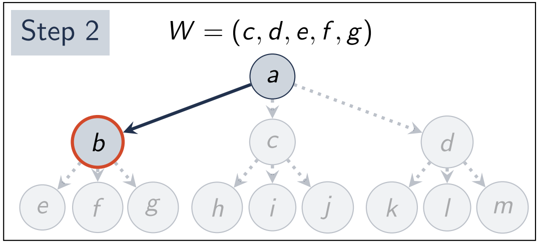
Procedural

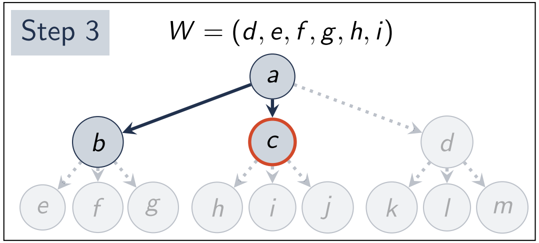
```
Procedure dfs($q_0, \delta, A$)
W ← ($q_0$) ; // Stack
$T[q_0]$ ← nil ; // Search Tree
while W do
    let $q$ = pop(W) in
        if $q \in A$ then
            return tree-path($T, q$);
        else
            foreach $q' \in \delta(q)$ do
                if $\neg$contains($T, q'$) then
                    $T[q'] \leftarrow q$;
                    $W \leftarrow$ push($q'$, $W$);
return nil;
```

Breadth-First Search





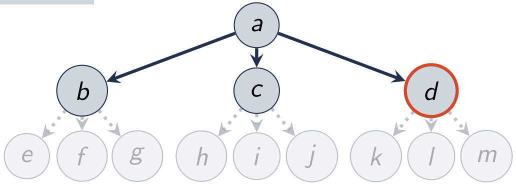




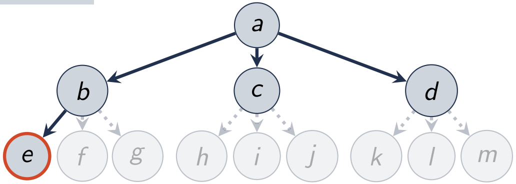
Breadth-First Search

continued – 1

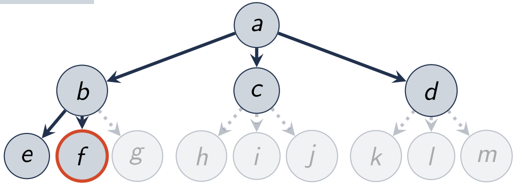
Step 4 $W = (e, f, g, h, i, j, k, l, m)$



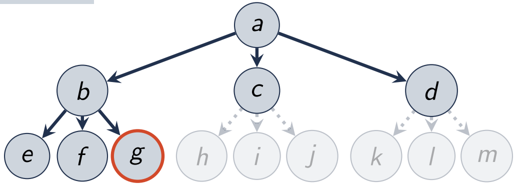
Step 5 $W = (f, g, h, i, j, k, l, m)$



Step 6 $W = (g, h, i, j, k, l, m)$



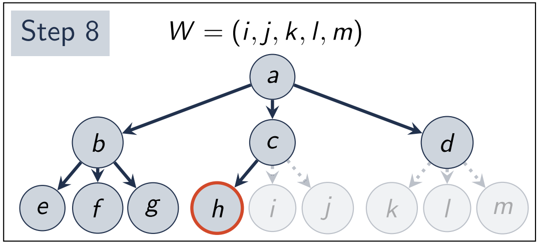
Step 7 $W = (h, i, j, k, l, m)$



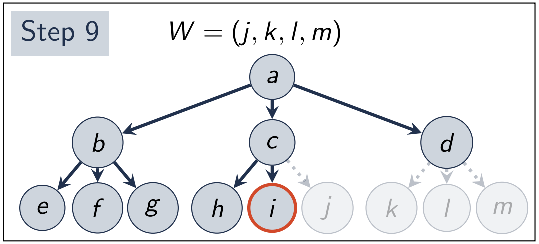
Breadth-First Search

continued – 2

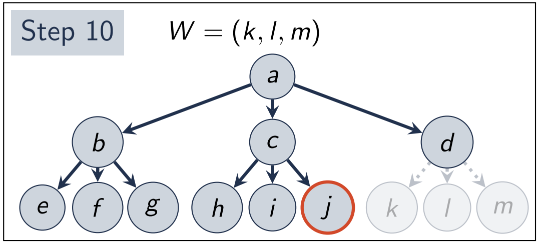
Step 8 $W = (i, j, k, l, m)$



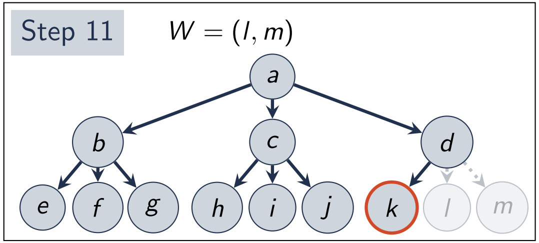
Step 9 $W = (j, k, l, m)$



Step 10 $W = (k, l, m)$

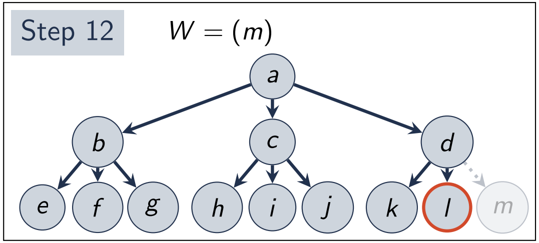


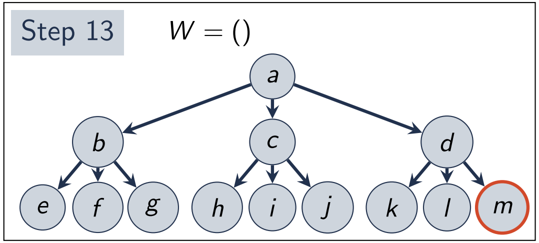
Step 11 $W = (l, m)$



Breadth-First Search

continued – 3





BFS Outline

Procedural

Track search progress via queue:

1. Dequeue node q from queue
2. Enqueue all neighbors of q
3. Repeat

BFS Algorithm

Procedural

```
Procedure bfs($q_0, \delta, A$)
$W \leftarrow (q_0)$; // Queue
$T[q_0] \leftarrow$ nil; // Search Tree
while $W$ do
    let $q =$ dequeue($W$) in
        if $q \in A$ then
            return tree-path($T, q$);
        else
            foreach $q' \in \delta(q)$ do
                if $\neg$contains($T$,$q'$) then
                    $T[q'] \leftarrow q$;
                    $W \leftarrow$ enqueue($q'$, $W$);
return nil;
```

Outline

Planning and Search Problems

Basic Search
- Depth-First Search
- Breadth-First Search

Properties of Search and Planning

More Search Variations
- Iterative-Deepening Search
- Backward Search
- Bidirectional Search

Planning Properties

Correctness:	Do we get a right answer?
Completeness:	Do we always get an answer?
Optimality:	Do we get the best answer?

Correctness

Definition (Correctness)

A planning algorithm is correct if every plan it produces is valid.

Given q_0, δ, A:

If $P(q_0, \delta, A) = (p_0, \ldots, p_n)$, then:

- $p_0 = q_0$
- $p_n \in A$
- For all p_i, $p_i \in \delta(p_{i-1})$

Completeness

Definition (Completeness)

A planning algorithm is complete if:

- When a solution exists, the planner always returns a solution,
- and when a solution does not exists, the planer returns false.

Given q_0, δ, A:

Solution Exists

There exists solution $(p_0, \ldots, p_n)$ where

- $p_0 = q_0$
- $p_n \in A$
- For all p_i, $p_i \in \delta(p_{i-1})$

Then $P(q_0, \delta, A)$ returns such.

No Solution Exists

Does not exist $(p_0, \ldots, p_n)$ where

- $p_0 = q_0$
- $p_n \in A$
- For all p_i, $p_i \in \delta(p_{i-1})$

Then $P(q_0, \delta, A)$ returns false.

Optimality

Definition (Optimality)

A planning algorithm is optimal if it produces the highest reward (/ lowest cost) plan.

Given $p_0,\ \delta,\ A,\quad \overbrace{V : Q \mapsto \mathbb{R}}^{\text{state reward function}}\quad :$

$$P(q_0, \delta, A) = \underset{p_0,\ldots,p_n}{\operatorname{argmax}} \overbrace{\left(\sum_{i=0}^{n} V(p_i) \right)}^{\text{path reward}}$$

Trade-offs

Why aren't we always Correct, Complete, and Optimal?

- Computation:
  - Time
  - Space
- Incomplete information / model limitations
- Infinite Spaces

Outline

Planning and Search Problems

Basic Search
- Depth-First Search
- Breadth-First Search

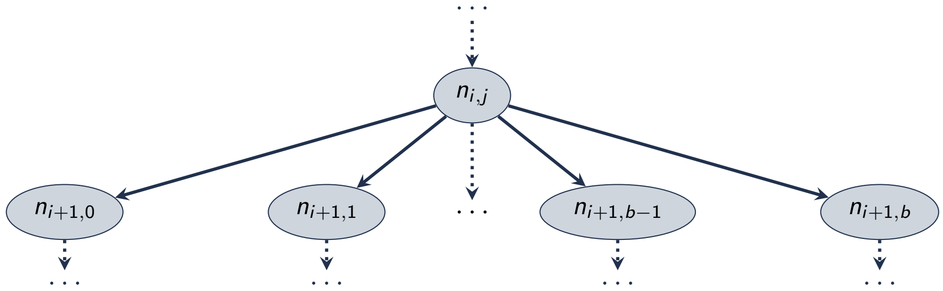
Properties of Search and Planning

More Search Variations
- Iterative-Deepening Search
- Backward Search
- Bidirectional Search

Branching Factor

Definition (Branching Factor)

Number of outgoing edges from a node of the search tree.

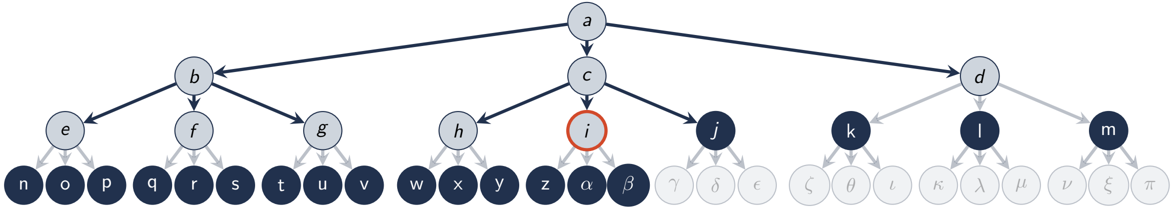


BFS Memory Use

b: branching factor d: current depth

$$|W| \sim b^d$$

$$W = (j, k, l, m, n, o.p, q, r, s, t, u, b, w, x, y, z, \alpha, \beta)$$



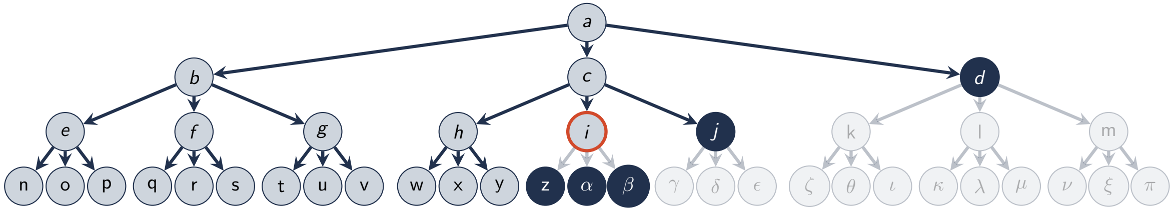
Memory use dominated by queued "next" level.

DFS Memory Use

b: branching factor d: current depth

$$|W| \sim b * d$$

$$W = (z, \alpha, \beta, j, d)$$


a
b
c
d
e
f
g
h
i
j
k
l
m
n o p q r s t u v w x y z
α β γ δ ϵ ζ θ ι κ λ μ ν ξ π

Small work list
(still have visited set)

DFS vs. BFS

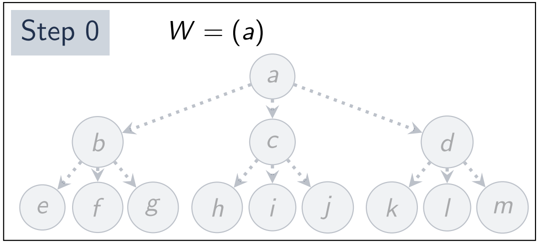
DFS	BFS
Pro: Compact work list (linear)	Pro: Optimal (for equal step cost)
Con: Not optimal	Con: Exponential work list

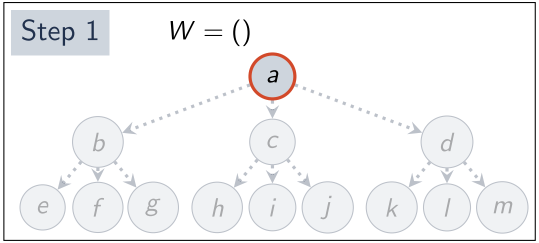
Iterative-Deepening Search (IDS) Overview

1. Set depth limit to 0
2. Run depth-first search up to depth-limit
   2.1 If goal found, return
   2.2 Otherwise, increment depth limit and repeat

Iterative-Deepening Search

Start with depth limit 0

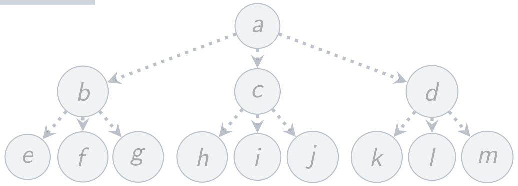




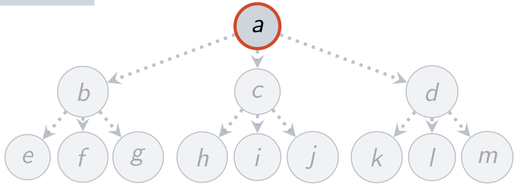
Iterative-Deepening Search

Retry with depth limit 1

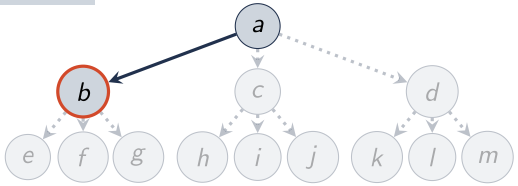
Step 2 $W = (a)$



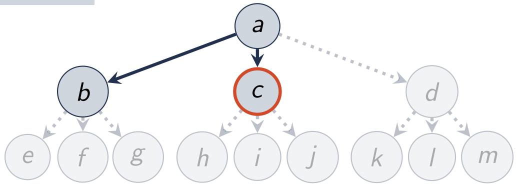
Step 3 $W = (b, c, d)$



Step 4 $W = (c, d)$

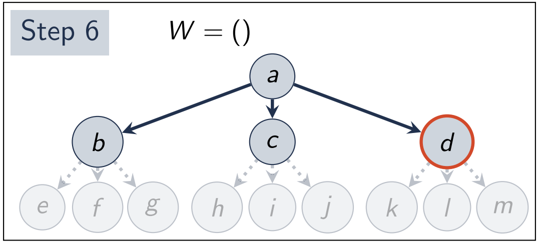


Step 5 $W = (d)$



Iterative-Deepening Search

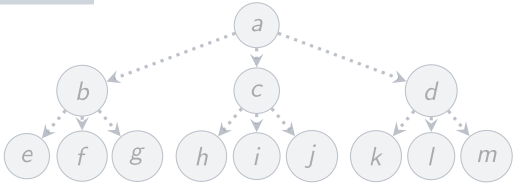
Retry with depth limit 1 – continued 1



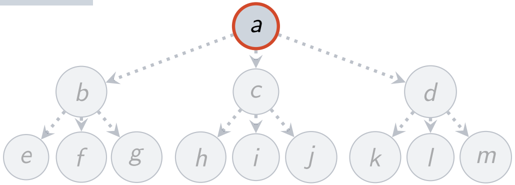
Iterative-Deepening Search

Retry with depth limit 2

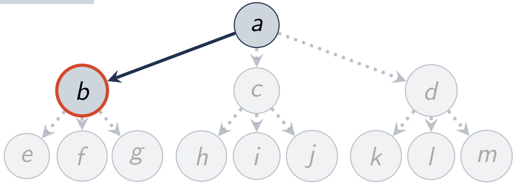
Step 0 $W = (a)$



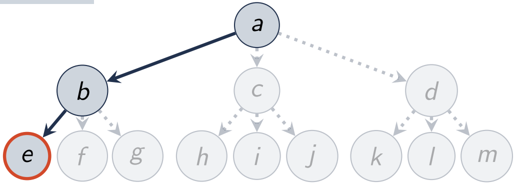
Step 1 $W = (b, c, d)$



Step 2 $W = (e, f, g, c, d)$

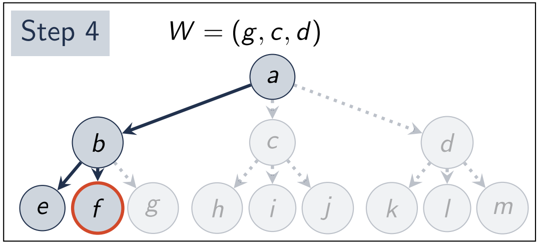


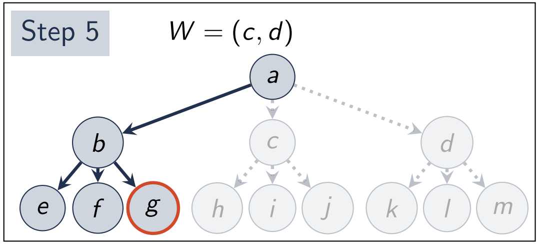
Step 3 $W = (f, g, c, d)$

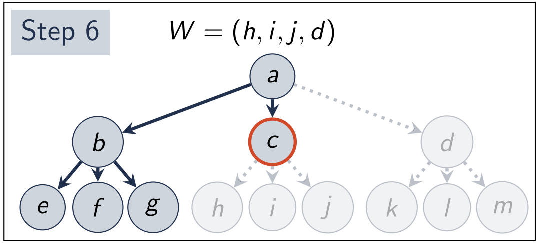


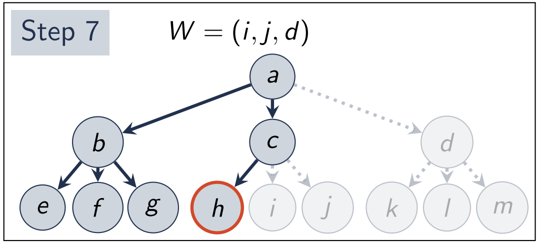
Iterative-Deepening Search

Retry with depth limit 2 – continued 1





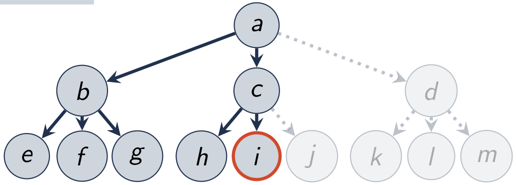




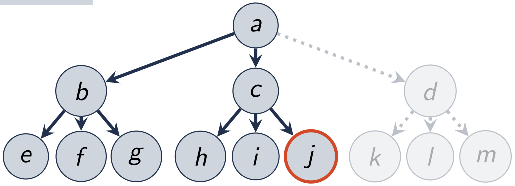
Iterative-Deepening Search

Retry with depth limit 2 – continued 2

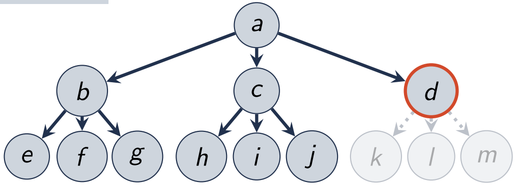
Step 8 $W = (j, d)$



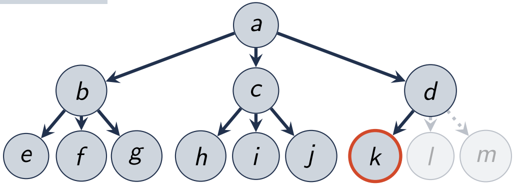
Step 9 $W = (d)$



Step 10 $W = (k, l, m)$

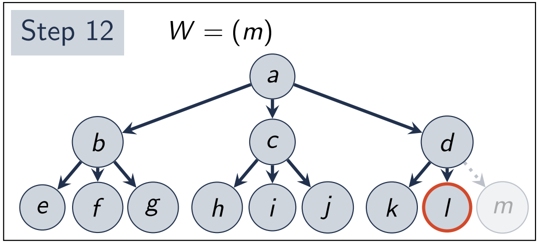


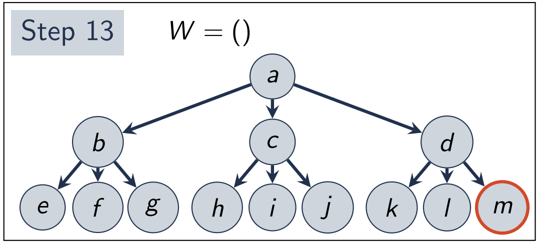
Step 11 $W = (l, m)$



Iterative-Deepening Search

Retry with depth limit 2 – continued 3





Depth-Limited Search Algorithm

Procedure dls($q_0, \delta, A, d_{\max}$)

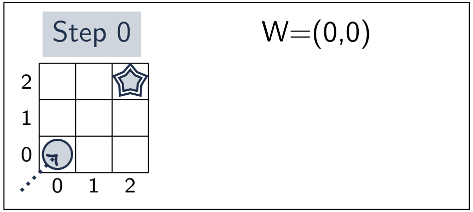
```
T[q_0] ← nil ; // Search Tree
function visit(q, d) is
    if q ∈ A then return tree-path(T, q) ;
    if (d ≤ 0) then return nil;
    else
        foreach q' ∈ δ(q) do
            if ¬contains(T,q') then
                T[q'] ← q;
                let p = visit(q', d − 1) in
                    if p then return p ;
        return nil
return (visit(q_0, d_max), T);
```

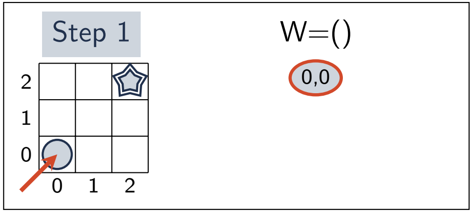
Iterative-Deepening Search Algorithm

```
Procedure ids($q_0, \delta, A$)
1 $d \leftarrow 0$;
2 repeat
3     $(p, T) \leftarrow$ dls $(q_0, \delta, A, d)$;
4     $d \leftarrow d + 1$;
5 until $p$ is non-empty or $T$ contains all nodes;
6 return $p$
```

Example: IDS Gridworld

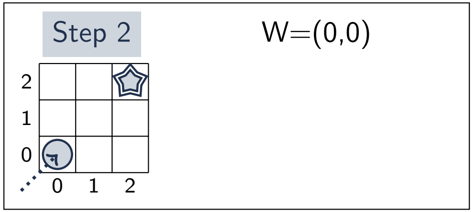
Depth limit 0

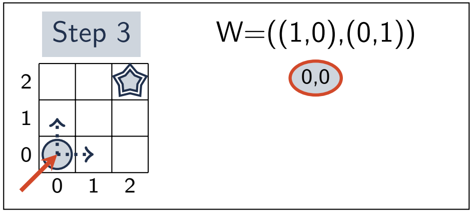


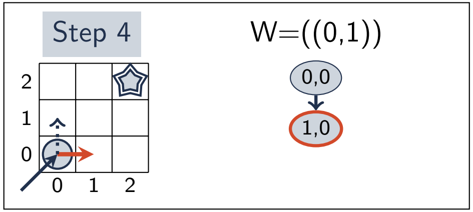


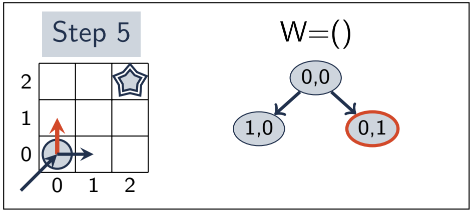
Example: IDS Gridworld

Depth limit 1



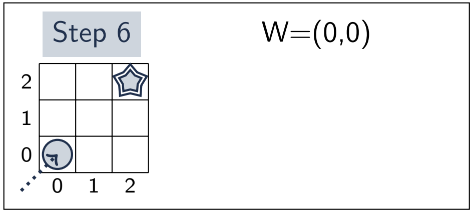


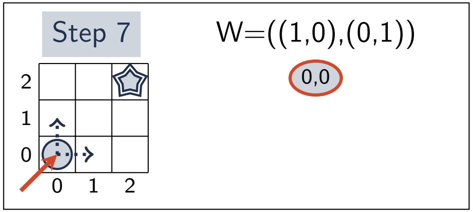


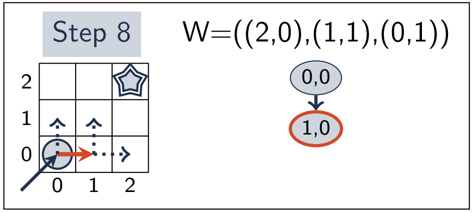


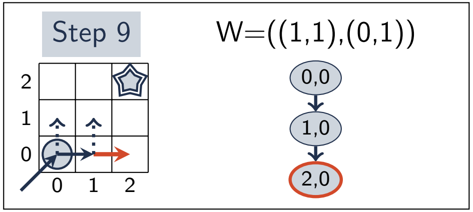
Example: IDS Gridworld

Depth limit 2



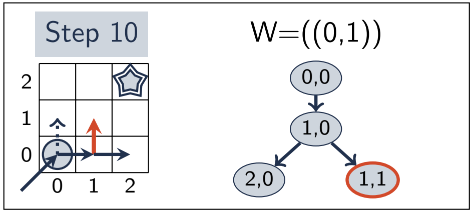


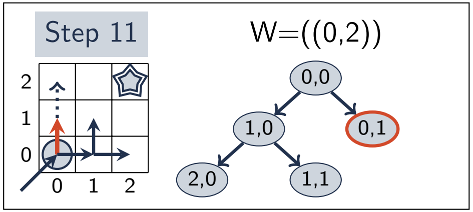


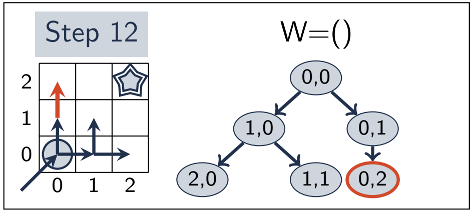


Example: IDS Gridworld

Depth limit 2, continued 1

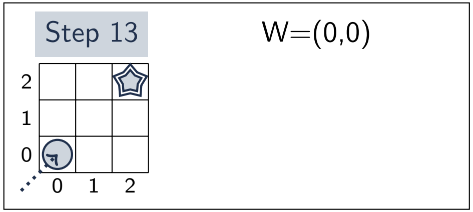


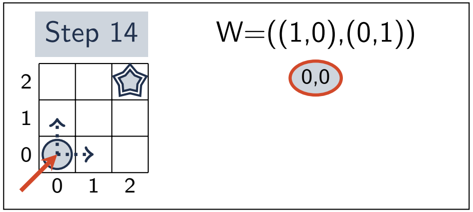


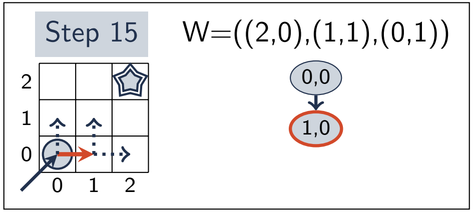


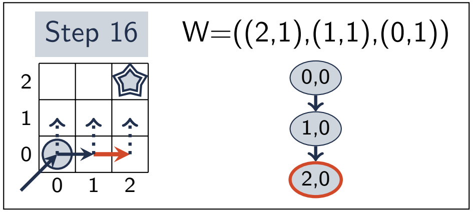
Example: IDS Gridworld

Depth limit 3



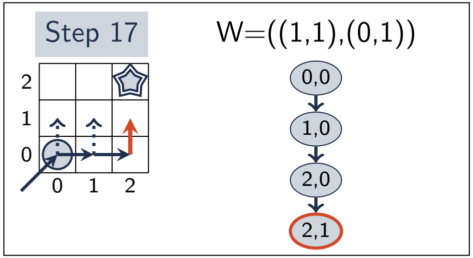


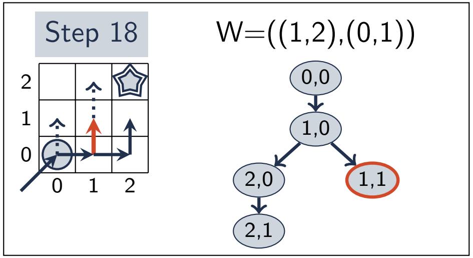


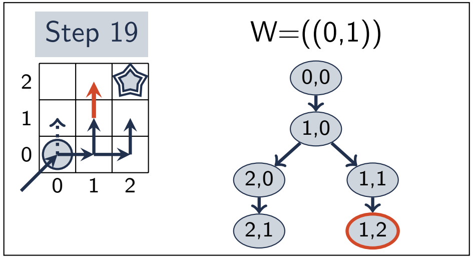


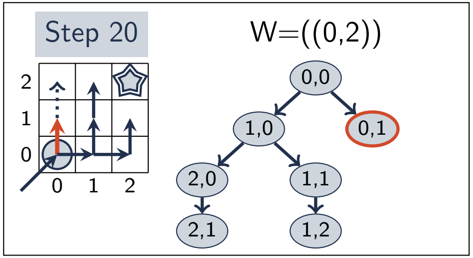
Example: IDS Gridworld

Depth limit 3, continued 1



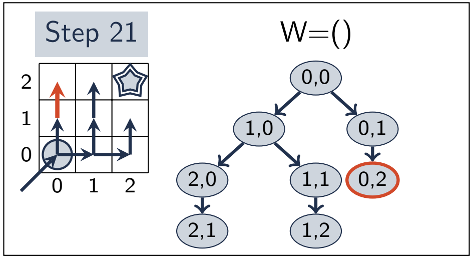






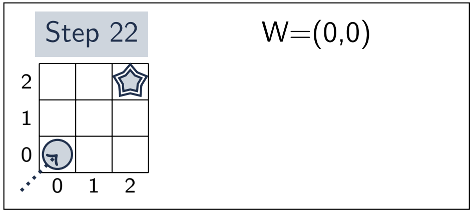
Example: IDS Gridworld

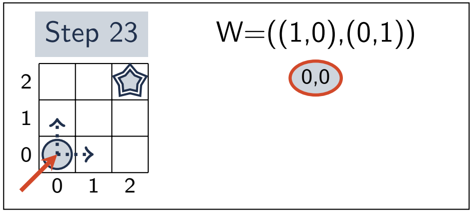
Depth limit 3, continued 2

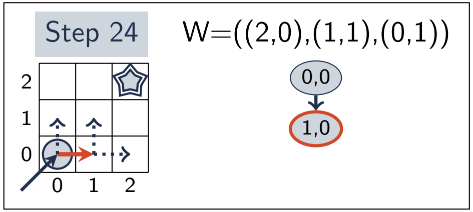


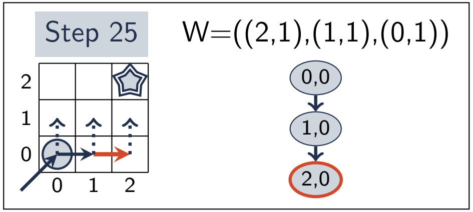
Example: IDS Gridworld

Depth limit 4



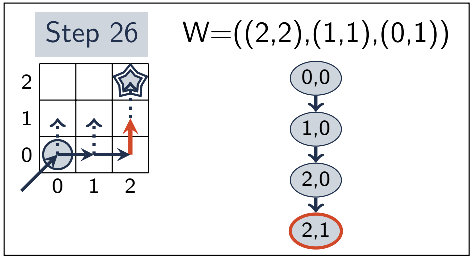


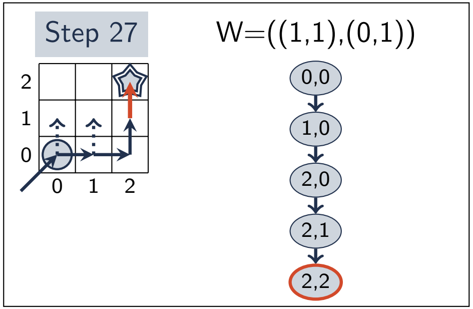




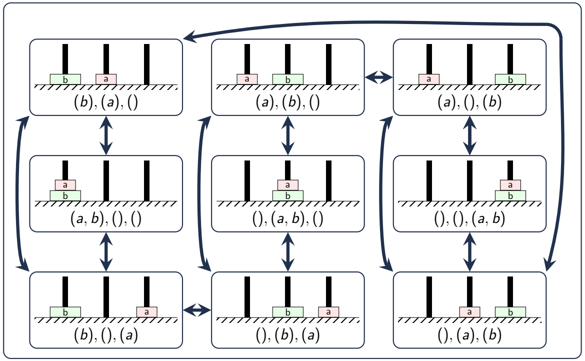
Example: IDS Gridworld

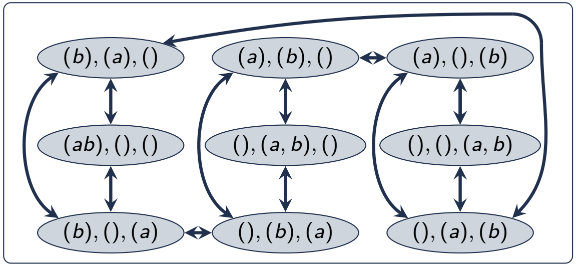
Depth limit 4, continued 1





Hanoi Graph





Exercise: IDS Hanoi

Depth limit 0

Exercise: IDS Hanoi

Depth limit 1

Exercise: IDS Hanoi

Depth limit 2

Exercise: IDS Hanoi

Depth limit 2, continued 1

Exercise: IDS Hanoi

Depth limit 3

Exercise: IDS Hanoi

Depth limit 3, continued 1

Backward Search Outline

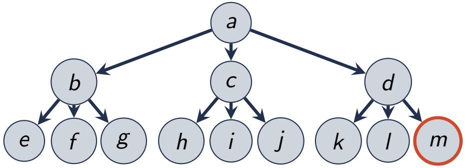
1. Start from goal set
2. Follow transitions backward to start
3. End at start state

Backward Search Algorithm

```
Procedure back-bfs($q_0, \delta_b, A$)
foreach $q \in A$ do
    $W_b \leftarrow$ enqueue ($q$, $W_b$); // Queue
    $T_b[q] \leftarrow$ nil; // Tree
while $W_b$ do
    let $q =$ dequeue ($W_b$) in
        if $q = q_0$ then
            return reverse-tree-path ($T_b, q$);
        else
            foreach $q' \in \delta_b(q)$ do
                if $\neg$contains ($T$, $q'$) then
                    $T_b[q'] \leftarrow q$;
                    $W_b \leftarrow$ enqueue ($q'$, $W_b$);
    return nil;
```

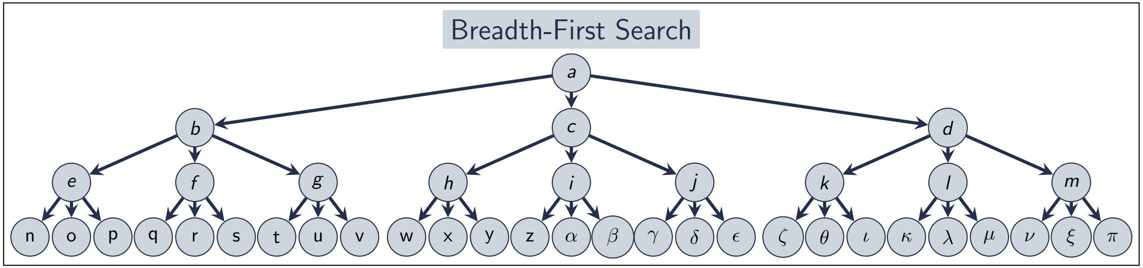
Reverse-Tree-Path

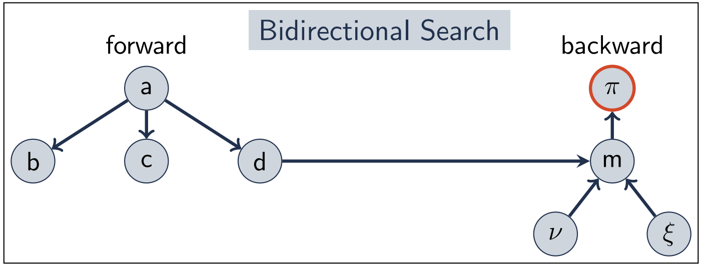
```
Procedure reverse-tree-path($T$, $q$)
function $rec(q)$ is
    if $q$ then // Recursive Case
        return cons($q$, rec ($T[q]$))
    else // Base Case: at the root
        return nil
return rec ($q$);
```



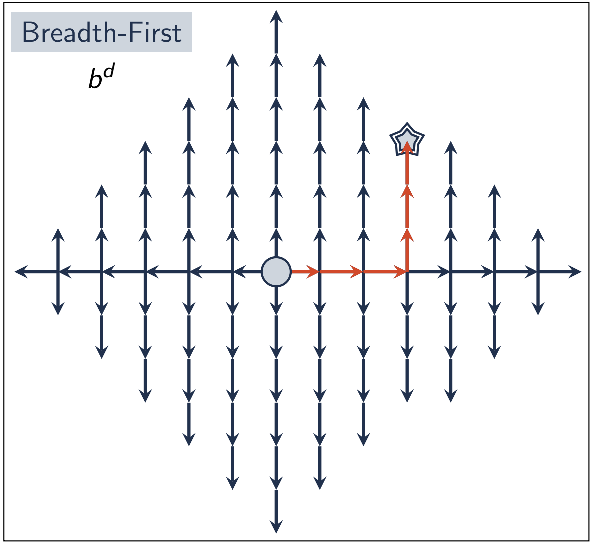
1. `rec(m)`
2. `cons(m, rec(d))`
3. `cons(m, cons(d, rec(a)))`
4. `cons(m, cons(d, cons(a, rec(nil))))`
5. `cons(m, cons(d, cons(a, nil)))`
6. `(m, d, a)`

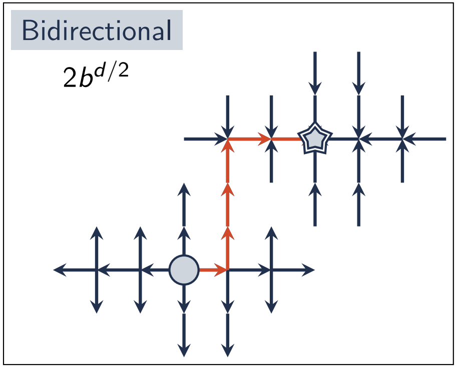
Bidirectional Search





Breadth-First vs Bidirectional Search Trees





BDS Algorithm

```
Procedure bds($q_0, \delta_f, \delta_b, A$)
1  $W_f \leftarrow (q_0)$; // Init Forward Queue
2  $T_f[q] \leftarrow$ nil; // Init Forward Tree
3  foreach $q \in A$ do // Init Backwards
4      $W_b \leftarrow$ enqueue($q$, $W_b$); // Backward Queue
5      $T_b[q] \leftarrow$ nil; // Backward Tree
6  while $W_f \wedge W_b$ do
       /* Forward Step */
7      $(q, W_f) \leftarrow$ grow-bds($W_f, T_f, T_b, \delta_f$) ;
8      if $q$ then return bds-result($T_f, T_b, q$) ;
       /* Backward Step */
9      $(q, W_b) \leftarrow$ grow-bds($W_b, T_b, T_f, \delta_b$) ;
10     if $q$ then return bds-result($T_f, T_b, q$) ;
11 return nil;
```

BDS Algorithm

Subroutines

Procedure bds-grow(W, T, T_o, δ)

```
1 let $q$ = dequeue($W$) in
2     foreach $q' \in \delta(q)$ do
3         if $\neg$contains($T$,$q'$) then
4             $T[q'] \leftarrow q$;
5             if contains($T_o$,$q'$) then
                  // Found shared node
6                 return (q',W);
7             else
8                 $W \leftarrow$ enqueue($q'$, $W$)
9 return (nil,W);
```

Procedure bds-result(T_f, T_b, q)

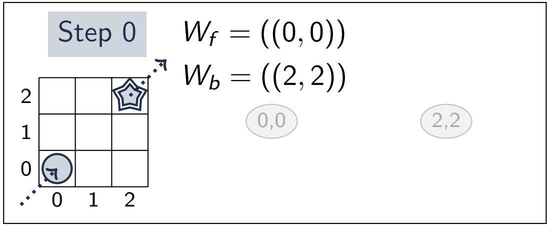
```
  /* Path from root to q                */
1 $p_f \leftarrow$ tree-path($T_f$, $q$);
  /* Path from q+1 to goal              */
2 $p_b \leftarrow$ reverse-tree-path($T_b$, $T_b[q]$);
3 return append($p_f$, $p_b$);
```

Example: BDS Gridworld

Step 0

$W_f = ((0,0))$

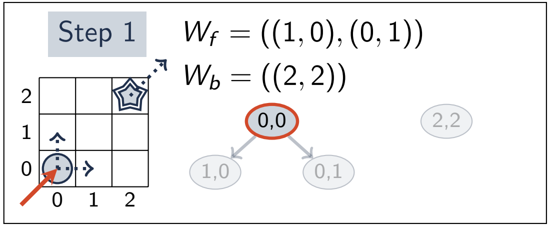
$W_b = ((2,2))$



Step 1

$W_f = ((1,0),(0,1))$

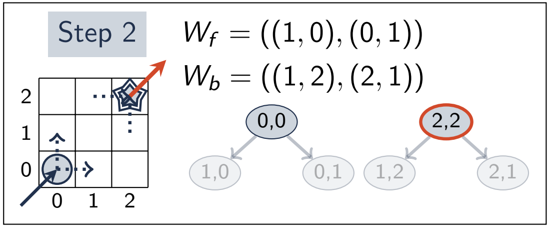
$W_b = ((2,2))$



Step 2

$W_f = ((1,0),(0,1))$

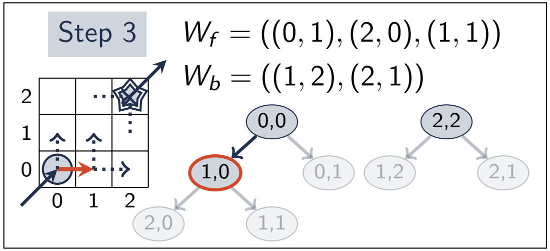
$W_b = ((1,2),(2,1))$



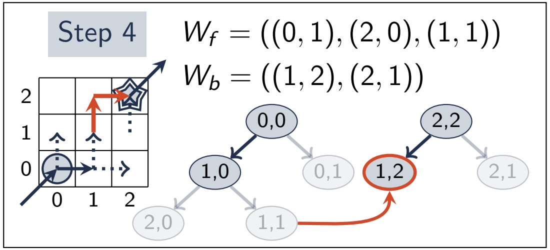
Step 3

$W_f = ((0,1),(2,0),(1,1))$

$W_b = ((1,2),(2,1))$



Example: BDS Gridworld



$W_f = ((0,1),(2,0),(1,1))$

$W_b = ((1,2),(2,1))$

Exercise: BDS Hanoi

Summary

Planning and Search Problems

Basic Search
- Depth-First Search
- Breadth-First Search

Properties of Search and Planning

More Search Variations
- Iterative-Deepening Search
- Backward Search
- Bidirectional Search