Search

Dr. Neil T. Dantam

CSCI-534, Colorado School of Mines

Spring 2020

MINES

Introduction

Definition (Search)

Progressively visit nodes (of a graph or tree) until reaching a goal.

Outcomes

- Know the formal definition of **search** and **planning**
- Know definitions search/planning **properties**
- Understand **uninformed search** algorithms
- Evaluate the trade-offs among uninformed search algorithms

MINES

Outline

MINES

Outline

The Planning / Search Problem

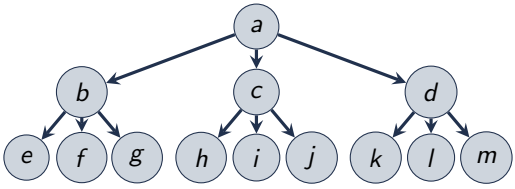
Given:
1. State space: $\mathcal{Q}$
2. Transition function $\delta : \mathcal{Q} \mapsto \mathcal{P}(\mathcal{Q})$
3. Start state: $q_0 \in \mathcal{Q}$
4. Goal set: $A \subseteq \mathcal{Q}$

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: $p_{k+1} \in \delta(p_k)$

MINES

Search Trees



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

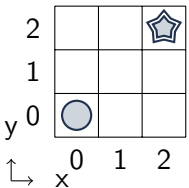
Transition Function:

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

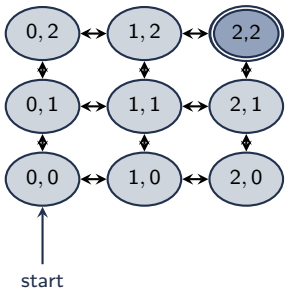
Explore / construct search tree until finding the goal

Example Domain 0: Gridworld

Cartoon



Graph



Symbols

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

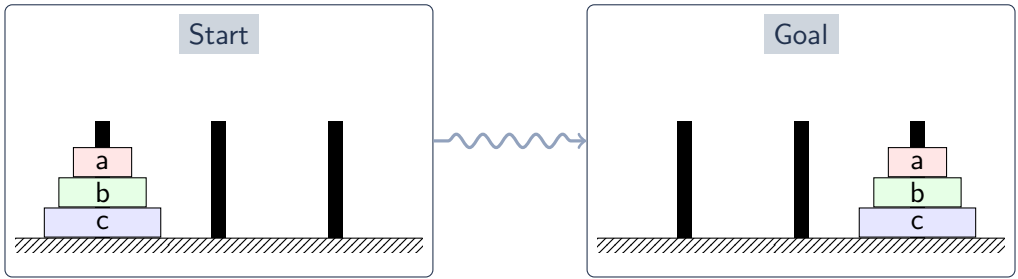
Transitions:

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

Start: $q_0 = (0,0)$

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

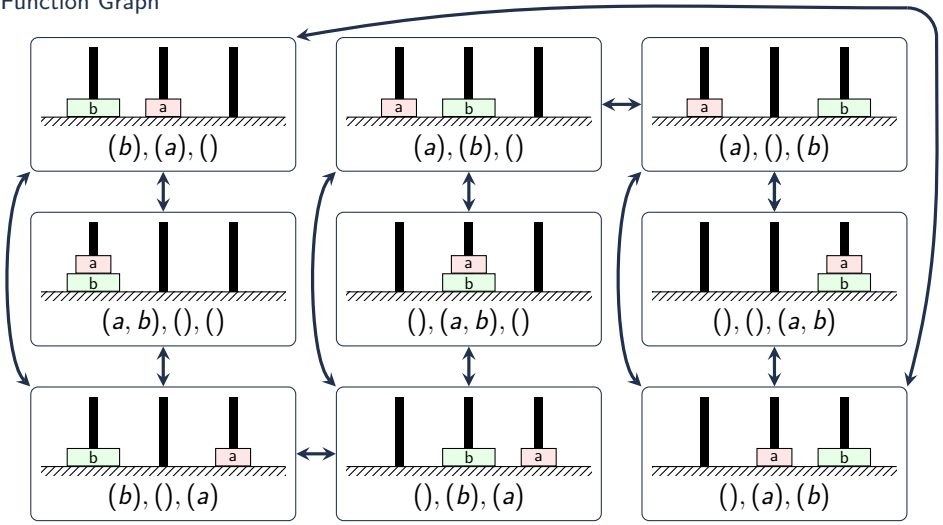
Example Domain 1: Towers of Hanoi



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.

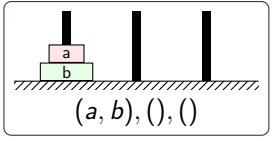
Example Domain 1: Towers of Hanoi

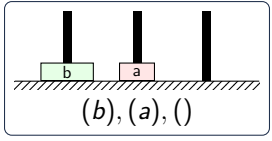
Transition Function Graph

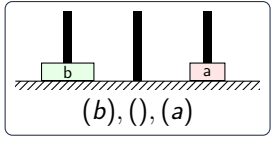


Example Domain 1: Towers of Hanoi

Symbolic Transition Function







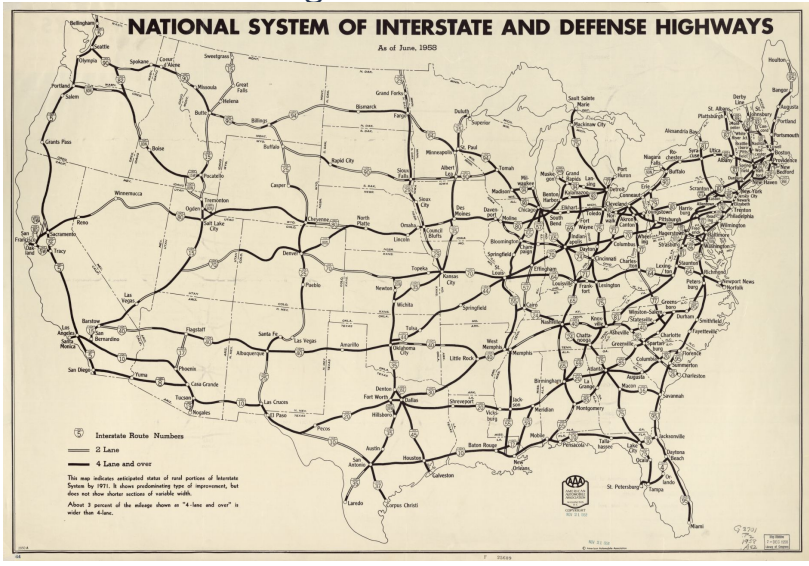
Transition Function:

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

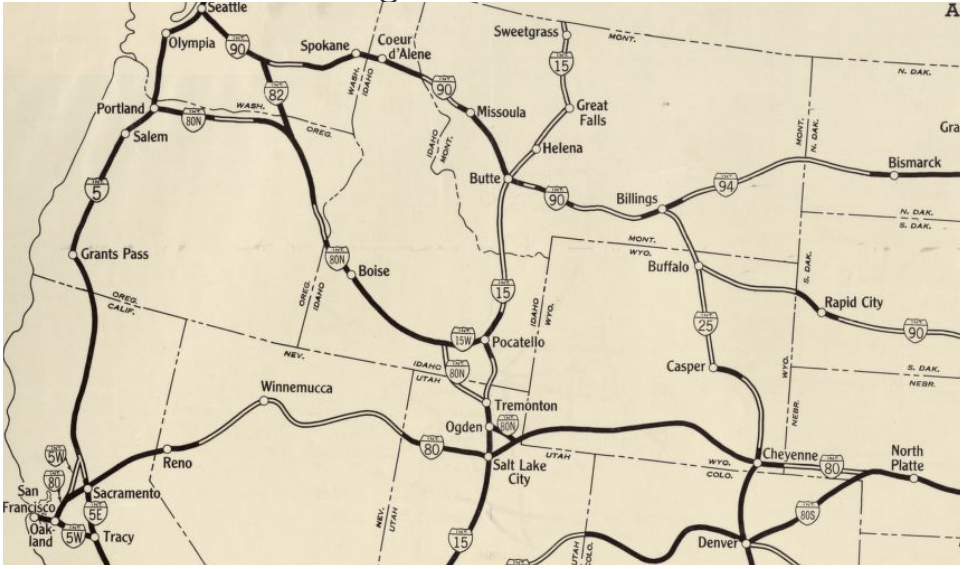
Start: $q_0 = \left((a,b),(),()\right)$

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

Exercise Domain 2: Auto Navigation



Exercise Domain 2: Auto Navigation



Exercise Domain 2: Auto Navigation

Outline

Uninformed vs. Informed Search

Definition: Uninformed Search

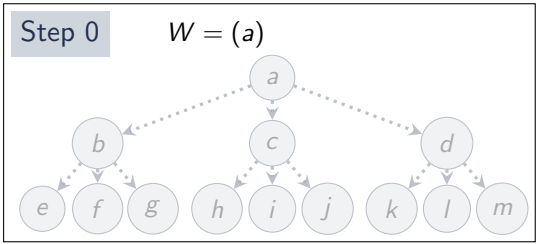
Search that has no additional information about states beyond the problem definition, i.e., can only distinguish between goal and non-goal states. Also called *blind search*.

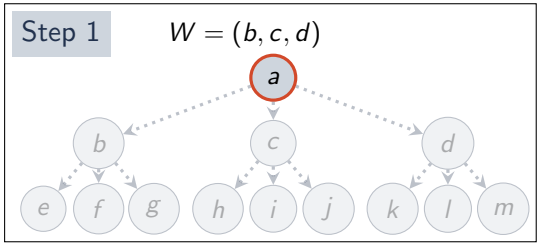
Definition: Informed Search

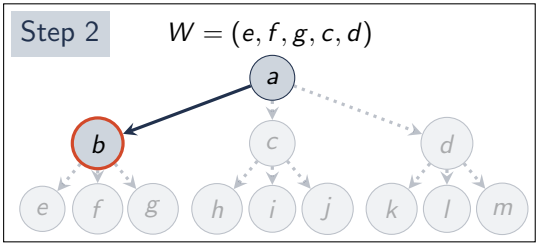
Search that has can considers whether some non-goal states are "more promising" than others. Also called *heuristic search*.

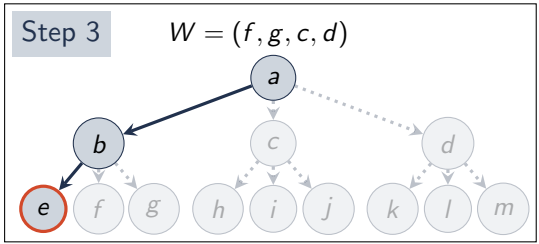
(more in informed search next lecture)

Depth-First Search



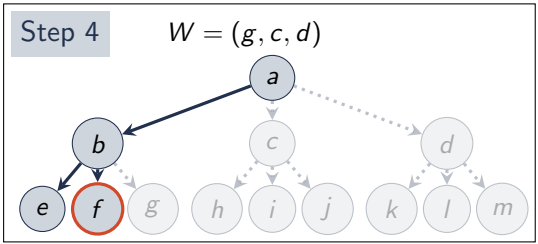


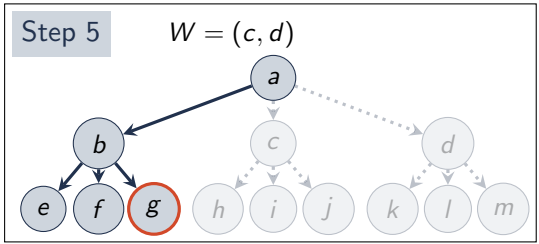


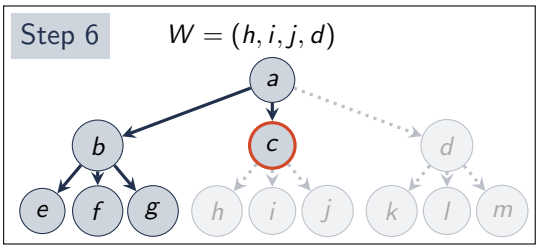


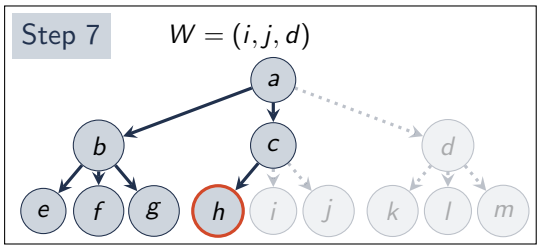
Depth-First Search

continued – 1



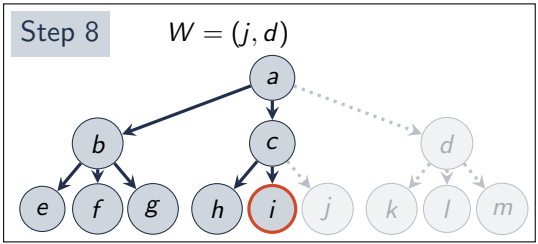


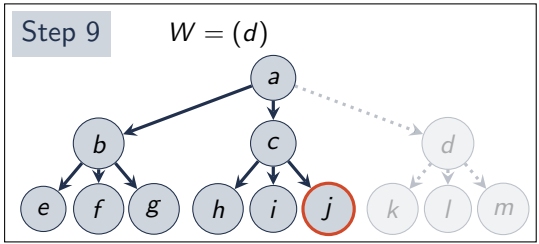


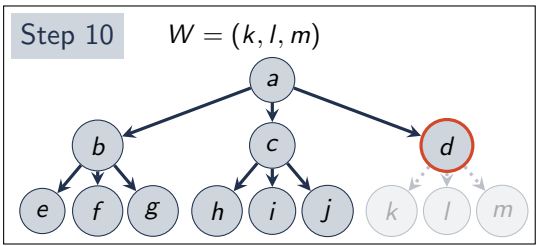


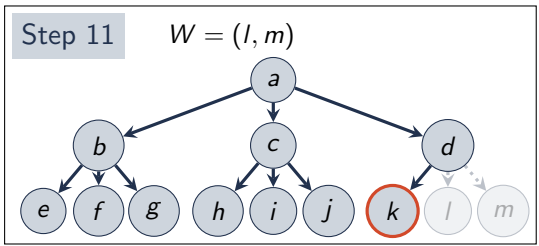
Depth-First Search

continued – 2



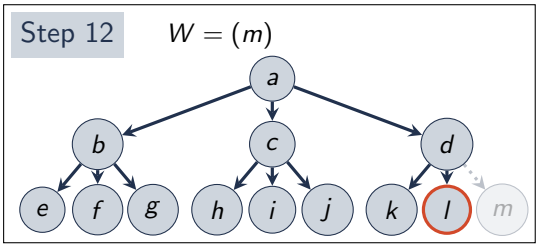


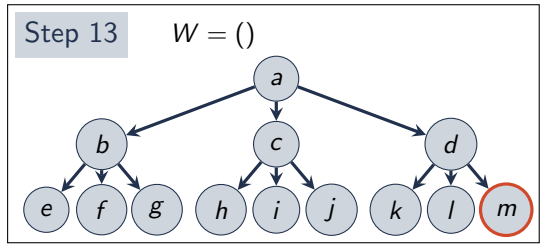




Depth-First Search

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

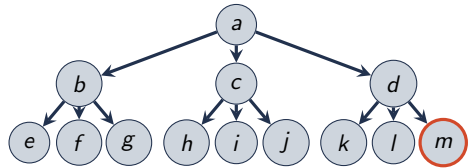
Procedure dfs(q_0, δ, A)

```
1  T[q_0] ← NIL ; // Search Tree:  state ↦ parent
2  function visit(q) is
3      if q ∈ A then // Base case:  Reached the goal
4          return tree-path(T, q);
5      else // Recursive case:  visit neighbors of q
6          foreach q' ∈ δ(q) do
7              if ¬contains(T,q') then
8                  T[q'] ← q;
9                  let p ← visit(q') in
10                     if p then return p ;
11         return NIL;
12 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( T[q] , cons(q, path) );
4     else // Base Case: at the root
5         return path;
6 return rec(q, NIL);
```



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

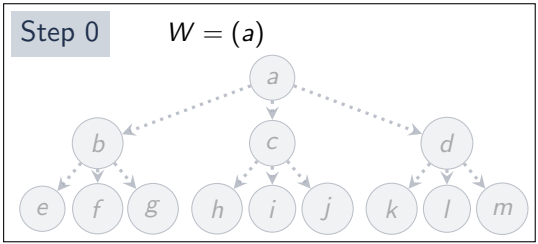
DFS Algorithm

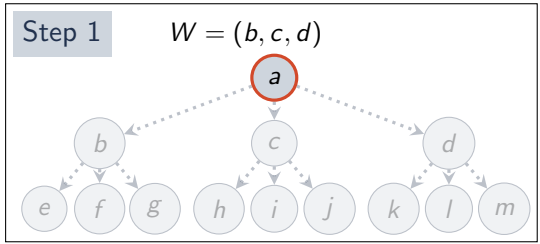
Procedural

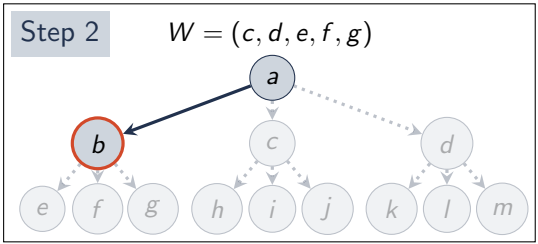
Procedure dfs(q_0, δ, A)

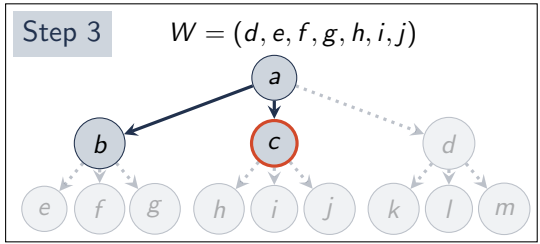
```
1  W ← (q_0) ; // Stack
2  T[q_0] ← NIL ; // Search Tree
3  while W do
4      let q ← pop(W) in
5          if q ∈ A then
6              return tree-path(T, q);
7          else
8              foreach q' ∈ δ(q) do
9                  if ¬contains(T,q') then
                       /* Schedule visits by pushing neighbors onto W   */
10                     T[q'] ← q;
11                     W ← push(q', W);
12 return NIL;
```

Breadth-First Search



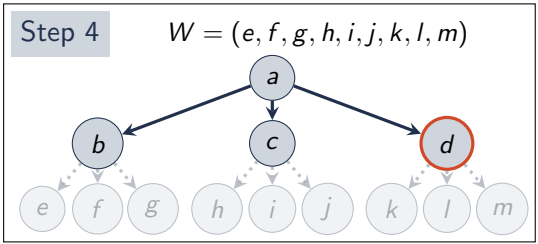


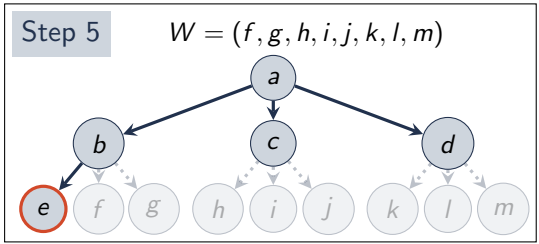


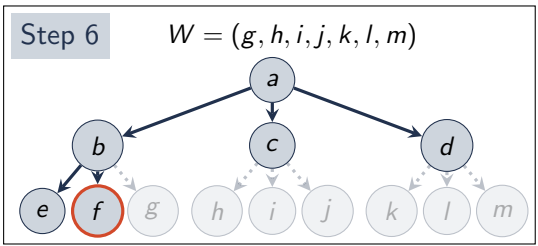


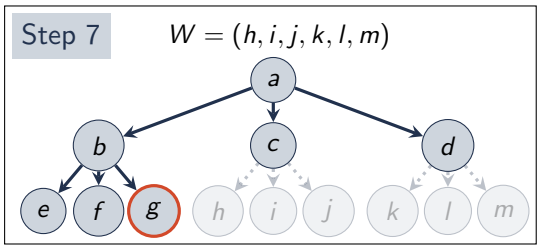
Breadth-First Search

continued – 1



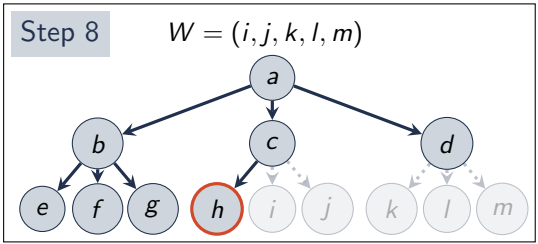


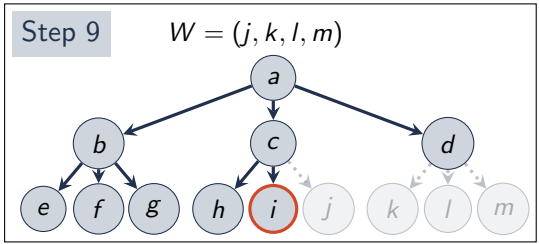


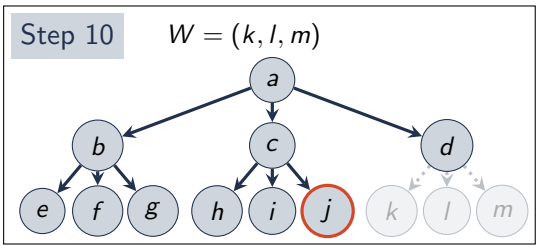


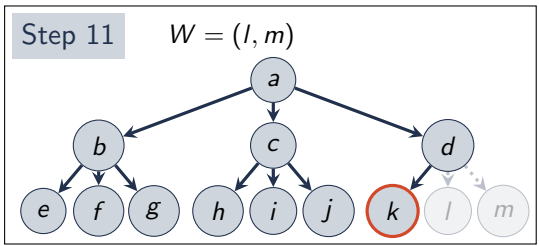
Breadth-First Search

continued – 2



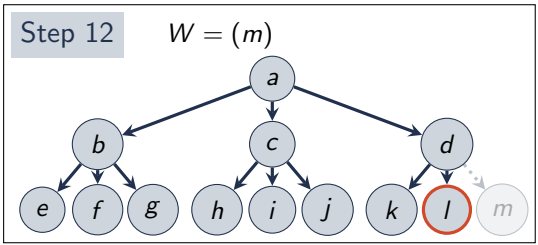


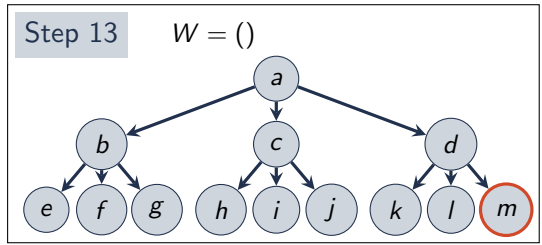




Breadth-First Search

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, δ, A)

```
1  W ← (q_0); // Queue
2  T[q_0] ← NIL; // Search Tree
3  while W do
4      let q ← dequeue(W) in
5          if q ∈ A then
6              return tree-path(T, q);
7          else
8              foreach q' ∈ δ(q) do
9                  if ¬contains(T,q') then
10                     T[q'] ← q;
11                     W ← enqueue(q', W);
12 return NIL;
```

Outline

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

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

Trade-offs

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

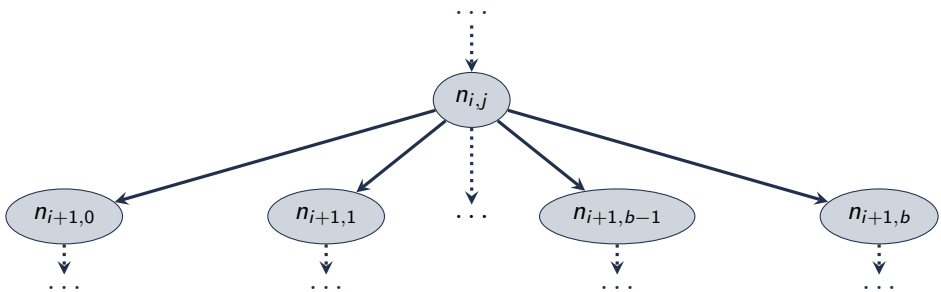
- Computation:
  - Time
  - Space
- Incomplete information / model limitations
- Infinite Spaces

Outline

Branching Factor

Definition (Branching Factor)

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

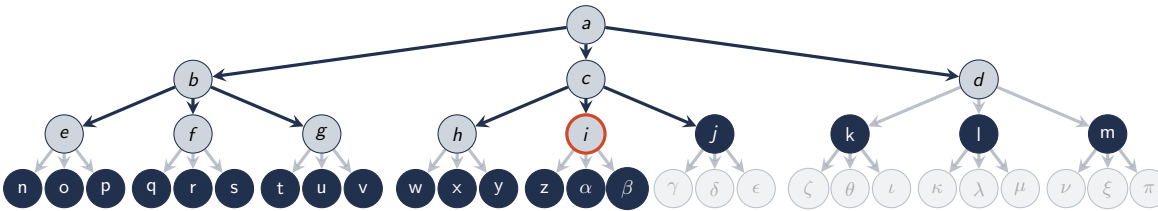


BFS Memory Use

b: branching factor d: current depth

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

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



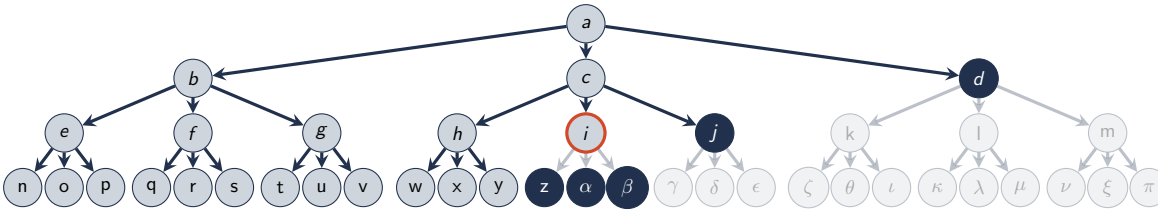
Memory use dominated by queued "next" level.

DFS Memory Use

b: branching factor d: current depth

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

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



Small work list
(still have visited set)

DFS vs. BFS

DFS

Pro: Compact work list (linear)

Con: Not optimal

BFS

Pro: Optimal (for equal step cost)

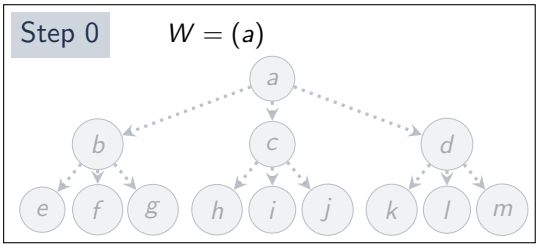
Con: Exponential work list

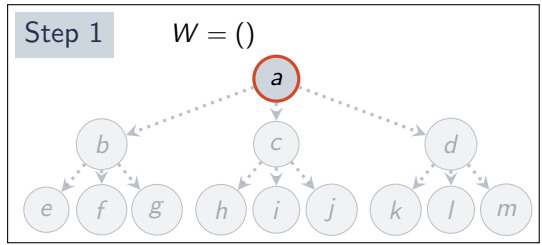
Iterative-Deepening Search (IDS) Overview

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

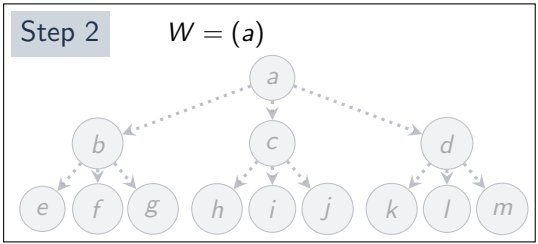
Start with depth limit 0

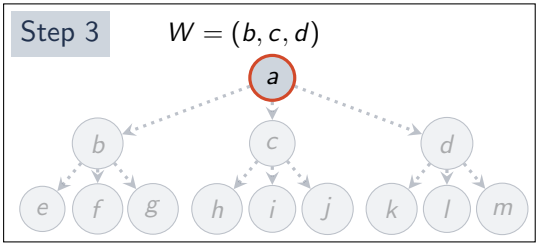


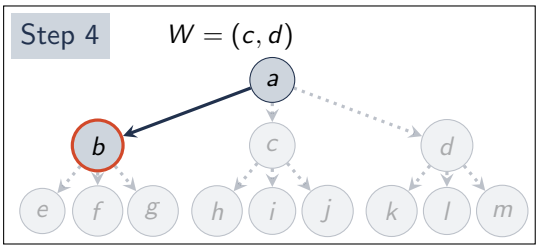


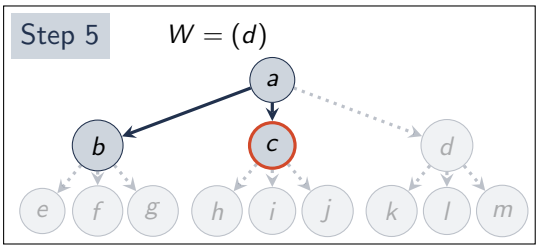
Iterative-Deepening Search

Retry with depth limit 1



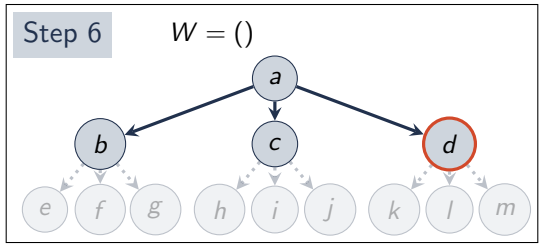






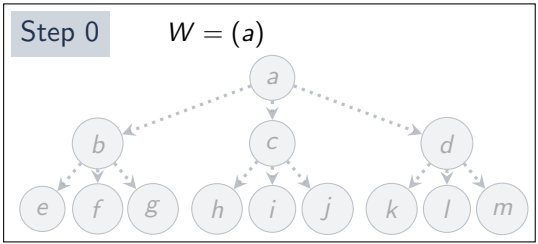
Iterative-Deepening Search

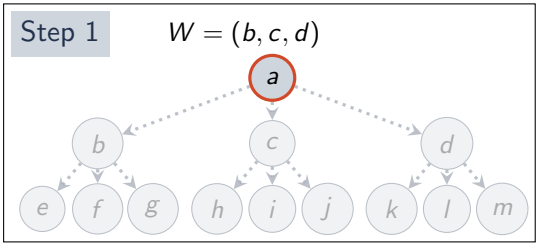
Retry with depth limit 1 – continued 1

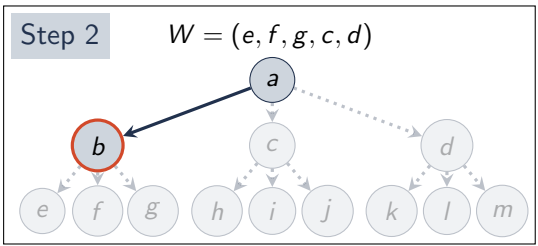


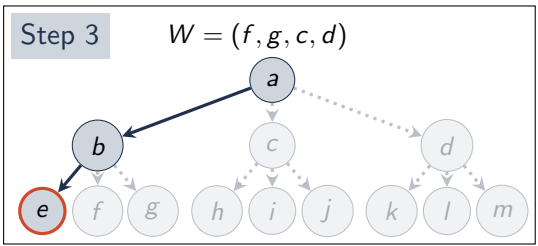
Iterative-Deepening Search

Retry with depth limit 2



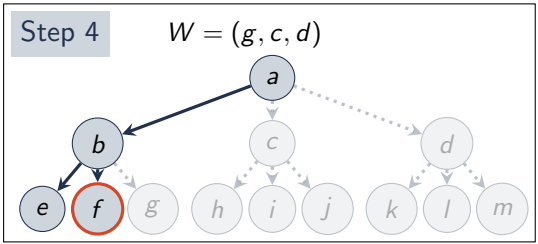


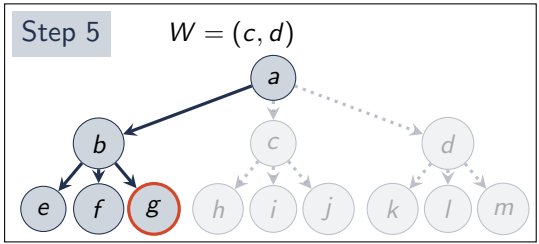


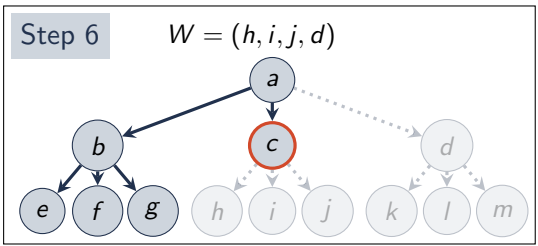


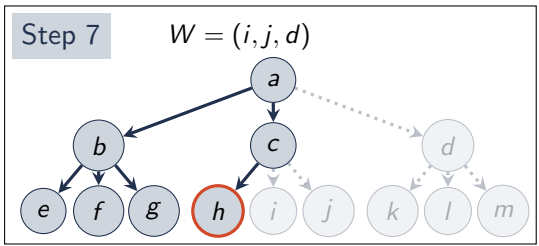
Iterative-Deepening Search

Retry with depth limit 2 – continued 1



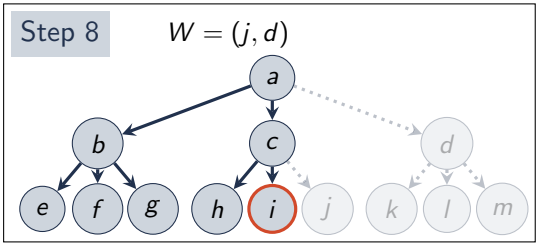


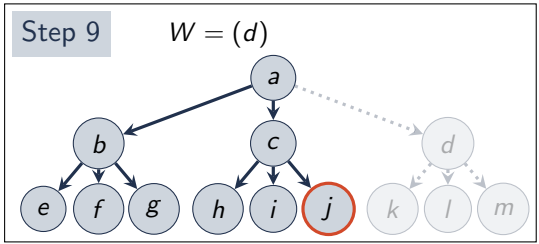


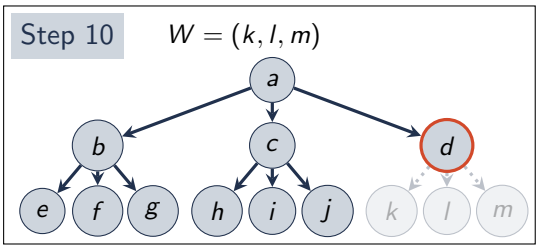


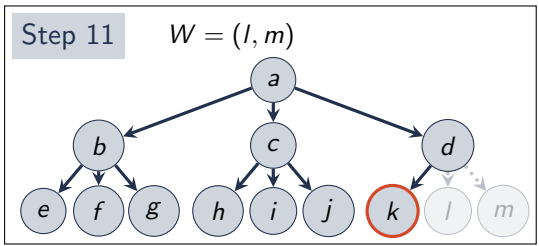
Iterative-Deepening Search

Retry with depth limit 2 – continued 2



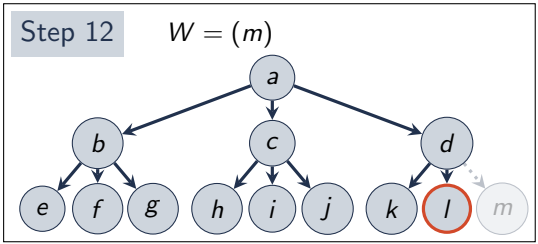


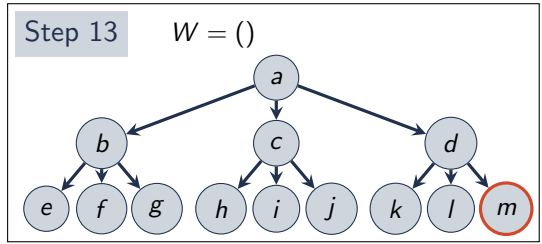




Iterative-Deepening Search

Retry with depth limit 2 – continued 3





Depth-Limited Search Algorithm

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

```
1  T[q_0] ← NIL ; // Search Tree
2  function visit(q, d) is
3      if q ∈ A then return tree-path(T, q) ;
4      if (d ≤ 0) then return NIL;
5      else
6          foreach q' ∈ δ(q) do
7              if ¬contains(T,q') then
8                  T[q'] ← q;
9                  let p ← visit(q', d − 1) in
10                     if p then return p ;
11         return NIL
12 return (visit(q_0, d_max), T);
```

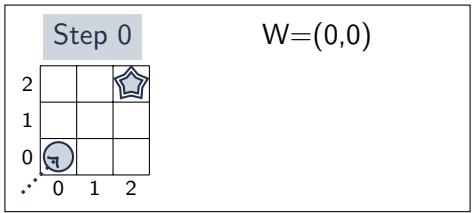
Iterative-Deepening Search Algorithm

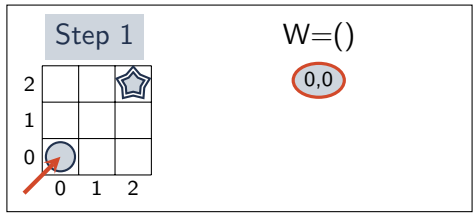
Procedure ids(q_0, δ, A)

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

Example: IDS Gridworld

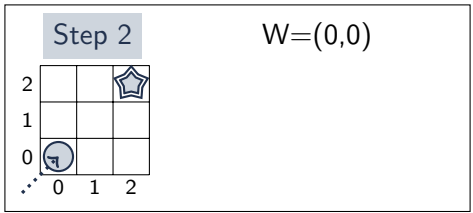
Depth limit 0

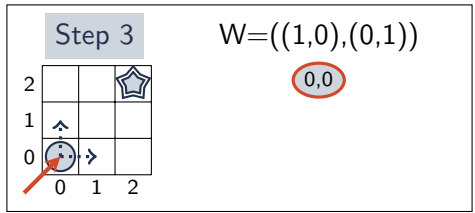


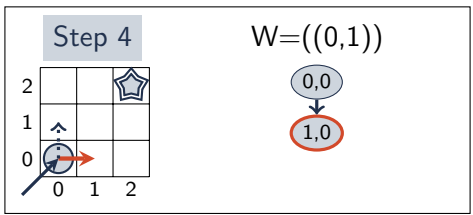


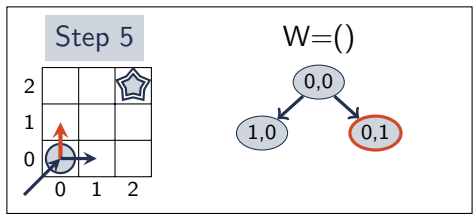
Example: IDS Gridworld

Depth limit 1



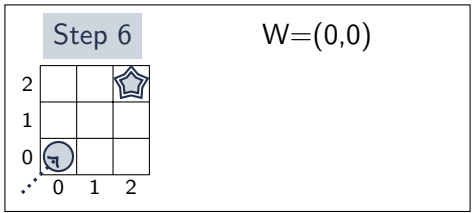


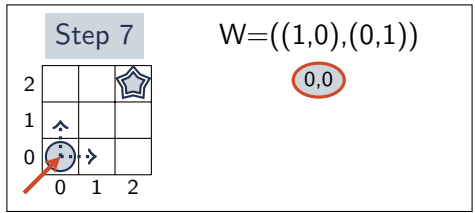


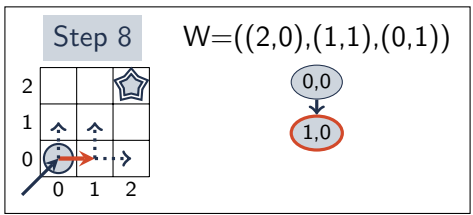


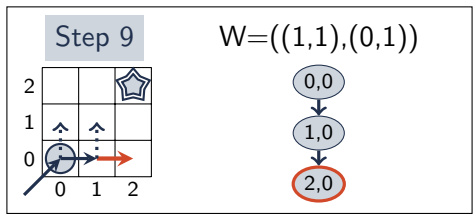
Example: IDS Gridworld

Depth limit 2



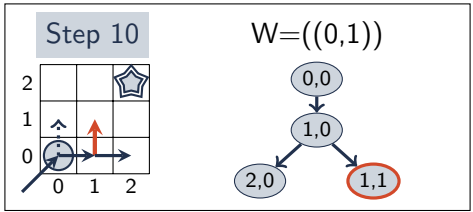


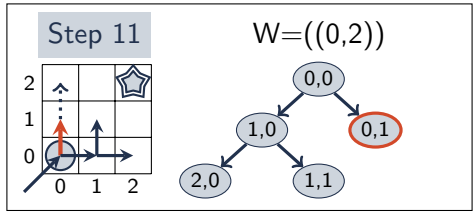


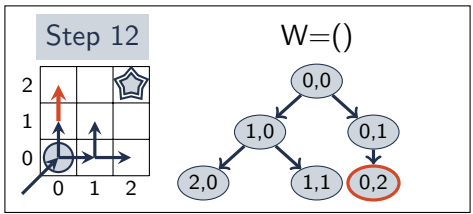


Example: IDS Gridworld

Depth limit 2, continued 1

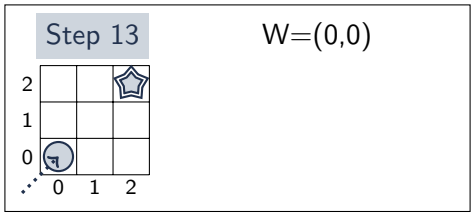


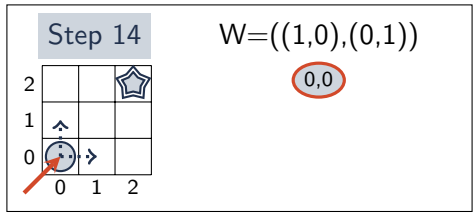


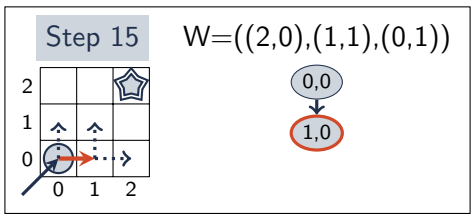


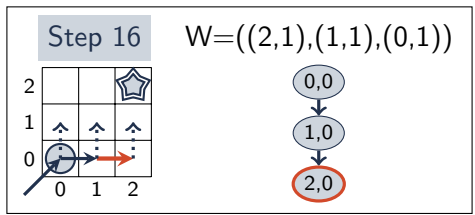
Example: IDS Gridworld

Depth limit 3



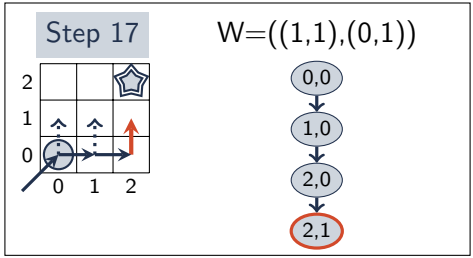


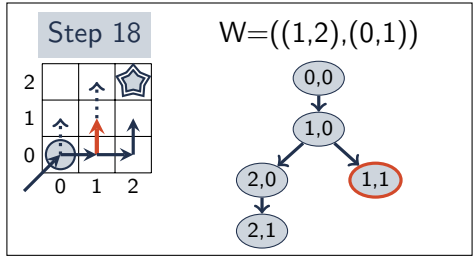


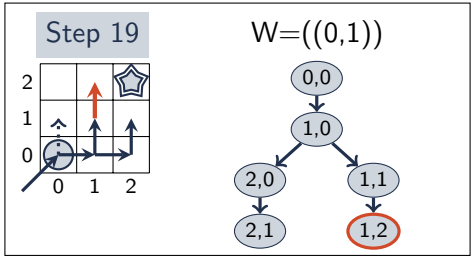


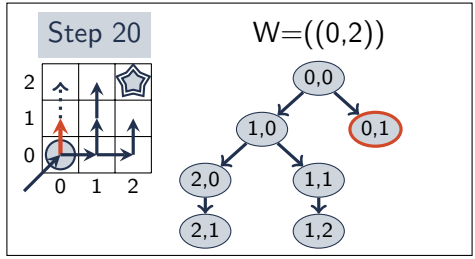
Example: IDS Gridworld

Depth limit 3, continued 1



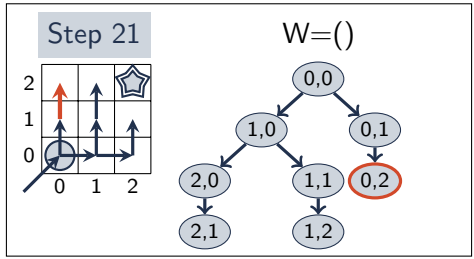






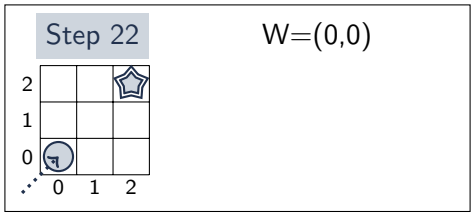
Example: IDS Gridworld

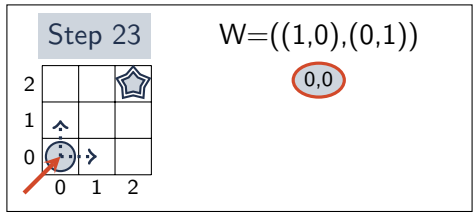
Depth limit 3, continued 2

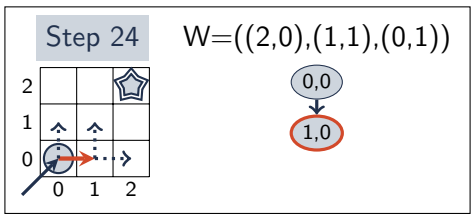


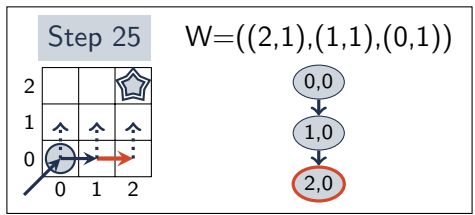
Example: IDS Gridworld

Depth limit 4



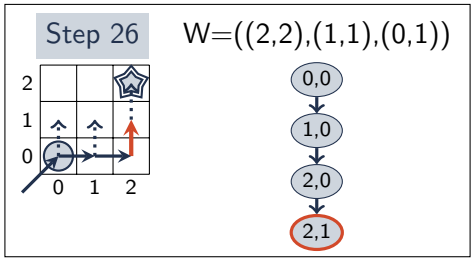


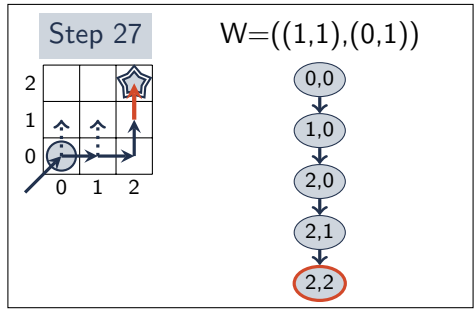




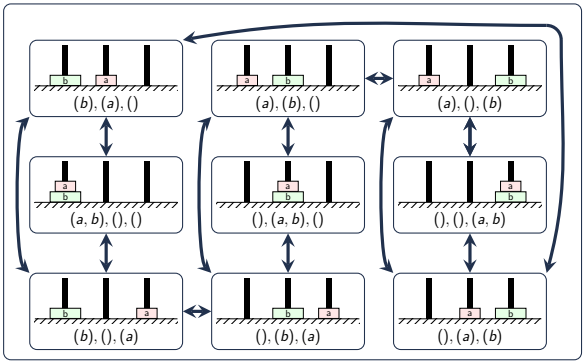
Example: IDS Gridworld

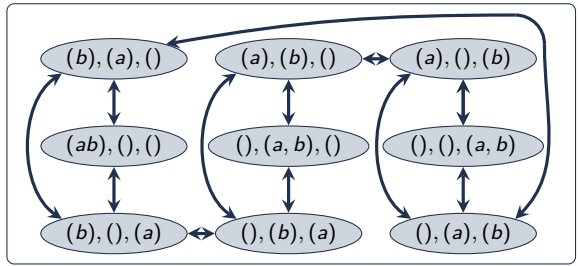
Depth limit 4, continued 1





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

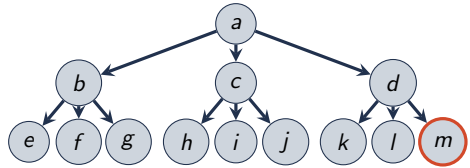
Procedure back-bfs(q_0, δ_b, A)

```
1  foreach q ∈ A do
2      W_b ← enqueue(q, W_b); // Queue
3      T_b[q] ← NIL; // Tree
4  while W_b do
5      let q ← dequeue(W_b) in
6          if q = q_0 then
7              return reverse-tree-path(T_b, q);
8          else
9              foreach q' ∈ δ_b(q) do
10                 if ¬contains(T, q') then
11                     T_b[q'] ← q;
12                     W_b ← enqueue(q', W_b);
13     return NIL;
```

Reverse-Tree-Path

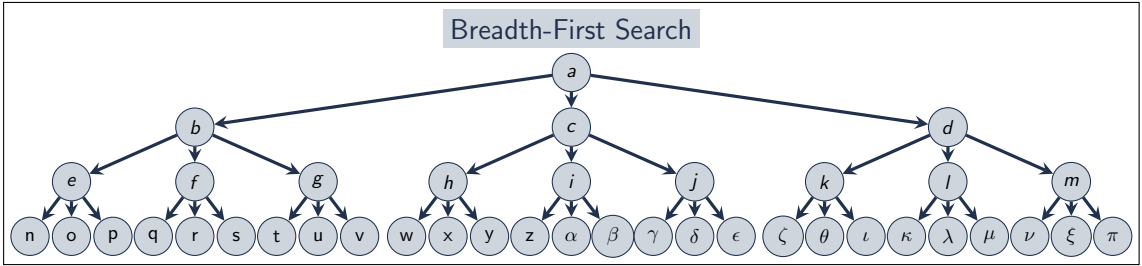
Procedure reverse-tree-path(T, q)

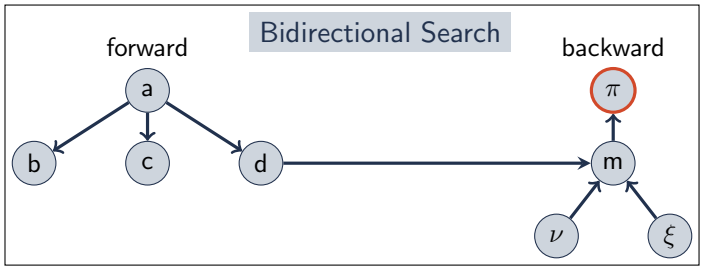
```
1 function rec(q) is
2     if q then // Recursive Case
3         return cons(q, rec(T[q]))
4     else // Base Case:  at the root
5         return NIL
6 return rec(q);
```



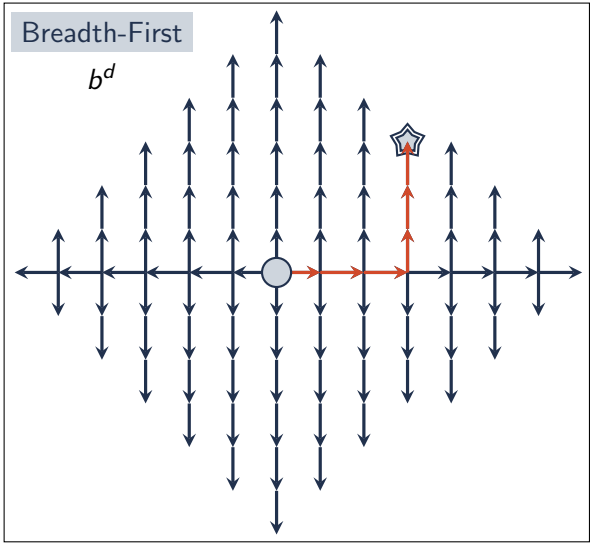
1. `rec(m)`
2. `cons(m, rec(d))`
3. `cons(m, cons(d, rec(a)))`
4. `cons(m, cons(d, cons(a, rec(nil))))`
5. `cons(m, cons(d, cons(a, nil)))`
6. `(m, d, a)`

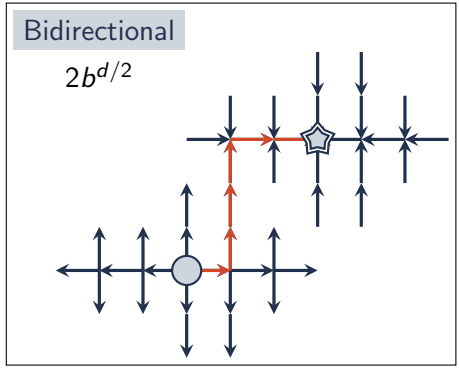
Bidirectional Search





Breadth-First vs Bidirectional Search Trees





BDS Algorithm

Procedure bds($q_0, \delta_f, \delta_b, A$)

```
1  W_f ← (q_0); // Init Forward Queue
2  T_f[q] ← NIL; // Init Forward Tree
3  foreach q ∈ A do // Init Backwards
4      W_b ← enqueue(q, W_b); // Backward Queue
5      T_b[q] ← NIL; // Backward Tree
6  while W_f ∧ W_b do
       /* Forward Step                                   */
7      (q, W_f) ← grow-bds(W_f, T_f, T_b, δ_f) ;
8      if q then return bds-result(T_f, T_b, q) ;
       /* Backward Step                                  */
9      (q, W_b) ← grow-bds(W_b, T_b, T_f, δ_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' ∈ δ(q) do
3         if ¬contains(T,q') then
4             T[q'] ← q;
5             if contains(T_o,q') then
                  // Found shared node
6                 return (q',W);
7             else
8                 W ← enqueue(q', W)
9 return (NIL,W);
```

Procedure bds-result(T_f, T_b, q)

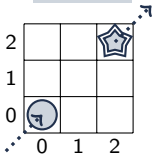
```
  /* Path from root to q                */
1 p_f ← tree-path(T_f, q);
  /* Path from q+1 to goal              */
2 p_b ←
    reverse-tree-path(T_b, T_b[q]);
3 return append(p_f, p_b);
```

Example: BDS Gridworld

Step 0

$W_f = ((0,0))$

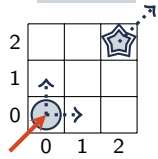
$W_b = ((2,2))$



Step 1

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

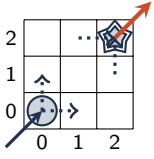
$W_b = ((2,2))$



Step 2

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

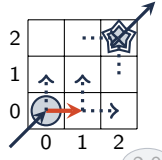
$W_b = ((1,2),(2,1))$



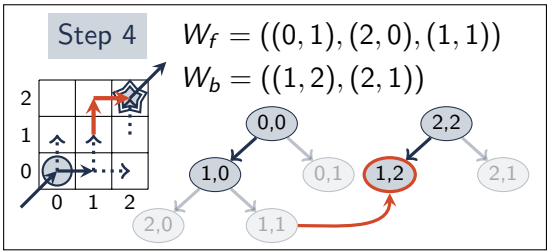
Step 3

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

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



Example: BDS Gridworld



Exercise: BDS Hanoi

Summary

- "Basic" Uninformed Search:
  - Depth-first Search (DFS)
  - Breadth-first Search (BFS)
- Search and Planning Properties:
  - Correctness
  - Completeness
  - Optimality
- "Advanced" Uninformed Search:
  - Iterative-Deepening
  - Backward Search
  - Bidirectional Search

MINES

References

Textbook Russell & Norvig.

- Ch 3.3. Searching for Solutions
- Ch 3.4. Uninformed Search Strategies