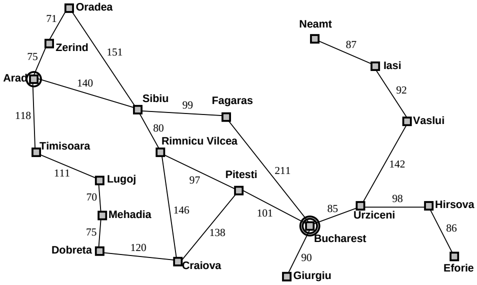
Problem solving by search
Finding the optimal sequence of states/decisions/actions

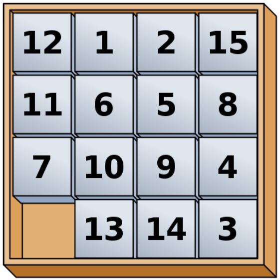
Tomáš Svoboda, Petr Pošík

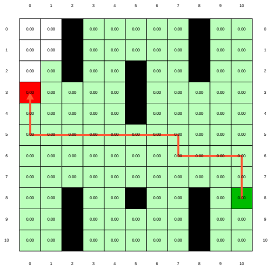
Vision for Robots and Autonomous Systems, Center for Machine Perception
Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University in Prague

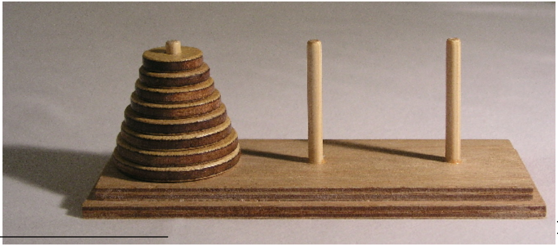
February 21, 2024

Problems to solve



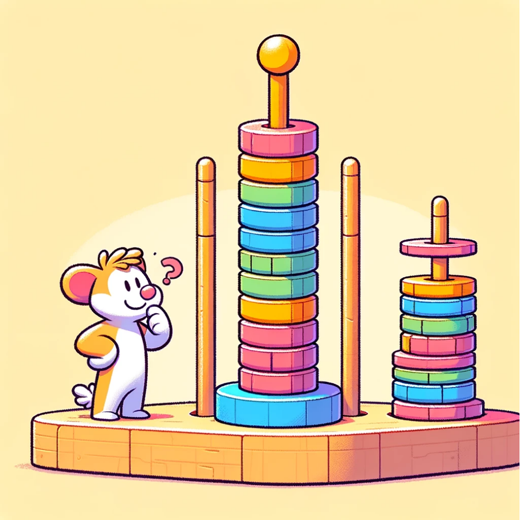





[1]

[1] CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=228623

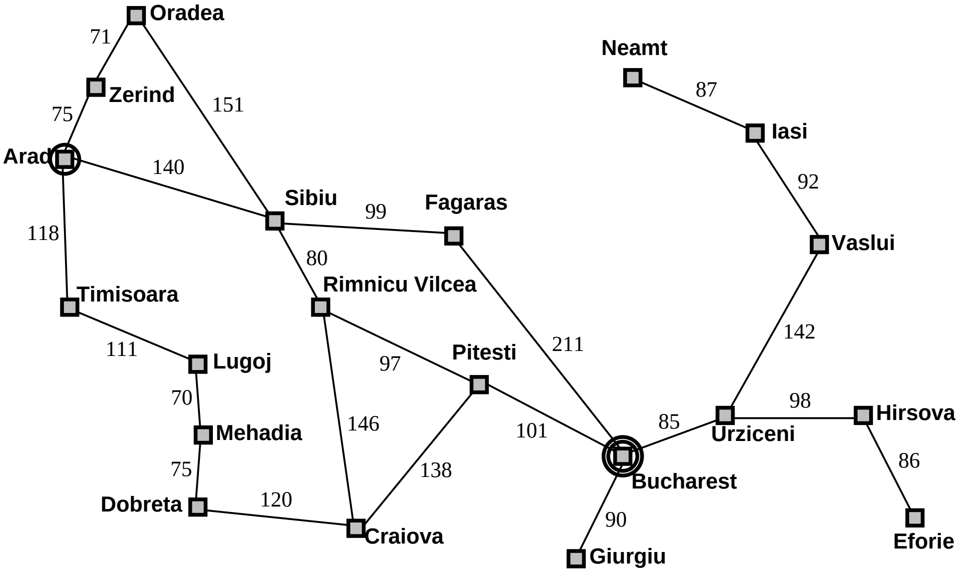
Understanding the problem is the key, DALL-E.



Outline

- Search problem. *What do you want to solve?*
- State space graphs. *How do you formalize/represent the problem? Problem abstraction.*
- Search trees. *Visualization of the algorithm run.*
- Strategies: which tree branches to choose?
- Strategy/Algorithm properties. *Memory, time, ...*
- Programming infrastructure.

Example: Traveling in Romania



Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

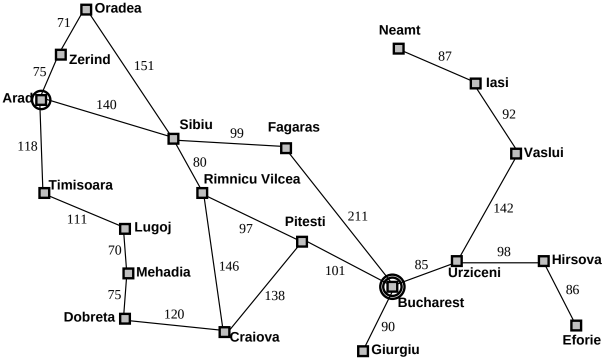
actions (decisions): select a road

Solution:

Sequence of cities (path)
(action/decision sequence [2])

Optimality – Cost:

Energy, time, tolls, ...



Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

actions (decisions): select a road

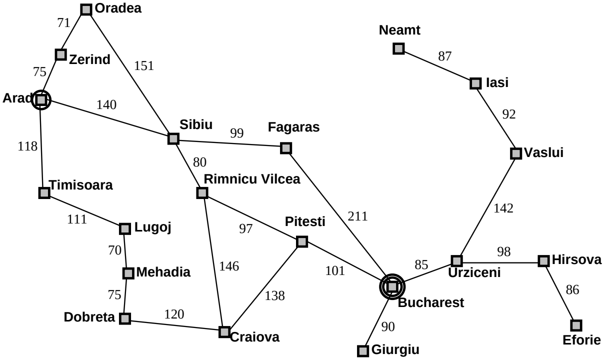
Solution:

Sequence of cities (path)

(action/decision sequence [2])

Optimality – Cost:

Energy, time, tolls, . . .



Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

actions (decisions): select a road

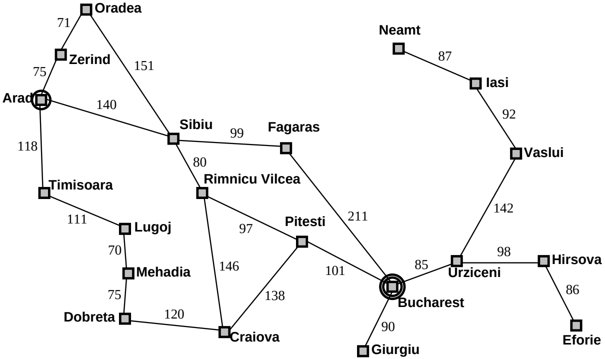
Solution:

Sequence of cities (path)

(action/decision sequence [2])

Optimality – Cost:

Energy, time, tolls, . . .



Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

actions (decisions): select a road

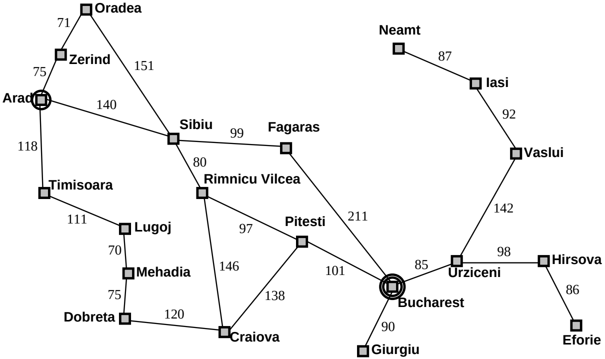
Solution:

Sequence of cities (path)

(action/decision sequence [2])

Optimality – Cost:

Energy, time, tolls, . . .



Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

actions (decisions): select a road

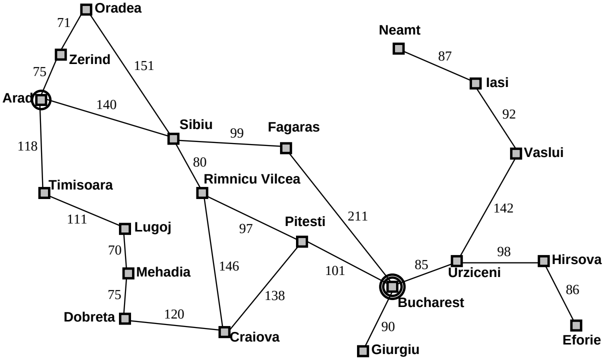
Solution:

Sequence of cities (path)

(action/decision sequence [2])

Optimality – Cost:

Energy, time, tolls, . . .



Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

actions (decisions): select a road

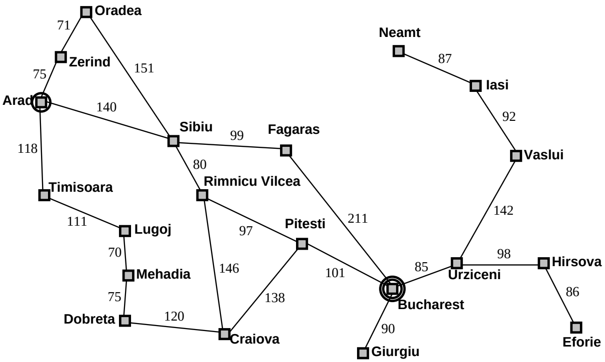
Solution:

Sequence of cities (path)

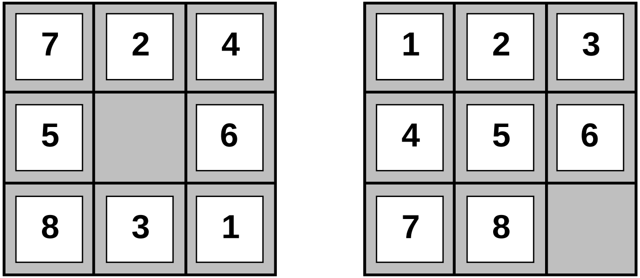
(action/decision sequence [2])

Optimality – Cost:

Energy, time, tolls, . . .



Example: The 8-puzzle



Start State **Goal State**

states?
actions?
solution?
cost?

A Search Problem

- State space (including Start/Initial state): position, board configuration,
- Actions : drive to, Up, Down, Left . . .
- Transition model : Given state and action return state (and cost)
- Goal test : Are we done?

A Search Problem

- State space (including Start/Initial state): position, board configuration,
- Actions : drive to, Up, Down, Left ...
- Transition model : Given state and action return state (and cost)
- Goal test : Are we done?

A Search Problem

- State space (including Start/Initial state): position, board configuration,
- Actions : drive to, Up, Down, Left . . .
- Transition model : Given state and action return state (and cost)
- Goal test : Are we done?

A Search Problem

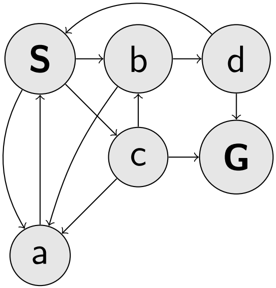
- State space (including Start/Initial state): position, board configuration,
- Actions : drive to, Up, Down, Left ...
- Transition model : Given state and action return state (and cost)
- Goal test : Are we done?

Discrete State Space

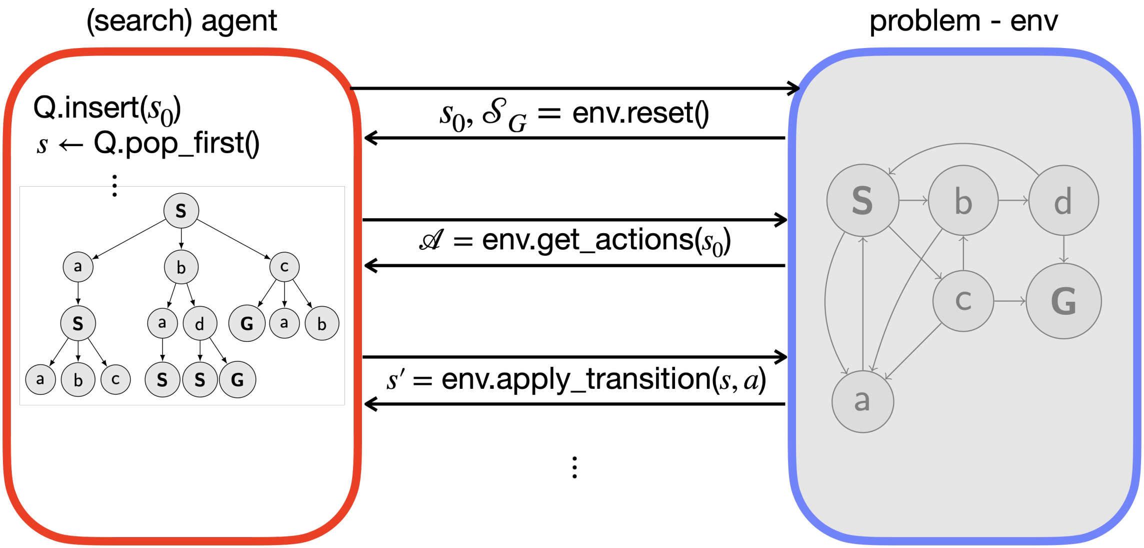
State space graph: a representation of a search problem

- States $s \in \mathcal{S} = \{\mathbf{S}, \mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathbf{G}\}$ (finite set)
- Arcs represent actions a, for each state s, $a \in \mathcal{A}(s)$ ($\mathcal{A}$ is also finite)
- State transition function $s' = \mathrm{result}(s, a)$
- Start (initial) state $s_0 \in \mathcal{S}$, $s_0 = \mathbf{S}$.
- Goal set $\mathcal{S}_G \subset \mathcal{S}$.

Each state occurs only *once* in a state (search) space.

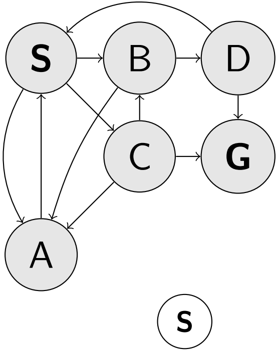


agent – problem dialog (a programmer's viewpoint)



BFS, Q is FIFO data structure (queue)

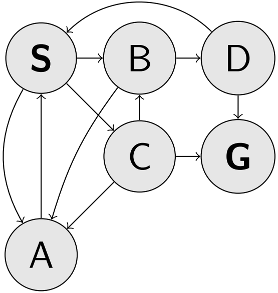
```
function FORWARD_SEARCH
    Q.insert(s0) and mark s0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s' ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure
```



Q: (_,**S**)
visited: **S**

BFS, Q is FIFO data structure (queue)

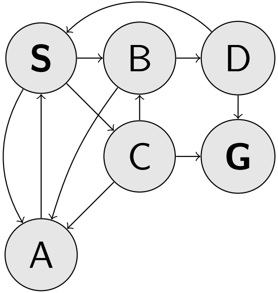
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

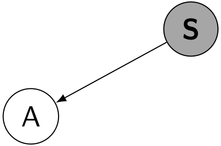


Q:
visited: **S**

BFS, Q is FIFO data structure (queue)

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ 𝒮_G then return Success
        for all a ∈ 𝒜(s) do
            s' ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure
```



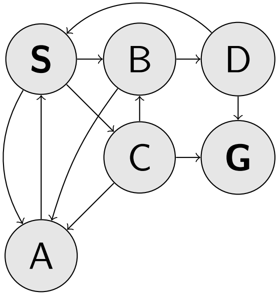


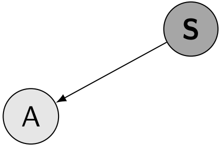
Q:

visited: **S**

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

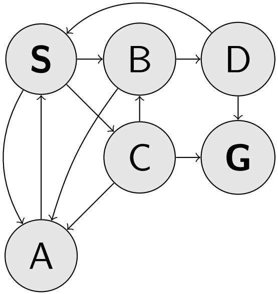


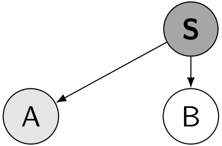


Q: (S,A)
visited: **S** A

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

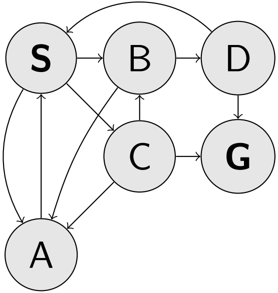


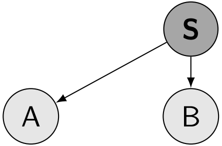


Q: (S,A)
visited: **S** A

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

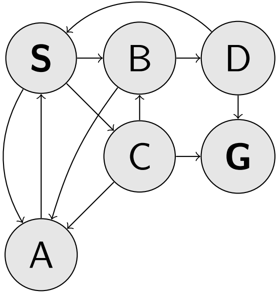


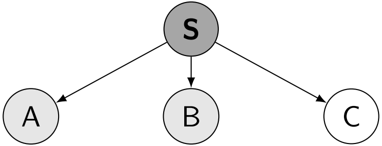


Q: (S,A) (S,B)
visited: **S** A B

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

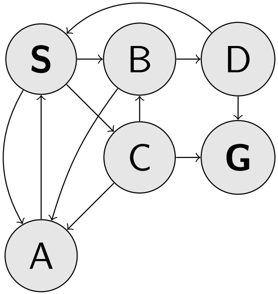


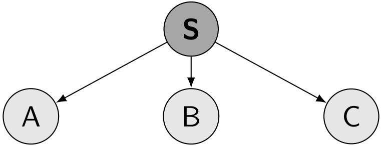


Q: (S,A) (S,B)
visited: **S** A B

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

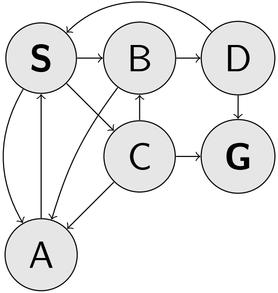


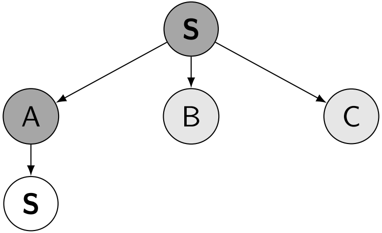


Q: (S,A) (S,B) (S,C)
visited: **S** A B C

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

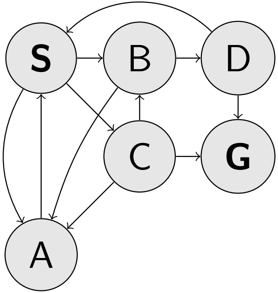


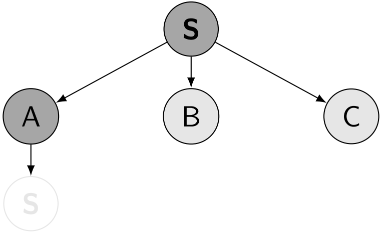


Q: (S,B) (S,C)
visited: **S** A B C

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

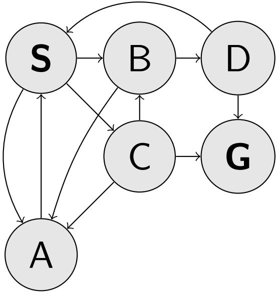


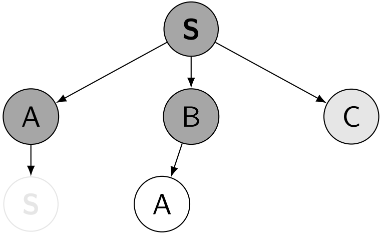


Q: (S,B) (S,C)
visited: **S** A B C

BFS, Q is FIFO data structure (queue)

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s' ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure
```



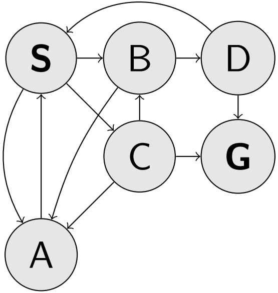


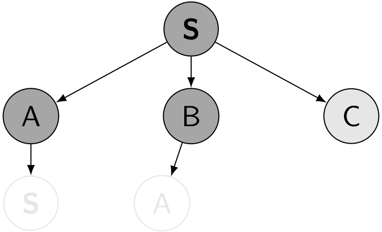
Q: (S,C)

visited: **S** A B C

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
 Q.insert(s_0) and mark s_0 as visited
 while Q not empty **do**
  $p, s \leftarrow$ Q.pop_first()
  parent[s] $\leftarrow p$
  if $s \in \mathcal{S}_G$ **then return** Success
  for all $a \in \mathcal{A}(s)$ **do**
   $s' \leftarrow$ result(s, a)
   if s' not visited **then**
    Mark s' as visited
    Q.insert(s, s')
   else
    Resolve duplicate s'
return Failure

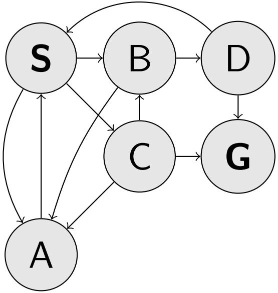


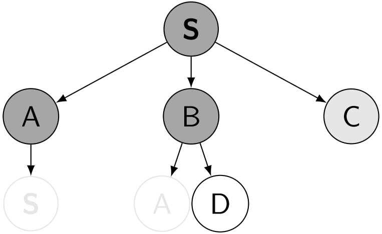


Q: (S,C)
visited: **S** A B C

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

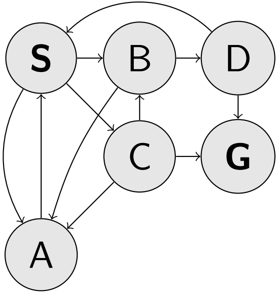


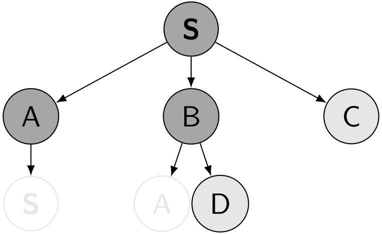


Q: (S,C)
visited: **S** A B C

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

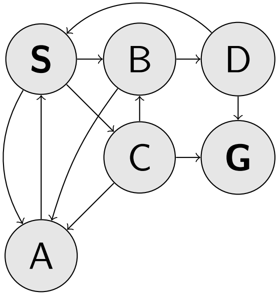


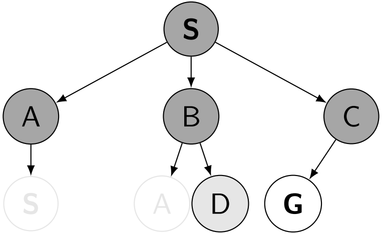


Q: (S,C) (B,D)
visited: **S** A B C D

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

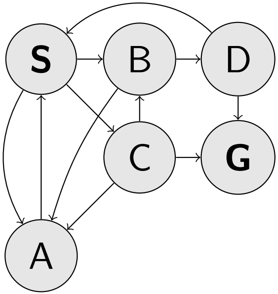


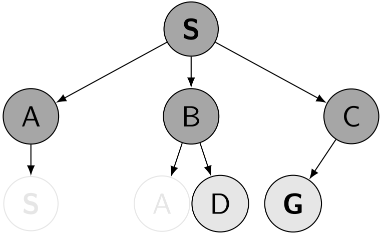


Q: (B,D)
visited: **S** A B C D

BFS, Q is FIFO data structure (queue)

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s' ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure
```



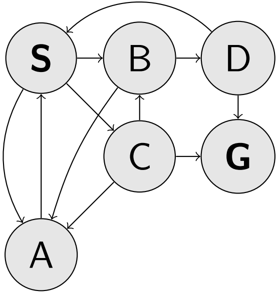


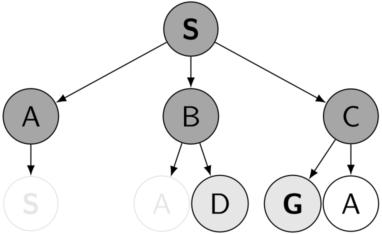
Q: (B,D) (C,**G**)

visited: **S** A B C D **G**

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

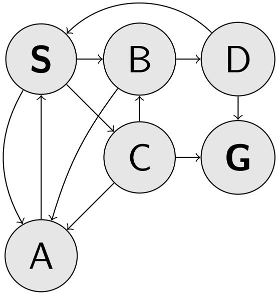


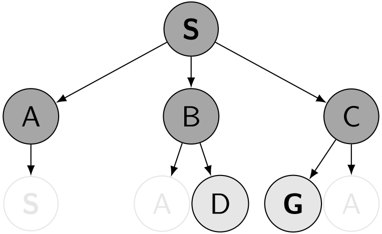


Q: (B,D) (C,**G**)
visited: **S** A B C D **G**

BFS, Q is FIFO data structure (queue)

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s' ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure
```



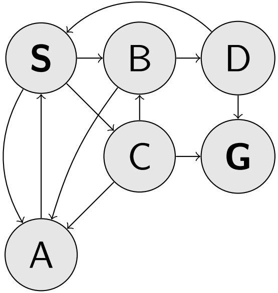


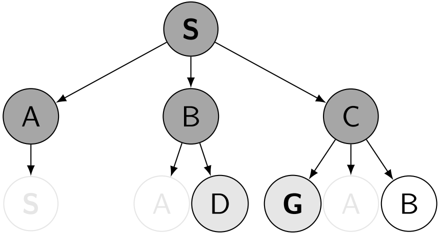
Q: (B,D) (C,**G**)

visited: **S** A B C D **G**

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

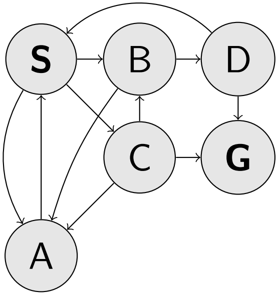


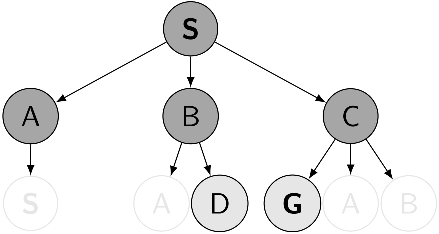


Q: (B,D) (C,**G**)
visited: **S** A B C D **G**

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

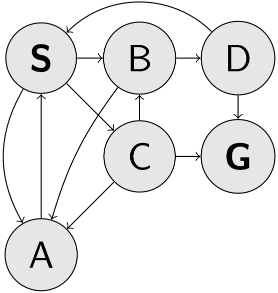


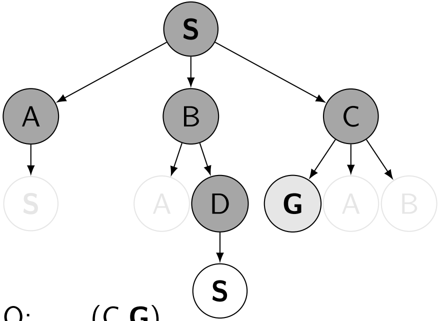


Q: (B,D) (C,**G**)
visited: **S** A B C D **G**

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

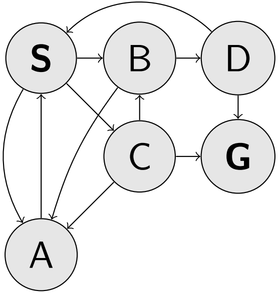


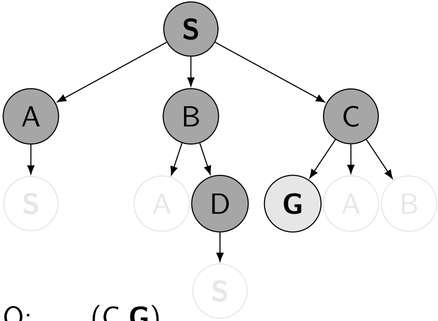


Q: (C,**G**)
visited: **S** A B C D **G**

BFS, Q is FIFO data structure (queue)

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s' ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure
```



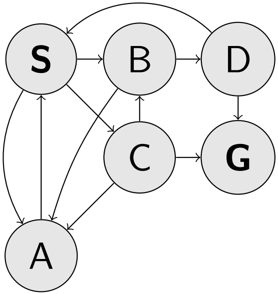


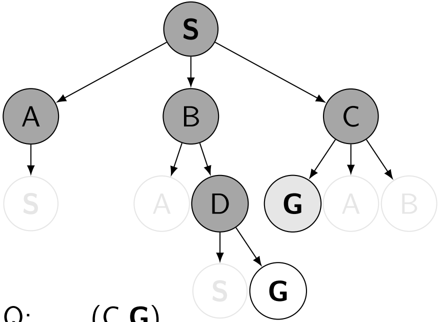
Q: (C,**G**)

visited: **S** A B C D **G**

BFS, Q is FIFO data structure (queue)

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ 𝒮_G then return Success
        for all a ∈ 𝒜(s) do
            s′ ← result(s, a)
            if s′ not visited then
                Mark s′ as visited
                Q.insert(s, s′)
            else
                Resolve duplicate s′
return Failure
```



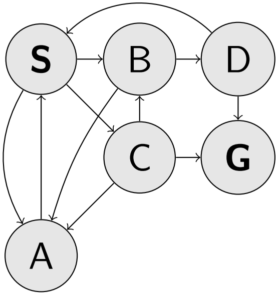


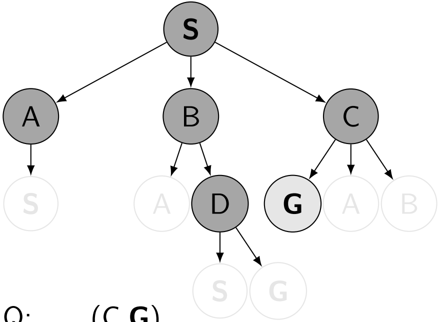
Q: (C,**G**)

visited: **S** A B C D **G**

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

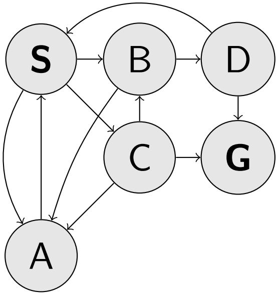


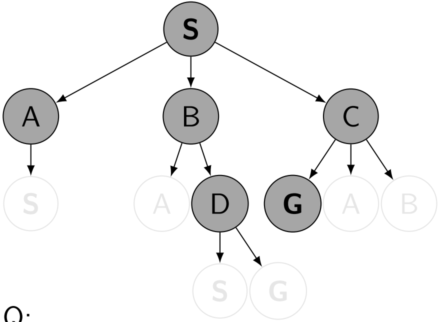


Q: (C,**G**)
visited: **S** A B C D **G**

BFS, Q is FIFO data structure (queue)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure





Q:
visited: **S** A B C D **G**

Search (algorithm) properties

- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Search (algorithm) properties

- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Search (algorithm) properties

- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Search (algorithm) properties

- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Search (algorithm) properties

- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Search (algorithm) properties

- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Search (algorithm) properties

- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Search (algorithm) properties

- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Search (algorithm) properties

- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Search (algorithm) properties

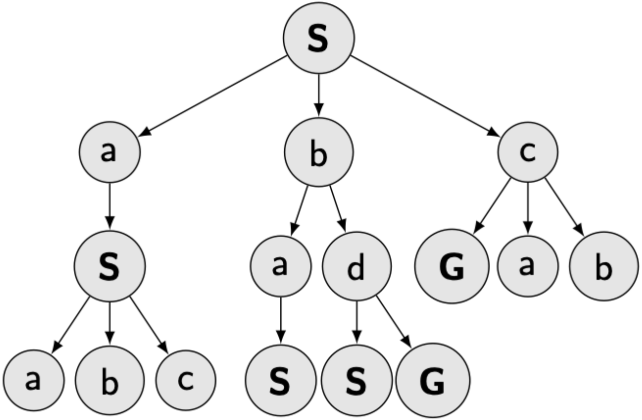
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

How many nodes in a (search) tree? What are tree parameters?

Strategies

How to traverse/build a search tree?

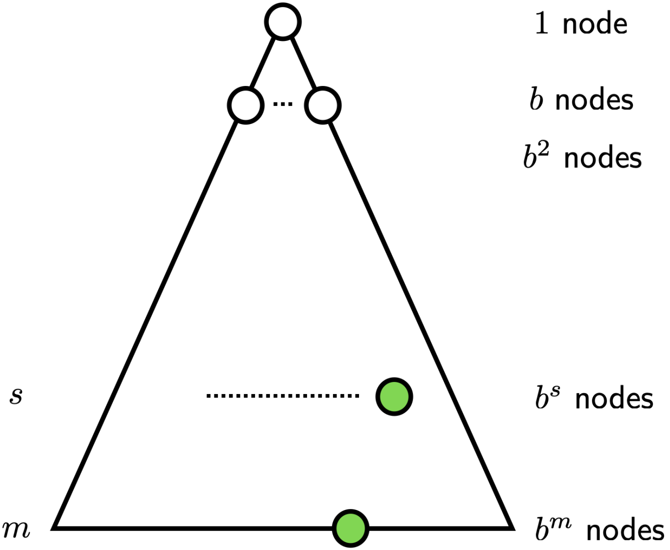
- Depth of the tree d. A node is in depth d.
- Max-Depth of the tree m. Can be ∞.
- (Average) Branching factor b.
- s denotes the depth of the shallowest Goal .
- How many nodes in the whole tree?



Strategies

How to traverse/build a search tree?

- Depth of the tree d. A node is in depth d.
- Max-Depth of the tree m. Can be ∞.
- (Average) Branching factor b.
- s denotes the depth of the shallowest Goal .
- How many nodes in the whole tree?



BFS properties

Complete?

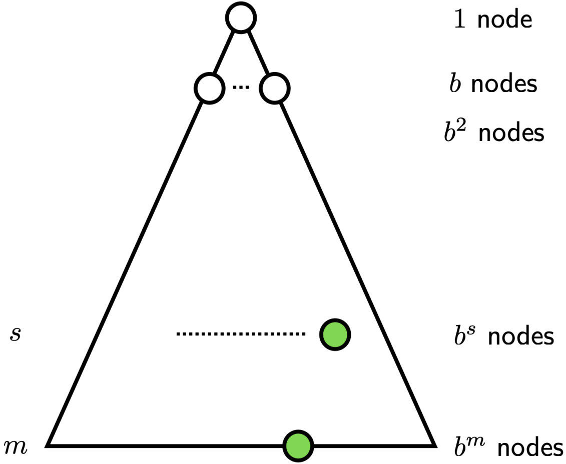
Optimal?

Space complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$

Time complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



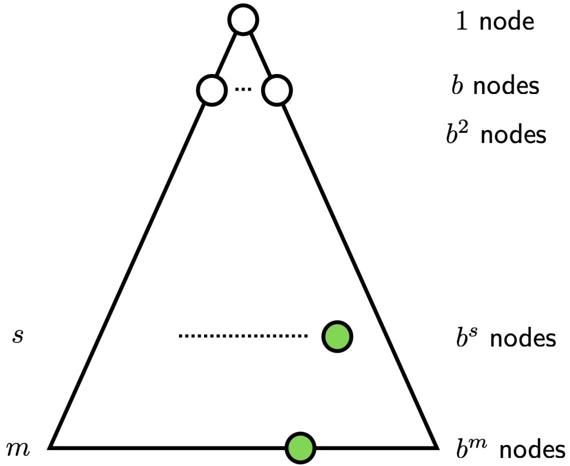
BFS properties

Complete?
Optimal?
Space complexity?

- A $\mathcal{O}(bm)$
- B $\mathcal{O}(b^m)$
- C $\mathcal{O}(m^b)$
- D $\mathcal{O}(b^s)$

Time complexity?

- A $\mathcal{O}(bm)$
- B $\mathcal{O}(b^m)$
- C $\mathcal{O}(m^b)$
- D $\mathcal{O}(b^s)$



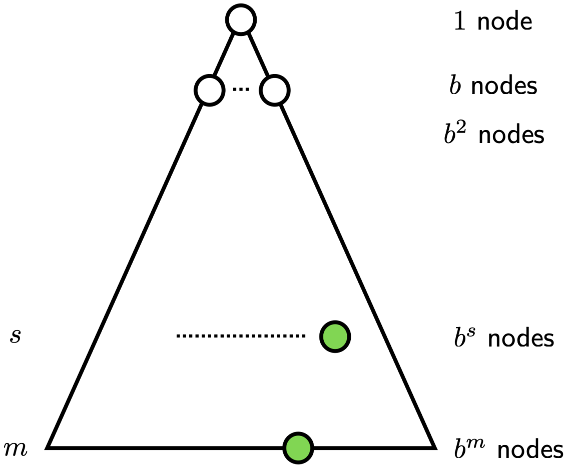
BFS properties

Complete?
Optimal?
Space complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$

Time complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



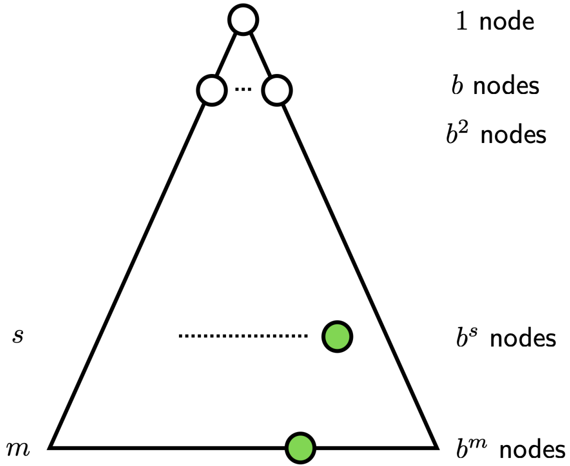
BFS properties

Complete?
Optimal?
Space complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$

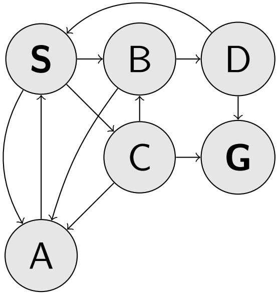
Time complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



DFS, Q is LIFO data structure (stack)

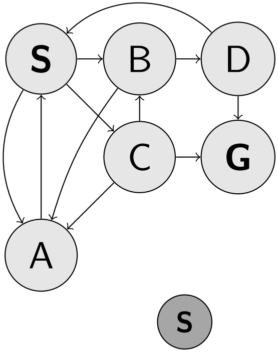
```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s' ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure
```



Q: (_,**S**)
visited: **S**

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

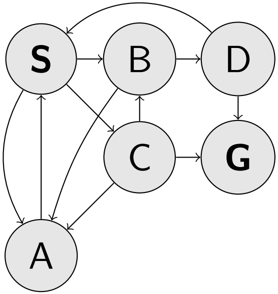


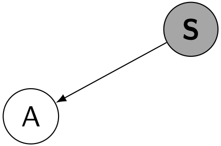
Q:

visited: **S**

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

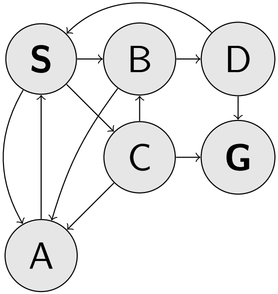


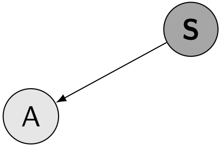


Q:
visited: **S**

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

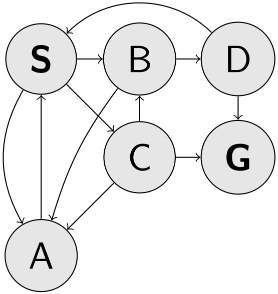


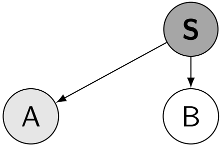


Q: (S,A)
visited: **S** A

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

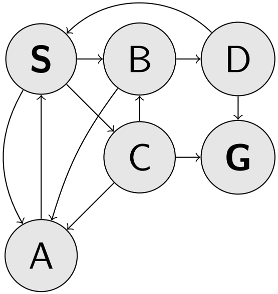


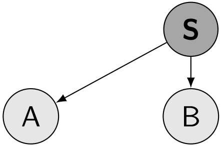


Q: (S,A)
visited: **S** A

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure

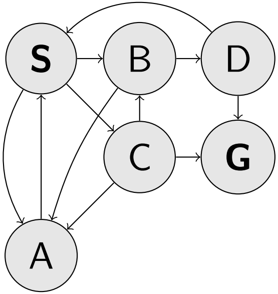


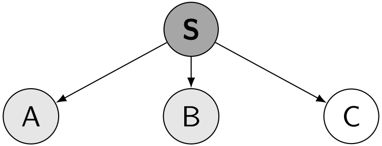


Q: (S,A) (S,B)
visited: **S** A B

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

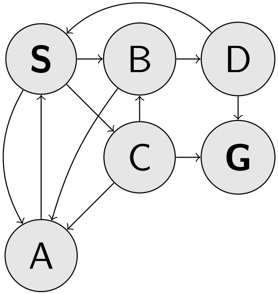


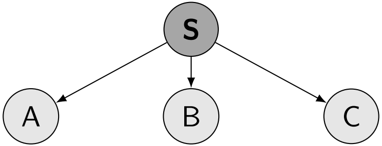


Q: (S,A) (S,B)
visited: **S** A B

DFS, Q is LIFO data structure (stack)

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ 𝒮_G then return Success
        for all a ∈ 𝒜(s) do
            s′ ← result(s, a)
            if s′ not visited then
                Mark s′ as visited
                Q.insert(s, s′)
            else
                Resolve duplicate s′
return Failure
```

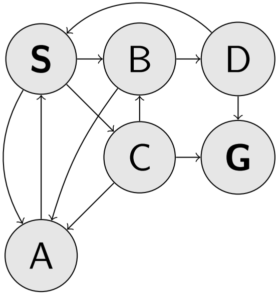


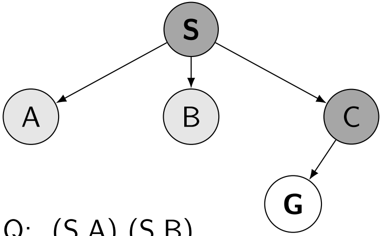


Q: (S,A) (S,B) (S,C)
visited: **S** A B C

DFS, Q is LIFO data structure (stack)

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s′ ← result(s, a)
            if s′ not visited then
                Mark s′ as visited
                Q.insert(s, s′)
            else
                Resolve duplicate s′
return Failure
```



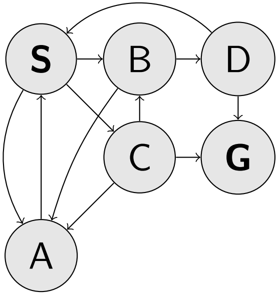


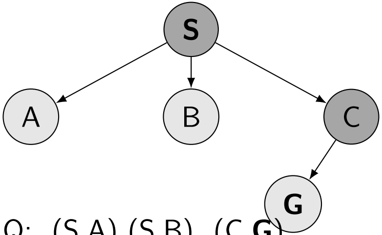
Q: (S,A) (S,B)

visited: **S** A B C

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

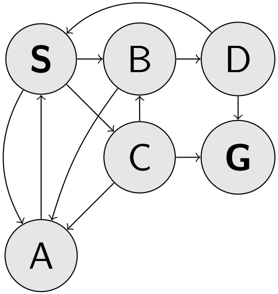


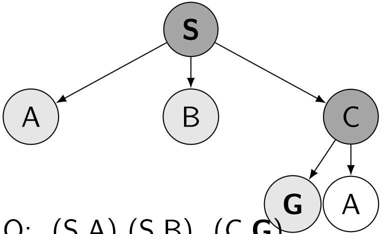


Q: (S,A) (S,B) (C,**G**)
visited: **S** A B C **G**

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

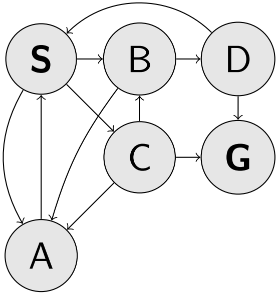


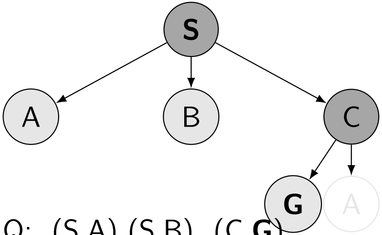


Q: (S,A) (S,B) (C,**G**)
visited: **S** A B C **G**

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

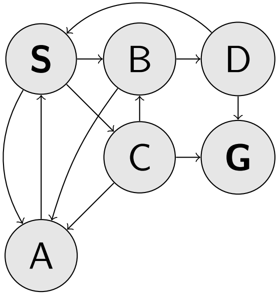


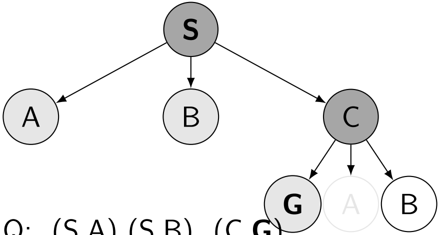


Q: (S,A) (S,B) (C,**G**)
visited: **S** A B C **G**

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s' \leftarrow$ result(s, a)
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure

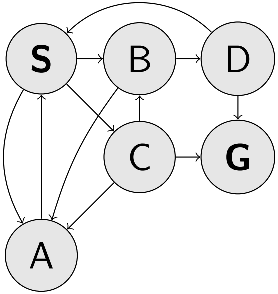


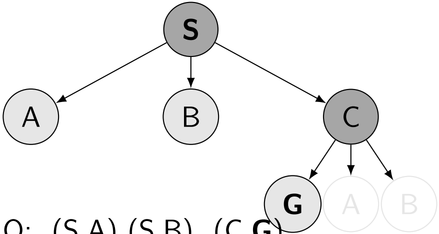


Q: (S,A) (S,B) (C,**G**)
visited: **S** A B C **G**

DFS, Q is LIFO data structure (stack)

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s ← Q.pop_first()
        parent[s] ← p
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s' ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s')
            else
                Resolve duplicate s'
return Failure
```



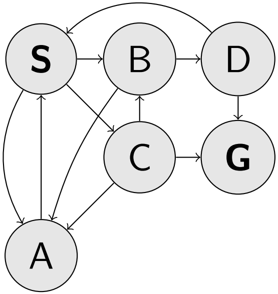


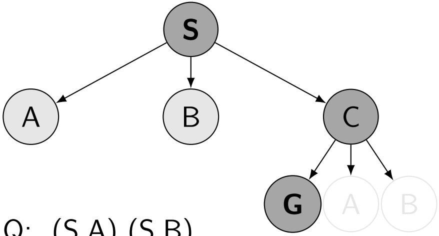
Q: (S,A) (S,B) (C,**G**)

visited: **S** A B C **G**

DFS, Q is LIFO data structure (stack)

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s' \leftarrow$ result(s, a)
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s')
      else
        Resolve duplicate s'
return Failure





Q: (S,A) (S,B)
visited: **S** A B C **G**

DFS properties

Complete?

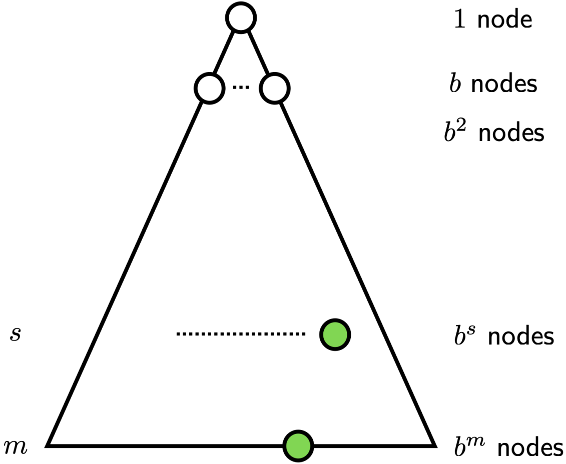
Optimal?

Space complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$

Time complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



DFS properties

Complete?

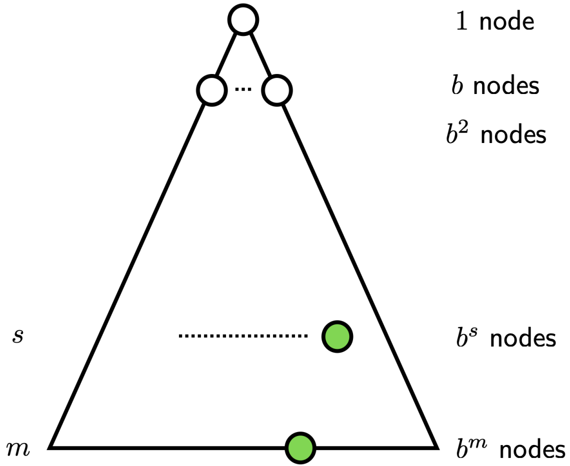
Optimal?

Space complexity?

- A $\mathcal{O}(bm)$
- B $\mathcal{O}(b^m)$
- C $\mathcal{O}(m^b)$
- D $\mathcal{O}(b^s)$

Time complexity?

- A $\mathcal{O}(bm)$
- B $\mathcal{O}(b^m)$
- C $\mathcal{O}(m^b)$
- D $\mathcal{O}(b^s)$



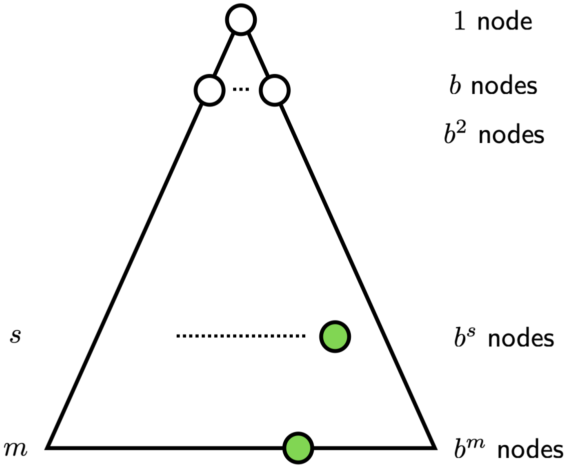
DFS properties

Complete?
Optimal?
Space complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$

Time complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



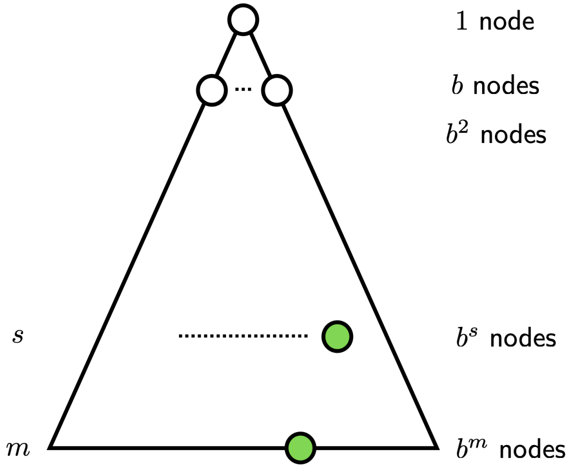
DFS properties

Complete?
Optimal?
Space complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$

Time complexity?

- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



Iterative deepening DFS (ID-DFS)

- Start with `maxdepth = 1`
- Perform DFS with limited depth. Report success or failure.
- If failure, forget everything, increase `maxdepth` and repeat DFS

Is it not a terrible waste to forget everything between steps?

Iterative deepening DFS (ID-DFS)

- Start with `maxdepth = 1`
- Perform DFS with limited depth. Report success or failure.
- If failure, forget everything, increase `maxdepth` and repeat DFS

Is it not a terrible waste to forget everything between steps?

Iterative deepening DFS (ID-DFS)

- Start with `maxdepth = 1`
- Perform DFS with limited depth. Report success or failure.
- If failure, forget everything, increase `maxdepth` and repeat DFS

Is it not a terrible waste to forget everything between steps?

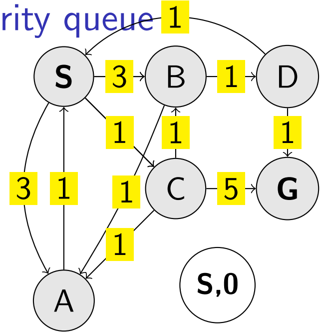
Iterative deepening DFS (ID-DFS)

- Start with `maxdepth = 1`
- Perform DFS with limited depth. Report success or failure.
- If failure, forget everything, increase `maxdepth` and repeat DFS

Is it not a terrible waste to forget everything between steps?

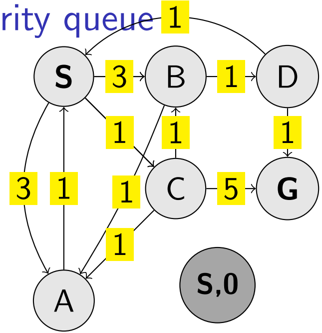
Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s', c ← result(s, a)                    ▷ c cost
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure
```



Q: (_,**S**,0)
visited: **S**

Uniform Cost Search (Dijkstra), Q is priority queue



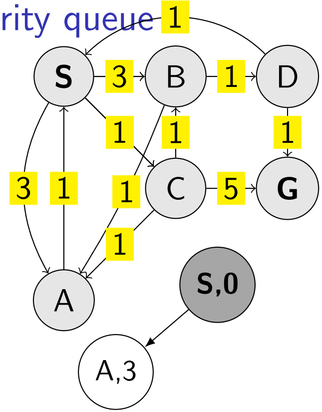

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s', c ← result(s, a)                    ▷ c cost
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure
```

Q:

visited: **S**

Uniform Cost Search (Dijkstra), Q is priority queue

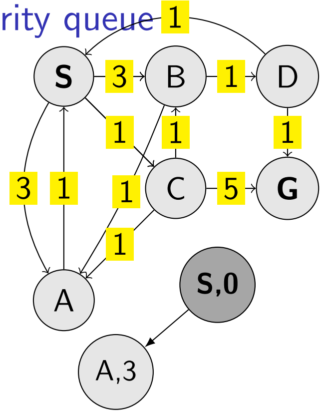
```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ 𝒮_G then return Success
        for all a ∈ 𝒜(s) do
            s', c ← result(s, a)                    ▷ c cost
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure
```



Q:

visited: **S**

Uniform Cost Search (Dijkstra), Q is priority queue

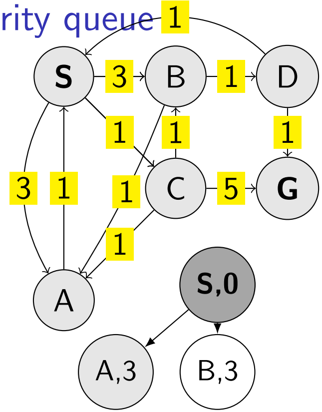


function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure

Q: (S,A,3)
visited: **S** A

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s', c ← result(s, a)                         ▷ c cost
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure
```

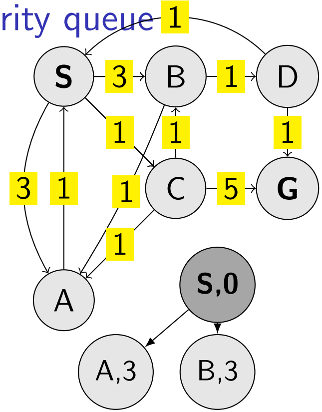


Q: (S,A,3)

visited: **S** A

Uniform Cost Search (Dijkstra), Q is priority queue

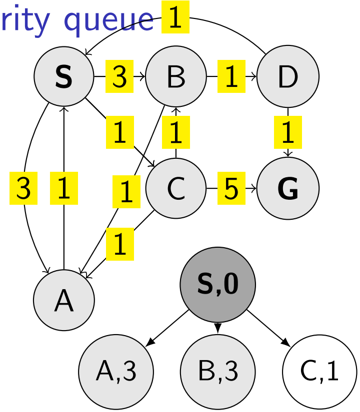
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s, \_ \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow s$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s', c \leftarrow$ result(s, a) ▷ c cost
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure



Q: (S,A,3) (S,B,3)
visited: **S** A B

Uniform Cost Search (Dijkstra), Q is priority queue

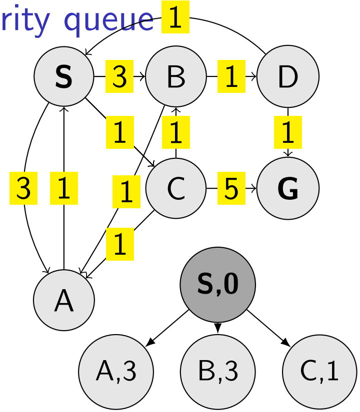
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (S,A,3) (S,B,3)
visited: **S** A B

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s′, c ← result(s, a)                    ▷ c cost
            if s′ not visited then
                Mark s′ as visited
                Q.insert(s, s′, cost_from_start)
            else
                Resolve duplicate s′
    return Failure
```

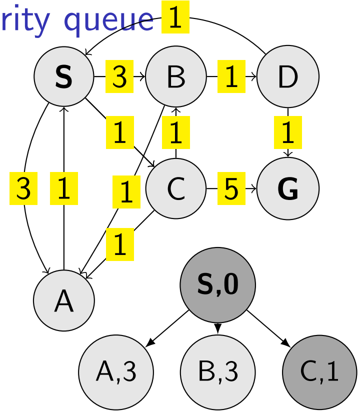


Q: (S,C,1) (S,A,3) (S,B,3)

visited: **S** A B C

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s', c ← result(s, a)                    ▷ c cost
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure
```

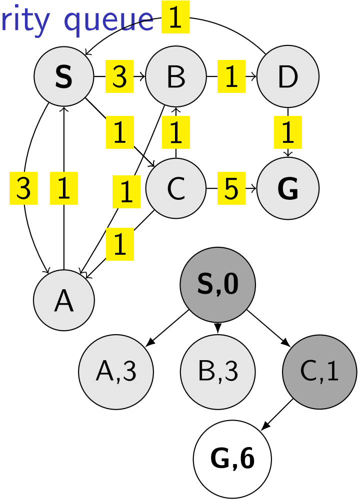


Q: (S,A,3) (S,B,3)

visited: **S** A B C

Uniform Cost Search (Dijkstra), Q is priority queue

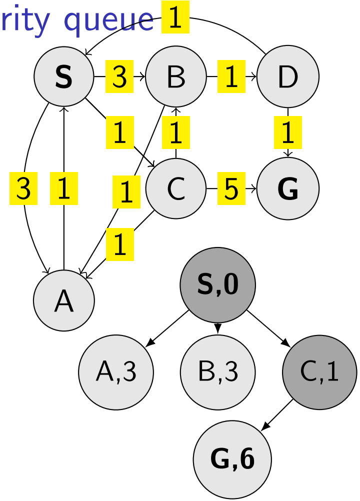
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow$ result(s, a)   ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (S,A,3) (S,B,3)
visited: **S** A B C

Uniform Cost Search (Dijkstra), Q is priority queue

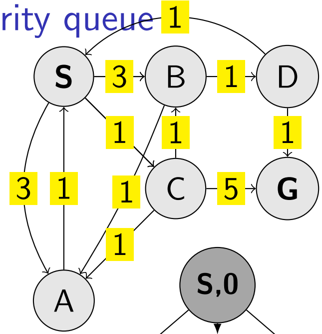
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure

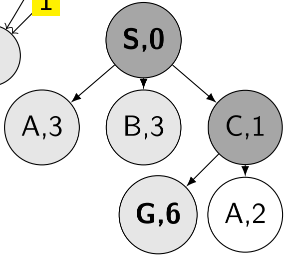


Q: (S,A,3) (S,B,3) (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow$ result(s, a) ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure

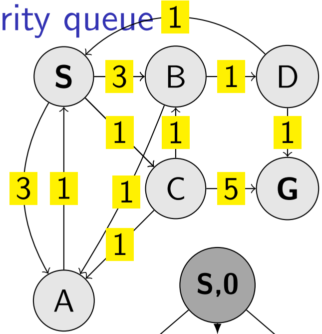


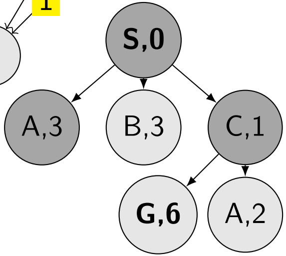


Q: (S,A,3) (S,B,3) (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s', c ← result(s, a)                      ▷ c cost
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure
```



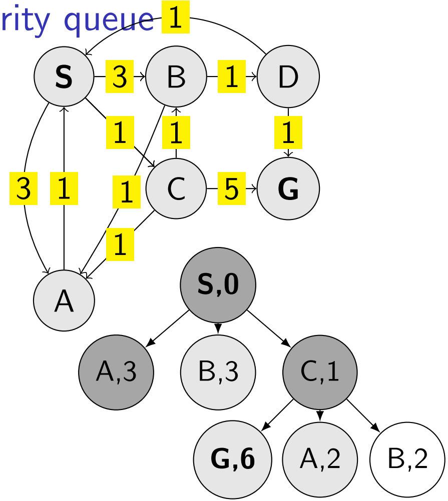


Q: (C,A,2) (S,B,3) (C,**G**,6)

visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

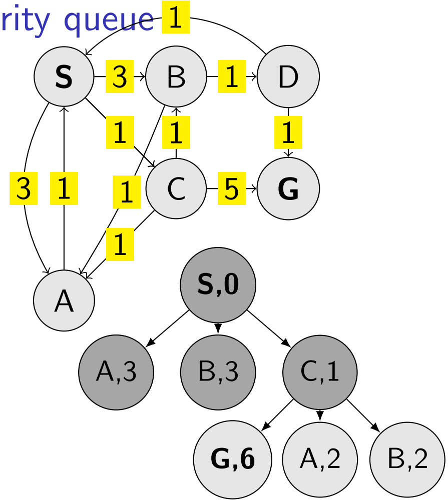
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (C,A,2) (S,B,3) (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

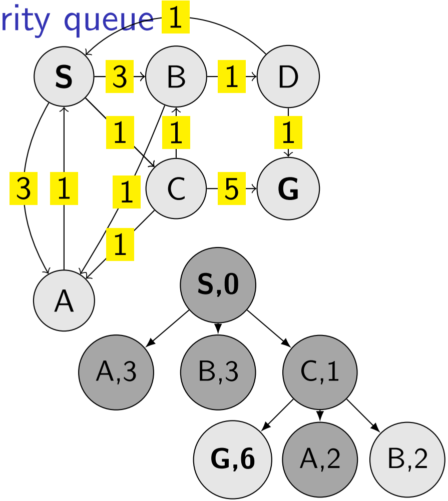
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (C,A,2) (C,B,2) (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ 𝒮_G then return Success
        for all a ∈ 𝒜(s) do
            s′, c ← result(s, a)                    ▷ c cost
            if s′ not visited then
                Mark s′ as visited
                Q.insert(s, s′, cost_from_start)
            else
                Resolve duplicate s′
    return Failure
```

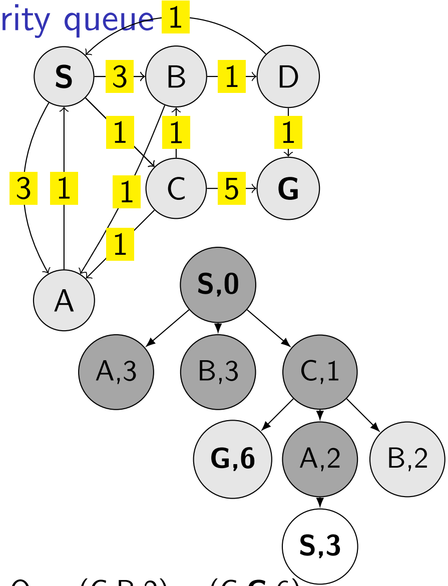


Q: (C,B,2) (C,**G**,6)

visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

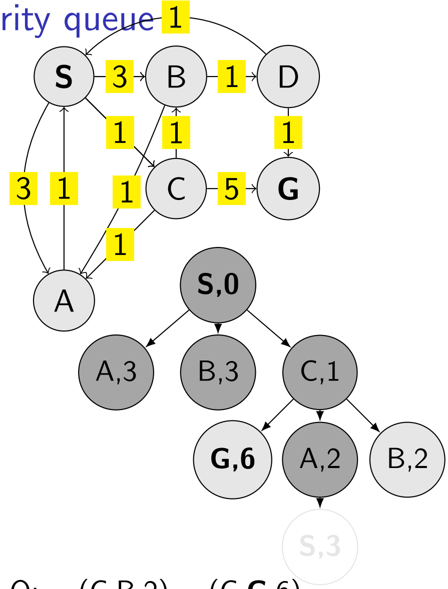
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (C,B,2) (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

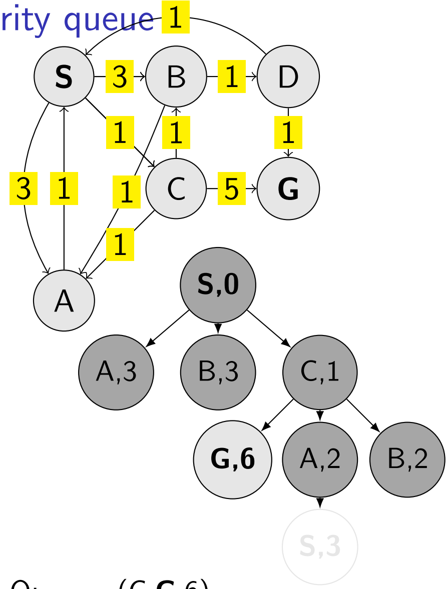
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s, \_ \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow s$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure



Q: (C,B,2) (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

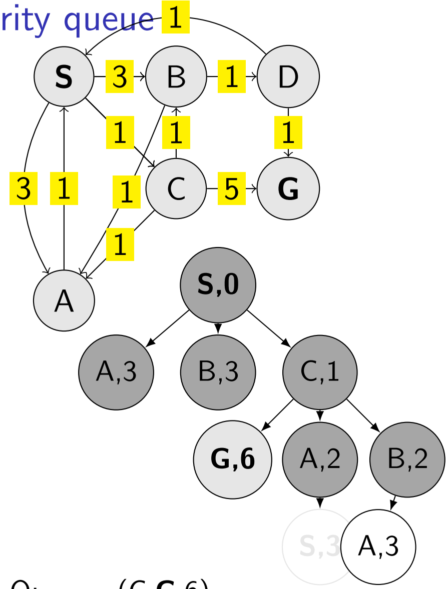
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

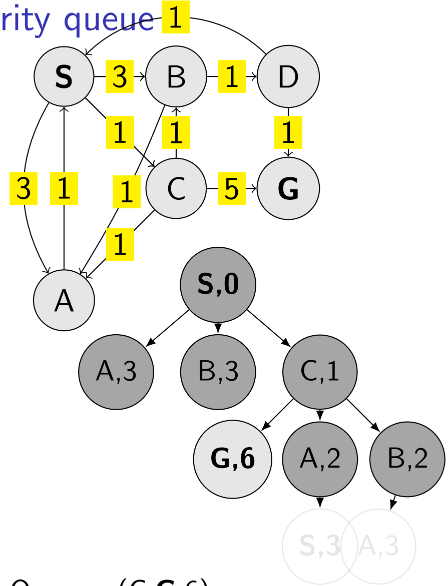
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

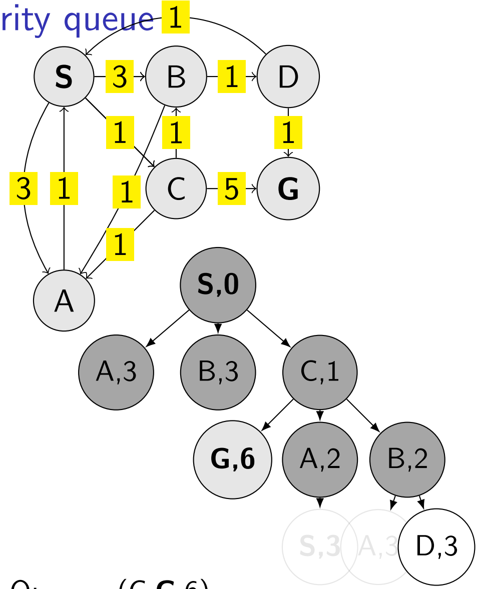
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

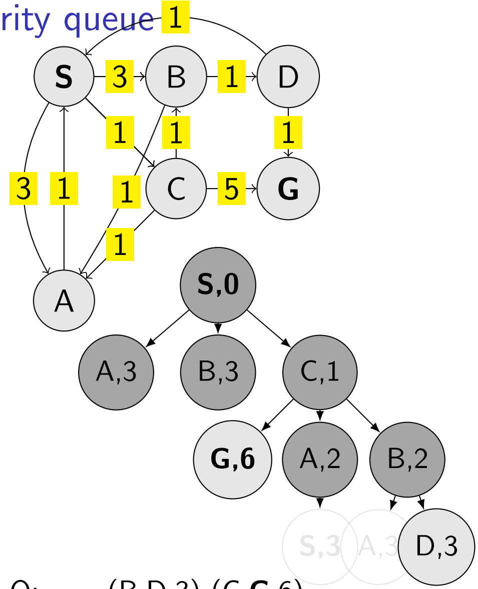
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (C,**G**,6)
visited: **S** A B C **G**

Uniform Cost Search (Dijkstra), Q is priority queue

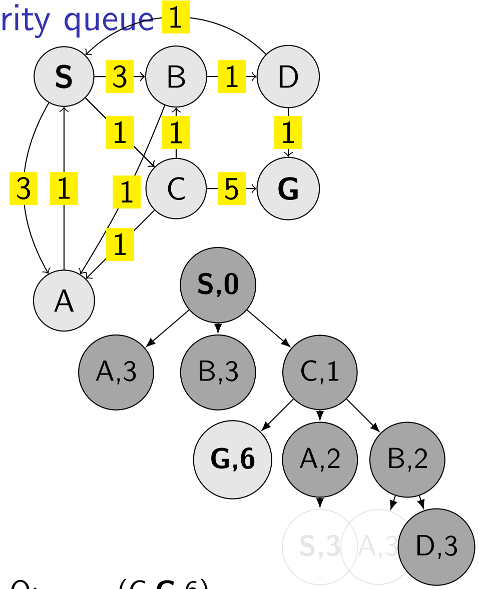
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s, \_ \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow s$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s', c \leftarrow$ result(s, a)   ▷ c cost
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure



Q: (B,D,3) (C,**G**,6)
visited: **S** A B C **G** D

Uniform Cost Search (Dijkstra), Q is priority queue

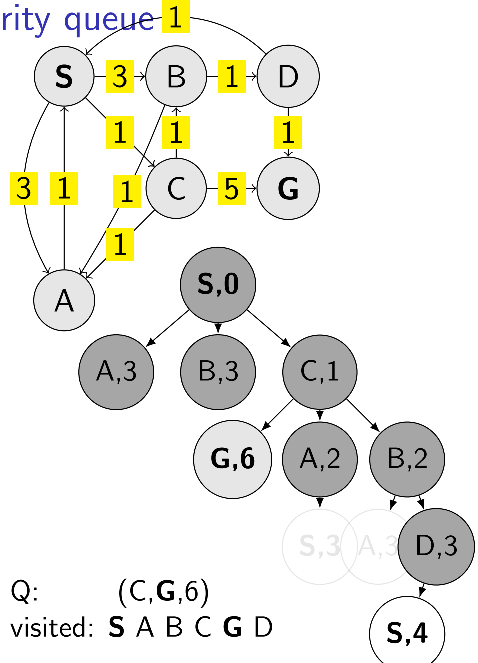
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Q: (C,**G**,6)
visited: **S** A B C **G** D

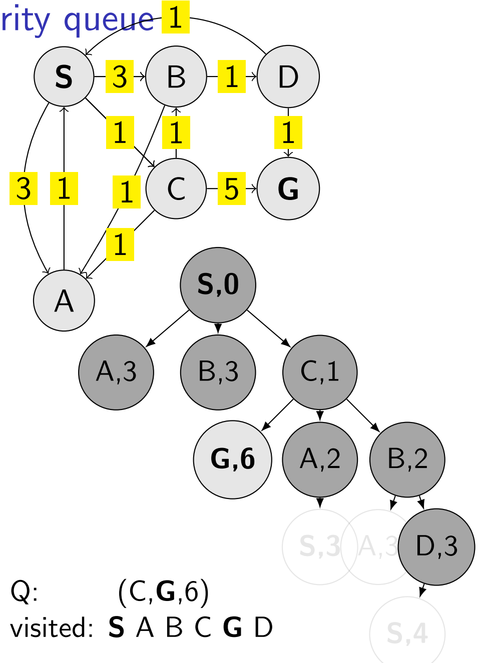
Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow$ result(s, a) ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



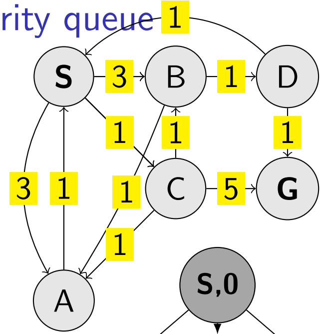
Uniform Cost Search (Dijkstra), Q is priority queue

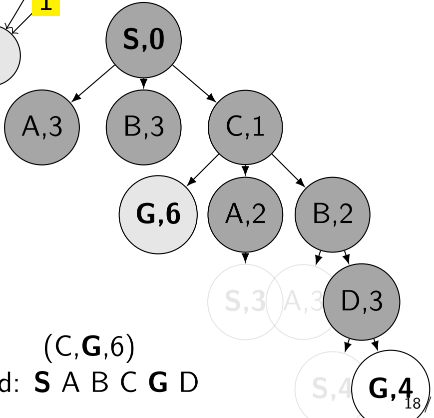
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \mathrm{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s, \_ \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow s$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s', c \leftarrow$ result(s, a)   ▷ c cost
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure

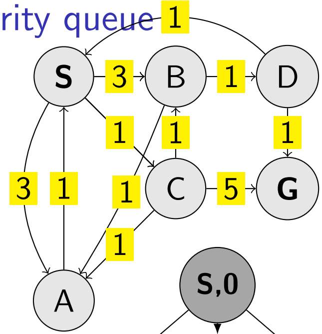


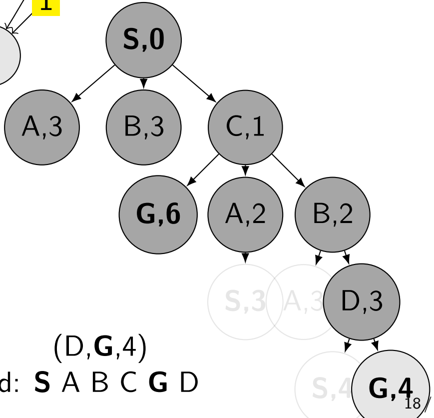


Q: (C,**G**,6)
visited: **S** A B C **G** D

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s, \_ \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow s$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure

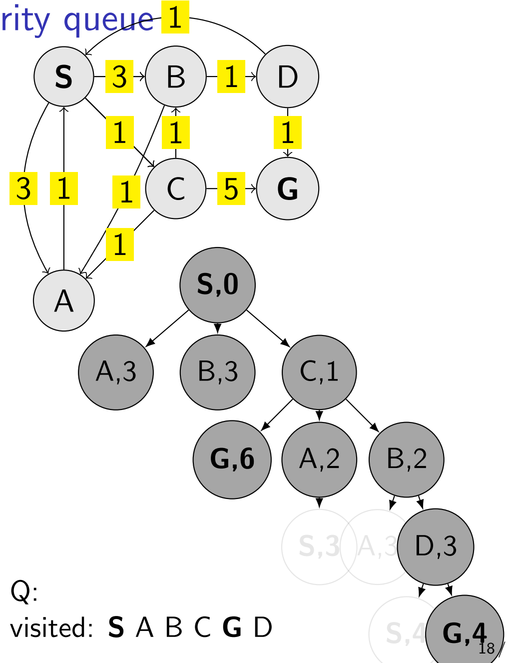




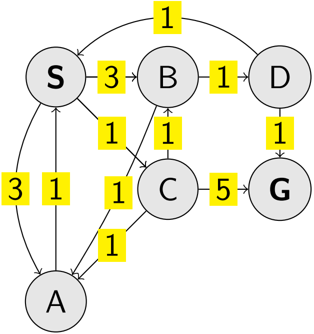
Q: (D,**G**,4)
visited: **S** A B C **G** D

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s, \_ \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow s$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s', c \leftarrow$ result(s, a) ▷ c cost
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s', cost_from_start)
            else
                Resolve duplicate s'
    return Failure



UCS properties

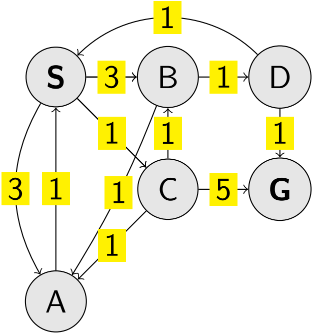


Complete?

Optimal?

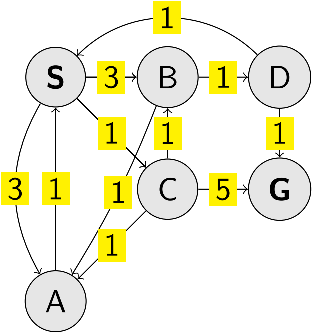
Complexities?

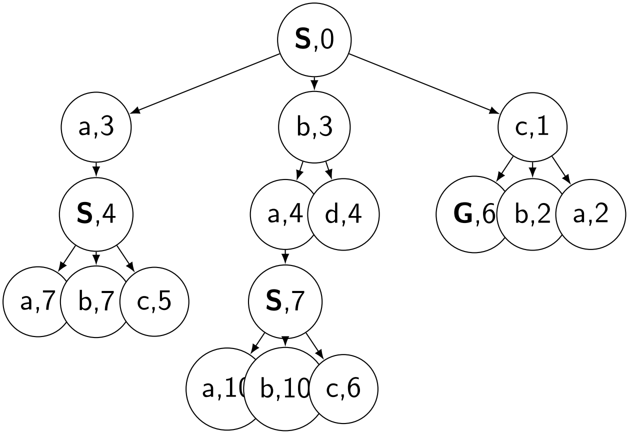
UCS properties



Complete?
Optimal?
Complexities?

UCS properties





Complete?
Optimal?
Complexities?

Node selection, take argmin $f(n)$. Search Node: $n = (p, s, \text{priority\_value})$

Selecting next node to explore (`pop` operation):

$$\texttt{node} \leftarrow \operatorname*{argmin}_{n \in \texttt{Q}} f(n)$$

What is $f(n)$ for DFS, BFS, and UCS?

- DFS:
- BFS:
- UCS:

- $f(n) = n.\text{cost\_from\_start}$
- $f(n) = n.\text{depth}$
- $f(n) = -n.\text{depth}$

The good: (one) frontier as a priority queue
(I.e., priority queue will work universally. Still, stack (LIFO) and queue (FIFO) are (conceptually) the perfect data structures for DFS and BFS, respectively.)
The bad: All the $f(n)$ correspond to the cost from n to the start cost-to-come (to n).

Node selection, take argmin $f(n)$. Search Node: $n = (p, s, \text{priority\_value})$

Selecting next node to explore (pop operation):

$$\texttt{node} \leftarrow \operatorname*{argmin}_{n \in \texttt{Q}} f(n)$$

What is $f(n)$ for DFS, BFS, and UCS?

- DFS:
- BFS:
- UCS:

- $f(n) = n.\text{cost\_from\_start}$
- $f(n) = n.\text{depth}$
- $f(n) = -n.\text{depth}$

The good: (one) frontier as a priority queue
(I.e., priority queue will work universally. Still, stack (LIFO) and queue (FIFO) are (conceptually) the perfect data structures for DFS and BFS, respectively.)
The bad: All the $f(n)$ correspond to the cost from n to the start cost-to-come (to n).

Node selection, take argmin $f(n)$. Search Node: $n = (p, s, \text{priority\_value})$

Selecting next node to explore (`pop` operation):

$$\texttt{node} \leftarrow \operatorname*{argmin}_{n \in \texttt{Q}} f(n)$$

What is $f(n)$ for DFS, BFS, and UCS?

- DFS:
- BFS:
- UCS:

- $f(n) = n.\text{cost\_from\_start}$
- $f(n) = n.\text{depth}$
- $f(n) = -n.\text{depth}$

The good: (one) frontier as a priority queue
(I.e., priority queue will work universally. Still, stack (LIFO) and queue (FIFO) are (conceptually) the perfect data structures for DFS and BFS, respectively.)
The bad: All the $f(n)$ correspond to the cost from n to the start cost-to-come (to n).

Node selection, take argmin $f(n)$. Search Node: $n = (p, s, \text{priority\_value})$

Selecting next node to explore (`pop` operation):

$$\texttt{node} \leftarrow \operatorname*{argmin}_{n \in \texttt{Q}} f(n)$$

What is $f(n)$ for DFS, BFS, and UCS?

- DFS:
- BFS:
- UCS:

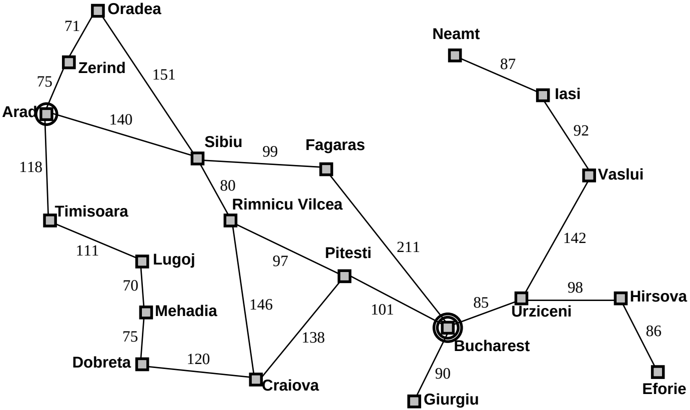
- $f(n) = n.\text{cost\_from\_start}$
- $f(n) = n.\text{depth}$
- $f(n) = -n.\text{depth}$

The good: (one) frontier as a priority queue
(I.e., priority queue will work universally. Still, stack (LIFO) and queue (FIFO) are (conceptually) the perfect data structures for DFS and BFS, respectively.)
The bad: All the $f(n)$ correspond to the cost from n to the start cost-to-come (to n).

How far are we from the goal cost-to-go ? – Heuristics

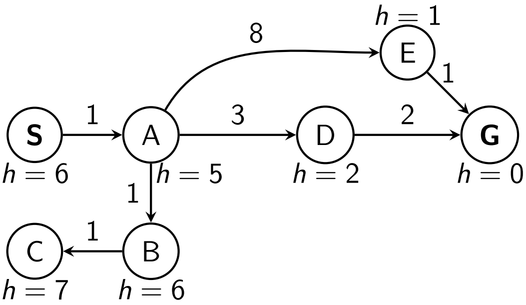
- A function that estimates how close a *state* is to the goal.
- Designed for a particular problem.
- $h(n.state)$ – it is function of the state (attribute of node)
- It is often shortened as $h(n)$ – heuristic value of node n.

Example of heuristics



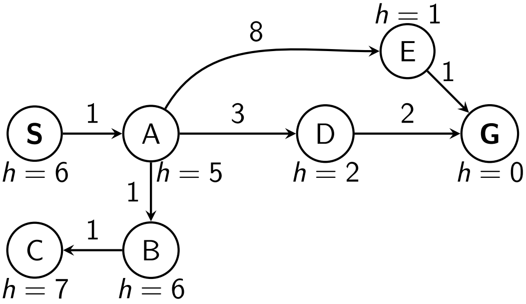
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Greedy, take the $n^* = \operatorname{argmin}_{n \in \mathtt{Q}} h(n)$



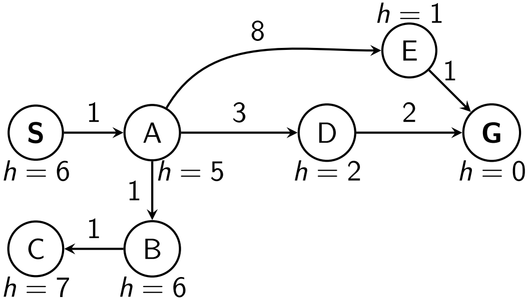
What is wrong (and nice) with the Greedy?

Greedy, take the $n^* = \operatorname{argmin}_{n \in \mathtt{Q}} h(n)$



What is wrong (and nice) with the Greedy?

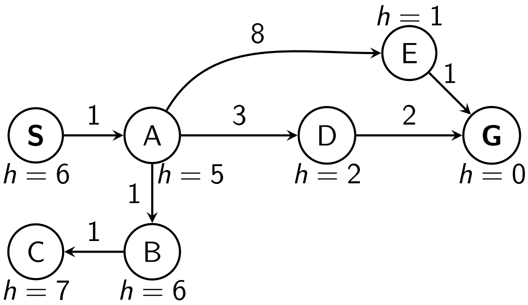
A* combines UCS and Greedy



UCS orders by backward (path) cost $g(n)$
Greedy uses heuristics (goal proximity) $h(n)$

A* orders nodes by: $f(n) = g(n) + h(n)$

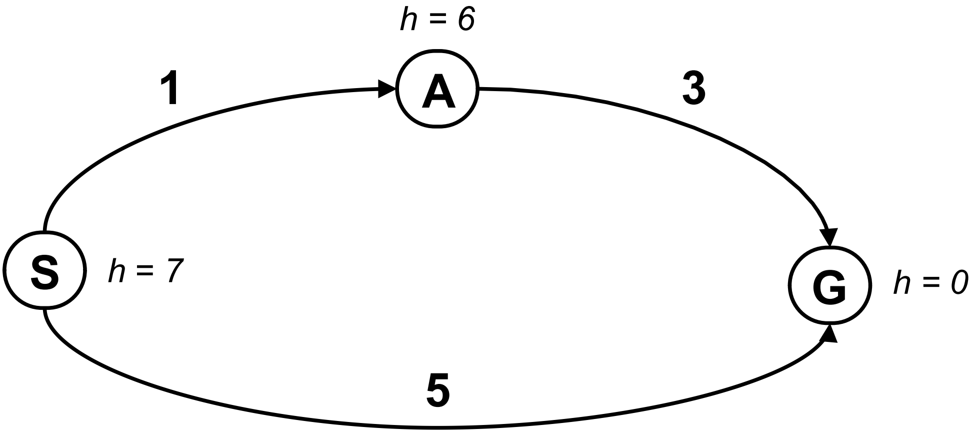
A* combines UCS and Greedy



UCS orders by backward (path) cost $g(n)$
Greedy uses heuristics (goal proximity) $h(n)$

A* orders nodes by: $f(n) = g(n) + h(n)$

Is A* optimal?

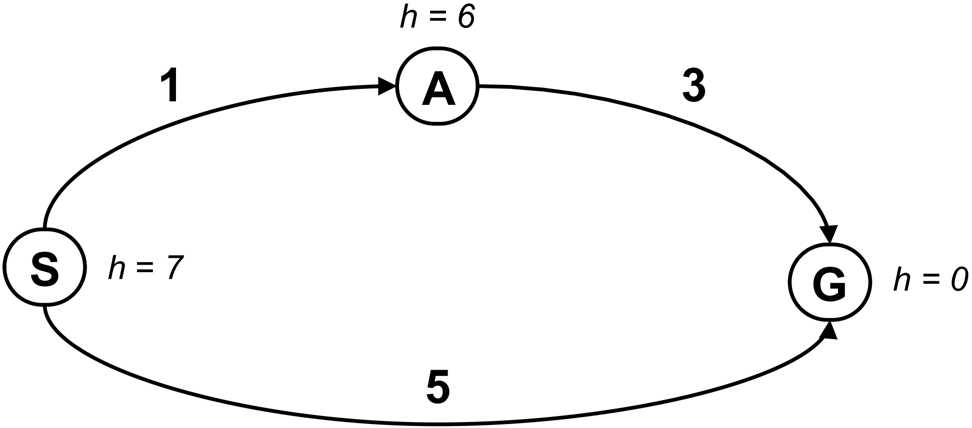


[2]

What is the problem?

[2] Graph example: Dan Klein and Pieter Abbeel

Is A* optimal?



2

What is the problem?

[2] Graph example: Dan Klein and Pieter Abbeel

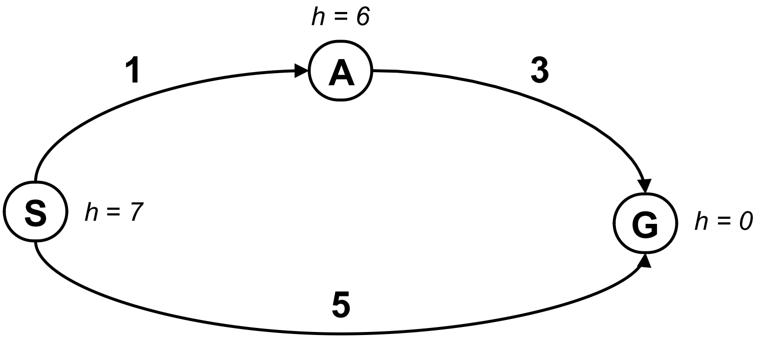
What is the right $h(A)$?

A: $0 \leq h(A) \leq 4$

B: $h(A) \leq 3$

C: $0 \leq h(A) \leq 3$

D: $0 \leq h(A)$



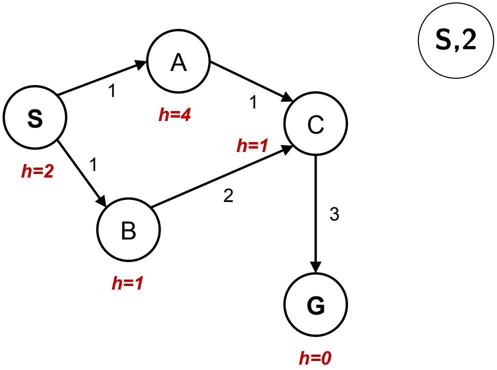
Admissible heuristics

A heuristic function h is admissible if:

$$
\begin{aligned}
h(n) &\leq \text{cost}(n.state, \text{Goal}_{\text{nearest}}) \\
h(\text{Goal}) &= 0
\end{aligned}
$$

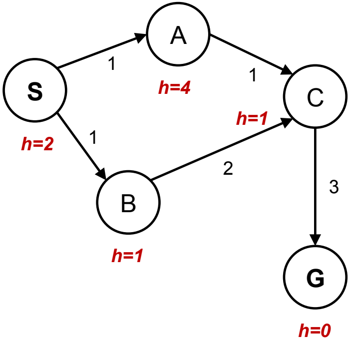
Consistent heuristic

function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$
      if s' not visited **then**
        Mark s' as visited
        $\text{Q.insert}(s, s', \text{cost\_from\_start} + h(s'))$
      else
        Resolve duplicate s'
  return Failure



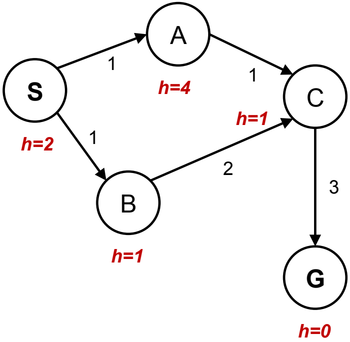
Consistent heuristic

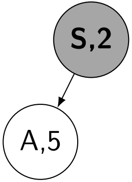
```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s', c ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s', cost_from_start + h(s'))
            else
                Resolve duplicate s'
    return Failure
```



Consistent heuristic

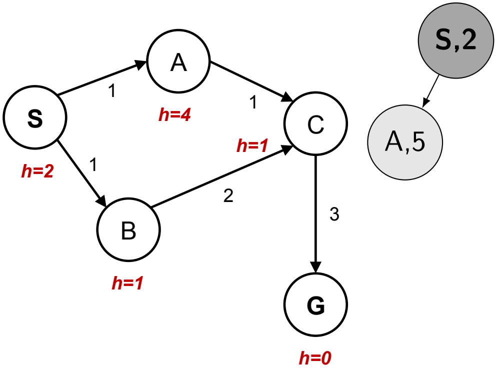
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$
      if s' not visited **then**
        Mark s' as visited
        $\text{Q.insert}(s, s', \text{cost\_from\_start} + h(s'))$
      else
        Resolve duplicate s'
  return Failure





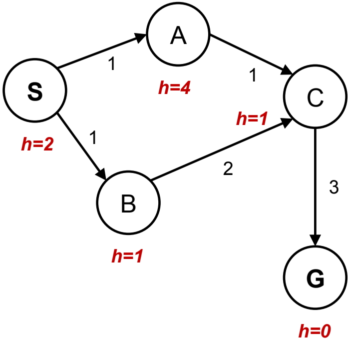
Consistent heuristic

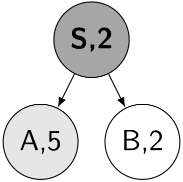
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$
      if s' not visited **then**
        Mark s' as visited
        $\text{Q.insert}(s, s', \text{cost\_from\_start} + h(s'))$
      else
        Resolve duplicate s'
  return Failure



Consistent heuristic

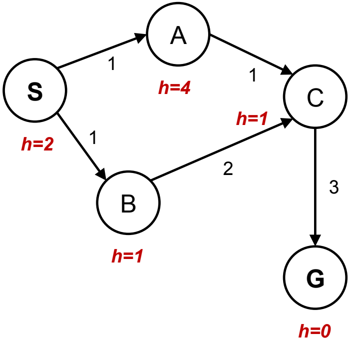
```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s', c ← result(s, a)
            if s' not visited then
                Mark s' as visited
                Q.insert(s, s', cost_from_start + h(s'))
            else
                Resolve duplicate s'
    return Failure
```

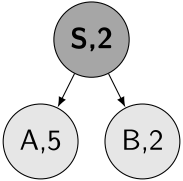




Consistent heuristic

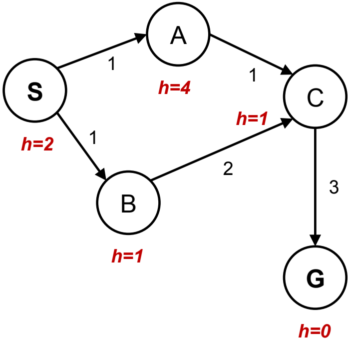
```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s′, c ← result(s, a)
            if s′ not visited then
                Mark s′ as visited
                Q.insert(s, s′, cost_from_start + h(s′))
            else
                Resolve duplicate s′
    return Failure
```

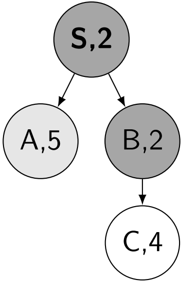




Consistent heuristic

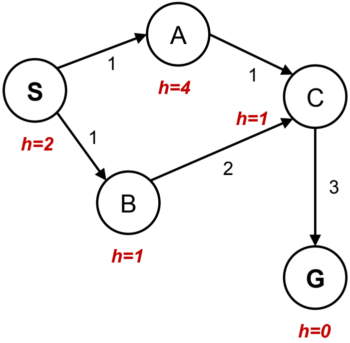
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$
      if s' not visited **then**
        Mark s' as visited
        $\text{Q.insert}(s, s', \text{cost\_from\_start} + h(s'))$
      else
        Resolve duplicate s'
  return Failure

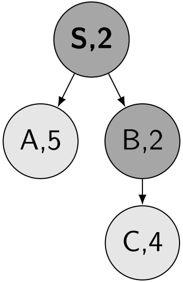




Consistent heuristic

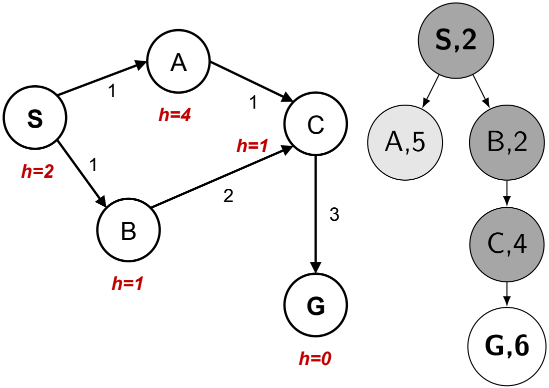
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$
      if s' not visited **then**
        Mark s' as visited
        Q.insert($s, s', \text{cost\_from\_start} + h(s')$)
      else
        Resolve duplicate s'
  return Failure





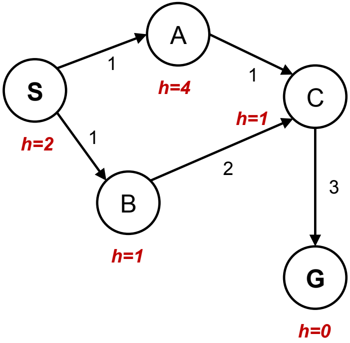
Consistent heuristic

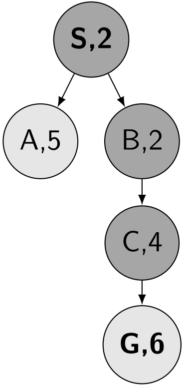
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$
      if s' not visited **then**
        Mark s' as visited
        $\text{Q.insert}(s, s', \text{cost\_from\_start} + h(s'))$
      else
        Resolve duplicate s'
  return Failure



Consistent heuristic

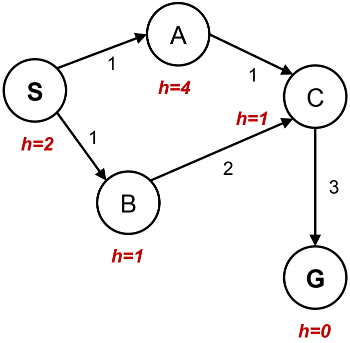
function FORWARD_SEARCH
  Q.insert(s_0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow s$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow \text{result}(s, a)$
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start $+ h(s')$)
      else
        Resolve duplicate s'
  return Failure

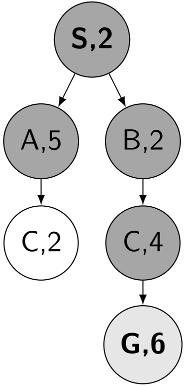




Consistent heuristic

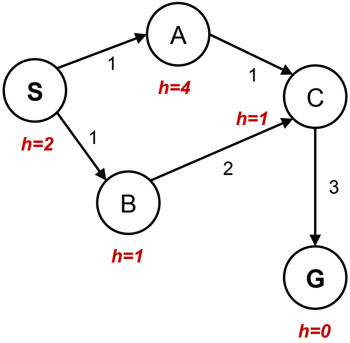
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty **do**
        $p, s, \_ \leftarrow$ Q.pop_first()
        parent[s] $\leftarrow s$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s', c \leftarrow \text{result}(s, a)$
            if s' not visited **then**
                Mark s' as visited
                Q.insert(s, s', cost_from_start $+ h(s')$)
            else
                Resolve duplicate s'
    return Failure

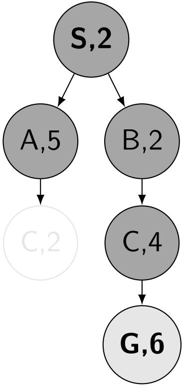




Consistent heuristic

```
function FORWARD_SEARCH
    Q.insert(s_0) and mark s_0 as visited
    while Q not empty do
        p, s, _ ← Q.pop_first()
        parent[s] ← s
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s′, c ← result(s, a)
            if s′ not visited then
                Mark s′ as visited
                Q.insert(s, s′, cost_from_start + h(s′))
            else
                Resolve duplicate s′
    return Failure
```





What would be the proper $h(A)$?

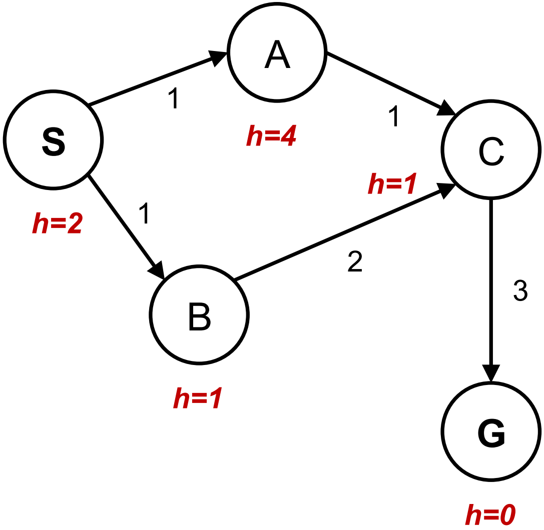
Consider other $h(s)$ fixed.

A: $h(A) = 1$

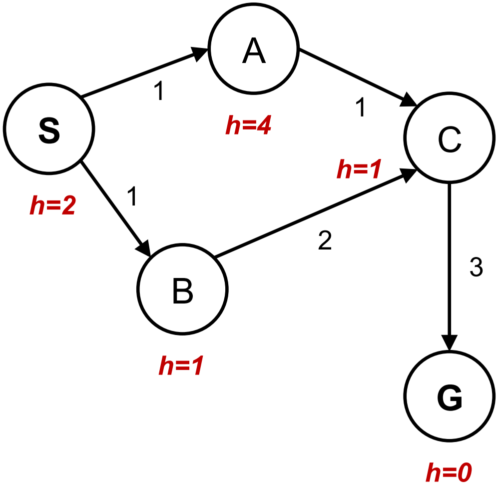
B: $h(A) = 2$

C: $1 \leq h(A) \leq 2$

D: $0 \leq h(A) \leq 1$



Consistent heuristics

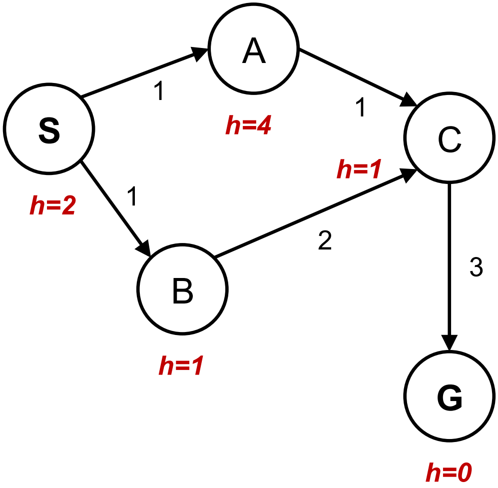


Admissible h:
$h(A) \leq$ true cost $A \to G$

Consistent h:
$h(A) - h(C) \leq$ true cost $A \to C$
in general:
$h(p) - h(s) \leq$ true cost $p \to s$ for any pair: parent p and its successor s

$f(n) = g(n) + h(n)$ along a path never decreases!

Consistent heuristics



Admissible h:

$h(A) \leq$ true cost $A \rightarrow G$

Consistent h:

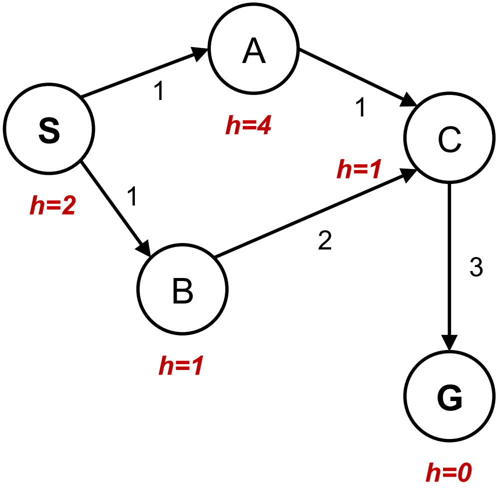
$h(A) - h(C) \leq$ true cost $A \rightarrow C$

in general:

$h(p) - h(s) \leq$ true cost $p \rightarrow s$ for any pair: parent p and its successor s

$f(n) = g(n) + h(n)$ along a path never decreases!

Consistent heuristics

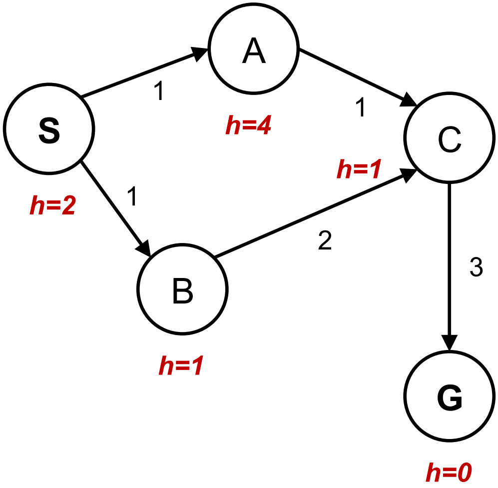


Admissible h:
$h(A) \leq$ true cost $A \rightarrow G$

Consistent h:
$h(A) - h(C) \leq$ true cost $A \rightarrow C$
in general:
$h(p) - h(s) \leq$ true cost $p \rightarrow s$ for any pair: parent p and its successor s

$f(n) = g(n) + h(n)$ along a path never decreases!

Consistent heuristics



Admissible h:
$h(A) \leq$ true cost $A \rightarrow G$

Consistent h:
$h(A) - h(C) \leq$ true cost $A \rightarrow C$
in general:
$h(p) - h(s) \leq$ true cost $p \rightarrow s$ for any pair: parent p and its successor s

$f(n) = g(n) + h(n)$ along a path never decreases!

Summary

- Effectivness – adding heuristic estimates of cost-to-go
- Not all heuristics are equally good (admissibility, consistence, informativeness)

References, further reading

Some figures from [2]. Chapter 2 in [1] provides a compact/dense intro into search algorithms.

[1] Steven M. LaValle.
Planning Algorithms.
Cambridge, 1st edition, 2006.
Online version available at: http://planning.cs.uiuc.edu.

[2] Stuart Russell and Peter Norvig.
Artificial Intelligence: A Modern Approach.
Prentice Hall, 4th edition, 2021.
http://aima.cs.berkeley.edu/.