ARTIFICIAL INTELLIGENCE

RECAP – UNIFORMED VS INFORMED SEARCH

uninformed search algorithms—algorithms that are given no information about the problem other than its definition.

Breadth First Search

Uniform Cost Search

Depth First Search

Depth Limited Search

Iterative Deepening Depth First Search

Bi Directional Search

Informed search algorithms, on the other hand, can do quite well given some guidance on where to look for solutions

ITERATIVE DEEPENING SEARCH

To avoid the infinite depth problem of DFS:

- Only search until depth L
- i.e, don’t expand nodes beyond depth L
- Depth-Limited Search

What if solution is deeper than L?

- Increase depth iteratively
- Iterative Deepening Search

IDS — GENERALLY THE PREFERRED UNINFORMED SEARCH

- Inherits the memory advantage of depth-first search
- Has the completeness property of breadth-first search

DEPTH-LIMITED SEARCH & IDS

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   else if DEPTH[node] = limit then return cutoff
   else for each successor in EXPAND(node, problem) do
       result ← RECURSIVE-DLS(successor, problem, limit)
       if result = cutoff then cutoff-occurred? ← true
       else if result ≠ failure then return result
   if cutoff-occurred? then return cutoff else return failure
```

Goal test in recursive call, one-at-a-time

DEPTH-LIMITED SEARCH & IDS

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*)

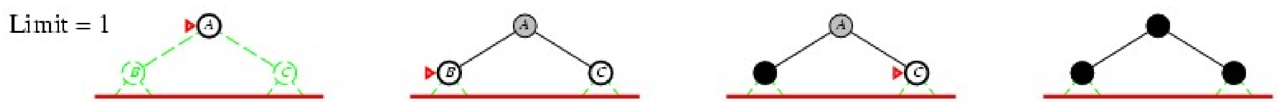
if *result* ≠ cutoff **then return** *result*

At *depth* = 0, IDS only goal-tests the start node.
The start node is is not expanded at *depth* = 0.

ITERATIVE DEEPENING SEARCH, L=0

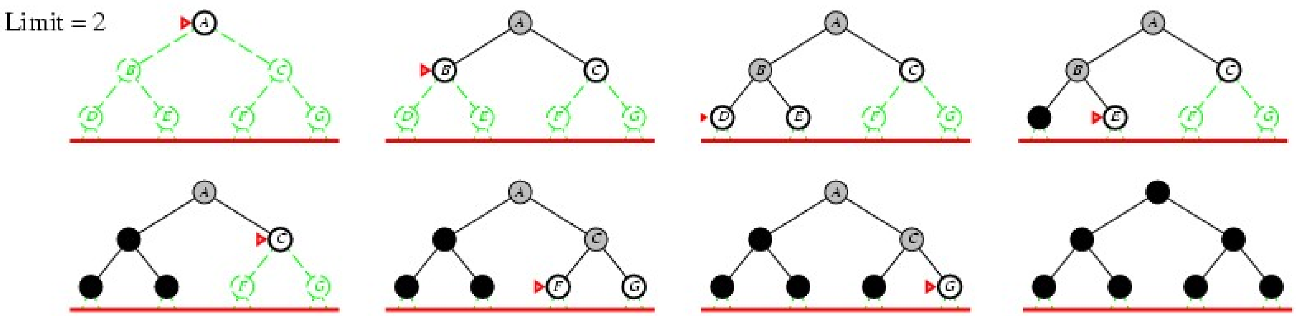
At L=0, the start node is goal-tested but no nodes are expanded. This is so that you can solve trick problems like, “Starting in Arad, go to Arad.”

ITERATIVE DEEPENING SEARCH, L=1



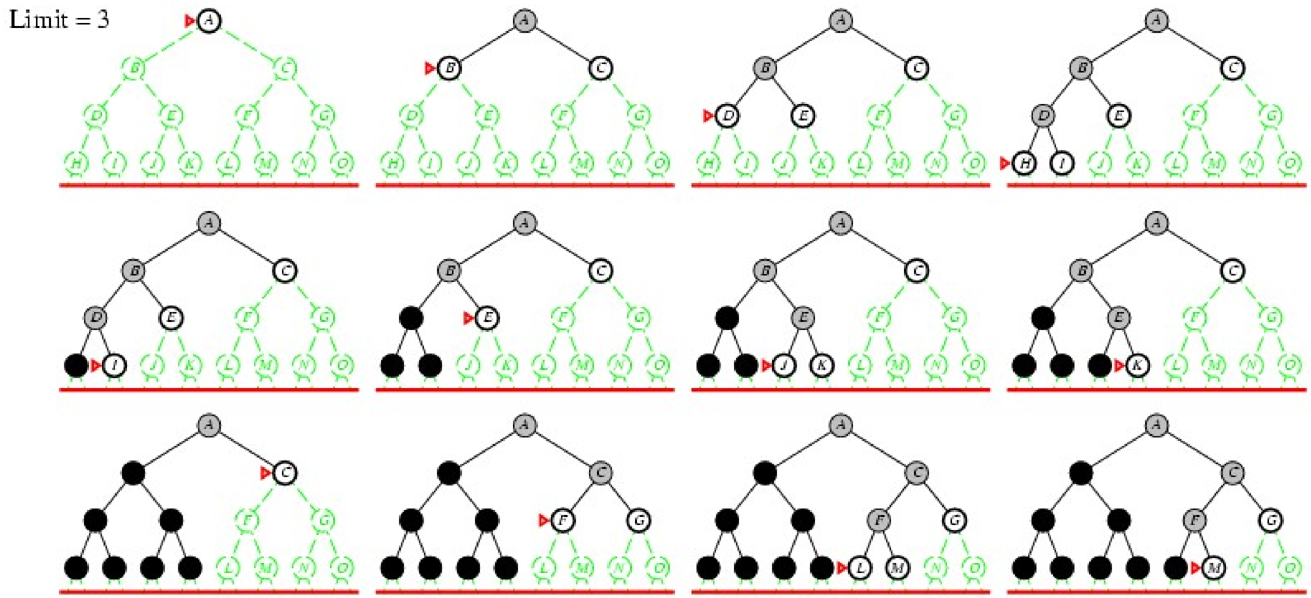
At L=1, the start node is expanded. Its children are goal-tested, but not expanded. Recall that to expand a node means to generate its children.

ITERATIVE DEEPENING SEARCH, L=2



At L=2, the start node and its children are expanded.
Its grand-children are goal-tested, but not expanded.

ITERATIVE DEEPENING SEARCH, L=3



At L=3, the start node, its children, and its grand-children are expanded. Its great-grand-children are goal-tested, but not expanded.

PROPERTIES OF ITERATIVE DEEPENING SEARCH

Complete? Yes

Time? $O(b^d)$

Space? $O(bd)$

Optimal? No, for general cost functions.

Yes, if cost is a non-decreasing function only of depth.

Generally the preferred uninformed search strategy.

BIDIRECTIONAL SEARCH

bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet.

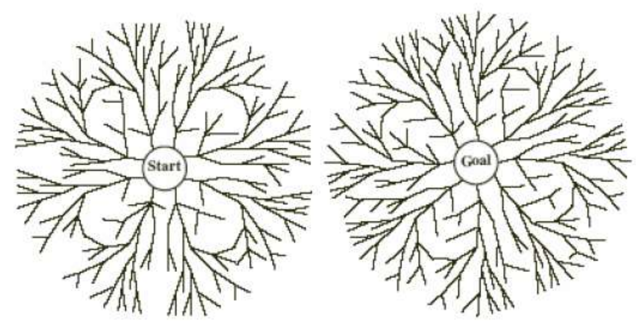
- **Simultaneously:**
  - **Search forward from start**
  - **Search backward from the goal**

Stop when the two searches meet.

- **If branching factor = b in each direction,**

with solution at depth d

➔ only $O(2\ b^{d/2})$= $O(2\ b^{d/2})$



BIDIRECTIONAL SEARCH

Key limitations:

How to search backwards can be an issue (e.g., in Chess)? What's tricky?

Problem: lots of states satisfy the goal; don't know which one is relevant.

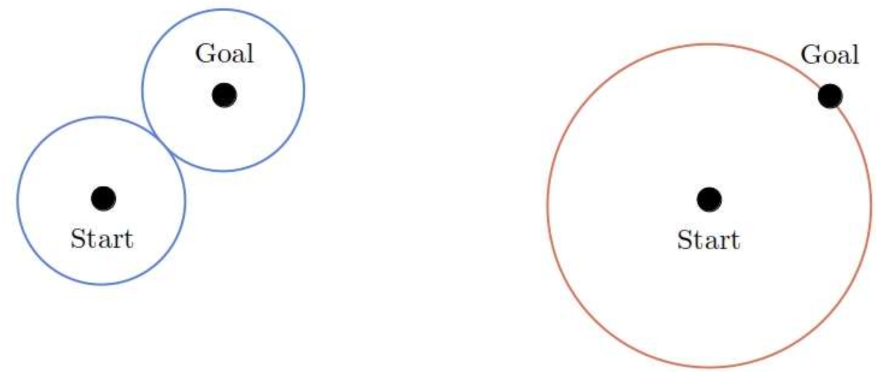
Aside: The predecessor of a node should be easily computable (i.e., actions are easily reversible).

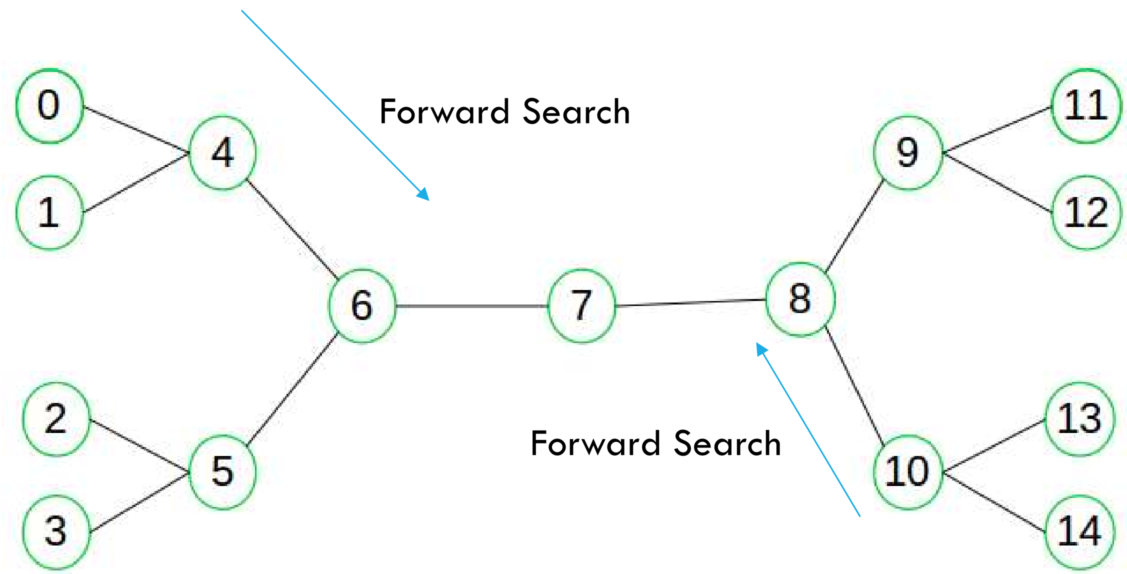
BIDIRECTIONAL SEARCH

Consider node 0 to be initial state

While node 14 is the goal node

Nodes expanded in a bidirectional search vs. those expanded in a unidirectional search.





BIDIRECTIONAL SEARCH

For this to work, we need to keep track of two frontiers and two tables of reached states,

and we need to be able to reason backwards: if state s0 is a successor of s in the forward

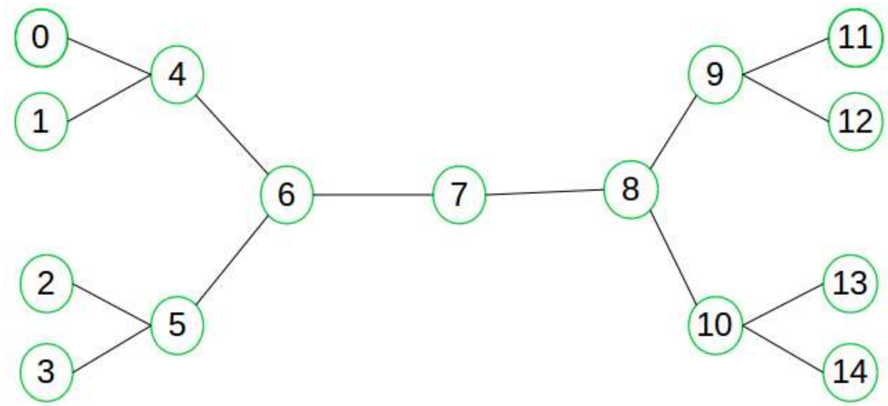
direction, then we need to know that s is a successor of s0 in the backward direction.

We have a solution when the two frontiers collide

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                        // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                        // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F, frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
          // Expand node on frontier; check against the other frontier in reached_2.
          // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

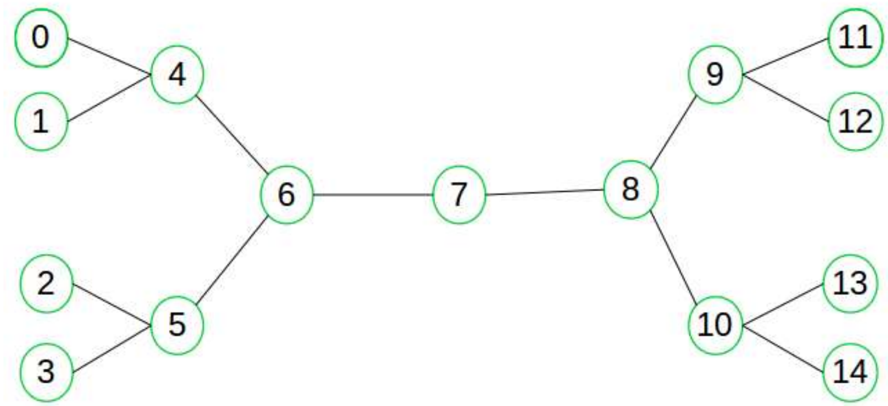
Figure 3.14 Bidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.



Nodef = 0

Nodeb = 14

function BIBF-SEARCH($problem_F$, f_F, $problem_B$, f_B) **returns** a solution node, or *failure*
  $node_F \leftarrow$ NODE($problem_F$.INITIAL) // *Node for a start state*
  $node_B \leftarrow$ NODE($problem_B$.INITIAL) // *Node for a goal state*
  $frontier_F \leftarrow$ a priority queue ordered by f_F, with $node_F$ as an element
  $frontier_B \leftarrow$ a priority queue ordered by f_B, with $node_B$ as an element
  $reached_F \leftarrow$ a lookup table, with one key $node_F$.STATE and value $node_F$
  $reached_B \leftarrow$ a lookup table, with one key $node_B$.STATE and value $node_B$
  $solution \leftarrow failure$
  while not TERMINATED($solution$, $frontier_F$, $frontier_B$) **do**
    if f_F(TOP($frontier_F$)) < f_B(TOP($frontier_B$)) **then**
      $solution \leftarrow$ PROCEED(F, $problem_F$, $frontier_F$, $reached_F$, $reached_B$, $solution$)
    else $solution \leftarrow$ PROCEED(B, $problem_B$, $frontier_B$, $reached_B$, $reached_F$, $solution$)
  return $solution$

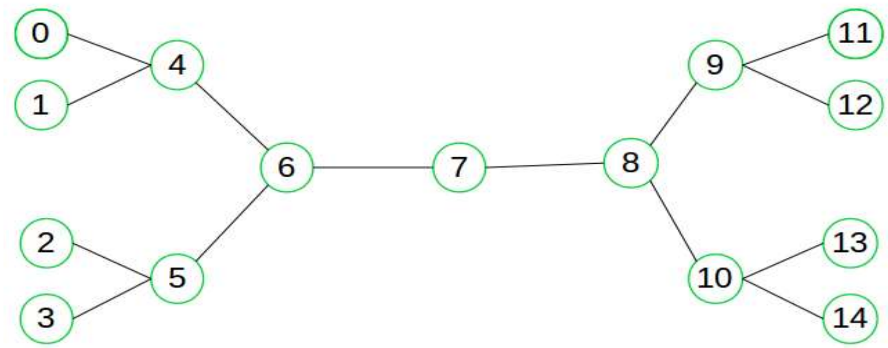


Frontierf = Nodef
frontier = Nodeb

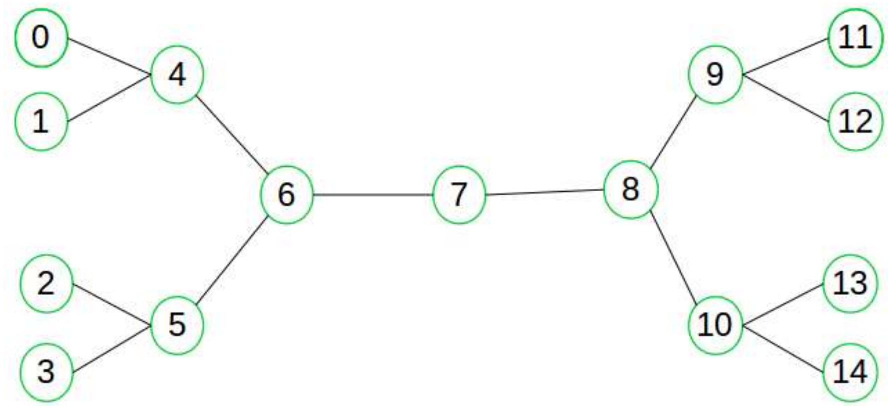
function BIBF-SEARCH(*problem*$_F$, f_F, *problem*$_B$, f_B) **returns** a solution node, or *failure*
  node$_F$ ← NODE(*problem*$_F$.INITIAL) // *Node for a start state*
  node$_B$ ← NODE(*problem*$_B$.INITIAL) // *Node for a goal state*
  frontier$_F$ ← a priority queue ordered by f_F, with *node*$_F$ as an element
  frontier$_B$ ← a priority queue ordered by f_B, with *node*$_B$ as an element
  reached$_F$ ← a lookup table, with one key *node*$_F$.STATE and value *node*$_F$
  reached$_B$ ← a lookup table, with one key *node*$_B$.STATE and value *node*$_B$
  solution ← *failure*
  while not TERMINATED(*solution*, *frontier*$_F$, *frontier*$_B$) **do**
    if f_F(TOP(*frontier*$_F$)) < f_B(TOP(*frontier*$_B$)) **then**
      solution ← PROCEED(*F*, *problem*$_F$, *frontier*$_F$, *reached*$_F$, *reached*$_B$, *solution*)
    else *solution* ← PROCEED(*B*, *problem*$_B$, *frontier*$_B$, *reached*$_B$, *reached*$_F$, *solution*)
  return *solution*

If cost(0-4) < cost(14-10)

- Then proceed forward
  else proceed backward

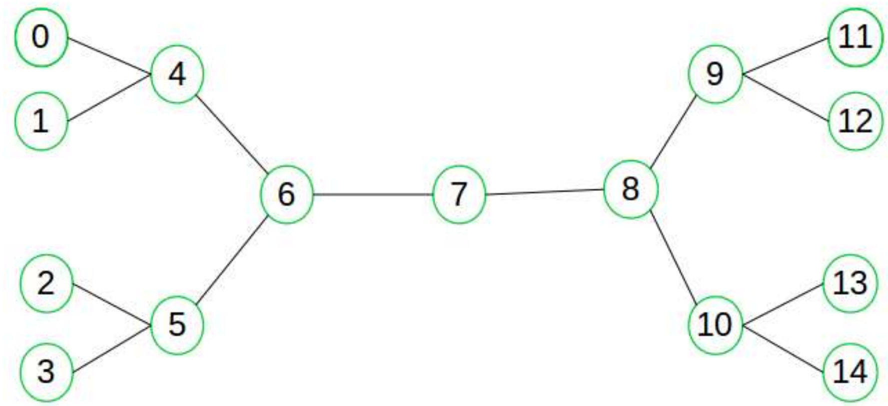


function BIBF-SEARCH($problem_F$, f_F, $problem_B$, f_B) **returns** a solution node, or *failure*
  $node_F \leftarrow$ NODE($problem_F$.INITIAL) *// Node for a start state*
  $node_B \leftarrow$ NODE($problem_B$.INITIAL) *// Node for a goal state*
  $frontier_F \leftarrow$ a priority queue ordered by f_F, with $node_F$ as an element
  $frontier_B \leftarrow$ a priority queue ordered by f_B, with $node_B$ as an element
  $reached_F \leftarrow$ a lookup table, with one key $node_F$.STATE and value $node_F$
  $reached_B \leftarrow$ a lookup table, with one key $node_B$.STATE and value $node_B$
  $solution \leftarrow failure$
  while not TERMINATED($solution$, $frontier_F$, $frontier_B$) **do**
    if f_F(TOP($frontier_F$)) $<$ f_B(TOP($frontier_B$)) **then**
      $solution \leftarrow$ PROCEED(F, $problem_F$, $frontier_F$, $reached_F$, $reached_B$, $solution$)
    else $solution \leftarrow$ PROCEED(B, $problem_B$, $frontier_B$, $reached_B$, $reached_F$, $solution$)
  return $solution$




```
function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

Get the next nodes – predecessor nodes or successor nodes



If not in reached -> haven't explored this before

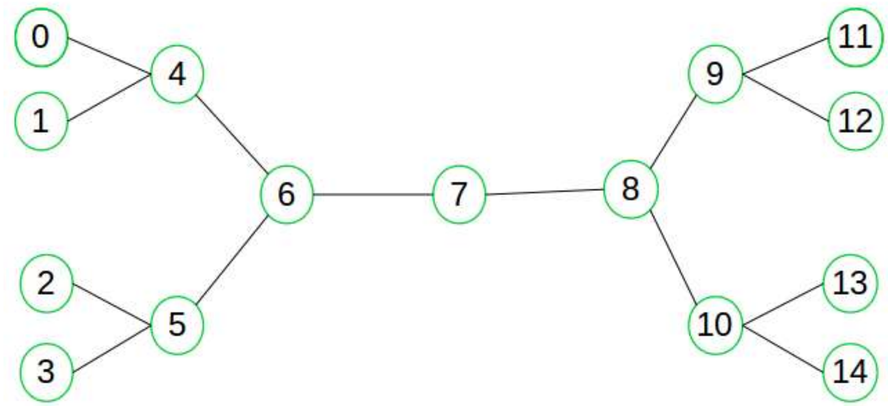
OR

It is in reached but with a higher path cost

-> update in reached (as UCS)

-> add to frontier

```
function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```



If not in reached and we just added it

OR

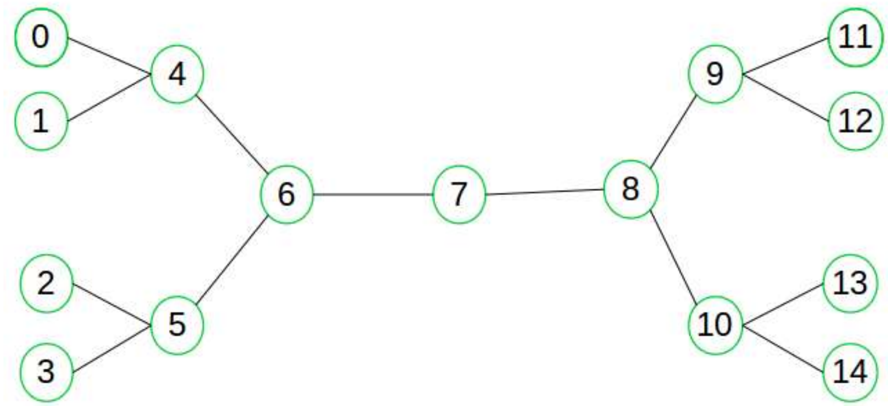
It is in reached and we updated it

Then check

If it is in reached2 (seen from other path too)

Join these path (keeping track of direction!)

```
function PROCEED(dir, problem, frontier, reached, reached2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached2 then
        solution2 ← JOIN-NODES(dir, child, reached2[s]))
        if PATH-COST(solution2) < PATH-COST(solution) then
          solution ← solution2
  return solution
```



Submit the lower cost solution as final solution

```
function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

SUMMARY OF ALGORITHMS

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

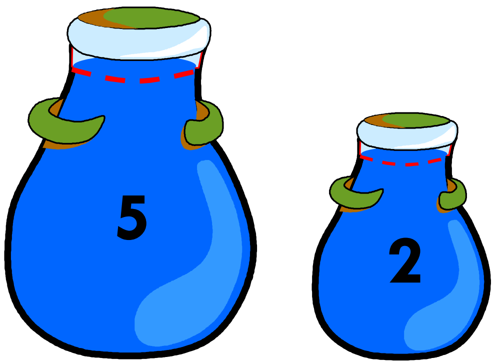
There are a number of footnotes, caveats, and assumptions.
See Fig. 3.21, p. 91.
[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon > 0$
[c] optimal if step costs are all identical
(also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
(also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Generally the preferred uninformed search strategy

Note that $d \leq \lfloor 1+C^*/\varepsilon \rfloor$

WATER JUG PROBLEM

Given a full 5-gallon jug and a full 2-gallon jug, fill the 2-gallon jug with exactly one gallon of water.



State: ?

Initial State: ?

Actions: ?

Goal State: ?

WATER JUG PROBLEM

State = (x,y), where x is the number of gallons of water in the 5-gallon jug and y is # of gallons in the 2-gallon jug

Initial State = (5,2)

Goal State = (*,1), where * means any amount

Actions table

Name	Cond.	Transition	Effect
Empty5	–	$(x,y)\to(0,y)$	Empty 5-gal. jug
Empty2	–	$(x,y)\to(x,0)$	Empty 2-gal. jug
2to5	$x \leq 3$	$(x,2)\to(x+2,0)$	Pour 2-gal. into 5-gal.
5to2	$x \geq 2$	$(x,0)\to(x-2,2)$	Pour 5-gal. into 2-gal.
5to2part	$y < 2$	$(1,y)\to(0,y+1)$	Pour partial 5-gal. into 2-gal.