Graph Search Algorithms

CS 583 - Spring 2019

Amarda Shehu

[amarda](AT)gmu.edu
Department of Computer Science
George Mason University

1. (Uninformed and Informed) Graph Search Algorithms
    - Uninformed Search
        - Breadth-first Search (BFS)
        - Depth-first Search (DFS)
        - Depth-limited Search (DLS)
        - Iterative Deepening Search (IDS)
    - Informed Graph Search Algorithms
    - Dijkstra's Graph Search Algorithm
    - A* Search

General Search Template

- **Important insight:**
  - Any search algorithm constructs a tree, adding to it vertices of graph G in some order
  - $G = (V, E)$ —– look at it as split in two: set S on one side and $V - S$ on the other
  - search proceeds as vertices are taken from $V - S$ and added to S
  - search ends when $V - S$ is empty or goal found
  - First vertex to be taken from $V - S$ and added to S?
  - Next vertex? (... expansion ...)
  - Where to keep track of these vertices? (... fringe/frontier ...)
- **Important ideas:**
  - Fringe (frontier into $V - S$/border between S and $V - S$)
  - Expansion (neighbor generation so can add to fringe)
  - Exploration strategy (what order to grow S?)
- **Main question:**
  - which fringe/frontier nodes to explore/expand next?
  - strategy distinguishes search algorithms from one another

Search Strategies

A strategy is defined by picking the **order of node expansion**

Strategies are evaluated along the following dimensions:

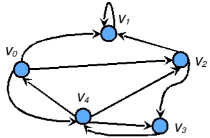
- completeness—does it always find a solution if one exists?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory
- optimality—does it always find a least-cost solution?

Time and space complexity are measured in terms of:

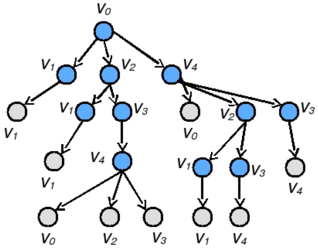
- b—maximum branching factor of the search tree
- d—depth of the least-cost solution
- m—maximum depth of the state space (may be ∞)

Characteristics of Uninformed Graph Search/Traversal:

- There is no additional information about states/vertices beyond what is provided in the problem definition.
- All that the search does is generate successors/neighbors and distinguish a **goal** state from a **non-goal state**.



The systematic search "lays out" all paths from initial vertex; it traverses the search tree of the graph.



F: search data structure (fringe)
parent array: stores “edge comes from” to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
   u ← F.extract()
   if isGoal(u) then
      return true
   for each v in outEdges(u) do
      if no parent[v] then
         F.insert(v)
         parent[v] ← u
```

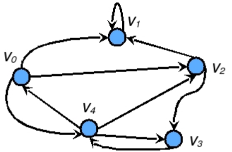
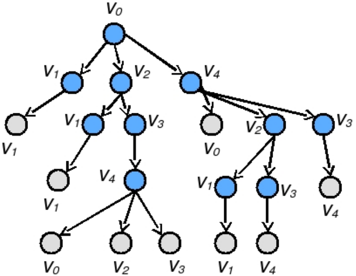


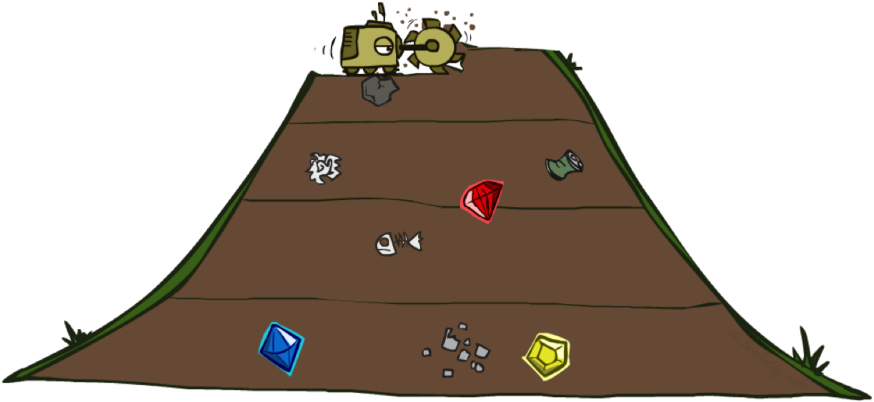
Figure: Graph



Uninformed Search Algorithms

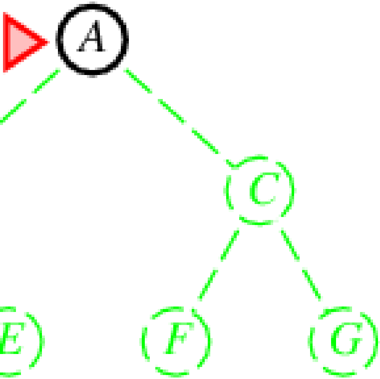
- Breadth-first Search (BFS)
- Depth-first Search (DFS)
- Depth-limited search (DLS)
- Iterative Deepening Search (IDS)

Breadth-first Search (BFS)



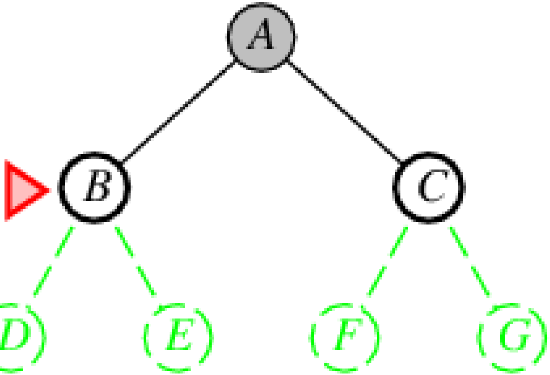
Strategy: **Expand shallowest unexpanded node**

Implementation:
fringe = first-in first-out (FIFO), i.e., unvisited successors go at end
F is a queue



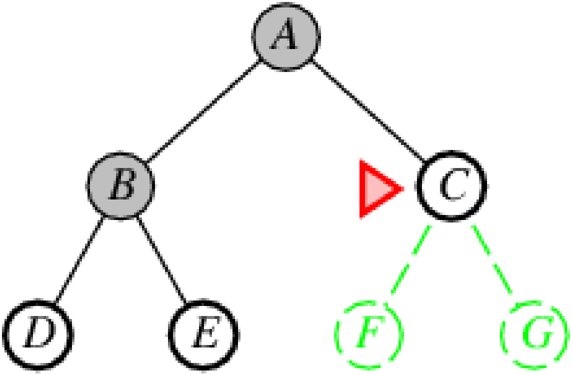
Strategy: **Expand shallowest unexpanded node**

Implementation:
fringe = first-in first-out (FIFO), i.e., unvisited successors go at end
F is a queue



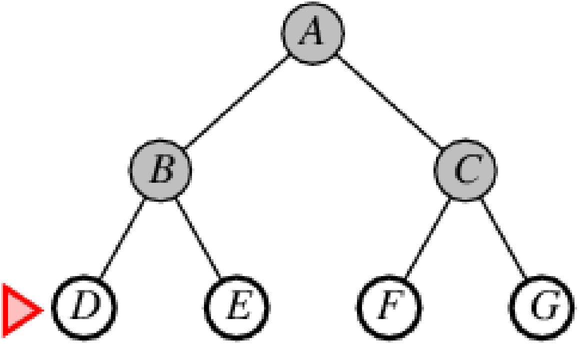
Strategy: Expand shallowest unexpanded node

Implementation:
fringe = first-in first-out (FIFO), i.e., unvisited successors go at end
F is a queue



Strategy: Expand shallowest unexpanded node

Implementation:
fringe = first-in first-out (FIFO), i.e., unvisited successors go at end
F is a queue



Breadth-first Search (BFS)

F: search data structure (fringe)

F is a queue (FIFO) in BFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
    u ← F.extract()
    if isGoal(u) then
        return true
    for each v in outEdges(u) do
        if no parent[v] then
            F.insert(v)
            parent[v] ← u
```

Running Time?

F: search data structure (fringe)

F is a queue (FIFO) in BFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
    u ← F.extract()
    if isGoal(u) then
        return true
    for each v in outEdges(u) do
        if no parent[v] then
            F.insert(v)
            parent[v] ← u
```

Running Time?

Let V and E be vertices and edges in search tree

Breadth-first Search (BFS)

F: search data structure (fringe)

F is a queue (FIFO) in BFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
    u ← F.extract()
    if isGoal(u) then
        return true
    for each v in outEdges(u) do
        if no parent[v] then
            F.insert(v)
            parent[v] ← u
```

Running Time?

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$

Breadth-first Search (BFS)

F: search data structure (fringe)

F is a queue (FIFO) in BFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
    u ← F.extract()
    if isGoal(u) then
        return true
    for each v in outEdges(u) do
        if no parent[v] then
            F.insert(v)
            parent[v] ← u
```

Running Time?

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$

What about in terms of b and m?

Breadth-first Search (BFS)

F: search data structure (fringe)

F is a queue (FIFO) in BFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
    u ← F.extract()
    if isGoal(u) then
        return true
    for each v in outEdges(u) do
        if no parent[v] then
            F.insert(v)
            parent[v] ← u
```

Running Time?

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$

What about in terms of b and m?

Properties of Breadth-first Search (BFS)

Complete??

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Time??

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space??

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal??

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost $= 1$ per step); not optimal in general

Space

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec

so 24hrs = 8640GB.

Properties of Breadth-first Search (BFS)

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec

so 24hrs = 8640GB.

BFS Summary

Basic Behavior:

- Expands all nodes at depth d before those at depth $d + 1$
- The sequence is root, then children, then grandchildren in the search tree.

Problems:

- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)

BFS Summary

Basic Behavior:

- Expands all nodes at depth d before those at depth $d + 1$
- The sequence is root, then children, then grandchildren in the search tree.

Problems:

- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)
- A graph with no weights can be considered to have edges of weight 1. In this case, BFS is optimal.

BFS Summary

Basic Behavior:

- Expands all nodes at depth d before those at depth $d + 1$
- The sequence is root, then children, then grandchildren in the search tree.

Problems:

- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)
- A graph with no weights can be considered to have edges of weight 1. In this case, BFS is optimal.
- BFS will find shallowest goal after expanding all shallower nodes (if branching factor is finite). Hence, BFS is complete.

BFS Summary

Basic Behavior:

- Expands all nodes at depth d before those at depth $d+1$
- The sequence is root, then children, then grandchildren in the search tree.

Problems:

- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)
- A graph with no weights can be considered to have edges of weight 1. In this case, BFS is optimal.
- BFS will find shallowest goal after expanding all shallower nodes (if branching factor is finite). Hence, BFS is complete.
- BFS is not very popular because time and space complexity are exponential: $O(b^{d+1})$ and $O(b^{d+1})$, respectively.
- Memory requirements of BFS are a bigger problem.

BFS Summary

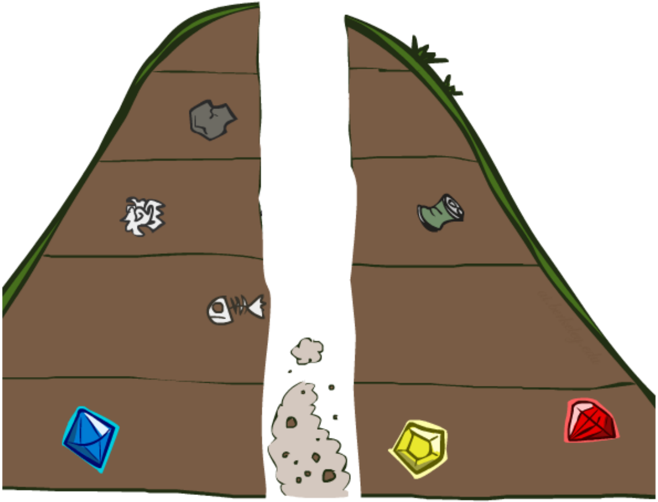
Basic Behavior:

- Expands all nodes at depth d before those at depth $d+1$
- The sequence is root, then children, then grandchildren in the search tree.

Problems:

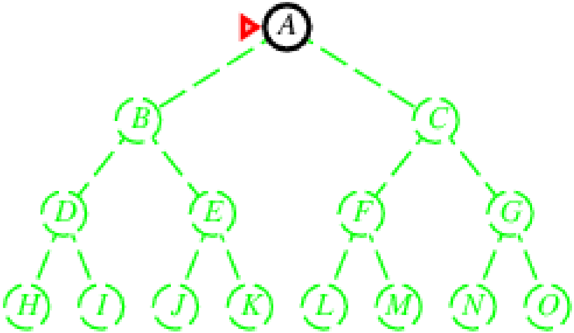
- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal (uniform cost search a generalization)
- A graph with no weights can be considered to have edges of weight 1. In this case, BFS is optimal.
- BFS will find shallowest goal after expanding all shallower nodes (if branching factor is finite). Hence, BFS is complete.
- BFS is not very popular because time and space complexity are exponential: $O(b^{d+1})$ and $O(b^{d+1})$, respectively.
- Memory requirements of BFS are a bigger problem.

Depth-first Search (DFS)



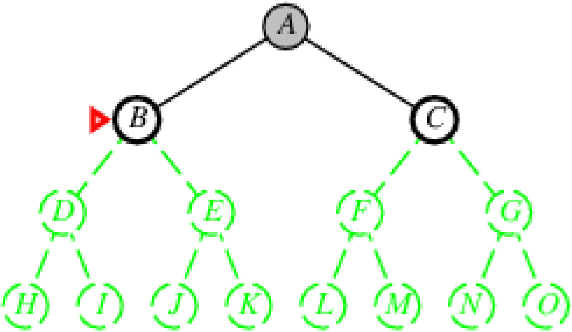
Strategy: Expand deepest unexpanded node

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



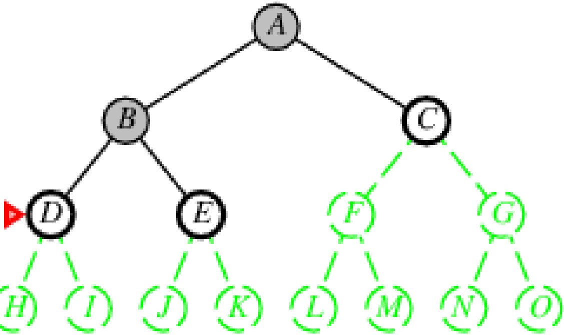
Strategy: Expand deepest unexpanded node

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



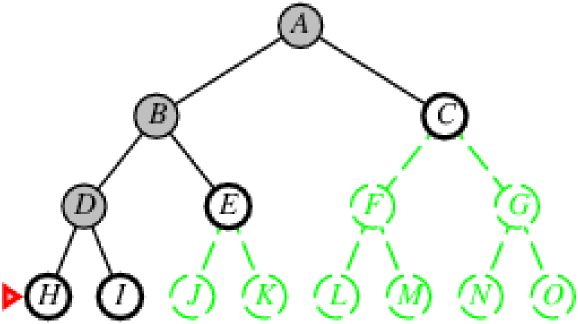
Strategy: **Expand deepest unexpanded node**

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



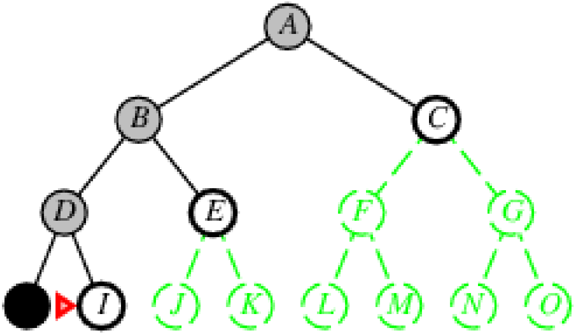
Strategy: Expand deepest unexpanded node

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



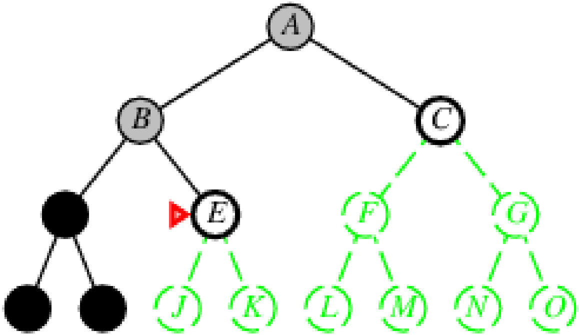
Strategy: Expand deepest unexpanded node

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



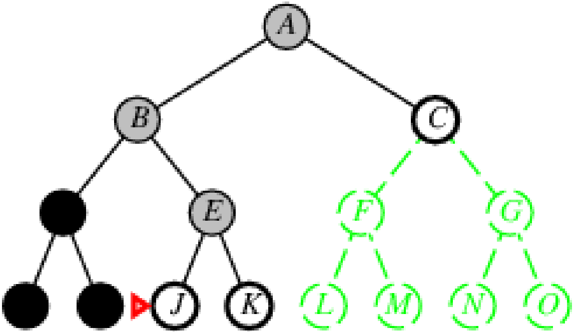
Strategy: **Expand deepest unexpanded node**

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



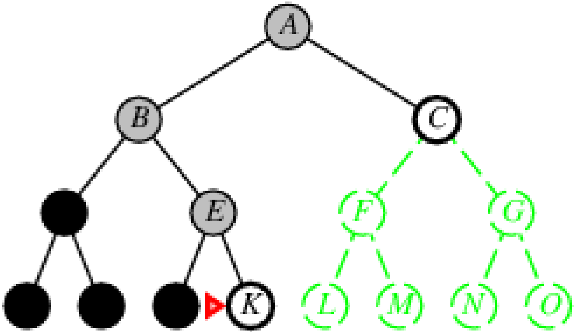
Strategy: **Expand deepest unexpanded node**

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



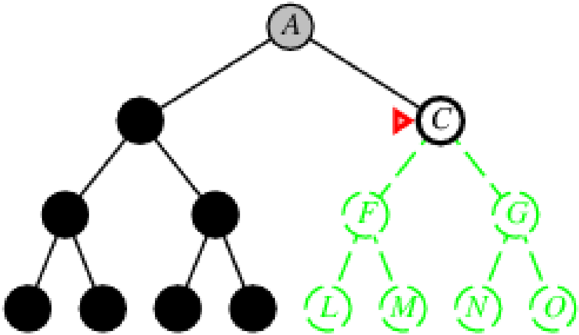
Strategy: Expand deepest unexpanded node

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



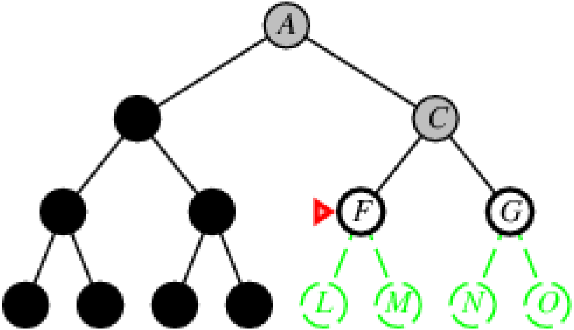
Strategy: Expand deepest unexpanded node

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



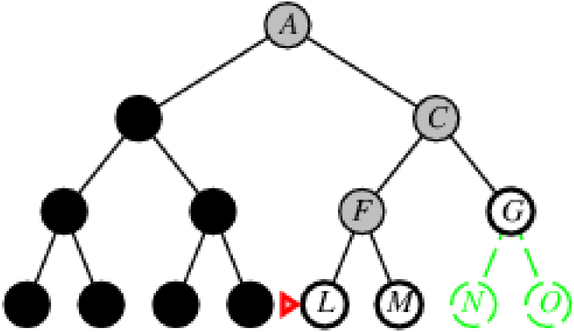
Strategy: Expand deepest unexpanded node

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



Strategy: Expand deepest unexpanded node

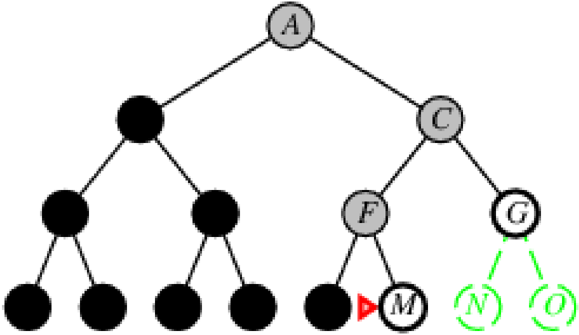
Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



Depth-first Search (DFS)

Strategy: **Expand deepest unexpanded node**

Implementation:
fringe = last-in first-out (LIFO), i.e., unvisited successors at front
F is a stack



Depth-first Search (DFS)

F: search data structure (fringe)

F is a stack (LIFO) in DFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
    u ← F.extract()
    if isGoal(u) then
        return true
    for each v in outEdges(u) do
        if no parent[v] then
            F.insert(v)
            parent[v] ← u
```

Running Time?

Depth-first Search (DFS)

F: search data structure (fringe)

F is a stack (LIFO) in DFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
   u ← F.extract()
   if isGoal(u) then
      return true
   for each v in outEdges(u) do
      if no parent[v] then
         F.insert(v)
         parent[v] ← u
```

Running Time?

Let V and E be vertices and edges in search tree

Depth-first Search (DFS)

F: search data structure (fringe)

F is a stack (LIFO) in DFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
    u ← F.extract()
    if isGoal(u) then
        return true
    for each v in outEdges(u) do
        if no parent[v] then
            F.insert(v)
            parent[v] ← u
```

Running Time?

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$

Depth-first Search (DFS)

F: search data structure (fringe)

F is a stack (LIFO) in DFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
    u ← F.extract()
    if isGoal(u) then
        return true
    for each v in outEdges(u) do
        if no parent[v] then
            F.insert(v)
            parent[v] ← u
```

Running Time?

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$

What about in terms of b and m?

Depth-first Search (DFS)

F: search data structure (fringe)

F is a stack (LIFO) in DFS!

parent array: stores "edge comes from" to record visited states

```
F.insert(v)
parent[v] ← true
while not F.isEmpty do
    u ← F.extract()
    if isGoal(u) then
        return true
    for each v in outEdges(u) do
        if no parent[v] then
            F.insert(v)
            parent[v] ← u
```

Running Time?

Let V and E be vertices and edges in search tree

$O(|V| + |E|)$

What about in terms of b and m?

Properties of Depth-first Search (DFS)

Complete??

Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time??

Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than BFS

Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than BFS

Space??

Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than BFS

Space?? $O(bm)$, i.e., linear space!

Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than BFS

Space?? $O(bm)$, i.e., linear space!

Optimal??

Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than BFS

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than BFS

Space?? $O(bm)$, i.e., linear space!

Optimal?? No Why?

Properties of Depth-first Search (DFS)

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than BFS

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Why?

DFS Summary

Basic Behavior:

- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:

- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal

DFS Summary

Basic Behavior:

- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:

- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.

DFS Summary

Basic Behavior:

- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:

- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.
- Hence, DFS is not complete

DFS Summary

Basic Behavior:

- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:

- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.
- Hence, DFS is not complete
- Let b be the maximum number of successors of any node (known as branching factor), d be depth of shallowest goal, and m be maximum length of any path in the search tree

DFS Summary

Basic Behavior:

- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:

- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.
- Hence, DFS is not complete
- Let b be the maximum number of successors of any node (known as branching factor), d be depth of shallowest goal, and m be maximum length of any path in the search tree
- Time complexity is $O(b^m)$ and space complexity is $O(b \cdot m)$

DFS Summary

Basic Behavior:

- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:

- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.
- Hence, DFS is not complete
- Let b be the maximum number of successors of any node (known as branching factor), d be depth of shallowest goal, and m be maximum length of any path in the search tree
- Time complexity is $O(b^m)$ and space complexity is $O(b \cdot m)$

- When will BFS outperform DFS?
- When will DFS outperform BFS?

Another Advantage of DFS

RecursiveDFS(v)

if v is unmarked **then**
  mark v
  for each edge v, u **do**
    RecursiveDFS(u)

Color arrays can be kept to indicate that a vertex is undiscovered, the first time it is discovered, when its neighbors are in the process of being considered, and when all its neighbors have been considered.

DFS can be used to timestamp vertices with when they are discovered and when they are finished. These start and finish times are useful in various applications of DFS regarding constraint satisfaction.

Depth-limited Search (DLS)

- Problem with DFS is presence of infinite paths
- DLS limits the depth of a path in search tree of DFS
- Modifies *DFS* by using a predetermined depth limit $\mathrm{d_l}$
- DLS is incomplete if the shallowest goal is beyond the depth limit d_l
- DLS is not optimal if $d < d_l$
- Time complexity is $O(b^{d_l})$ and space complexity is $O(b \cdot d_l)$

= DFS with depth limit d_l [i.e., nodes at depth d_l are not expanded]

Recursive implementation:

```
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 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
```

Iterative Deepening Search (IDS)

- Finds the best depth limit by incrementing d_l until goal is found at $d_l = d$
- Can be viewed as running DLS with consecutive values of d_l
- IDS combines the benefits of both DFS and BFS
- Like DFS, its space complexity is $O(b \cdot d)$
- Like BFS, it is complete when the branching factor is finite, and it is optimal if the path cost is a non-decreasing function of the depth of the goal node
- Its time complexity is $O(b^d)$
- IDS is the preferred uninformed search when the state space is large, and the depth of the solution is not known

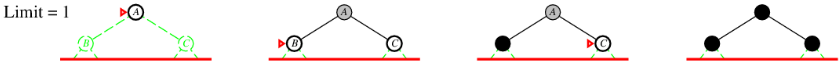
Iterative Deepening Search (IDS)

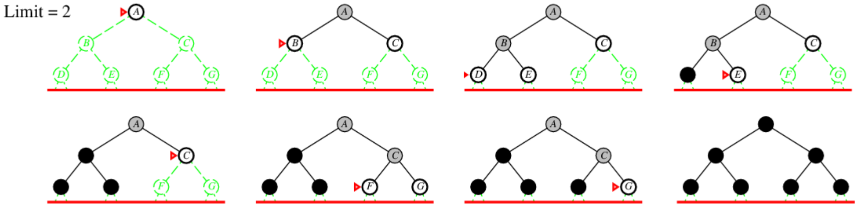
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution
  inputs: *problem*, a problem

  for *depth* $\leftarrow$ 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*
  end

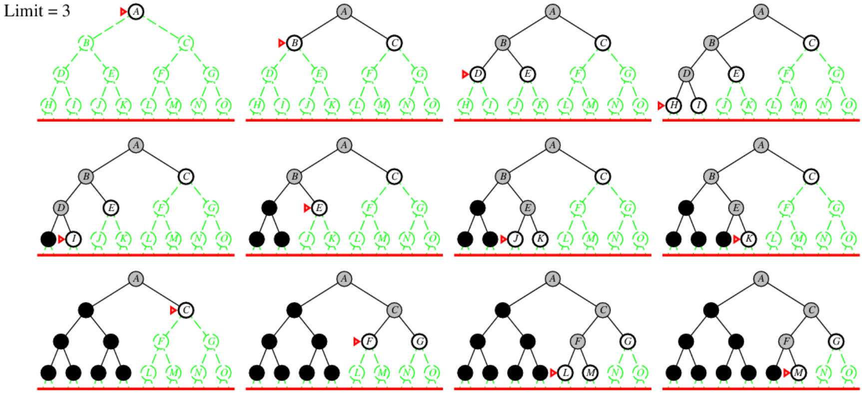
Iterative Deepening Search (IDS) @ $d_l = 0$

Iterative Deepening Search (IDS) @ $d_l = 1$




Limit = 2
A
B
C
D
E
F
G

Iterative Deepening Search (IDS) @ $d_l = 3$



Summary of Uninformed Search Algorithms

Criterion	Breadth-First	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	No	Yes, if $d_l \geq d$	Yes
Time	b^{d+1}	b^m	b^{d_l}	b^d
Space	b^{d+1}	bm	bd_l	bd
Optimal?	Yes*	No	No	Yes*

Uninformed Search Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- IDS uses only linear space and not much more time than other uninformed algorithms
- Graph search can be exponentially more efficient than tree search
- What about least-cost paths with non-uniform state-state costs?
    - That is next

Informed Graph Search Algorithms

- Make use of costs/weights in state-space graph
- **Informed**/Greedy graph search algorithms:
  - Dijkstra's Search [Edsger Dijkstra 1959]
  - Uniform-cost Search (a variant of Dijkstra's)
  - Best-First Search [Judea Pearl 1984]
  - A* Search [Petter Hart, Nils Nilsson, Bertram Raphael 1968]
  - B* Search [Hans Berliner 1979]
  - D* Search [Stenz 1994]
  - More variants of the above
- Other Algorithms:
  - What to do if weights are negative
  - Dynamic Programming rather than greedy paradigm
  - Bellman-Ford's, Floyd-Warshall's

Most popular: Dijkstra and A*

Differences from uninformed search algorithms:

- work with weighted graphs
- process nodes in order of attachment cost
- employ priority queue (min-heap) for this purpose instead of stack or queue
- Dijkstra: overkill, finds least-cost path from a given start node to all nodes in graph
- A*: works only with given start and goal pair
- Dijkstra: attachment cost of a node is current least cost from given start to that node
- A*: adds to this the estimated distance to goal node, where esimation uses an optimistic heuristic

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time

  Can terminate earlier? When? How does it relate to goal?

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time
  Can terminate earlier? When? How does it relate to goal?
- v extracted from F @ some iteration is one with lowest cost among all those in F

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time
  Can terminate earlier? When? How does it relate to goal?
- v extracted from F @ some iteration is one with lowest cost among all those in F
  ... so, vertices extracted from F in order of their costs

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time
  Can terminate earlier? When? How does it relate to goal?
- v extracted from F @ some iteration is one with lowest cost among all those in F
  ... so, vertices extracted from F in order of their costs
- When v extracted from F:

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time
  Can terminate earlier? When? How does it relate to goal?
- v extracted from F @ some iteration is one with lowest cost among all those in F
  ... so, vertices extracted from F in order of their costs
- When v extracted from F:
  v has been "removed" from $V - S$ and "added" to S

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time
  Can terminate earlier? When? How does it relate to goal?
- v extracted from F @ some iteration is one with lowest cost among all those in F
  ... so, vertices extracted from F in order of their costs
- When v extracted from F:
  v has been "removed" from $V - S$ and "added" to S
  get to reach/see v's neighbors and possibly update their costs

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time
  Can terminate earlier? When? How does it relate to goal?
- v extracted from F @ some iteration is one with lowest cost among all those in F
  ... so, vertices extracted from F in order of their costs
- When v extracted from F:
  v has been "removed" from $V - S$ and "added" to S
  get to reach/see v's neighbors and possibly update their costs

The rest are details, such as:

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time
  Can terminate earlier? When? How does it relate to goal?
- v extracted from F @ some iteration is one with lowest cost among all those in F
  ... so, vertices extracted from F in order of their costs
- When v extracted from F:
  v has been "removed" from $V - S$ and "added" to S
  get to reach/see v's neighbors and possibly update their costs

The rest are details, such as:

- What should $d[v]$ be? There are options...
  - backward cost (cost of $s \rightsquigarrow v$)
  - forward cost (estimate of cost of $v \rightsquigarrow g$)
  - back+for ward cost (estimate of $s \rightsquigarrow g$ through v)
- Which do I choose? This is how to you end up with different search algorithms

Essence of All Informed Search Algorithms

All you need to remember about informed search algorithms

- Associate a(n attachment) cost $d[v]$ with each vertex v
- F becomes a priority queue: F keeps frontier vertices, prioritized by $d[v]$
- Until F is empty, one vertex extracted from F at a time
  Can terminate earlier? When? How does it relate to goal?
- v extracted from F @ some iteration is one with lowest cost among all those in F
  ... so, vertices extracted from F in order of their costs
- When v extracted from F:
  v has been "removed" from $V - S$ and "added" to S
  get to reach/see v's neighbors and possibly update their costs

The rest are details, such as:

- What should $d[v]$ be? There are options...
  - backward cost (cost of $s \rightsquigarrow v$)
  - forward cost (estimate of cost of $v \rightsquigarrow g$)
  - back+for ward cost (estimate of $s \rightsquigarrow g$ through v)
- Which do I choose? This is how to you end up with different search algorithms

Dijkstra's Algorithm in Pseudocode

- Fringe: F is a priority queue/min-heap
- arrays: d stores attachment (backward) costs, $\pi[v]$ stores parents
- S not really needed, only for clarity below

Dijkstra(G, s, w)

$F \leftarrow s$, $S \leftarrow \{\}$
d[v] $\leftarrow \infty$ for all $v \in V$
$d[s] \leftarrow 0$
while $F \neq \{\}$ **do**
    $u \leftarrow$ Extract-Min(F)
    $S \leftarrow S \cup \{u\}$
    for each $v \in \mathrm{Adj(u)}$ **do**
        $F \leftarrow v$
        Relax(u, v, w)

Relax(u, v, w)

if $d[v] > d[u] + w(u, v)$ **then**
    $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[\mathrm{v}] \leftarrow \mathrm{u}$

- The process of relaxing tests whether one can improve the shortest-path estimate $d[v]$ by going through the vertex u in the shortest path from s to v
- If $d[u] + w(u, v) < d[v]$, then u replaces the predecessor of v
- Where would you put an earlier termination to stop when $s \rightsquigarrow g$ found?

- Fringe: F is a priority queue/min-heap
- arrays: d stores attachment (backward) costs, $\pi[v]$ stores parents
- S not really needed, only for clarity below

Dijkstra(G, s, w)

1: $F \leftarrow s$, $S \leftarrow \{\}$
2: d[v] $\leftarrow \infty$ for all $v \in V$
3: $d[s] \leftarrow 0$
4: **while** $F \neq \{\}$ **do**
5: $\quad u \leftarrow$ Extract-Min(F)
6: $\quad S \leftarrow S \cup \{u\}$
7: $\quad$ **for each** $v \in \mathrm{Adj(u)}$ **do**
8: $\quad\quad F \leftarrow v$
9: $\quad\quad$ Relax(u, v, w)

Relax(u, v, w)

1: **if** $d[v] > d[u] + w(u, v)$ **then**
2: $\quad d[v] \leftarrow d[u] + w(u, v)$
3: $\quad \pi[\mathrm{v}] \leftarrow \mathrm{u}$

in another implementation, F is initialized with all V, and line 8 is removed.

- The process of relaxing tests whether one can improve the shortest-path estimate $d[v]$ by going through the vertex u in the shortest path from s to v
- If $d[u] + w(u, v) < d[v]$, then u replaces the predecessor of v
- Where would you put an earlier termination to stop when $s \rightsquigarrow g$ found?

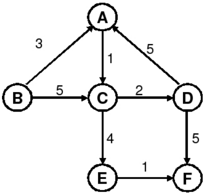


Figure: Graph $G = (V, E)$

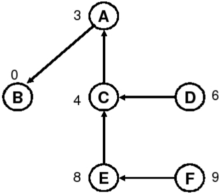


Figure: Shortest paths from B

	Initial		Pass1		Pass2		Pass3		Pass4		Pass5		Pass6	
Vertex	d	π	d	π	d	π	d	π	d	π	d	π	d	π
A	∞		3	B	3	B	3	B	3	B	3	B	3	B
B	0	—	0	—	0	—	0	—	0	—	0	—	0	—
C	∞		5	B	4	A	4	A	4	A	4	A	4	A
D	∞		∞		∞		6	C	6	C	6	C	6	C
E	∞		∞		∞		8	C	8	C	8	C	8	C
F	∞		∞		∞		∞		11	D	9	E	9	E

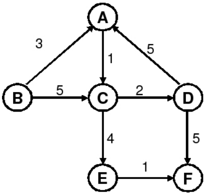


Figure: Graph $G = (V, E)$

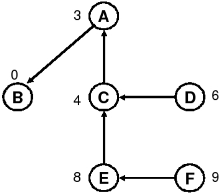
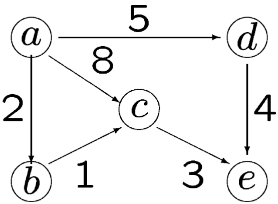


Figure: Shortest paths from B

	Initial		Pass1		Pass2		Pass3		Pass4		Pass5		Pass6	
Vertex	d	π	d	π	d	π	d	π	d	π	d	π	d	π
A	∞		3	B	3	B	3	B	3	B	3	B	3	B
B	0	–	0	–	0	–	0	–	0	–	0	–	0	–
C	∞		5	B	4	A	4	A	4	A	4	A	4	A
D	∞		∞		∞		6	C	6	C	6	C	6	C
E	∞		∞		∞		8	C	8	C	8	C	8	C
F	∞		∞		∞		∞		11	D	9	E	9	E

If not earlier goal termination criterion, Dijkstra's search tree is spanning tree of shortest paths from s to any vertex in the graph.

Take-home Exercise



	Initial		Pass1		Pass2		Pass3		Pass4		Pass5	
Vertex	*d*	π	*d*	π	*d*	π	*d*	π	*d*	π	*d*	π
a	0	—										
b	∞											
c	∞											
d	∞											
e	∞											

Correctness of Dijkstra's Search Algorithm

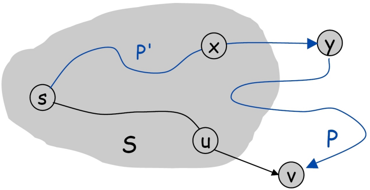
Dijkstra extracts vertices from fringe (adds to S) in order of their backward costs

Claim: When a vertex v is extracted from fringe F (thus "added" to S), the shortest path from s to v has been found. $\leftarrow$ invariant

Proof: by induction on $|S|$ (Base case $|S| = 1$ is trivial).
Assume invariant holds for $|S| = k \geq 1$.

- Let v be vertex about to be extracted from fringe (added to S), so has lowest backward cost
- Last time $d[v]$ updated when parent u extracted from fringe
- When $d[v]$ is lowest in the fringe, should we extract v or wait?
- Could $d[v]$ get lower later through some other vertex y in fringe?

$w(P) \geq w(P') + w(x, y)$	nonnegative weights
$\geq d[x] + w(x, y)$	inductive hypothesis
$\geq d[y]$	definition of $d[y]$
$\geq d[v]$	Dijkstra chose v over y



Running Time Analysis of Dijkstra's Algorithm

- Updating the heap takes at most $O(lg(|V|))$ time
- The number of updates equals the total number of edges
- So, the total running time is $O(|E| \cdot lg(|V|))$
- Running time can be improved depending on the actual implementation of the priority queue

$$\text{Time} = \theta(V) \cdot T(\mathrm{Extract} - \mathrm{Min}) + \theta(\mathrm{E}) \cdot \mathrm{T}(\mathrm{Decrease} - \mathrm{Key})$$

F	T(Extr.-Min)	T(Decr.-Key)	Total
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(1)$	$O(lg\|V\|)$	$O(\|E\| \cdot lg\|V\|)$
Fib. heap	$O(lg\|V\|)$	$O(1)$	$O(\|E\| + \|V\| \cdot lg\|V\|)$

How does this compare with BFS?
How does BFS get away from a $lg(|V|)$ factor?

A* Search

Idea: avoid expanding paths that are already expensive

Evaluation function $f(v) = g(v) + h(v)$:
Combines Dijkstra's/uniform cost with greedy best-first search
$g(v)$ = (actual) cost to reach v from s
$h(v)$ = estimated lowest cost from v to goal
$f(v)$ = estimated lowest cost from s through v to goal

Same implementation as before, but prioritize vertices in min-heap by $f[v]$

A* is both complete and optimal provided h satisfies certain conditions:
- for searching in a tree: admissible/optimistic
- for searching in a graph: consistent (which implies admissibility)

Admissible Heuristic

What do we want from $f[v]$?
  not to overestimate cost of path from source to goal that goes through v

Since $g[v]$ is actual cost from s to v, this "do not overestimate" criterion is for the forward cost heuristic, $h[v]$

A* search uses an admissible/optimistic heuristic
i.e., $h(v) \leq h^*(v)$ where $h^*(v)$ is the **true** cost from v
(Also require $h(v) \geq 0$, so $h(G) = 0$ for any goal G)

Example of an admissible heuristic: crow-fly distance never overestimates the actual road distance

A stronger, consistent heuristic: estimated cost of reaching goal from a vertex n is not greater than cost to go from n to its successors and then the cost from them to the goal

Admissible Heuristic

What do we want from $f[v]$?

not to overestimate cost of path from source to goal that goes through v

Since $g[v]$ is actual cost from s to v, this "do not overestimate" criterion is for the forward cost heuristic, $h[v]$

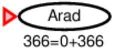
A* search uses an admissible/optimistic heuristic
i.e., $h(v) \leq h^*(v)$ where $h^*(v)$ is the **true** cost from v
(Also require $h(v) \geq 0$, so $h(G) = 0$ for any goal G)

Example of an admissible heuristic: crow-fly distance never overestimates the actual road distance

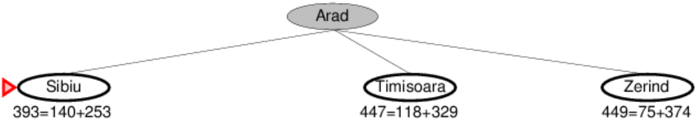
A stronger, consistent heuristic: estimated cost of reaching goal from a vertex n is not greater than cost to go from n to its successors and then the cost from them to the goal

Let's see A* with this heuristic in action

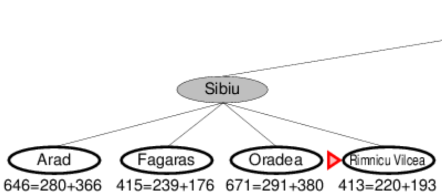
A* Search in Action



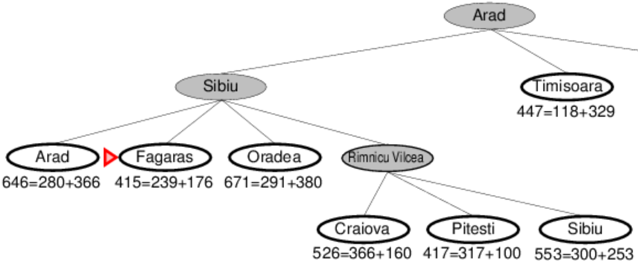
A* Search in Action



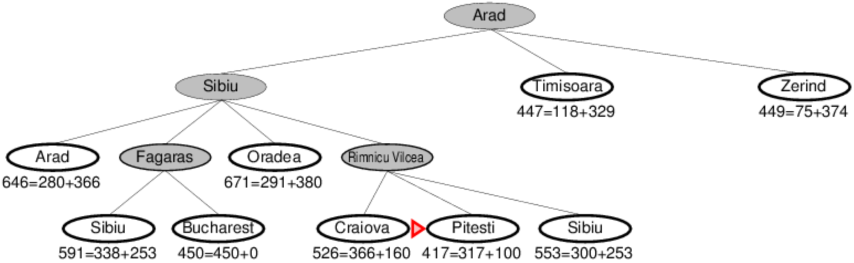
A* Search in Action



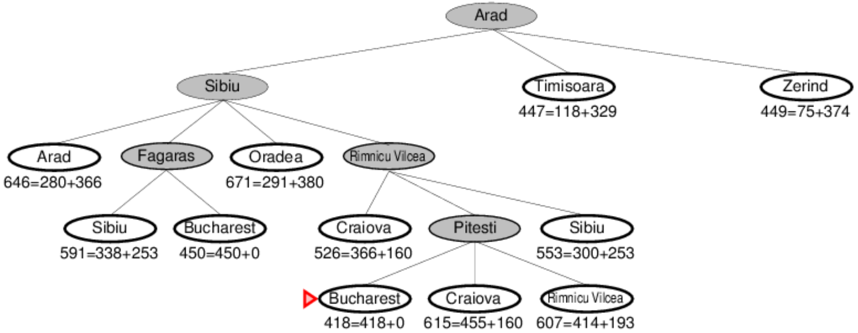
A* Search in Action



A* Search in Action



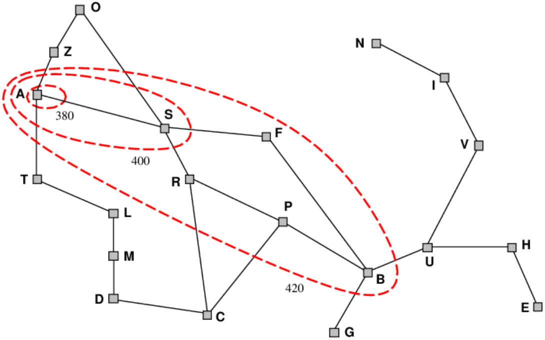
A* Search in Action



Optimality of A*

Skipping some details, but essentially if heuristic is consistent: A* expands nodes in order of increasing f value*

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



So, why does this guarantee optimality?
First time we see goal will be the time it has lowest f = g (h is 0)
Other occurrences have no lower f (f non-decreasing)

Summary of A* Search

Complete??

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time??

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g)-h(s)}{\delta(s,g)}$]

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g)-h(s)}{\delta(s,g)}$]

Space??

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g) - h(s)}{\delta(s,g)}$]

Space?? Keeps all generated nodes in memory (worse drawback than time)

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g)-h(s)}{\delta(s,g)}$]

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal??

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g)-h(s)}{\delta(s,g)}$]

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g) - h(s)}{\delta(s,g)}$]

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Optimally efficient for any given consistent heuristic:

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g)-h(s)}{\delta(s,g)}$]

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Optimally efficient for any given consistent heuristic:
A* expands all nodes with $f(v) < \delta(s,g)$

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g)-h(s)}{\delta(s,g)}$]

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Optimally efficient for any given consistent heuristic:
A* expands all nodes with $f(v) < \delta(s,g)$
A* expands some nodes with $f(v) = \delta(s,g)$

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g) - h(s)}{\delta(s,g)}$]

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Optimally efficient for any given consistent heuristic:
A* expands all nodes with $f(v) < \delta(s,g)$
A* expands some nodes with $f(v) = \delta(s,g)$
A* expands no nodes with $f(v) > \delta(s,g)$

Summary of A* Search

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [path length $\times \frac{\delta(s,g)-h(s)}{\delta(s,g)}$]

Space?? Keeps all generated nodes in memory (worse drawback than time)

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Optimally efficient for any given consistent heuristic:
A* expands all nodes with $f(v) < \delta(s,g)$
A* expands some nodes with $f(v) = \delta(s,g)$
A* expands no nodes with $f(v) > \delta(s,g)$

End of Graph Search Algorithms

CS583 additionally considers scenarios where greedy substructure does not lead to optimality

For instance, how can one modify Dijkstra and the other algorithms to deal with negative weights?

How does one efficiently find all pairwise shortest/least-cost paths?

Dynamic Programming is the right alternative in these scenarios

More graph exploration and search algorithms considered in CS583