Artificial Intelligence

Lec 5 - Search (contd.)

Pratik Mazumder

Content taken from various sources including
CS188 Intro to AI at UC Berkeley

Memory-bounded Heuristic Search

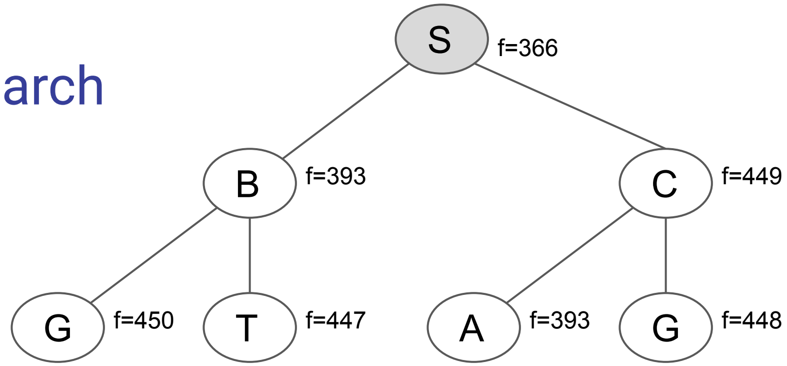
Iterative Deepening A* (IDA*) search

- Standard Iterative Deepening performs DFS upto a cutoff depth.
- If no solution is found then the cutoff depth is increased for the next iteration.
- The main difference between IDA* and standard iterative deepening is that the cutoff used is the f-cost (g + h) rather than the depth.
- At each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration.

Memory-bounded Heuristic Search

Iterative Deepening A* (IDA*) search

- Is IDA* optimal?



Memory-bounded Heuristic Search

Recursive Best-First Search (RBFS)

- Simple recursive algorithm that expands nodes in a best-first order, i.e., expands nodes with the lowest f-value first.
- Keeps **track** of the **f-value of the best alternative path** available from any ancestor of the current node.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path.
- As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value—the best f-value of its children.
- In this way, RBFS **remembers the f-value of the best leaf in the forgotten subtree** and can therefore decide whether it's worth re-expanding that subtree in the future.

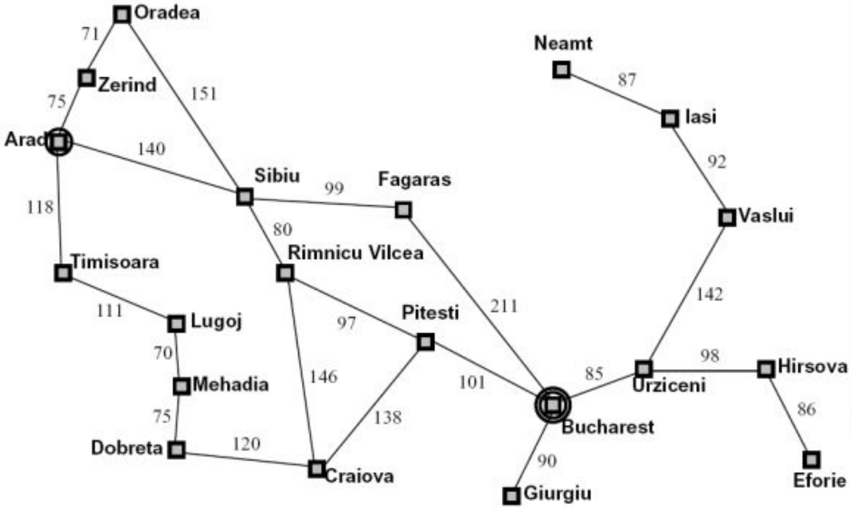
Memory-bounded Heuristic Search

Recursive Best-First Search (RBFS)

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
  return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
     add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do  /* update f with value from previous search, if any */
     s.f ← max(s.g + s.h, node.f))
  loop do
     best ← the lowest f-value node in successors
     if best.f > f_limit then return failure, best.f
     alternative ← the second-lowest f-value among successors
     result, best.f ← RBFS(problem, best, min(f_limit, alternative))
     if result ≠ failure then return result
```

Recursive Best-First Search (RBFS)

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
   return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do  /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```



Straight−line distance to Bucharest

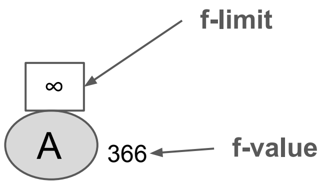
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Recursive Best-First Search (RBFS)

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
  return RBFS(*problem*, MAKE-NODE(*problem*.INITIAL-STATE), ∞)

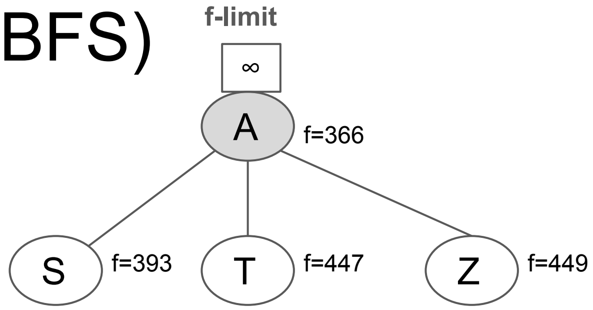
function RBFS(*problem*, *node*, *f_limit*) **returns** a solution, or failure and a new f-cost limit
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  successors $\leftarrow$ [ ]
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    add CHILD-NODE(*problem*, *node*, *action*) into *successors*
  if *successors* is empty **then return** *failure*, ∞
  for each *s* **in** *successors* **do** /* update f with value from previous search, if any */
    $s.f \leftarrow \max(s.g + s.h,\ node.f)$)
  loop do
    best $\leftarrow$ the lowest f-value node in *successors*
    if $best.f > f\_limit$ **then return** *failure*, $best.f$
    alternative $\leftarrow$ the second-lowest f-value among *successors*
    result, $best.f \leftarrow$ RBFS(*problem*, *best*, min(*f_limit*, *alternative*))
    if *result* $\neq$ *failure* **then return** *result*



Recursive Best-First Search (RBFS)

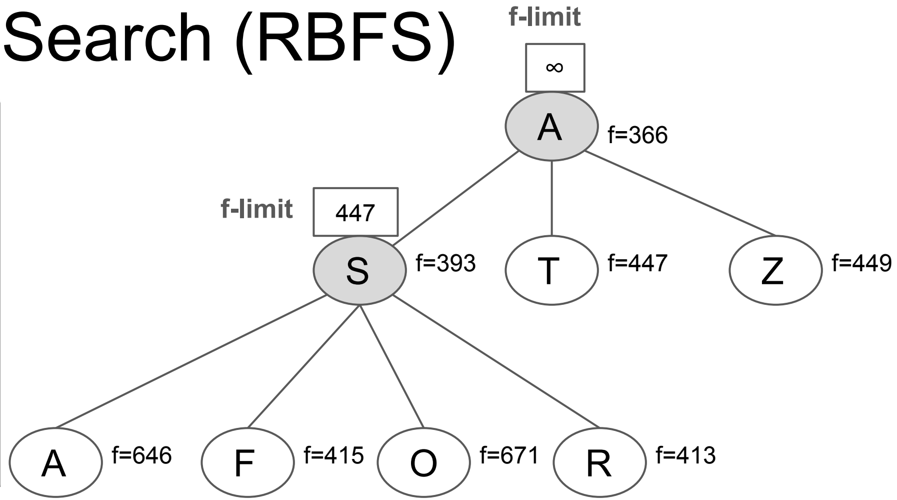
function RECURSIVE-BEST-FIRST-SEARCH($problem$) **returns** a solution, or failure
  return RBFS($problem$, MAKE-NODE($problem$.INITIAL-STATE), ∞)

function RBFS($problem$, $node$, f_limit) **returns** a solution, or failure and a new f-cost limit
  if $problem$.GOAL-TEST($node$.STATE) **then return** SOLUTION($node$)
  $successors \leftarrow$ [ ]
  for each $action$ **in** $problem$.ACTIONS($node$.STATE) **do**
    add CHILD-NODE($problem$, $node$, $action$) into $successors$
  if $successors$ is empty **then return** $failure$, ∞
  for each s **in** $successors$ **do** /* update f with value from previous search, if any */
    $s.f \leftarrow \max(s.g + s.h,\ node.f)$
  loop do
    $best \leftarrow$ the lowest f-value node in $successors$
    if $best.f > f\_limit$ **then return** $failure$, $best.f$
    $alternative \leftarrow$ the second-lowest f-value among $successors$
    $result$, $best.f \leftarrow$ RBFS($problem$, $best$, min(f_limit, $alternative$))
    if $result \neq failure$ **then return** $result$



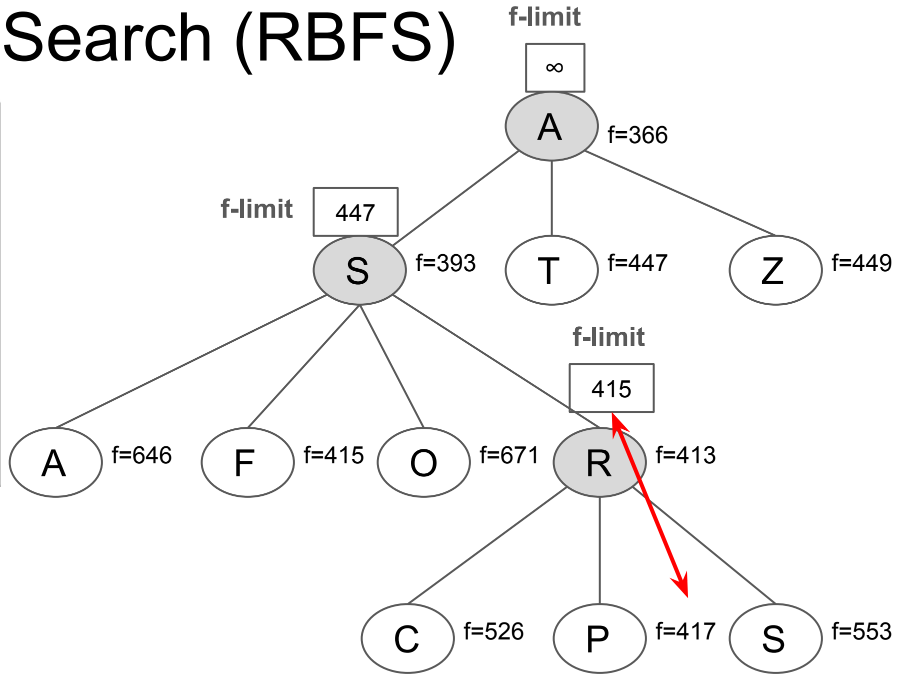
Recursive Best-First Search (RBFS)

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```



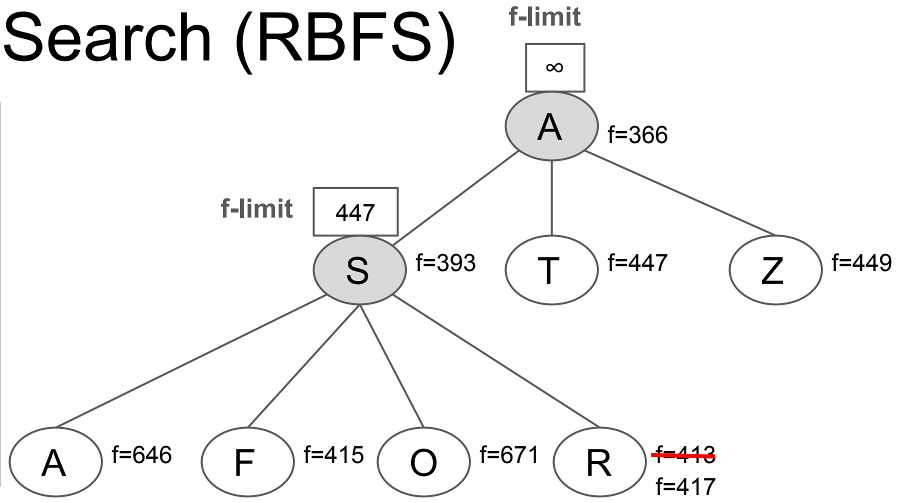
Recursive Best-First Search (RBFS)

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```



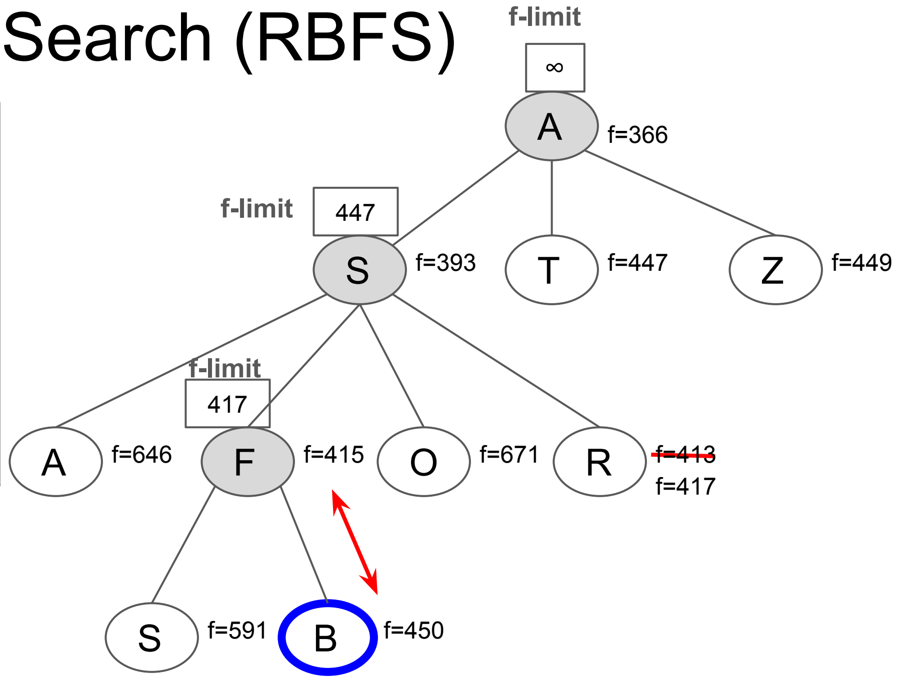
Recursive Best-First Search (RBFS)

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do  /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```



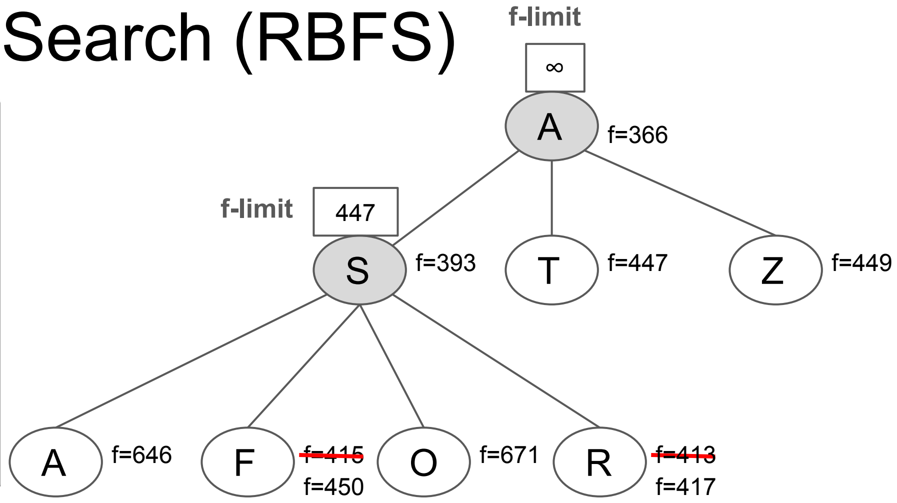
Recursive Best-First Search (RBFS)

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do  /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```



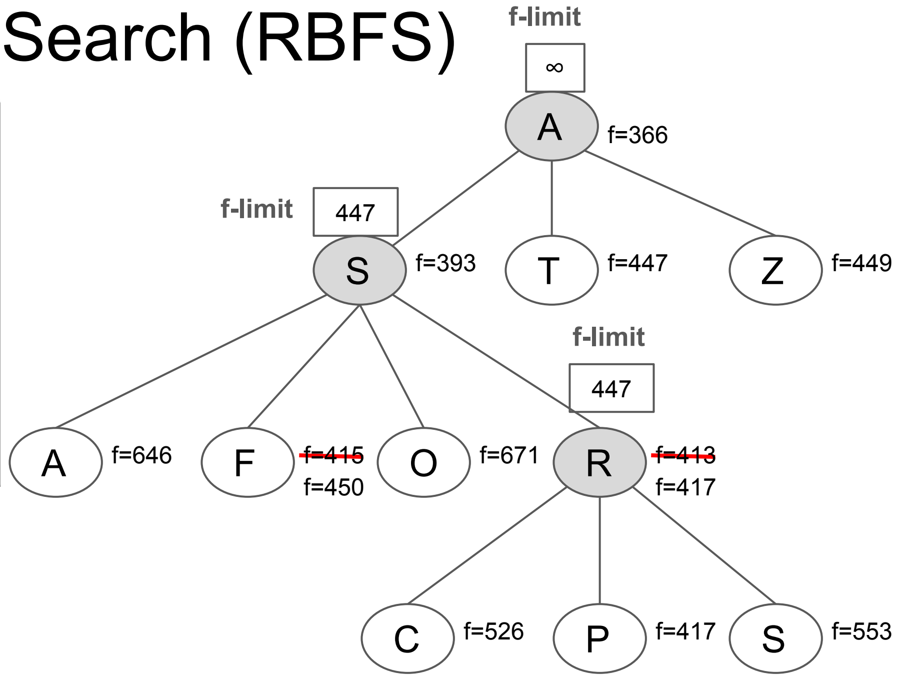
Recursive Best-First Search (RBFS)

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do  /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```



Recursive Best-First Search (RBFS)

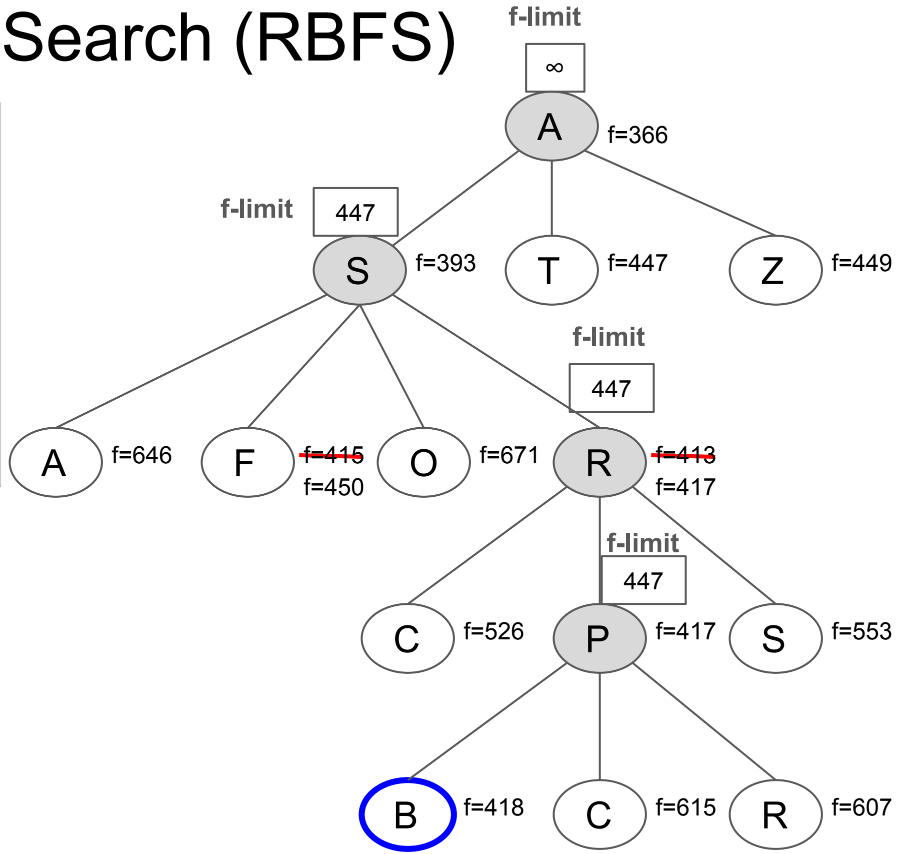
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```



Recursive Best-First Search (RBFS)

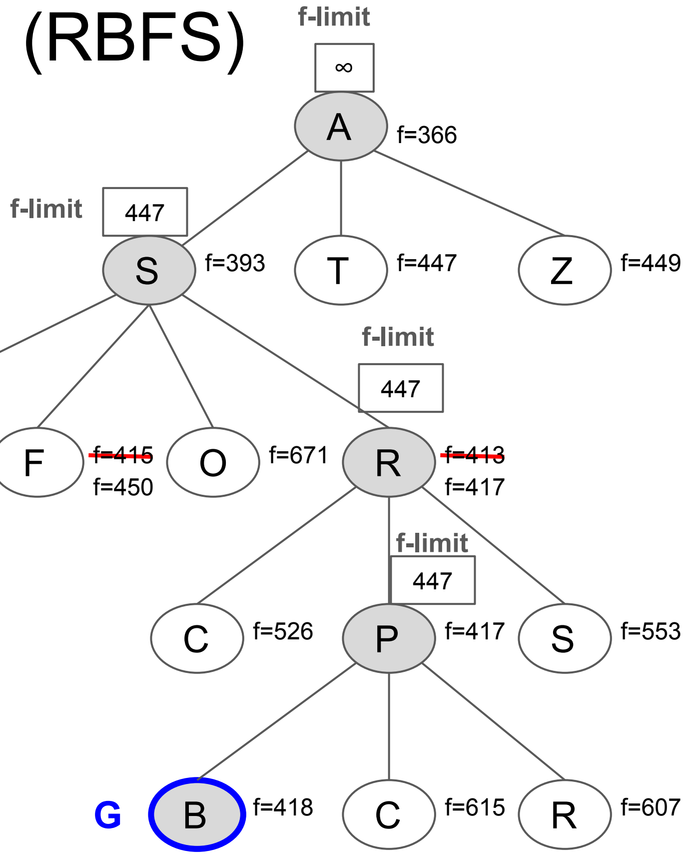
function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
  return RBFS(*problem*, MAKE-NODE(*problem*.INITIAL-STATE), ∞)

function RBFS(*problem*, *node*, *f_limit*) **returns** a solution, or failure and a new f-cost limit
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  successors ← [ ]
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    add CHILD-NODE(*problem*, *node*, *action*) into *successors*
  if *successors* is empty **then return** *failure*, ∞
  for each s **in** *successors* **do** /* update f with value from previous search, if any */
    $s.f \leftarrow \max(s.g + s.h,\ node.f)$
  loop do
    best ← the lowest f-value node in *successors*
    if $best.f > f\_limit$ **then return** *failure*, $best.f$
    alternative ← the second-lowest f-value among *successors*
    result, $best.f$ ← RBFS(*problem*, *best*, min(*f_limit*, *alternative*))
    if *result* ≠ *failure* **then return** *result*



Recursive Best-First Search (RBFS)

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do  /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```



Recursive Best-First Search (RBFS)

Difference between A* and RBFS

- A* keeps in memory all of the already generated nodes

- RBFS only keeps the current search path and the alternatives along the path

What does it do when a subtree is suspended?

- It forgets the subtree to save space

When RBFS suspends searching a subtree, what does it remember?

- An updated f-value of the root of the subtree

How does RBFS explore subtrees?

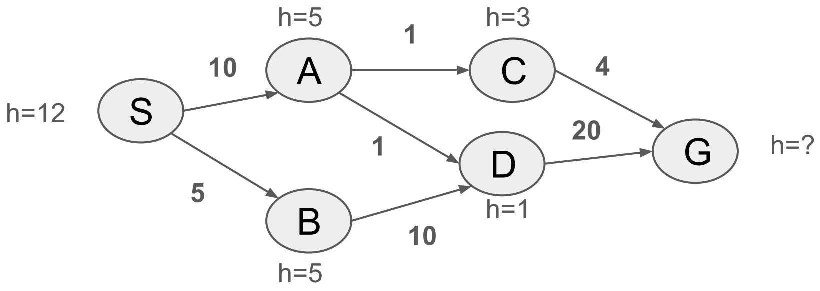
- As in A*, within a given f-bound

How is the bound determined?

- From the F-values of the siblings along the current search path
- The smallest F-value – The closest competitor

Practice

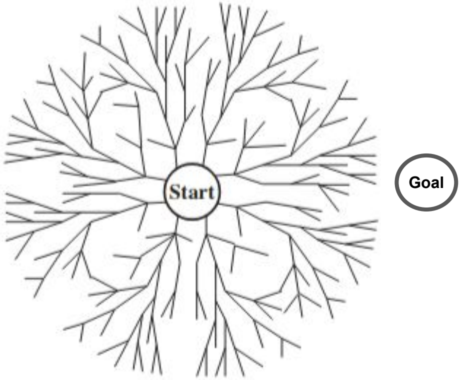
RBFS search



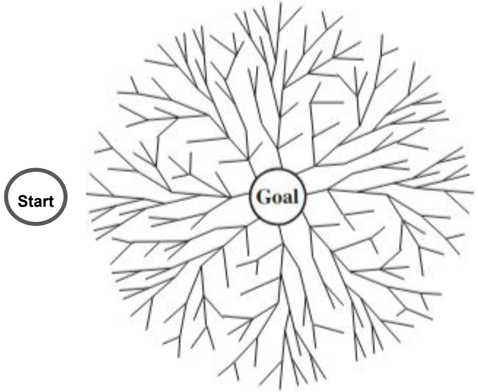
Forward Search and Backward Search

- Till now all the search algorithms started **searching from the start** state try to find the goal state among the successor states. **[Foward Search]**
- You can also **start searching from the goal state**, scan through the predecessors and try to find the start state. **[Backward Search]**
- **Branching factor**: average number of children that each node has in the search tree
- **Fan-out and Fan-in branching factor**, i.e., out and in branching factors
- **Forward search** more preferable if **Fan-out branching factor is lower**
- **Backward search** more preferable if **Fan-in branching factor is lower**

Forward Search



Backward Search

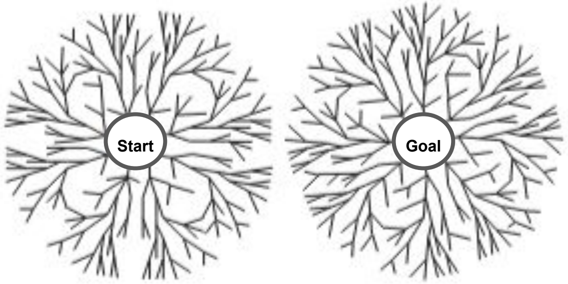


Bidirectional Search

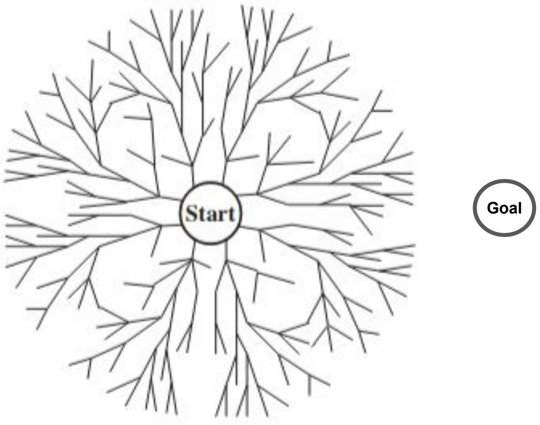
Forward and Backward Search Combined

- Idea: Run two **simultaneous** searches—one **forward** from the initial state and the other **backward** from the goal
    - hoping that the two searches meet in the middle
- Bidirectional search is implemented by **replacing the goal test with a check to see whether the frontiers/fringe of the two searches intersect.**
- Bidirectional search is preferable when:
  a) Generating predecessors is easy
  b) There are only 1 or few goal states

Bidirectional Search



Breadth First Search



Bidirectional Search

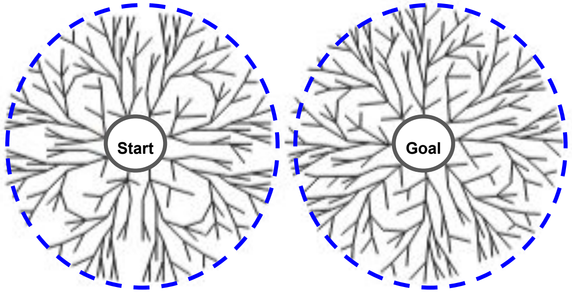
- **Motivation**: The **area of the two small circles** is **less than** the **area of one big circle** centered on the start state and reaching to the goal.

- E.g., if a problem has a solution depth of d = 6 and a branching factor of b = 10 and each direction runs BFS one node at a time, then, in the worst case, the two searches meet when they have generated all of the nodes at depth 3.

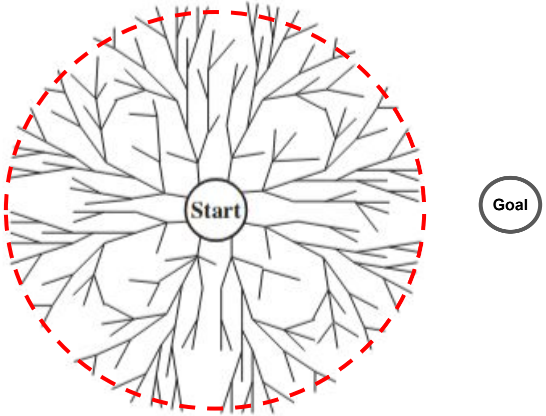
  <u>Node Generation:</u>
  Regular BFS: $1 + b + b^2 + b^3 + b^4 + b^5 + b^6$
  Bidirectional Search: $(1 + b + b^2 + b^3) + (1 + b + b^2 + b^3)$

- If there are **multiple explicitly listed goal states** then we can **construct a new dummy goal state** whose **immediate predecessors are all the actual goal states**.

Bidirectional Search



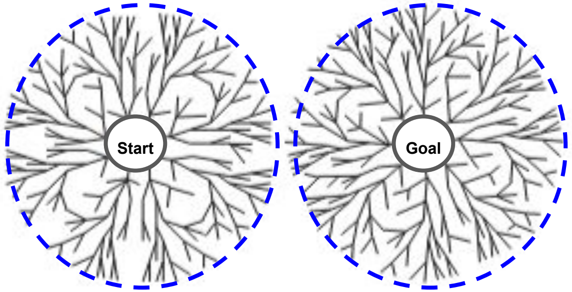
Breadth First Search



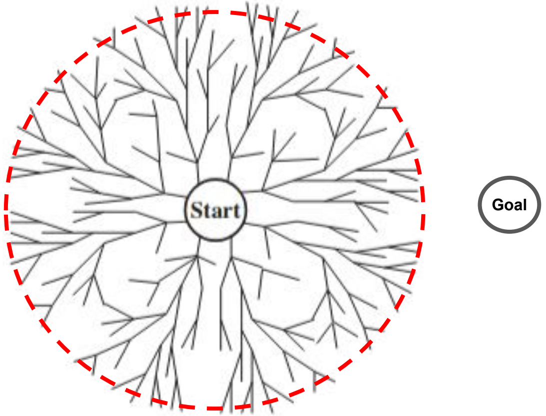
Bidirectional Search

- When you find an intersection at a node say D, follow these steps:
- Identify All Partial Plans:
    - Gather all partial plans that lead to D from the forward search and all plans that lead to D from the backward search.
- Select Shallowest/Least Cost Plans depending on BFS or UCS:
    - From the forward search, select the shallowest plan(s) that reach D.
    - From the backward search, select the shallowest plan(s) that reach D.
- Combine the Plans:
    - Construct the full path by concatenating the selected shallowest forward plan with the reversed shallowest backward plan.

Bidirectional Search



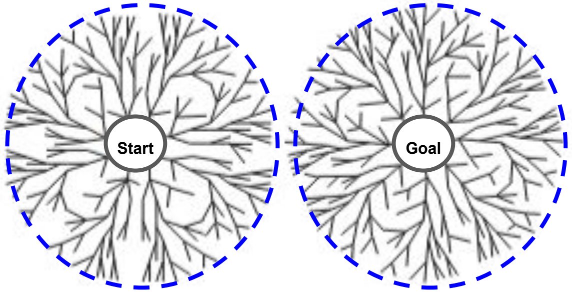
Breadth First Search



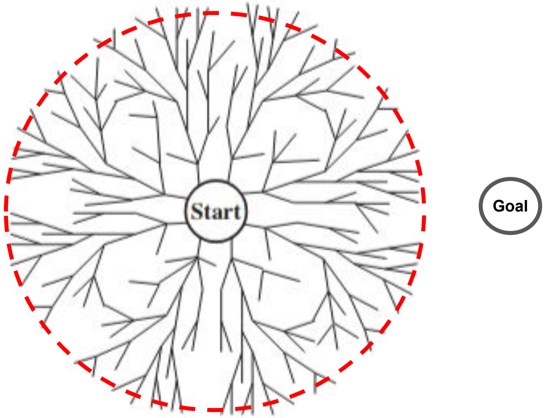
Bidirectional Search

- Is Bidirectional Search using BFS Complete?
    - Yes, since if not stopped each direction will scan the entire available search tree.
- Optimal?
    - Stepwise optimal when using BFS on uniformly weighted graph
    - Don’t stop checking for overlap before expanding all partial plans of the same smallest length.

Bidirectional Search

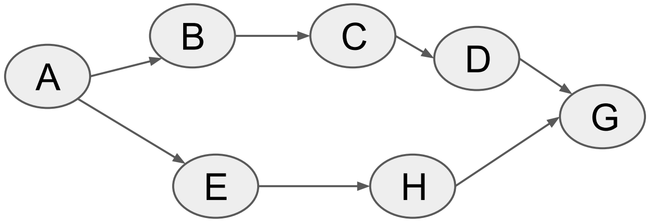


Breadth First Search

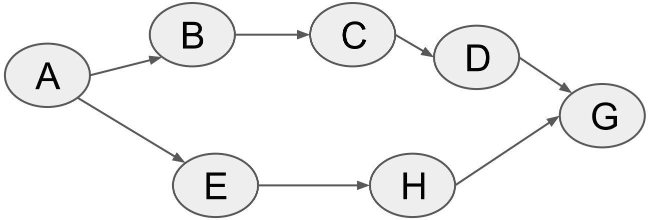


Bidirectional Search

- Is Bidirectional Search using BFS Complete?
    - Yes, since if not stopped each direction will scan the entire available search tree.
- Optimal?
    - Stepwise optimal when using BFS on uniformly weighted graph
    - Don’t stop checking for overlap before expanding all partial plans of the same smallest length.

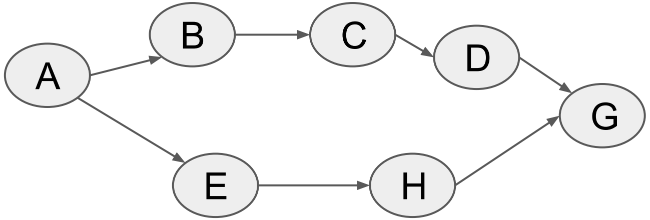


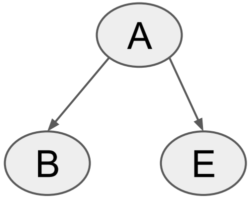
Bidirectional Search



	$\text{Fringe}_{\text{Forward}}$	$\text{PartPlan}_{\text{Forward}}$		$\text{Fringe}_{\text{Backward}}$	$\text{PartPlan}_{\text{Backward}}$
A	A	**A**	G	G	**G**

Bidirectional Search



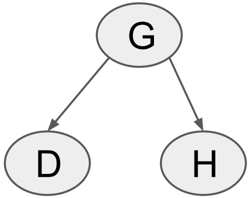


$\text{Fringe}_{\text{Forward}}$

~~A~~
A→B
A→E

$\text{PartPlan}_{\text{Forward}}$

A
A→**B**
A→**E**



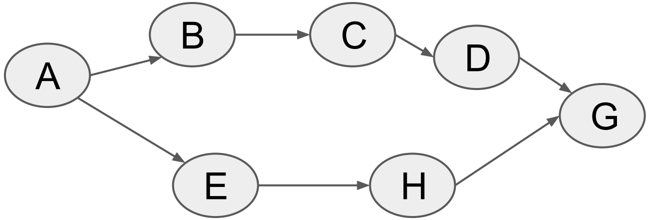
$\text{Fringe}_{\text{Backward}}$

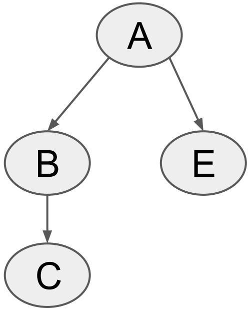
~~G~~
G→D
G→H

$\text{PartPlan}_{\text{Backward}}$

G
G→**D**
G→**H**

Bidirectional Search



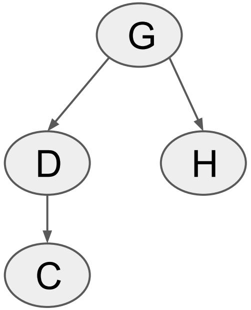


Fringe$_{\text{Forward}}$

~~A~~
~~A→B~~
A→E
A→B→C

PartPlan$_{\text{Forward}}$

A
A→**B**
A→**E**
A→B→**C**



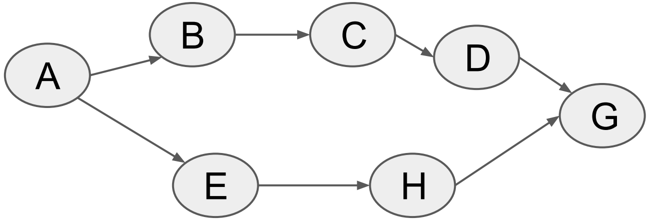
Fringe$_{\text{Backward}}$

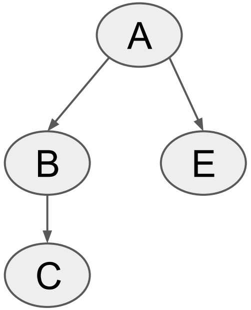
~~G~~
~~G→D~~
G→H
G→D→C

PartPlan$_{\text{Backward}}$

G
G→**D**
G→**H**
G→D→**C**

Bidirectional Search



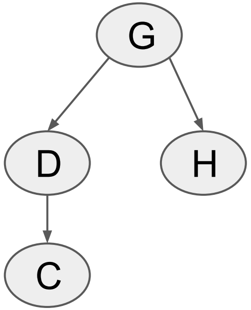


Fringe$_{Forward}$

~~A~~
~~A→B~~
~~A→E~~
A→B→C
A→E→H

PartPlan$_{Forward}$

A
A→**B**
A→**E**
A→B→**C**
A→E→**H**



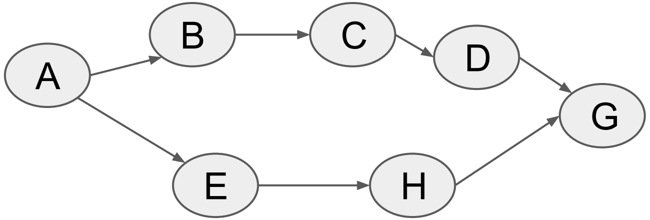
Fringe$_{Backward}$

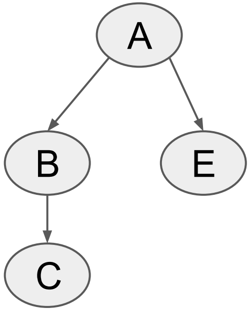
~~G~~
~~G→D~~
~~G→H~~
G→D→C
G→H→E

PartPlan$_{Backward}$

G
G→**D**
G→**H**
G→D→**C**
G→H→**E**

Bidirectional Search



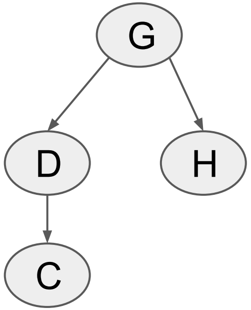


$\text{Fringe}_{\text{Forward}}$

~~A~~
~~A→B~~
~~A→E~~
A→B→C
A→E→H

$\text{PartPlan}_{\text{Forward}}$

A
A→**B**
A→**E** overlap
A→B→**C** overlap
A→E→**H** overlap



$\text{Fringe}_{\text{Backward}}$

~~G~~
~~G→D~~
~~G→H~~
G→D→C
G→H→E

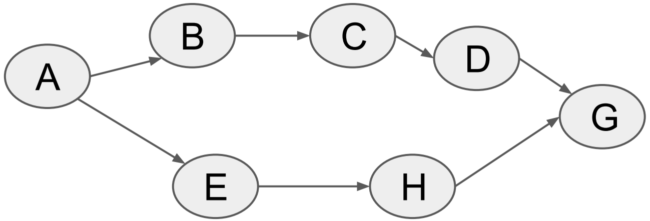
$\text{PartPlan}_{\text{Backward}}$

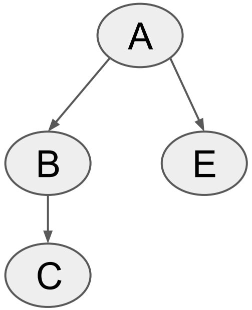
G
G→**D**
G→**H** overlap
G→D→**C** overlap
G→H→**E** overlap

FIND SHORTEST PATH THROUGH OVERLAP

A→**E**→H→G, 3 steps
A→B→**C**→D→G, 4 steps
A→**E**→H→G, 3 steps

Bidirectional Search



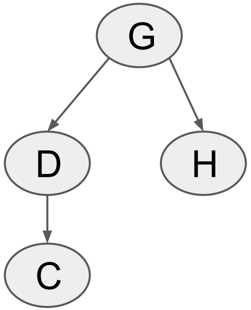


$\text{Fringe}_{\text{Forward}}$

~~A~~
~~A→B~~
~~A→E~~
A→B→C
A→E→H

$\text{PartPlan}_{\text{Forward}}$

A
A→**B**
A→**E** overlap
A→B→**C** overlap
A→E→**H** overlap



$\text{Fringe}_{\text{Backward}}$

~~G~~
~~G→D~~
~~G→H~~
G→D→C
G→H→E

$\text{PartPlan}_{\text{Backward}}$

G
G→**D**
G→**H** overlap
G→D→**C** overlap
G→H→**E** overlap

<u>FIND SHORTEST PATH THROUGH OVERLAP</u>

A→**E**→H→G, 3 steps
A→B→**C**→D→G, 4 steps
A→**E**→H→G, 3 steps