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

Tomáš Svoboda, Petr Pošík

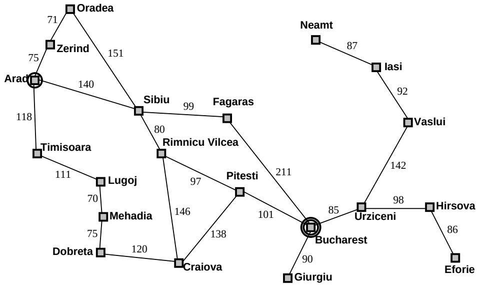
Vision for Robots and Autonomous Systems, Center for Machine Perception
Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University in Prague

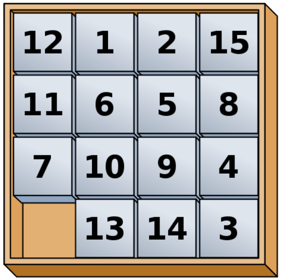
February 26, 2024

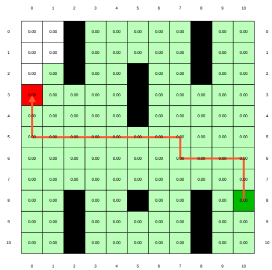
Notes

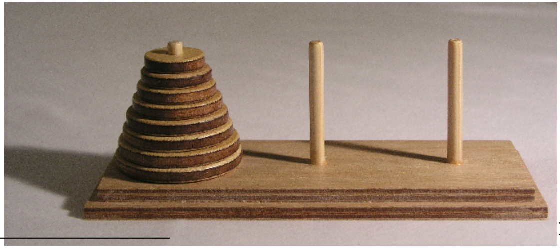
We will show that states/decisions/actions/control-commands are the same for deteriministic problems

Problems to solve



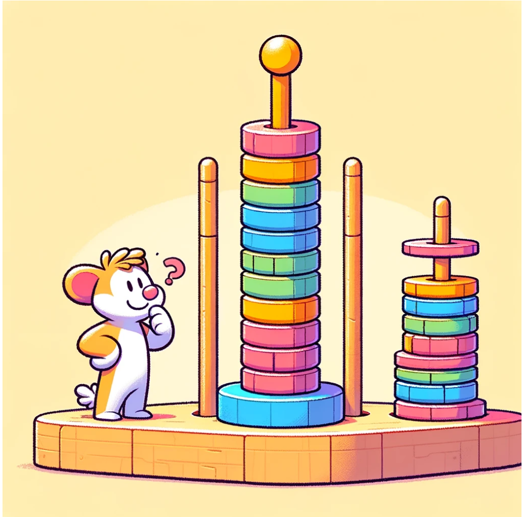




[1]

Notes

Understanding the problem is the key, DALL-E.



Notes

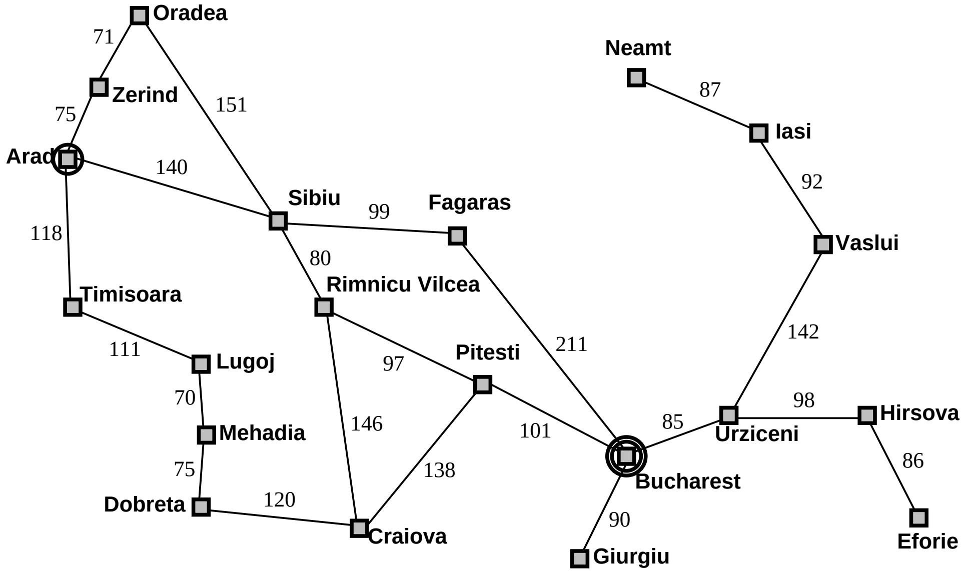
DALL-E creations after correctly explaing the problem itself.

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.

Notes

Example: Traveling in Romania



Notes

Ok, start with a simple one, almost everybody knows about the navigation - path planning problem. Waze, Garmin, . . . Here, the problem can be transferred into a graph quite directly - a map is a kind of a graph, states are location in a city.
Can you think about more problems?
For example:

- Touring problems. Special case: Traveling salesperson problem – each city must be visited exactly once.
- Planning robot movements – mobile robot or manipulator.
- VLSI (chip) layout.
- ...

Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

actions (decisions): select a road

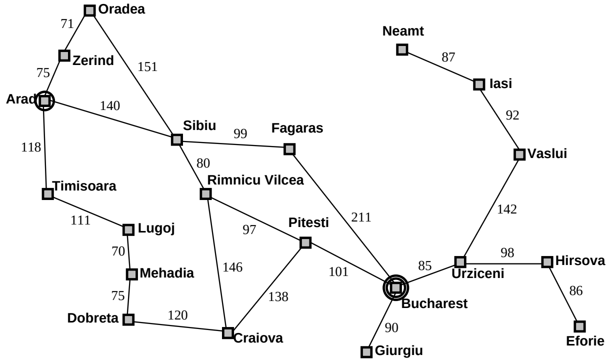
Solution:

Sequence of cities (path)

(sequence of actions/decisions [2])

Optimality – Cost, Loss, Utility, . . . :

Energy, time, tolls, . . .



Notes

Classical problem from the Book [2], we use it, too.

states and actions will be frequently discussed in several lectures and algorithms. It is important to fully understand them. At crossings, we need to decide about the next road - this is the action. We assume that we reach the next crossing - result of the action.

Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

actions (decisions): select a road

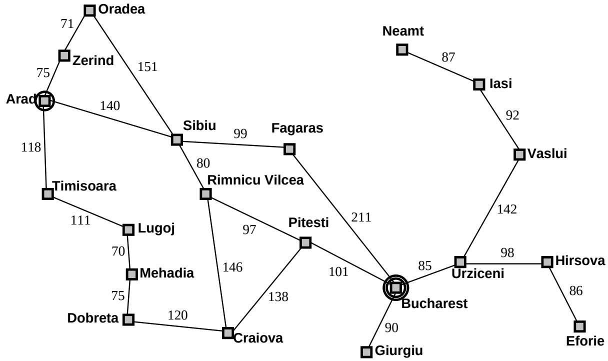
Solution:

Sequence of cities (path)

(sequence of actions/decisions [2])

Optimality – Cost, Loss, Utility, . . . :

Energy, time, tolls, . . .



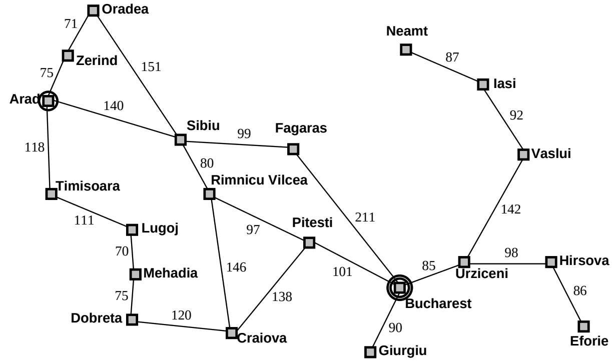
Notes

Classical problem from the Book [2], we use it, too.

states and actions will be frequently discussed in several lectures and algorithms. It is important to fully understand them. At crossings, we need to decide about the next road - this is the action. We assume that we reach the next crossing - result of the action.

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)
  (sequence of actions/decisions [2])
Optimality – Cost, Loss, Utility, . . . :
  Energy, time, tolls, . . .



Notes

Classical problem from the Book [2], we use it, too.

states and actions will be frequently discussed in several lectures and algorithms. It is important to fully understand them. At crossings, we need to decide about the next road - this is the action. We assume that we reach the next crossing - result of the action.

Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

actions (decisions): select a road

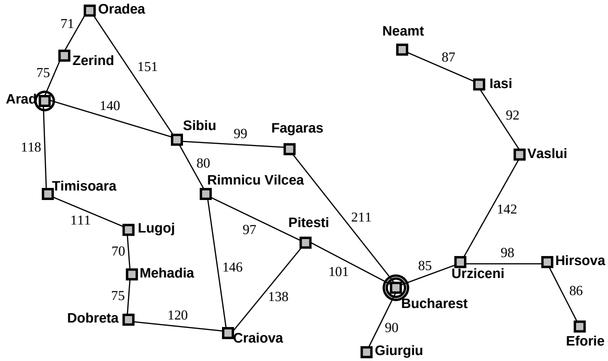
Solution:

Sequence of cities (path)

(sequence of actions/decisions [2])

Optimality – Cost, Loss, Utility, . . . :

Energy, time, tolls, . . .



Notes

Classical problem from the Book [2], we use it, too.

states and actions will be frequently discussed in several lectures and algorithms. It is important to fully understand them. At crossings, we need to decide about the next road - this is the action. We assume that we reach the next crossing - result of the action.

Traveling Example: State and Actions

Goal:

be in Bucharest

Problem formulation:

states: position in a city (cities)

actions (decisions): select a road

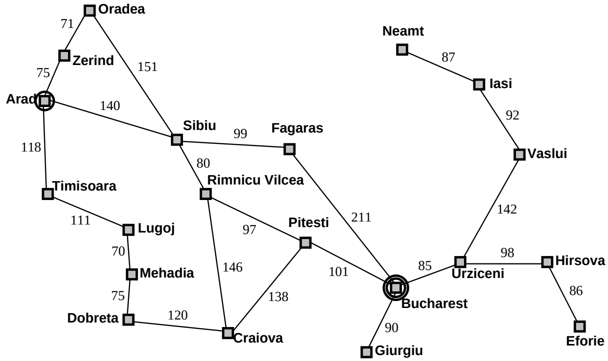
Solution:

Sequence of cities (path)

(sequence of actions/decisions [2])

Optimality – Cost, Loss, Utility, . . . :

Energy, time, tolls, . . .



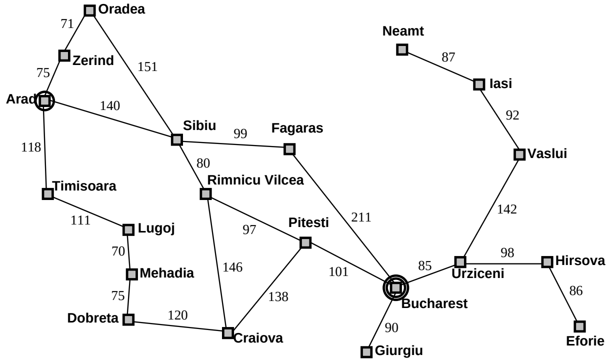
Notes

Classical problem from the Book [2], we use it, too.

states and actions will be frequently discussed in several lectures and algorithms. It is important to fully understand them. At crossings, we need to decide about the next road - this is the action. We assume that we reach the next crossing - result of the action.

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)
  (sequence of actions/decisions [2])
Optimality – Cost, Loss, Utility, . . . :
  Energy, time, tolls, . . .

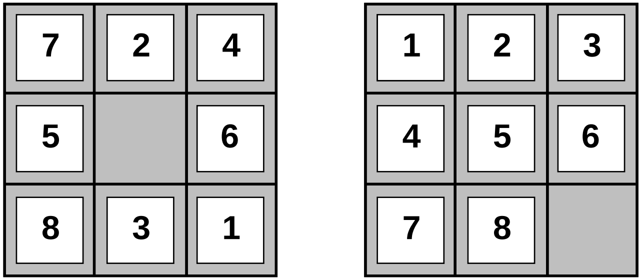


Notes

Classical problem from the Book [2], we use it, too.

states and actions will be frequently discussed in several lectures and algorithms. It is important to fully understand them. At crossings, we need to decide about the next road - this is the action. We assume that we reach the next crossing - result of the action.

Example: The 8-puzzle



Start State **Goal State**

states?
actions?
solution?
cost?

Notes

Also known as $n - 1$ puzzle.

- States: Location of each of the 8 tiles and the blank.
- Number of states: 9!
- Initial state: any state. (Note that any given goal state can be reached from exactly half of the initial states.)
- Actions: Movements of the blank space: Left, Right, Up, Down (or a subset of these)
- Solution / goal test: Check whether state matches the goal configuration.
- Path cost: nr. steps in the path (each step costs 1)

Toy problem (3.2.1) from [2].

A Search Problem

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

Notes

We will use the terminology throught the next 5-6 lectures; also for Markov (Sequential) Decision Processes, Reinforcement Learning.

Make a mental test: You are a robot, going from home to school. What would be states, actions, transition model, goal test?

Transition model can be also understood as a mapping between actions and results/outcome.

A Search Problem

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

Notes

We will use the terminology throught the next 5-6 lectures; also for Markov (Sequential) Decision Processes, Reinforcement Learning.

Make a mental test: You are a robot, going from home to school. What would be states, actions, transition model, goal test?

Transition model can be also understood as a mapping between actions and results/outcome.

A Search Problem

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

Notes

We will use the terminology throught the next 5-6 lectures; also for Markov (Sequential) Decision Processes, Reinforcement Learning.
Make a mental test: You are a robot, going from home to school. What would be states, actions, transition model, goal test?

Transition model can be also understood as a mapping between actions and results/outcome.

A Search Problem

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

Notes

We will use the terminology throught the next 5-6 lectures; also for Markov (Sequential) Decision Processes, Reinforcement Learning.

Make a mental test: You are a robot, going from home to school. What would be states, actions, transition model, goal test?

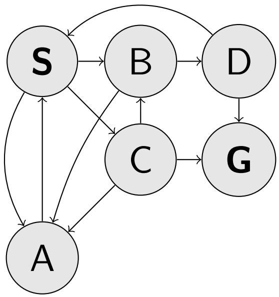
Transition model can be also understood as a mapping between actions and results/outcome.

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.

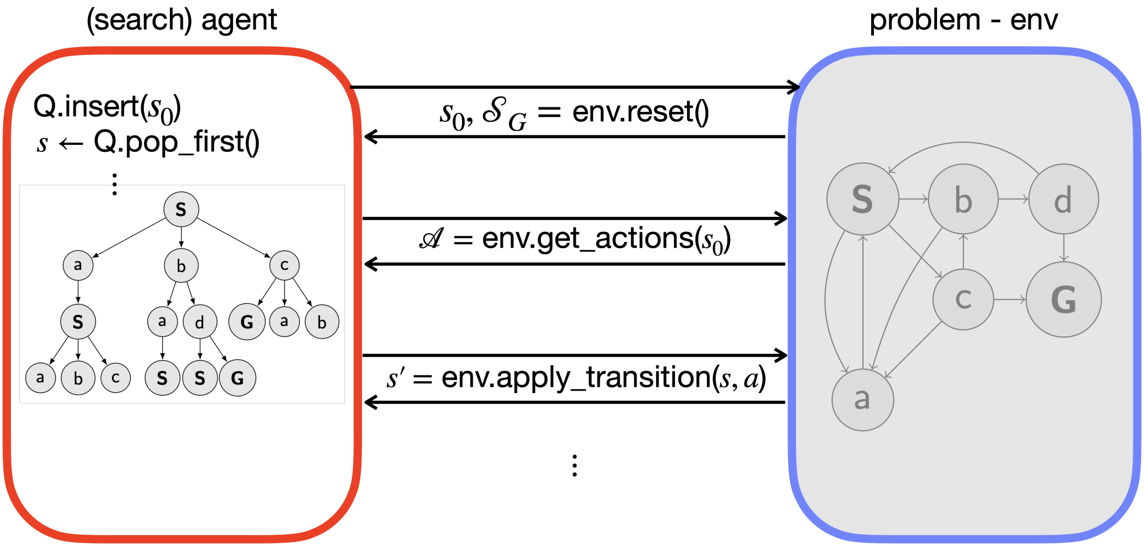


Notes

Formalizing a real world problem – (creating) a state space graph – could be a problem in itself. I put creating into brackets as it may be also infinite.
Close connection to graph algorithms like Dijkstra, Floyd-Warshall.

- Graph algorithms assume complete info about the graphs - the main input.
- For many real-world problems, the graph is not known in advance.
- The state space graph is *revealed during the search*. The graph serves as an abstraction - mental model - rather than as an actual data representation.
- Many real world problems have too many vertices, think about $n-1$ puzzle or chess, number of possible configurations is enormous.
- A solution can be actually quite shallow.

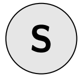
agent – problem dialog (a programmer's viewpoint)



Notes

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()
        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**

Notes

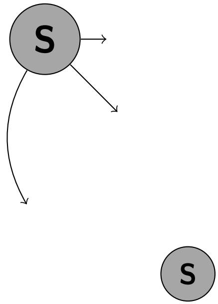
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
    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**

Notes

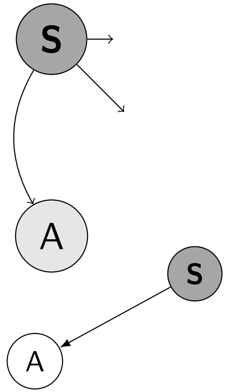
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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**

Notes

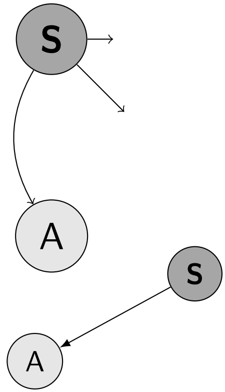
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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)
visited: **S** A

Notes

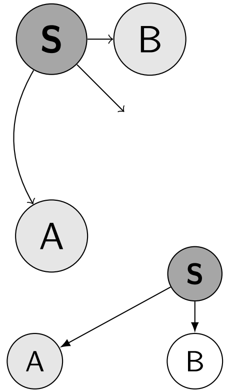
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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)
visited: **S** A

Notes

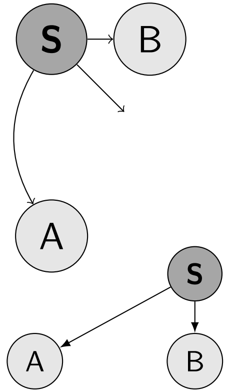
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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

Notes

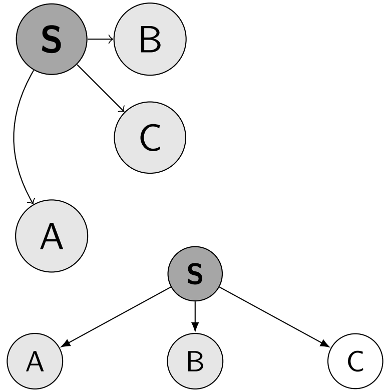
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
    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

Notes

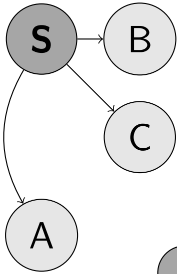
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

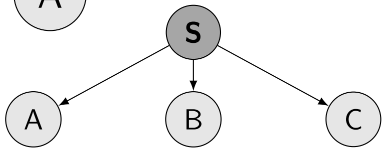
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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

Notes

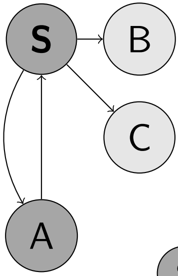
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

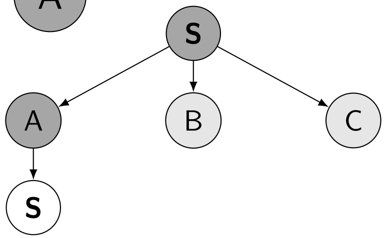
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
    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

Notes

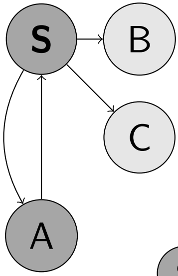
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

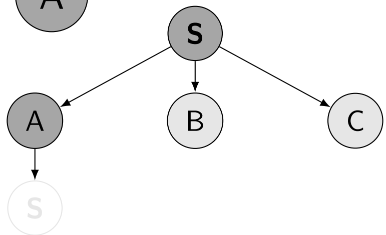
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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

Notes

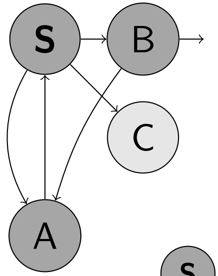
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

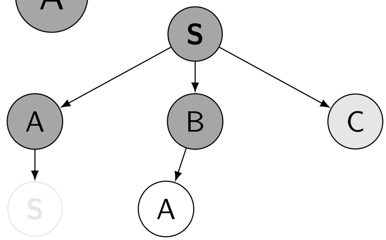
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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

Notes

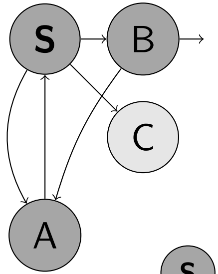
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

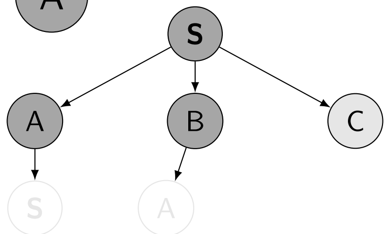
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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

Notes

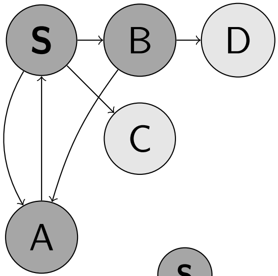
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

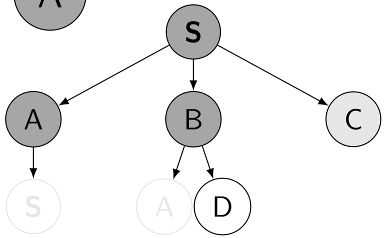
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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

Notes

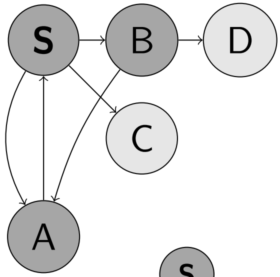
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

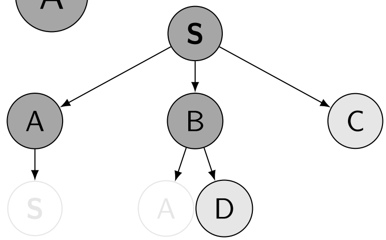
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
    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

Notes

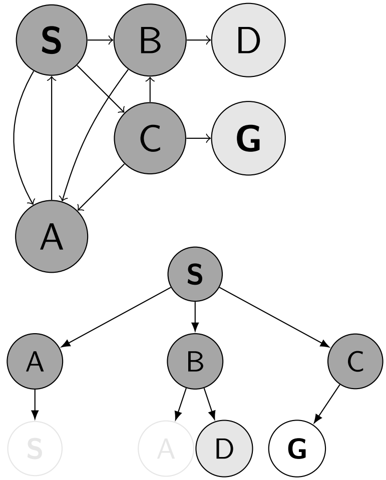
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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

Notes

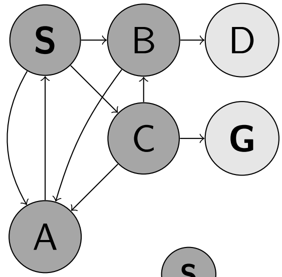
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

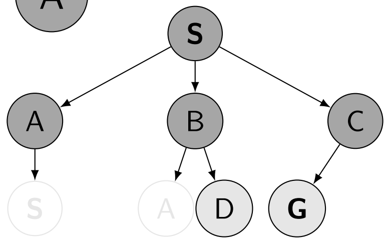
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
    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**

Notes

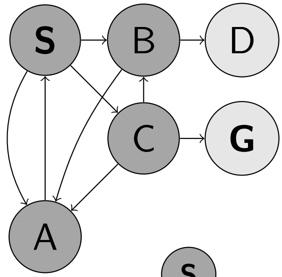
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

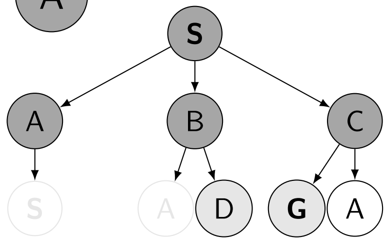
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
    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**

Notes

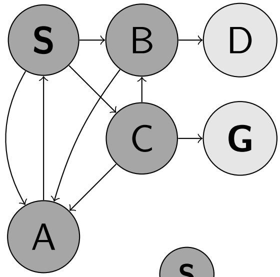
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

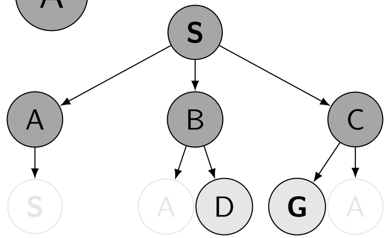
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
    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**

Notes

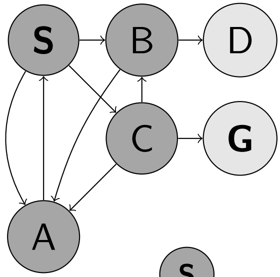
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

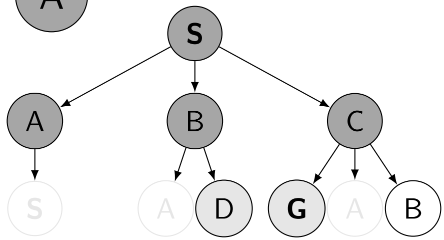
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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**

Notes

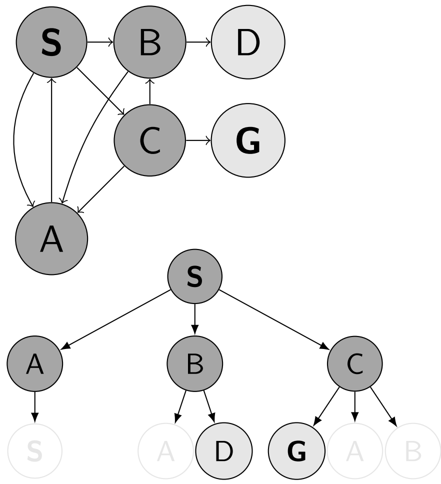
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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**

Notes

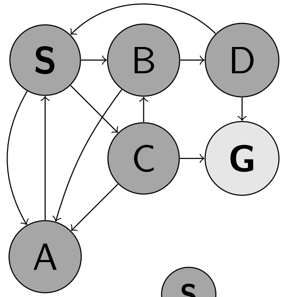
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

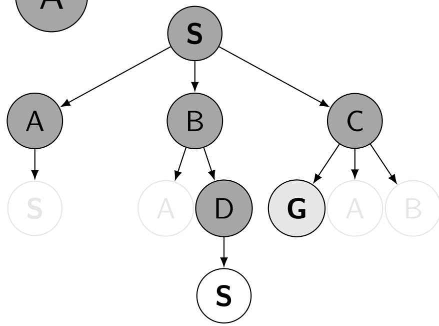
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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**

Notes

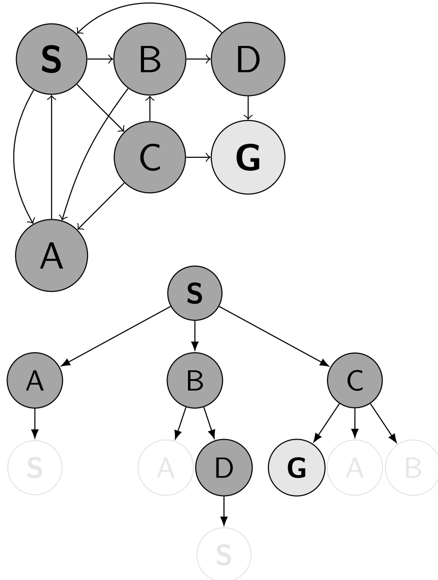
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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**

Notes

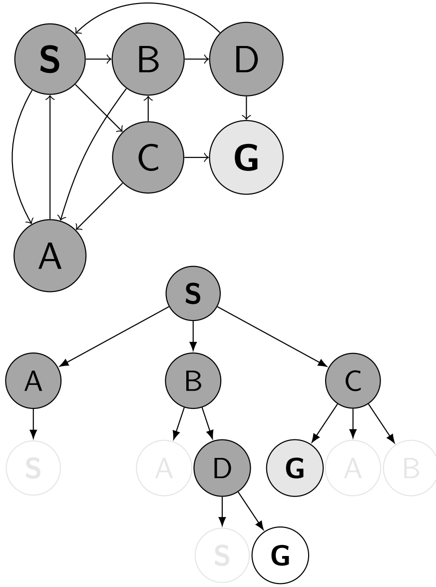
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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**

Notes

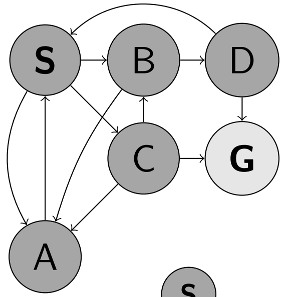
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

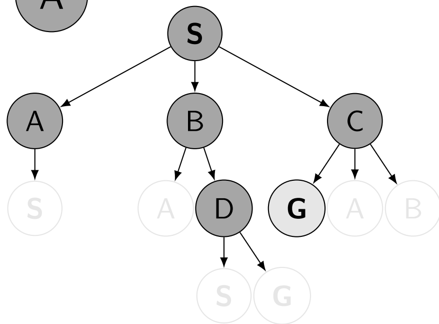
This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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**

Notes

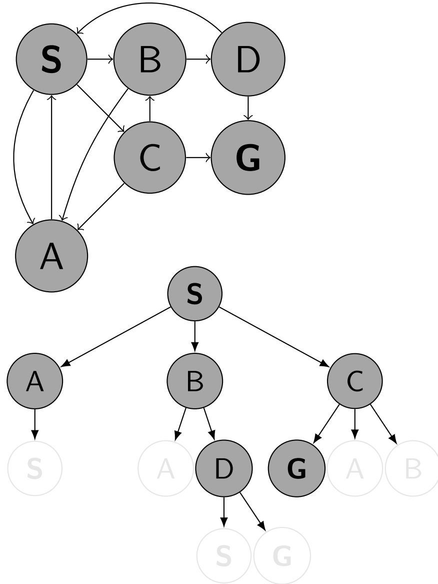
- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

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()
        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** A B C D **G**

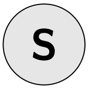
Notes

- What does the Q.pop() function/method do?
- Do we need to resolve duplicates somehow? If not, why?
- Could we stop and report success earlier?
- Howe to create the path?

This is the key slide to understand the difference between graph problem and tree search.
Create the search tree by pencil, think about Q and visited (whatever it may be).
How would you name the data structure Q? What kind of data structure?
When building the search tree:

- white - result of transition
- gray - visited and inside Q
- dark gray - visited and explored (outside Q)
- (made) invisible - forgotten

Search algorithm partitions state space into 3 disjoint sets



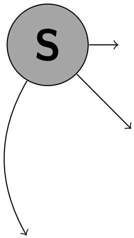
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



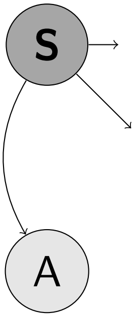
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search algorithm partitions state space into 3 disjoint sets



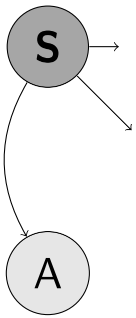
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



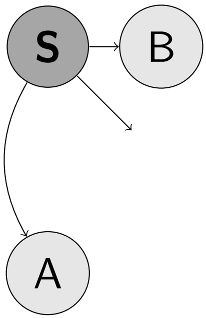
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search algorithm partitions state space into 3 disjoint sets



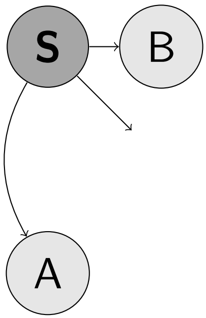
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search algorithm partitions state space into 3 disjoint sets



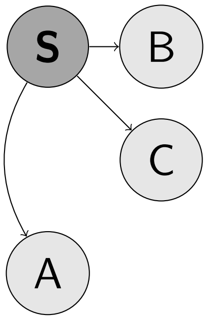
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



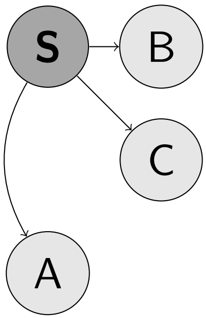
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search algorithm partitions state space into 3 disjoint sets



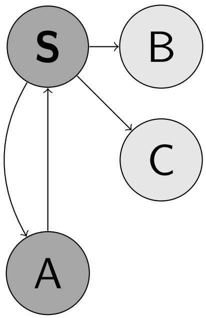
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



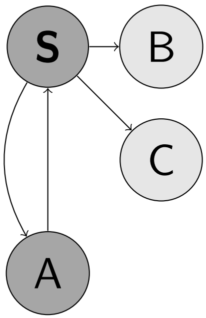
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search algorithm partitions state space into 3 disjoint sets



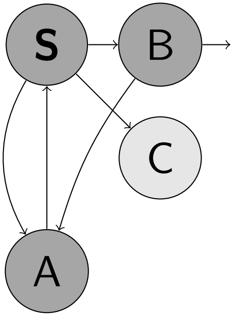
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



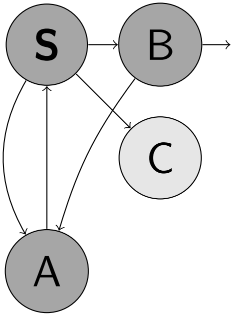
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



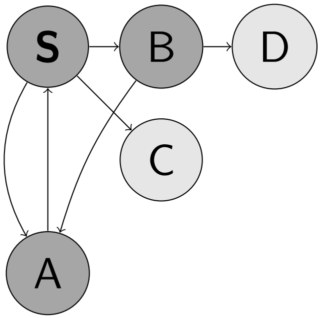
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search algorithm partitions state space into 3 disjoint sets



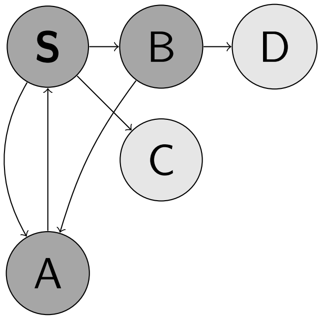
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



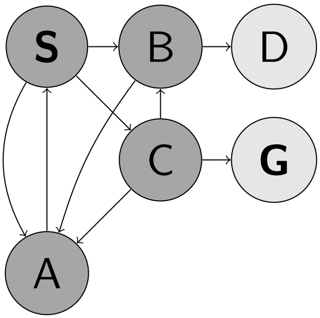
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



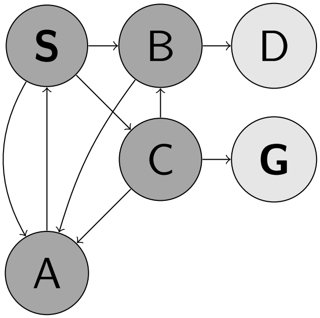
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



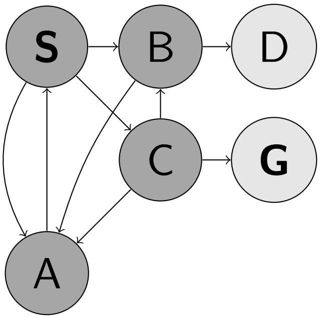
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



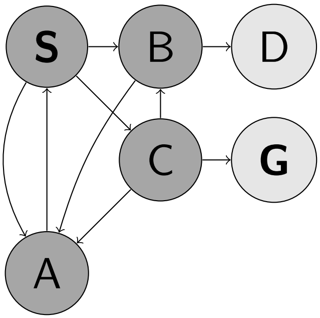
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search algorithm partitions state space into 3 disjoint sets



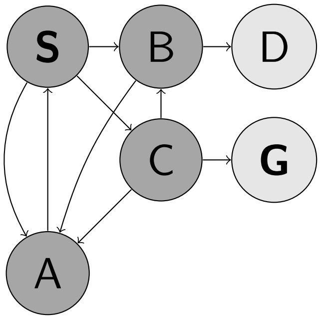
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



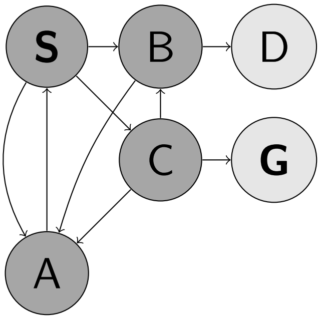
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



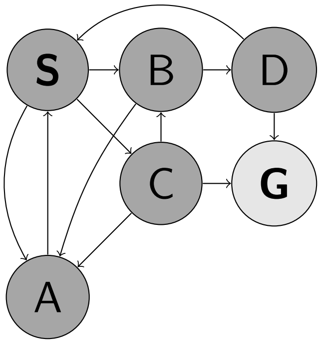
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



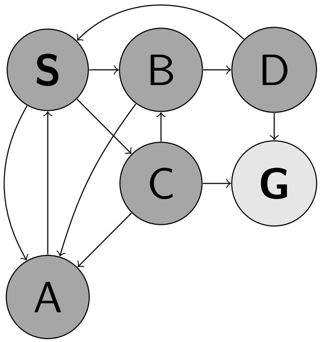
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search algorithm partitions state space into 3 disjoint sets



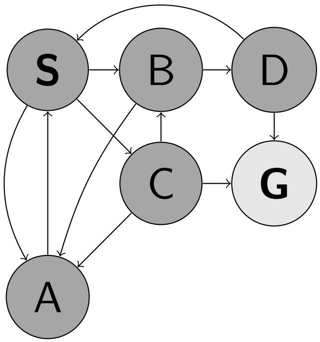
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search algorithm partitions state space into 3 disjoint sets



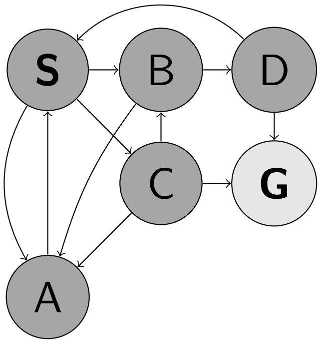
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



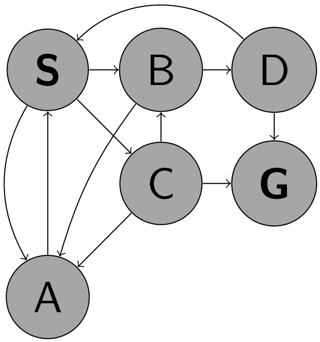
Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, . . .
- Dead, Closed, Explored, . . .
- Alive, Open, Frontier, . . .

Search algorithm partitions state space into 3 disjoint sets



Find good names

Notes

Few possible examples:

- Unknown, Unvisitied, ...
- Dead, Closed, Explored, ...
- Alive, Open, Frontier, ...

Search (algorithm) properties

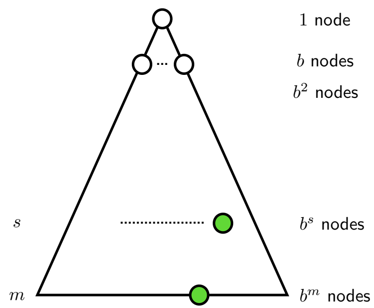
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

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



Search (algorithm) properties

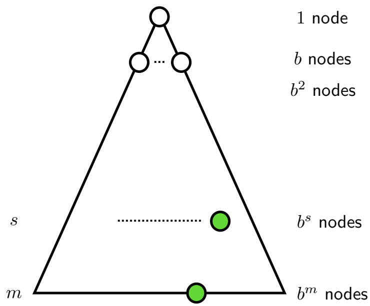
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

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



Search (algorithm) properties

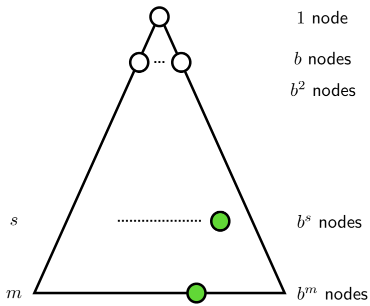
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

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



Search (algorithm) properties

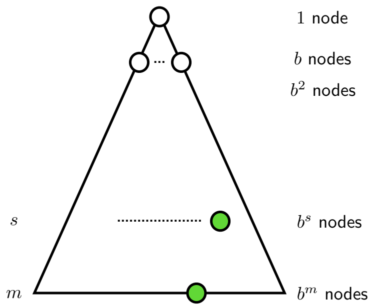
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

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



Search (algorithm) properties

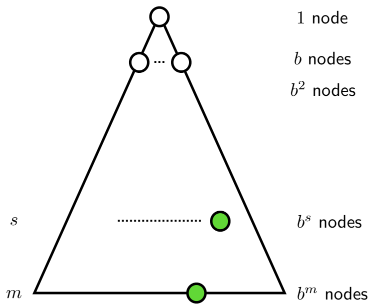
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

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



Search (algorithm) properties

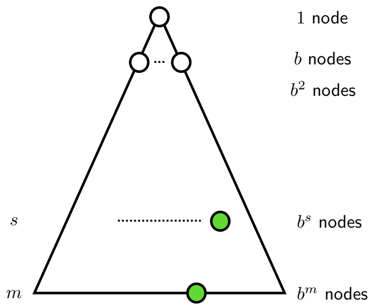
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

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



Search (algorithm) properties

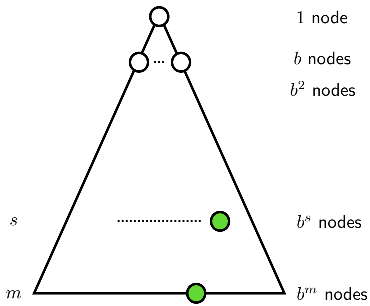
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

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



Search (algorithm) properties

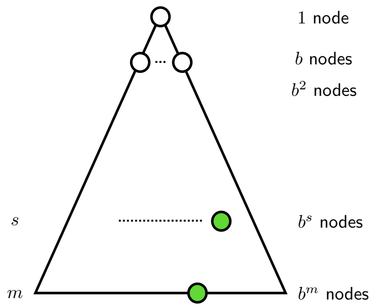
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

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



Search (algorithm) properties

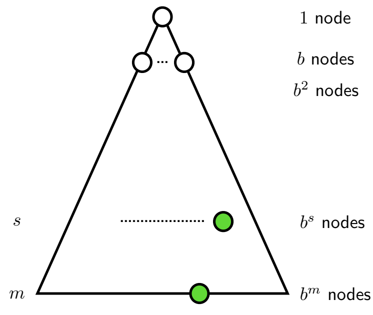
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

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



Search (algorithm) properties

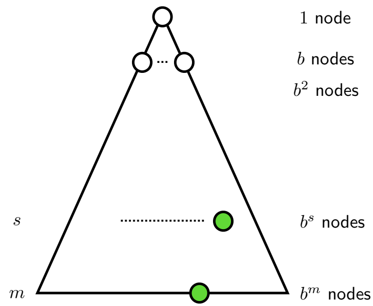
- Guaranteed to find a solution (if exists)? Complete?
- Guaranteed to find the least cost path? Optimal?
- How many steps - an operation with a node? Time complexity?
- How many nodes to remember? Space/Memory complexity?

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

Notes

Draw a (symbolic–think about a triangle) sketch of a (search) tree. It may grow upwards or downwards. How would you characterize/parametrize *size* of a tree.

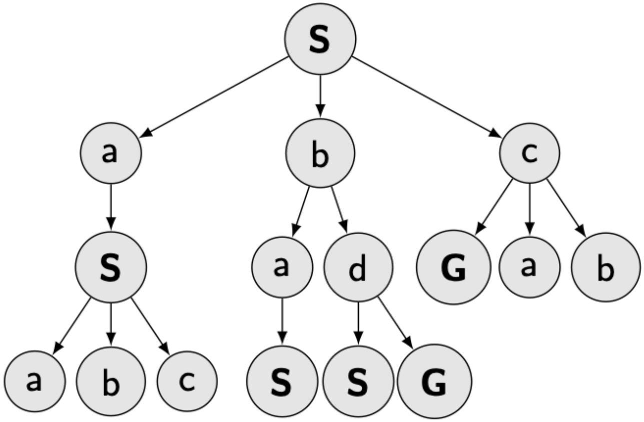
- Depth *d* of a node in the tree.
- Max-Depth of the tree *m*. Can be ∞.
- (Averege) Branching factor *b*.
- *s* denotes the depth of the shallowest Goal.
- How many nodes in the whole tree?



Strategies

How to traverse/build a search tree?

- Depth d of a node in the tree.
- 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?



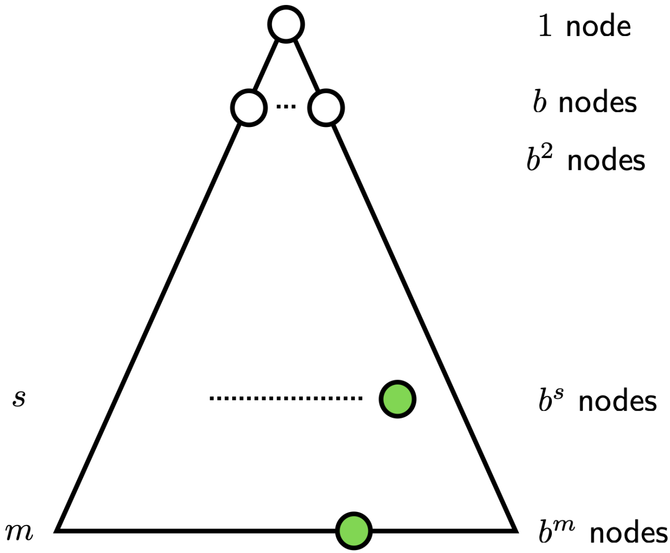
Notes

It is perhaps worth to remember that the search tree is built as the algorithm goes. Or better said, the tree is a human friendly representation of the machine run. Even small graphs (problems) may result in a large tree - depending on the search algorithm.

Strategies

How to traverse/build a search tree?

- Depth d of a node in the tree.
- 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?



Notes

It is perhaps worth to remember that the search tree is built as the algorithm goes. Or better said, the tree is a human friendly representation of the machine run. Even small graphs (problems) may result in a large tree - depending on the search algorithm.

BFS properties

Complete?

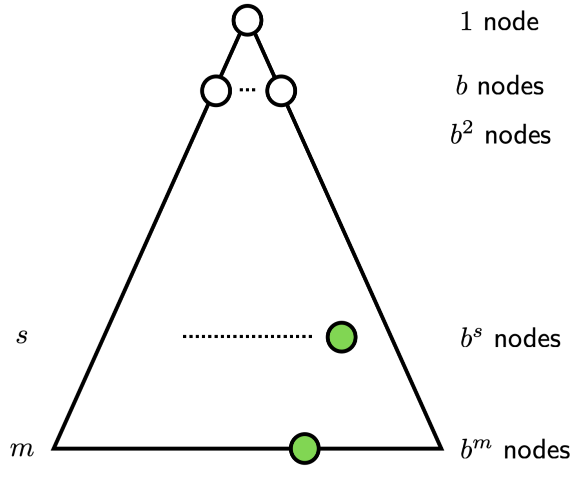
Optimal?

Time complexity?

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

Space complexity?

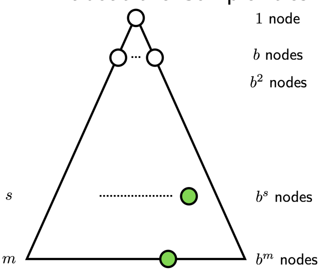
- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



Notes

- Time, can process the whole tree until s: b^s, well actually $b + b^2 + b^3 + \cdots + b^s$ but the last layer vastly dominates. Try some calculations for various b.
- Space, all the frontier: b^s
- Completness: Yes!
- Optimality, it does not miss the shallowest solution, hence if all the transition costs are 1: Yes!

Think about the Complexities in terms of $|\mathcal{S}|$ and $|\mathcal{A}|$ (Graph theory: vertices, edges)



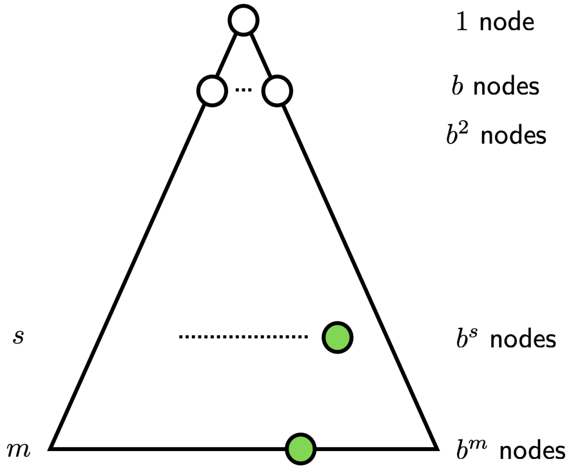
BFS properties

Complete?
Optimal?
Time complexity?

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

Space complexity?

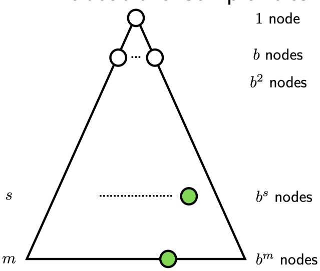
- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



Notes

- Time, can process the whole tree until s: b^s, well actually $b + b^2 + b^3 + \cdots + b^s$ but the last layer vastly dominates. Try some calculations for various b.
- Space, all the frontier: b^s
- Completness: Yes!
- Optimality, it does not miss the shallowest solution, hence if all the transition costs are 1: Yes!

Think about the Complexities in terms of $|\mathcal{S}|$ and $|\mathcal{A}|$ (Graph theory: vertices, edges)



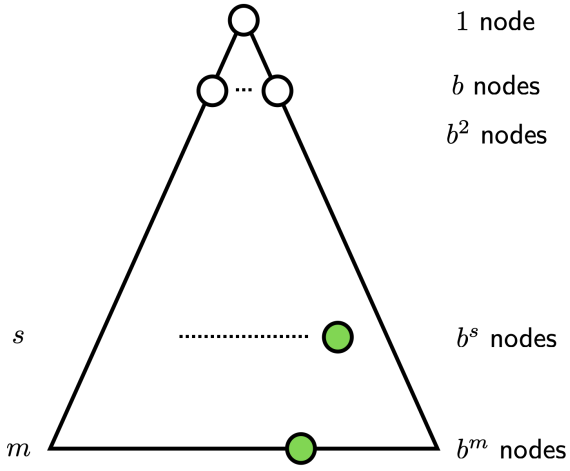
BFS properties

Complete?
Optimal?
Time complexity?

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

Space complexity?

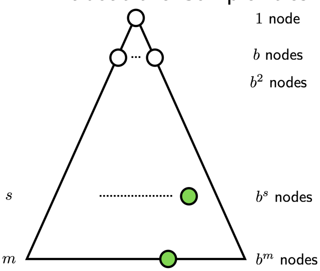
- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



Notes

- Time, can process the whole tree until s: b^s, well actually $b + b^2 + b^3 + \cdots + b^s$ but the last layer vastly dominates. Try some calculations for various b.
- Space, all the frontier: b^s
- Completness: Yes!
- Optimality, it does not miss the shallowest solution, hence if all the transition costs are 1: Yes!

Think about the Complexities in terms of $|\mathcal{S}|$ and $|\mathcal{A}|$ (Graph theory: vertices, edges)



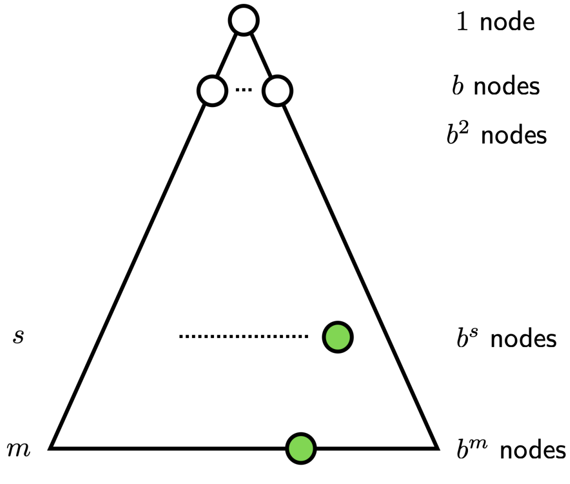
BFS properties

Complete?
Optimal?
Time complexity?

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

Space complexity?

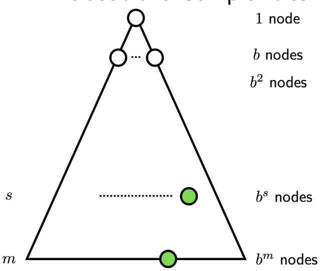
- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



Notes

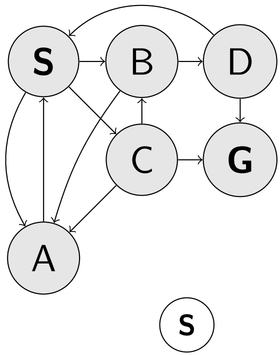
- Time, can process the whole tree until s: b^s, well actually $b + b^2 + b^3 + \cdots + b^s$ but the last layer vastly dominates. Try some calculations for various b.
- Space, all the frontier: b^s
- Completness: Yes!
- Optimality, it does not miss the shallowest solution, hence if all the transition costs are 1: Yes!

Think about the Complexities in terms of $|\mathcal{S}|$ and $|\mathcal{A}|$ (Graph theory: vertices, edges)



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()
        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
```



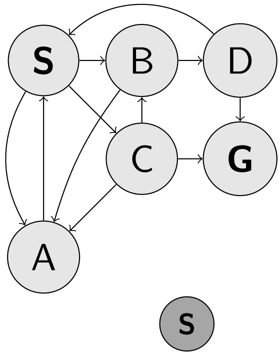
Q: (_,**S**)
visited: **S**

Notes

Do we need to resolve duplicates somehow? If not, why?

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()
    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



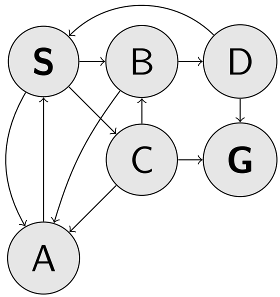
Q:
visited: **S**

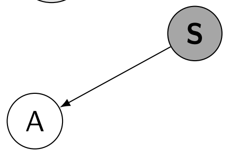
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
    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





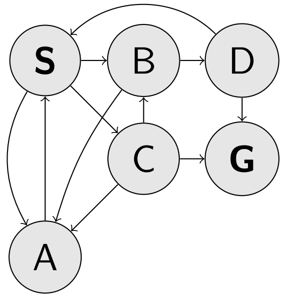
Q:
visited: **S**

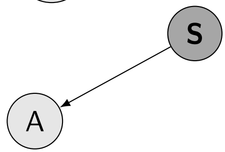
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
        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)

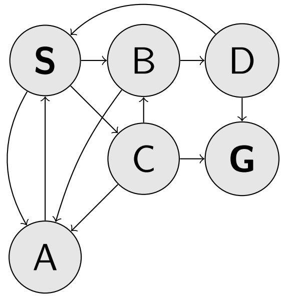
visited: **S** A

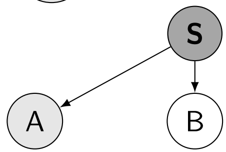
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
        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
```





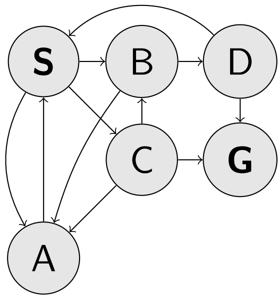
Q: (S,A)
visited: **S** A

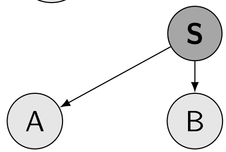
Notes

Do we need to resolve duplicates somehow? If not, why?

DFS, Q is LIFO data structure (stack)

```
function FORWARD_SEARCH
    Q.insert(_, s0) and mark s0 as visited
    while Q not empty do
        p, s ← Q.pop()
        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
```





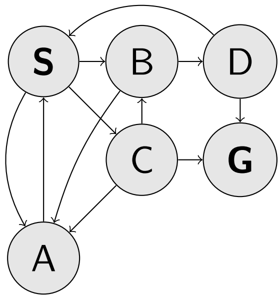
Q: (S,A) (S,B)
visited: **S** A B

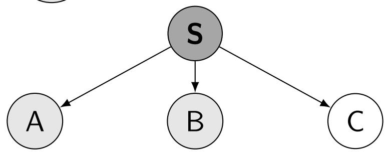
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
        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)

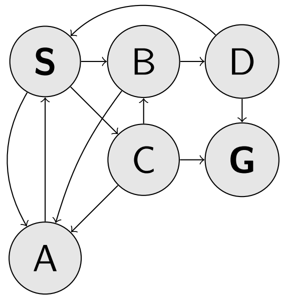
visited: **S** A B

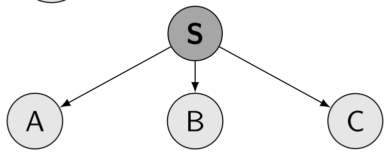
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
        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)

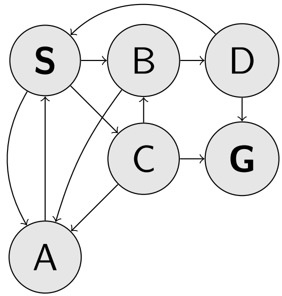
visited: **S** A B C

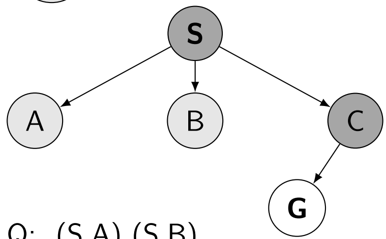
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
        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
```





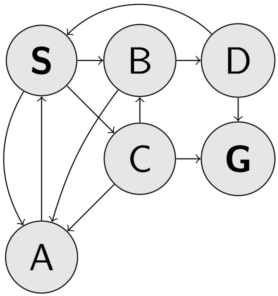
Q: (S,A) (S,B)
visited: **S** A B C

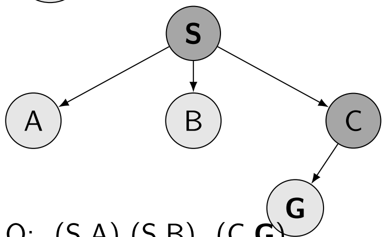
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
    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





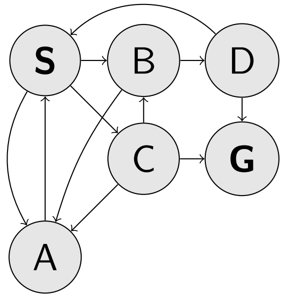
Q: (S,A) (S,B) (C,**G**)
visited: **S** A B C **G**

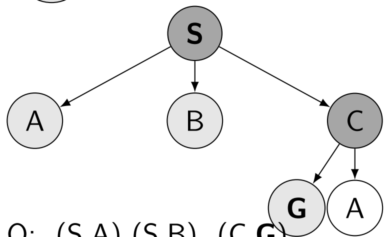
Notes

Do we need to resolve duplicates somehow? If not, why?

DFS, Q is LIFO data structure (stack)

```
function FORWARD_SEARCH
    Q.insert(_, s0) and mark s0 as visited
    while Q not empty do
        p, s ← Q.pop()
        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**)

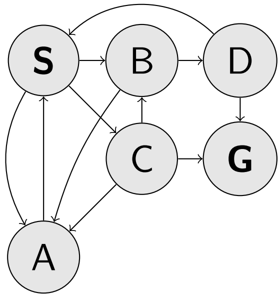
visited: **S** A B C **G**

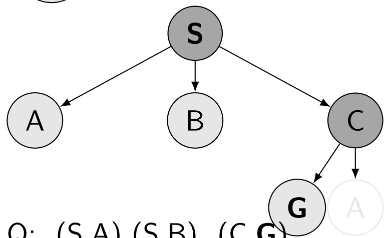
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
    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





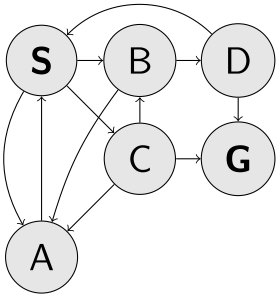
Q: (S,A) (S,B) (C,**G**)
visited: **S** A B C **G**

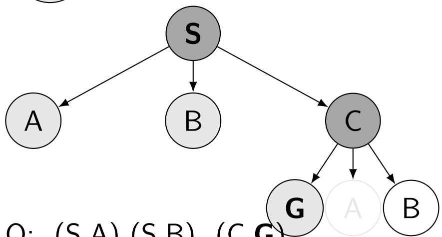
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
    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





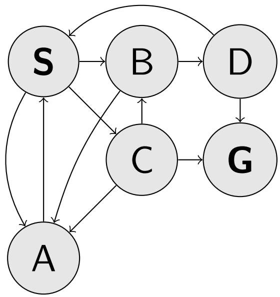
Q: (S,A) (S,B) (C,**G**)
visited: **S** A B C **G**

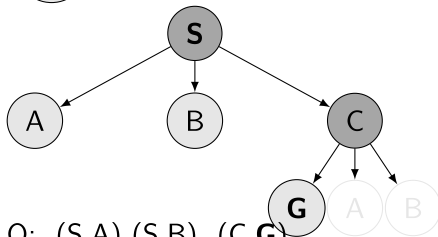
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
        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**)

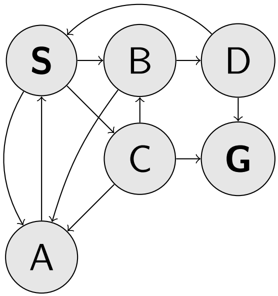
visited: **S** A B C **G**

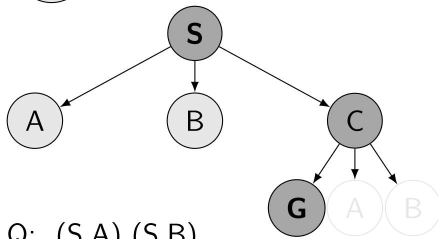
Notes

Do we need to resolve duplicates somehow? If not, why?

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()
        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 **G**

Notes

Do we need to resolve duplicates somehow? If not, why?

DFS properties

Complete?

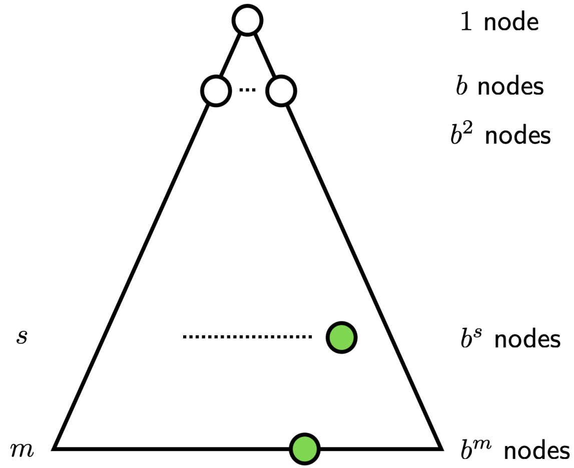
Optimal?

Time complexity?

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

Space complexity?

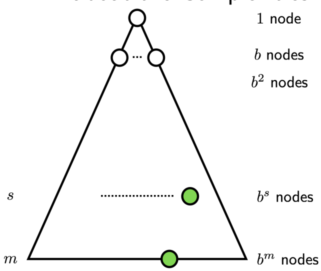
- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



Notes

- Time, can process the whole tree: b^m
- Space, only the path so far: bm (a path from root to leaf (m), plus siblings on the path are also on the frontier ($b \times m$))
- Completness: m may be ∞ hence, not in general
- Optimality: No! It just takes the first solution found.

Think about the Complexities in terms of $|\mathcal{S}|$ and $|\mathcal{A}|$ (Graph theory: vertices, edges)



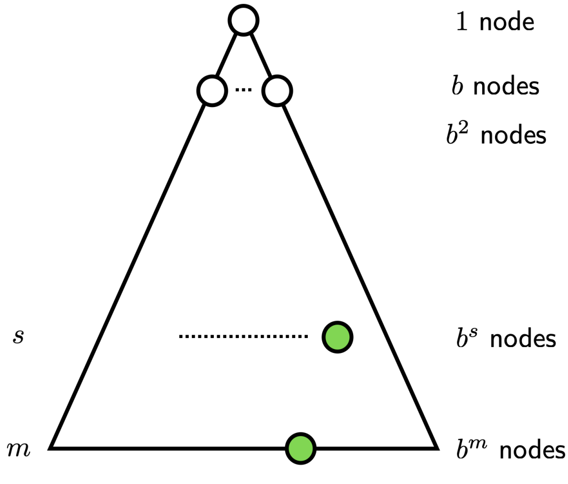
DFS properties

Complete?
Optimal?
Time complexity?

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

Space complexity?

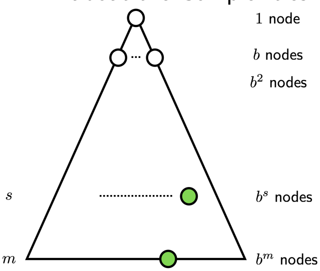
- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



Notes

- Time, can process the whole tree: b^m
- Space, only the path so far: bm (a path from root to leaf (m), plus siblings on the path are also on the frontier ($b \times m$))
- Completness: m may be ∞ hence, not in general
- Optimality: No! It just takes the first solution found.

Think about the Complexities in terms of $|\mathcal{S}|$ and $|\mathcal{A}|$ (Graph theory: vertices, edges)



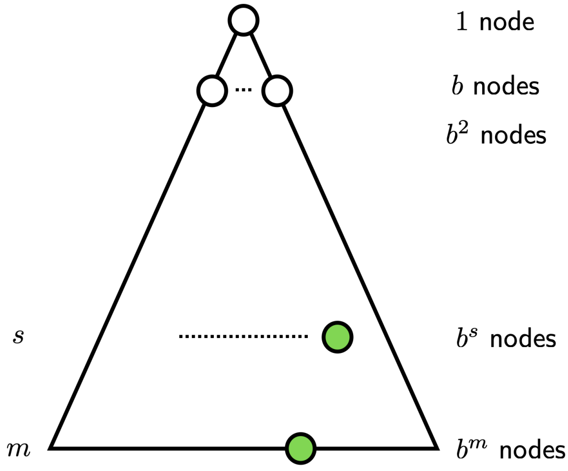
DFS properties

Complete?
Optimal?
Time complexity?

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

Space complexity?

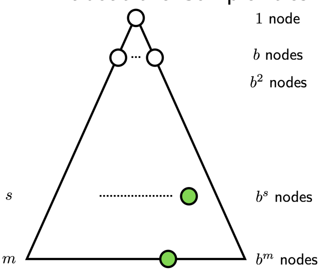
A $\mathcal{O}(bm)$
B $\mathcal{O}(b^m)$
C $\mathcal{O}(m^b)$
D $\mathcal{O}(b^s)$



Notes

- Time, can process the whole tree: b^m
- Space, only the path so far: bm (a path from root to leaf (m), plus siblings on the path are also on the frontier ($b \times m$))
- Completness: m may be ∞ hence, not in general
- Optimality: No! It just takes the first solution found.

Think about the Complexities in terms of $|\mathcal{S}|$ and $|\mathcal{A}|$ (Graph theory: vertices, edges)



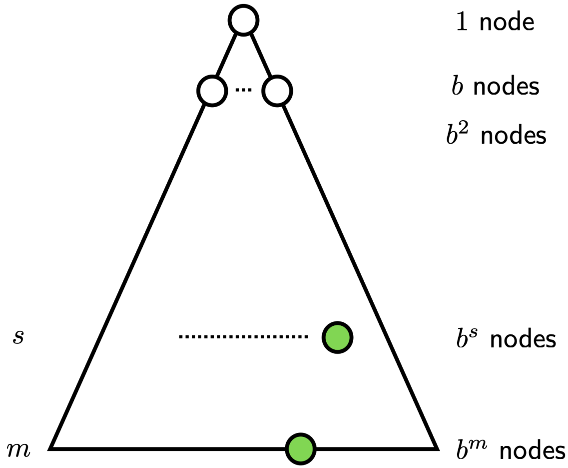
DFS properties

Complete?
Optimal?
Time complexity?

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

Space complexity?

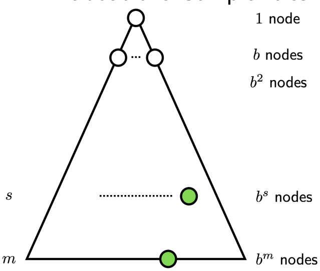
- **A** $\mathcal{O}(bm)$
- **B** $\mathcal{O}(b^m)$
- **C** $\mathcal{O}(m^b)$
- **D** $\mathcal{O}(b^s)$



Notes

- Time, can process the whole tree: b^m
- Space, only the path so far: bm (a path from root to leaf (m), plus siblings on the path are also on the frontier ($b \times m$))
- Completness: m may be ∞ hence, not in general
- Optimality: No! It just takes the first solution found.

Think about the Complexities in terms of $|\mathcal{S}|$ and $|\mathcal{A}|$ (Graph theory: vertices, edges)



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?

Notes

Really, how much do we repeat/waste? The “upper levels”, close to the root, are repeated many times. However, in a tree, most nodes are the bottom levels and nr. nodes traversed is what counts. More specifically, for a solution at depth s, the nodes on the bottm level are generated only once, those on the next-to-bottom level 2x … children of the root are generated $s\times$. Compare the number of nodes generated ID-DFS vs. BFS:

$$N(\text{ID-DFS}) = (s)b + (s-1)b^2 + (s-2)b^3 + \cdots + (1)b^s$$

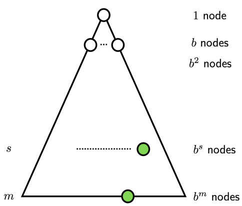
$$N(\text{BFS}) = b + b^2 + b^3 + \cdots + b^s$$

Try some calculations for various s and b. For $b = 10$ and $d = 5$:

$$N(\text{ID-DFS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$

(Example from [2].)



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?

Notes

Really, how much do we repeat/waste? The "upper levels", close to the root, are repeated many times. However, in a tree, most nodes are the bottom levels and nr. nodes traversed is what counts. More specifically, for a solution at depth s, the nodes on the bottm level are generated only once, those on the next-to-bottom level 2x ... children of the root are generated $s\times$. Compare the number of nodes generated ID-DFS vs. BFS:

$$N(\text{ID-DFS}) = (s)b + (s-1)b^2 + (s-2)b^3 + \cdots + (1)b^s$$

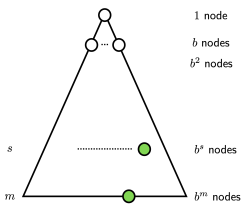
$$N(\text{BFS}) = b + b^2 + b^3 + \cdots + b^s$$

Try some calculations for various s and b. For $b = 10$ and $d = 5$:

$$N(\text{ID-DFS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$

(Example from [2].)



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?

Notes

Really, how much do we repeat/waste? The "upper levels", close to the root, are repeated many times. However, in a tree, most nodes are the bottom levels and nr. nodes traversed is what counts. More specifically, for a solution at depth s, the nodes on the bottm level are generated only once, those on the next-to-bottom level 2x … children of the root are generated $s\times$. Compare the number of nodes generated ID-DFS vs. BFS:

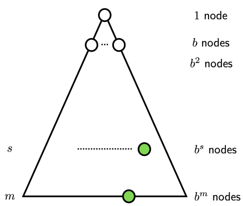
$$N(\text{ID-DFS}) = (s)b + (s-1)b^2 + (s-2)b^3 + \cdots + (1)b^s$$

$$N(\text{BFS}) = b + b^2 + b^3 + \cdots + b^s$$

Try some calculations for various s and b. For $b = 10$ and $d = 5$:

$$N(\text{ID-DFS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$



(Example from [2].)

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?

Notes

Really, how much do we repeat/waste? The "upper levels", close to the root, are repeated many times. However, in a tree, most nodes are the bottom levels and nr. nodes traversed is what counts. More specifically, for a solution at depth s, the nodes on the bottm level are generated only once, those on the next-to-bottom level 2x ... children of the root are generated $s\times$. Compare the number of nodes generated ID-DFS vs. BFS:

$$N(\text{ID-DFS}) = (s)b + (s-1)b^2 + (s-2)b^3 + \cdots + (1)b^s$$

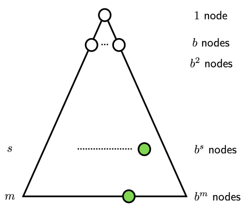
$$N(\text{BFS}) = b + b^2 + b^3 + \cdots + b^s$$

Try some calculations for various s and b. For $b = 10$ and $d = 5$:

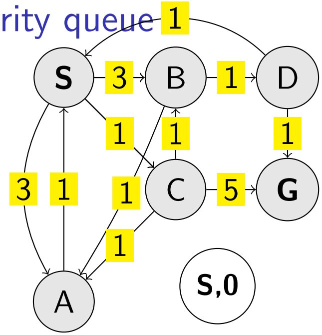
$$N(\text{ID-DFS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 = 111110$$

(Example from [2].)



Uniform Cost Search (Dijkstra), Q is priority queue




```
function FORWARD_SEARCH
    Q.insert(_, s_0, 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', 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
```

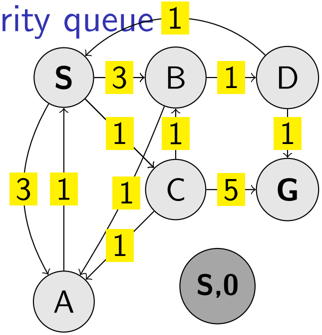
Q: (_,**S**,0)
visited: **S**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 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′, 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:

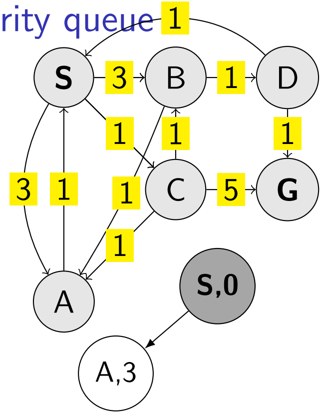
visited: **S**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(_, s_0, 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**
      $s', c \leftarrow$ 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



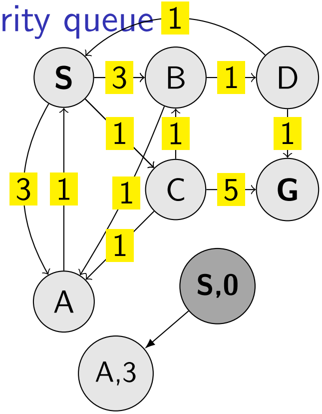
Q:
visited: **S**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert($\_, s_0, 0$) 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**
      $s', c \leftarrow$ 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



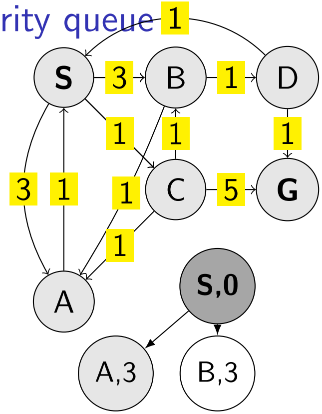
Q: (S,A,3)
visited: **S** A

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert($\_, s_0, 0$) 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**
      $s', c \leftarrow$ 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



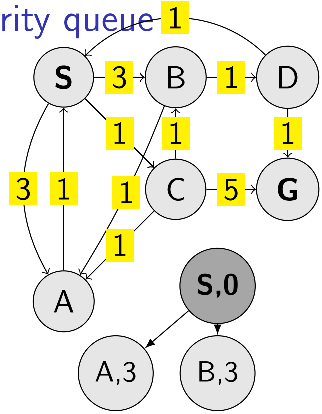
Q: (S,A,3)
visited: **S** A

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 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′, 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)

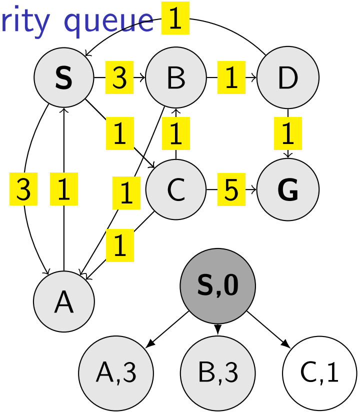
visited: **S** A B

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert($\_, s_0, 0$) 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**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



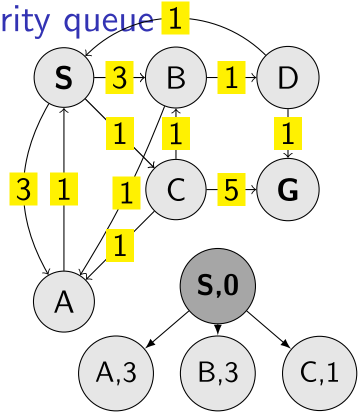
Q: (S,A,3) (S,B,3)
visited: **S** A B

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
 Q.insert($\_, s_0, 0$) 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**
   $s', c \leftarrow$ 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

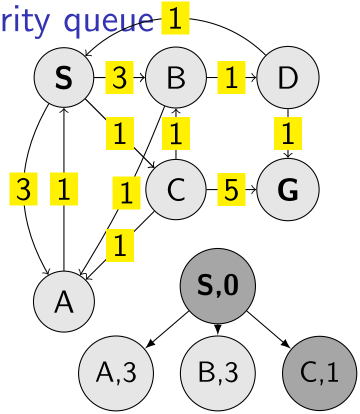


Q: (S,C,1) (S,A,3) (S,B,3)
visited: **S** A B C

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue



function FORWARD_SEARCH
    Q.insert($\_, s_0, 0$) 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**
            $s', c \leftarrow$ 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

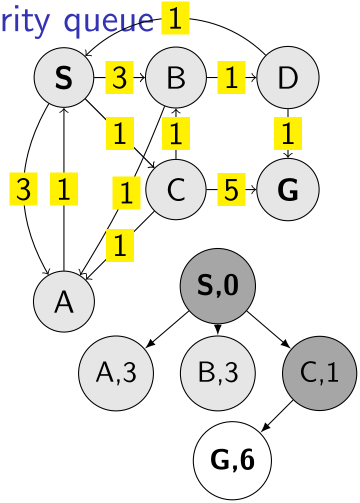
Q: (S,A,3) (S,B,3)
visited: **S** A B C

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert($\_, s_0, 0$) 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**
      $s', c \leftarrow$ 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



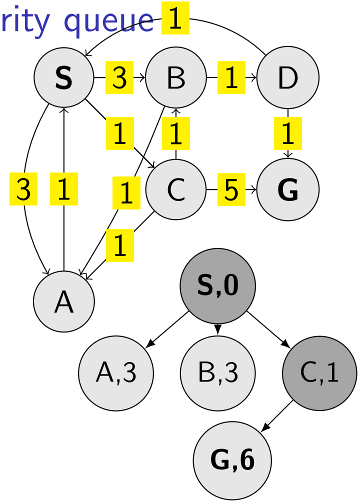
Q: (S,A,3) (S,B,3)
visited: **S** A B C

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(_, s_0, 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**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



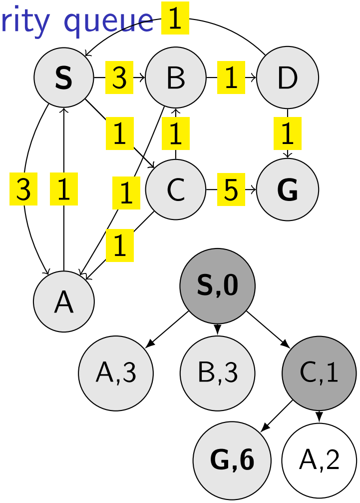
Q: (S,A,3) (S,B,3) (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(_, s_0, 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**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



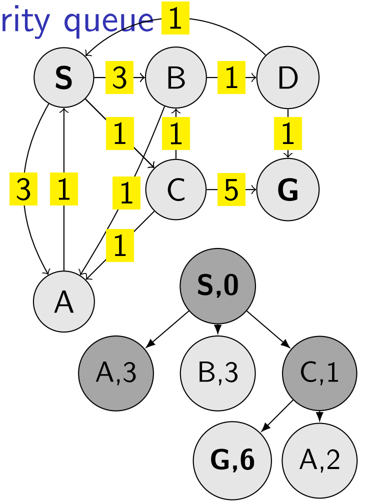
Q: (S,A,3) (S,B,3) (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 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', 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
```



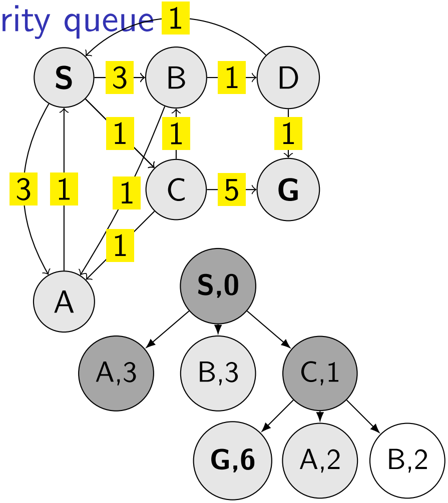
Q: (C,A,2) (S,B,3) (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(_ , s_0, 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**
      $s', c \leftarrow$ 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



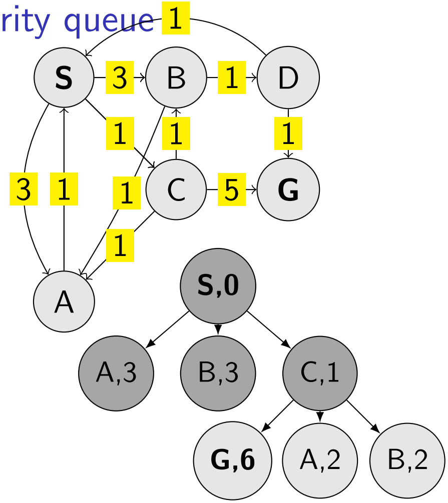
Q: (C,A,2) (S,B,3) (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 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', 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
```



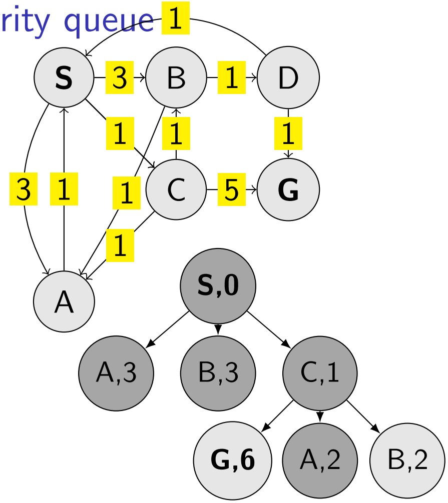
Q: (C,A,2) (C,B,2) (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert($\_, s_0, 0$) 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**
      $s', c \leftarrow \text{result}(s, a)$ ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



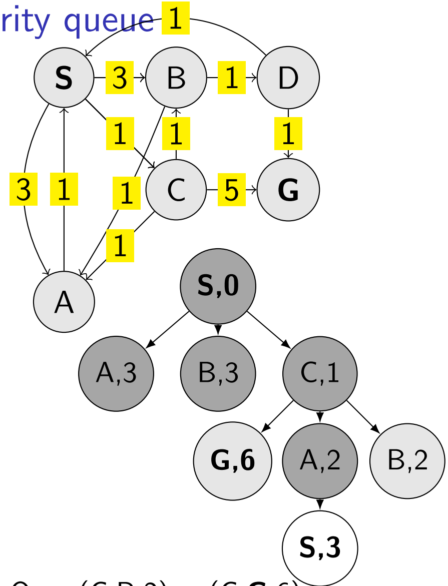
Q: (C,B,2) (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(_, s_0, 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**
      $s', c \leftarrow$ 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



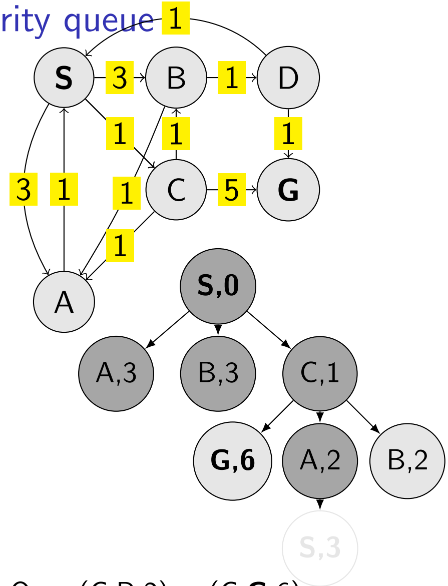
Q: (C,B,2) (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(_, s_0, 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**
      $s', c \leftarrow$ 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



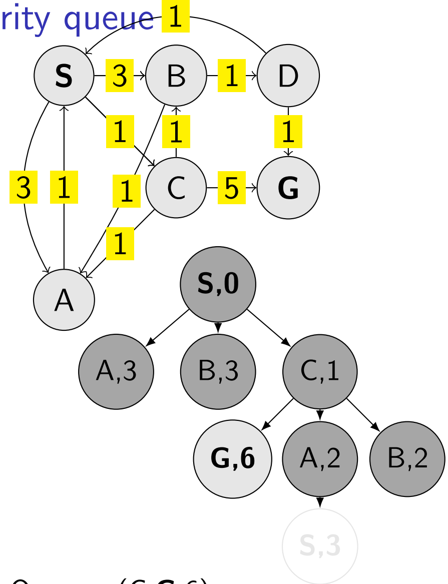
Q: (C,B,2) (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(_, s_0, 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**
      $s', c \leftarrow$ 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



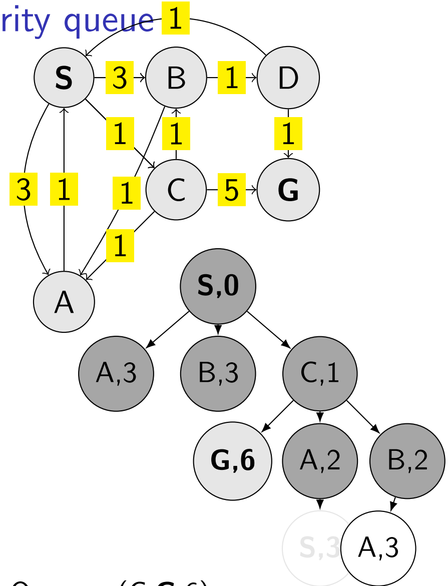
Q: (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(_, s_0, 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**
      $s', c \leftarrow$ 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



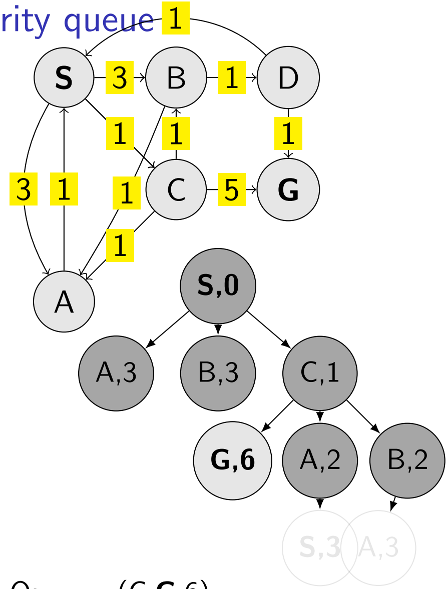
Q: (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert($\_, s_0, 0$) 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**
      $s', c \leftarrow$ 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



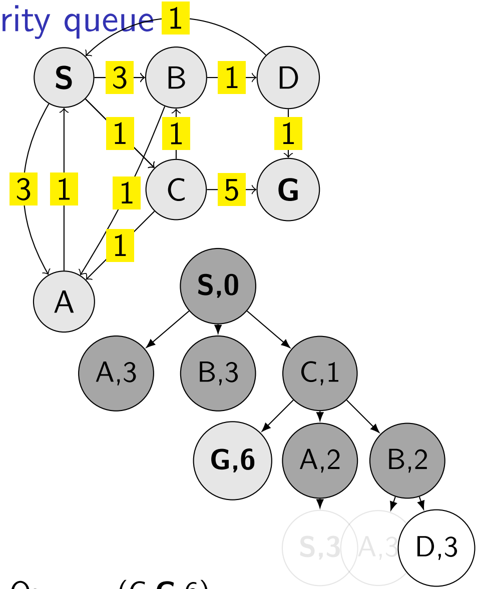
Q: (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert(_, s_0, 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**
      $s', c \leftarrow$ 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

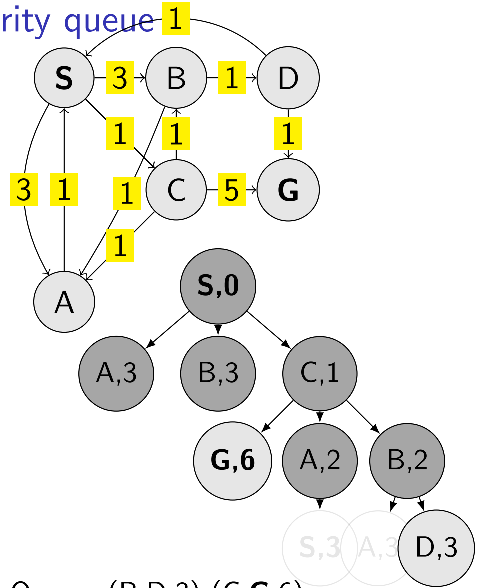


Q: (C,**G**,6)
visited: **S** A B C **G**

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue




```
function FORWARD_SEARCH
    Q.insert(_, s_0, 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′, 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
```

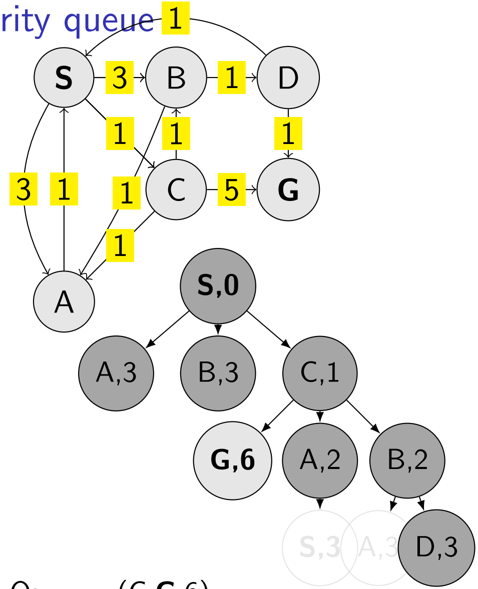
Q: (B,D,3) (C,**G**,6)
visited: **S** A B C **G** D

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 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', 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
```

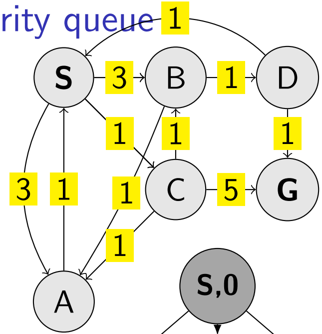


Q: (C,**G**,6)
visited: **S** A B C **G** D

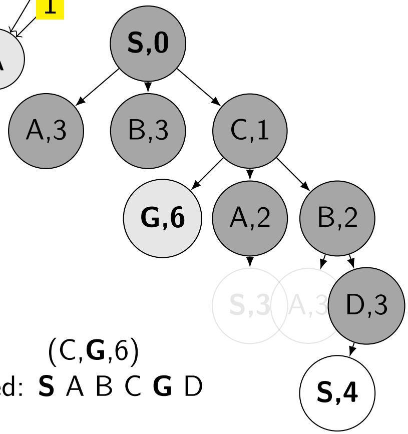
Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue



function FORWARD_SEARCH
    Q.insert(_, s_0, 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**
            $s', c \leftarrow$ 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



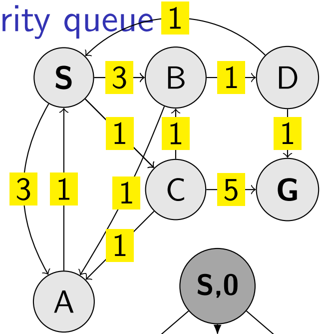
Q: (C,**G**,6)
visited: **S** A B C **G** D

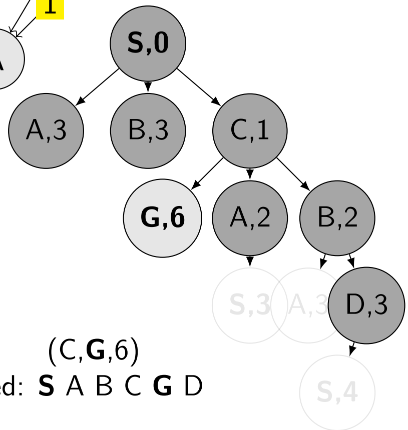
Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

function FORWARD_SEARCH
  Q.insert($\_, s_0, 0$) 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**
      $s', c \leftarrow$ 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





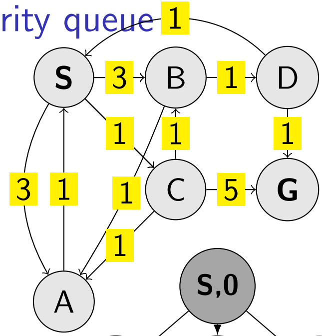
Q: (C,**G**,6)
visited: **S** A B C **G** D

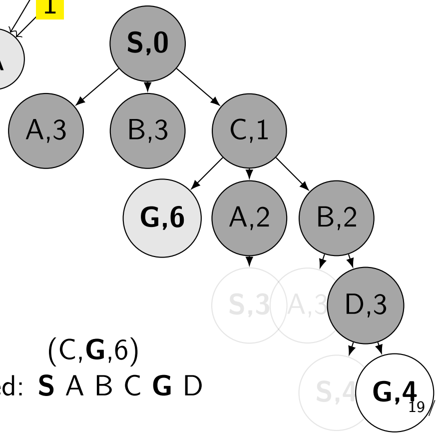
Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 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', 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
```





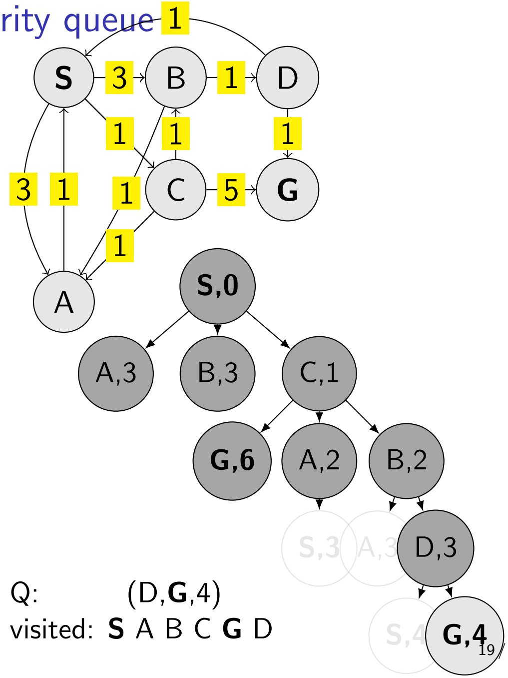
Q: (C,**G**,6)
visited: **S** A B C **G** D

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue

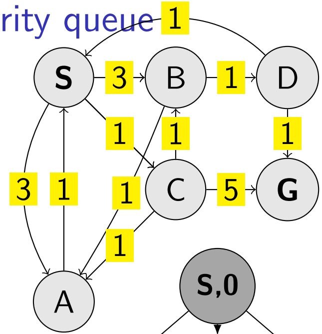
function FORWARD_SEARCH
  Q.insert(_, s_0, 0) and mark s_0 as visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop_first()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow$ result(s, a) ▷ c cost
      if s' not visited **then**
        Mark s' as visited
        Q.insert(s, s', cost_from_start)
      else
        Resolve duplicate s'
  return Failure



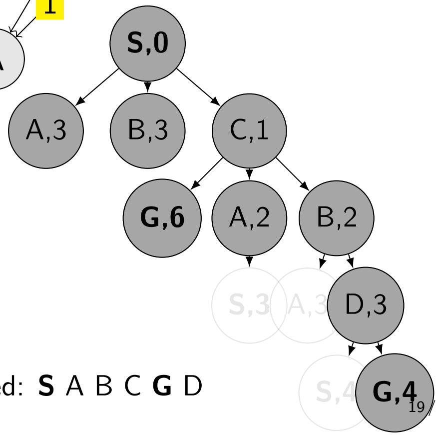
Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

Uniform Cost Search (Dijkstra), Q is priority queue



function FORWARD_SEARCH
  Q.insert(_, s_0, 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**
      $s', c \leftarrow$ 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

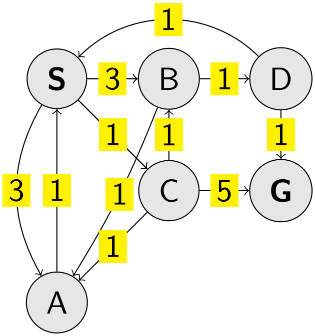


Q:
visited: **S** A B C **G** D

Notes

- Do we need to resolve duplicates somehow? If not, why?
- How is the cost_from_start computed?
- Why is it (sometimes) called Uniform cost search?

UCS properties



Complete?

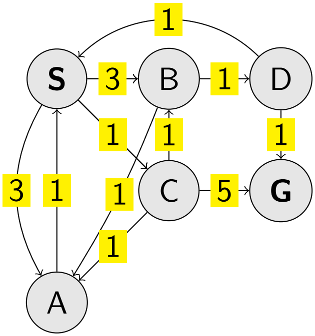
Optimal?

Complexities?

Notes

Parts of the (complete) search tree repeat, but with different costs

UCS properties

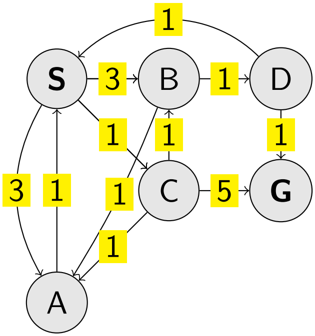


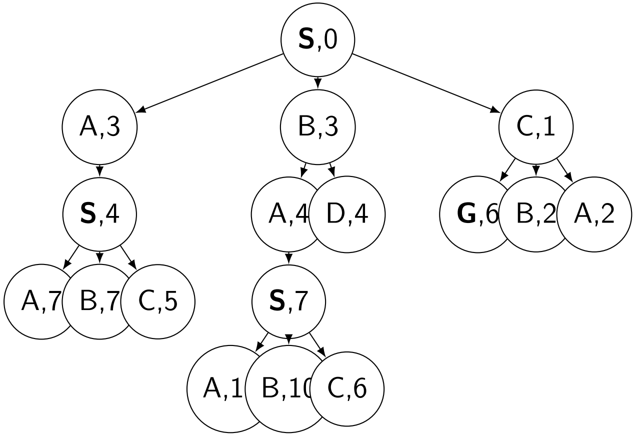
Complete?
Optimal?
Complexities?

Notes

Parts of the (complete) search tree repeat, but with different costs

UCS properties





Complete?
Optimal?
Complexities?

Notes

Parts of the (complete) search tree repeat, but with different costs

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

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

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

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 accumulated cost from start to n, cost_from_start .

Notes

- DFS: $f(n) = -n.depth$
- BFS: $f(n) = n.depth$
- UCS: $f(n) = n.path\_cost$

Do humans look back when planing path? Is looking back important at all? If yes, when?

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

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 accumulated cost from start to n, cost_from_start .

Notes

- DFS: $f(n) = -n.depth$
- BFS: $f(n) = n.depth$
- UCS: $f(n) = n.path\_cost$

Do humans look back when planing path? Is looking back important at all? If yes, when?

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

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 accumulated cost from start to n, cost_from_start .

Notes

- DFS: $f(n) = -n.depth$
- BFS: $f(n) = n.depth$
- UCS: $f(n) = n.path\_cost$

Do humans look back when planing path? Is looking back important at all? If yes, when?

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

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

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

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 accumulated cost from start to n, cost_from_start .

Notes

- DFS: $f(n) = -n.depth$
- BFS: $f(n) = n.depth$
- UCS: $f(n) = n.path\_cost$

Do humans look back when planing path? Is looking back important at all? If yes, when?

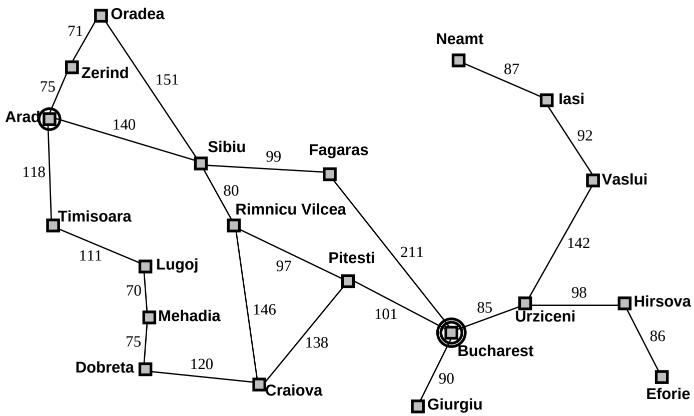
How far are we from the goal cost-to-go ? – Heuristics

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

Notes

What happens if $h(s) =$ true cost?

Example of heuristics



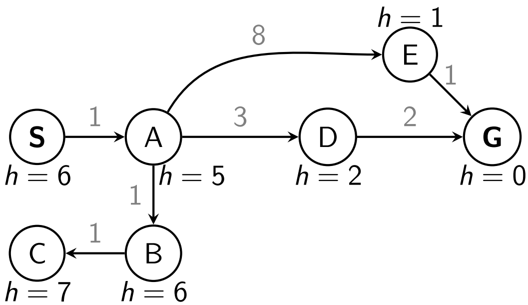
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Notes

Straight-line distance to Bucharest.

Illustration of *greedy* failing: Imagine going from Iasi to Fagaras. Neamt will be chosen for expansion. This will add Iasi back. Iasi is closer to Fagaras than Vaslui is and will be expanded again. Infinite loop... (3.5.1. in [**?**])

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



What is wrong (and nice) with the Greedy?

Notes

Also called "Greedy best-first search" [2].
What will happen in this example:

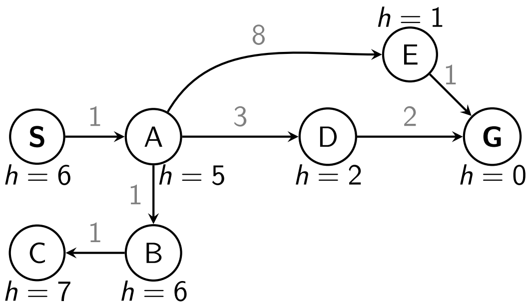
1. Expand "S". Add "A" to frontier.
2. Expand "A". Add "B","D","E".
3. Expand "E" ($h = 1$). Get "G".

Wrong:

- not optimal
- not complete (tree search version) (Can be shown on the Romania example – go back.)
- (graph search version is complete only in finite state spaces)

Nice: it is simple.

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



What is wrong (and nice) with the Greedy?

Notes

Also called "Greedy best-first search" [2].
What will happen in this example:

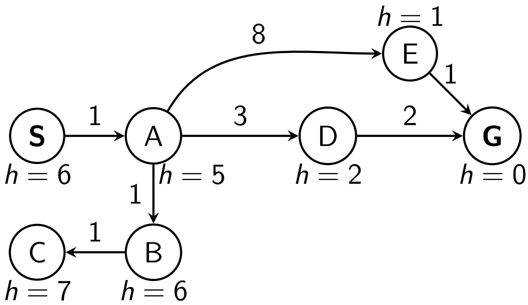
1. Expand "S". Add "A" to frontier.
2. Expand "A". Add "B","D","E".
3. Expand "E" ($h = 1$). Get "G".

Wrong:

- not optimal
- not complete (tree search version) (Can be shown on the Romania example – go back.)
- (graph search version is complete only in finite state spaces)

Nice: it is simple.

A^* combines UCS and Greedy



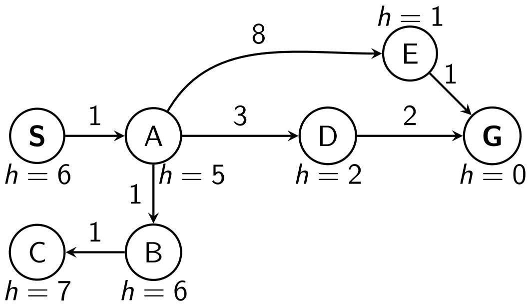
UCS orders by backward (path) cost $g(n)$
Greedy uses heuristics (goal proximity) $h(n)$

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

Notes

Trace the search algorithm on the paper. Does it find the shortest path?

A^* combines UCS and Greedy



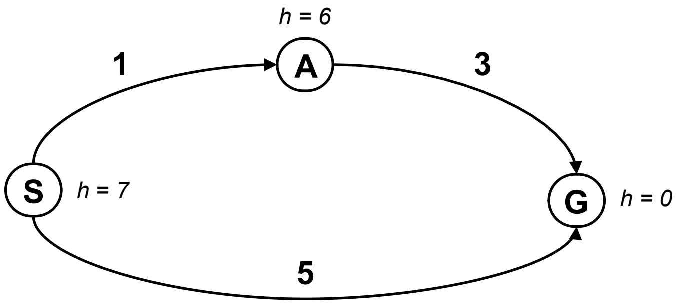
UCS orders by backward (path) cost $g(n)$
Greedy uses heuristics (goal proximity) $h(n)$

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

Notes

Trace the search algorithm on the paper. Does it find the shortest path?

Is A* optimal?



[2]

What is the problem?

---

[2]Graph example: Dan Klein and Pieter Abbeel

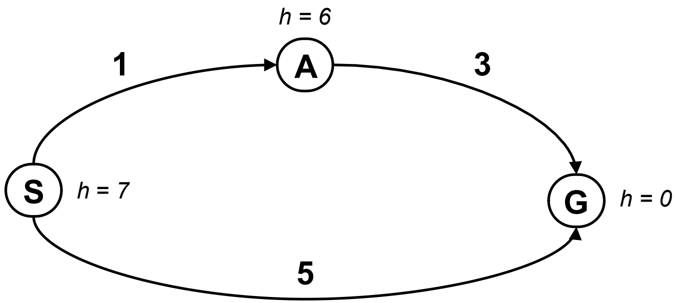
Notes

Try to answer the question before going to the next slide.

1. S
   - $f(S) = g(S) + h(S) = 0 + 7 = 7$
   - expanding/poping this one and crossing out (removing from frontier)
2. $S \rightarrow A$
   - $f(A) = g(A) + h(A) = 1 + 6 = 7$
3. $S \rightarrow G$
   - $f(G) = g(G) + h(G) = 5 + 0 = 5$
   - This is now cheapest on the frontier. I pop/expand and I'm done.

Ooops! That's not cheapest! What went wrong?
What follows – keep for next slide. Problem with $h(A) = 6$. Overestimating the expense. (Same problem for $h(S)$.)
Estimates need to be $\leq$ actual costs.

Is A* optimal?



2

What is the problem?

[2] Graph example: Dan Klein and Pieter Abbeel

Notes

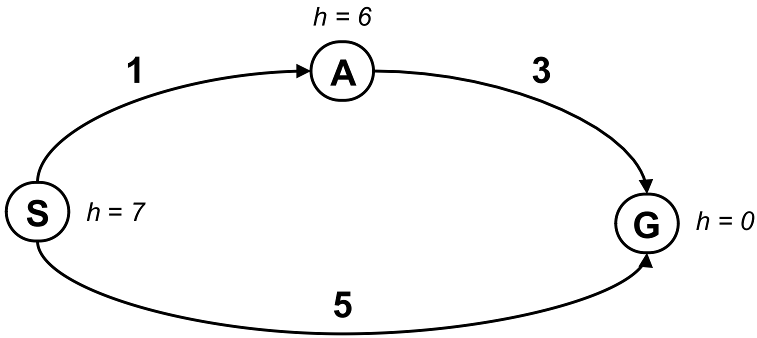
Try to answer the question before going to the next slide.

1. S
   - $f(S) = g(S) + h(S) = 0 + 7 = 7$
   - expanding/poping this one and crossing out (removing from frontier)
2. $S \to A$
   - $f(A) = g(A) + h(A) = 1 + 6 = 7$
3. $S \to G$
   - $f(G) = g(G) + h(G) = 5 + 0 = 5$
   - This is now cheapest on the frontier. I pop/expand and I'm done.

Ooops! That's not cheapest! What went wrong?
What follows – keep for next slide. Problem with $h(A) = 6$. Overestimating the expense. (Same problem for $h(S)$.)
Estimates need to be $\leq$ actual costs.

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: $0 \leq h(A)$



Notes

$h(A) \leq 3$ it means less than the actual cost of going from A to goal. Heuristic must not be overly pesimistic.
B is correct.
Negative $h(n)$ does not break the admissibility property but $h(\text{Goal}) = 0$ must be kept, always.
For a discussion, see, e.g.
https://stackoverflow.com/questions/30067813/are-heuristic-functions-that-produce-negative-values-inadmissible

Admissible heuristics

A heuristic function h is admissible if:

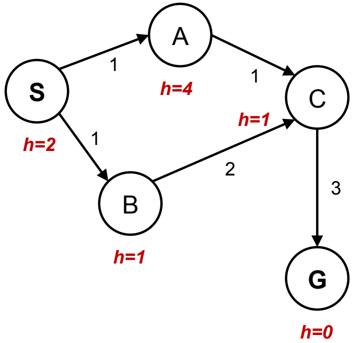
$$
\begin{aligned}
h(n) &\leq \text{cost}(n.state, \text{Goal}_{\text{nearest}}) \\
h(\text{Goal}) &= 0
\end{aligned}
$$

Notes

Even if negative heuristic value is allowed on the way to goal, does it make sense? How would you intepret $h(n) = 0$? Is it a meaningful minimum? Why?

Consistent heuristic

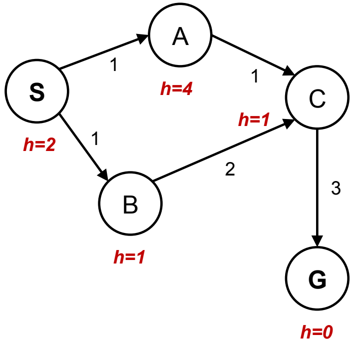
function FORWARD_SEARCH
    Q.insert($\_, s_0, 0$) and mark s_0 visited
    while Q not empty **do**
        $p, s, \_ \leftarrow$ Q.pop()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s', c \leftarrow$ 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



Notes

Consistent heuristic

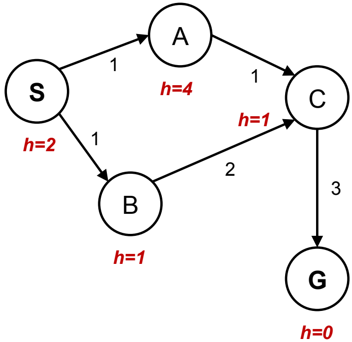
```
function FORWARD_SEARCH
    Q.insert(_, s_0, 0) and mark s_0 visited
    while Q not empty do
        p, s, _ ← Q.pop()
        parent[s] ← p
        if s ∈ S_G then return Success
        for all a ∈ A(s) do
            s′, c ← result(s, a)
            if s′ not visited then
                Mark s′ as visited
                Q.insert(s, s′, cost_from_start + h(s′))
            else
                Resolve duplicate s′
    return Failure
```

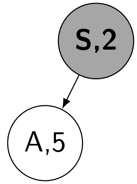


Notes

Consistent heuristic

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 0) and mark s_0 visited
    while Q not empty do
        p, s, _ ← Q.pop()
        parent[s] ← p
        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
```

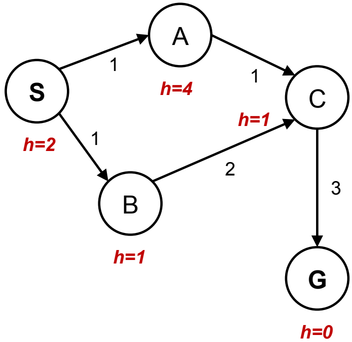


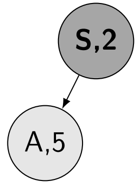


Notes

Consistent heuristic

function FORWARD_SEARCH
    Q.insert($\_, s_0, 0$) and mark s_0 visited
    while Q not empty **do**
        $p, s, \_ \leftarrow$ Q.pop()
        parent[s] $\leftarrow p$
        if $s \in \mathcal{S}_G$ **then return** Success
        for all $a \in \mathcal{A}(s)$ **do**
            $s', c \leftarrow$ 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

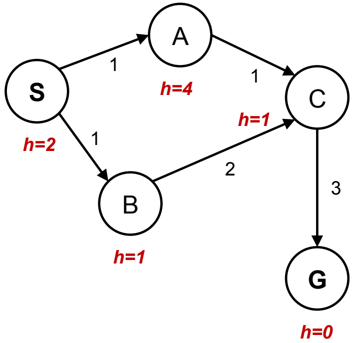


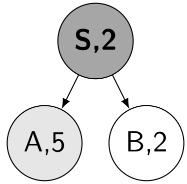


Notes

Consistent heuristic

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 0) and mark s_0 visited
    while Q not empty do
        p, s, _ ← Q.pop()
        parent[s] ← p
        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
```

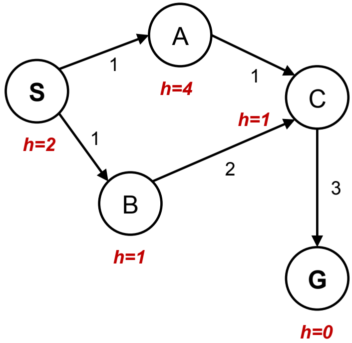


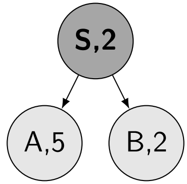


Notes

Consistent heuristic

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 0) and mark s_0 visited
    while Q not empty do
        p, s, _ ← Q.pop()
        parent[s] ← p
        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
```

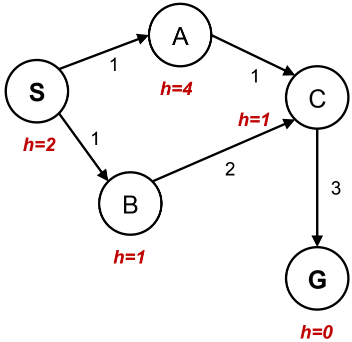


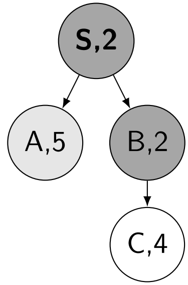


Notes

Consistent heuristic

```
function FORWARD_SEARCH
    Q.insert(_, s0, 0) and mark s0 visited
    while Q not empty do
        p, s, _ ← Q.pop()
        parent[s] ← p
        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
```

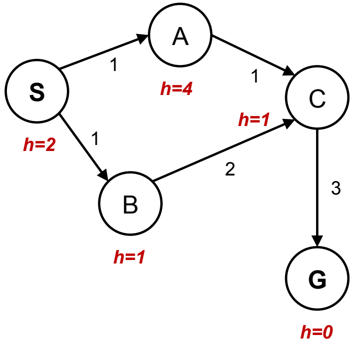


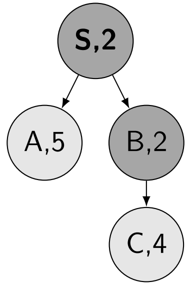


Notes

Consistent heuristic

function FORWARD_SEARCH
  Q.insert(_, s_0, 0) and mark s_0 visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow$ 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

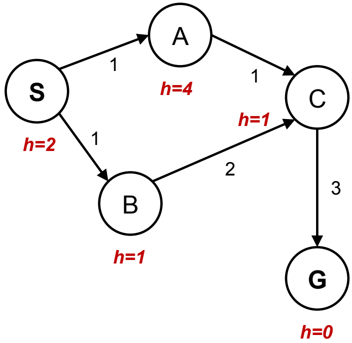


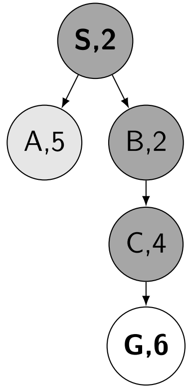


Notes

Consistent heuristic

function FORWARD_SEARCH
  Q.insert($\_, s_0, 0$) and mark s_0 visited
  while Q not empty **do**
    $p, s, \_ \leftarrow$ Q.pop()
    parent[s] $\leftarrow p$
    if $s \in \mathcal{S}_G$ **then return** Success
    for all $a \in \mathcal{A}(s)$ **do**
      $s', c \leftarrow$ 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

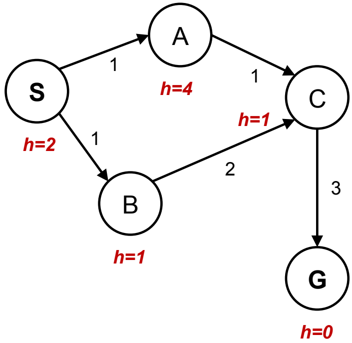


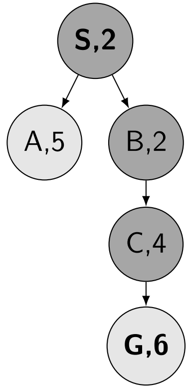


Notes

Consistent heuristic

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 0) and mark s_0 visited
    while Q not empty do
        p, s, _ ← Q.pop()
        parent[s] ← p
        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
```

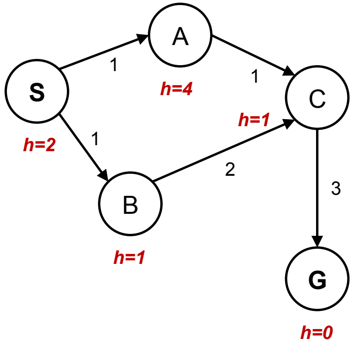


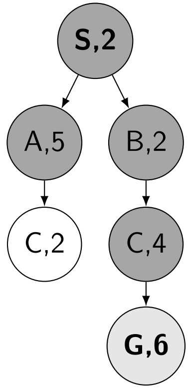


Notes

Consistent heuristic

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 0) and mark s_0 visited
    while Q not empty do
        p, s, _ ← Q.pop()
        parent[s] ← p
        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
```

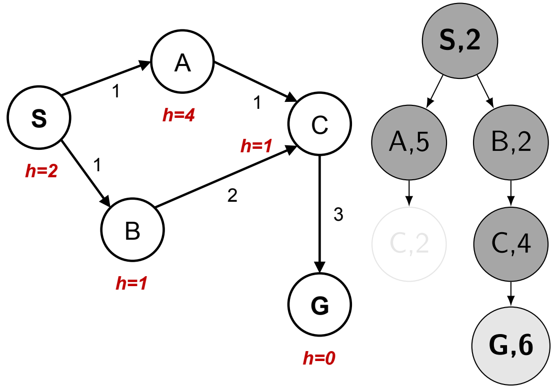




Notes

Consistent heuristic

```
function FORWARD_SEARCH
    Q.insert(_, s_0, 0) and mark s_0 visited
    while Q not empty do
        p, s, _ ← Q.pop()
        parent[s] ← p
        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
```



Notes

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

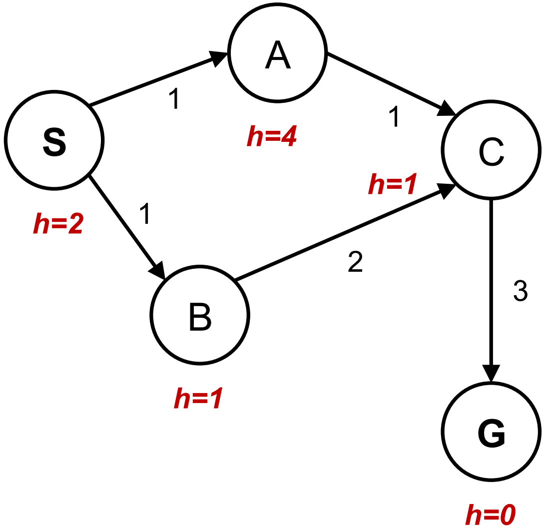
Consider other $h(s)$ fixed.

A: $h(A) = 1$

B: $h(A) = 2$

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

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

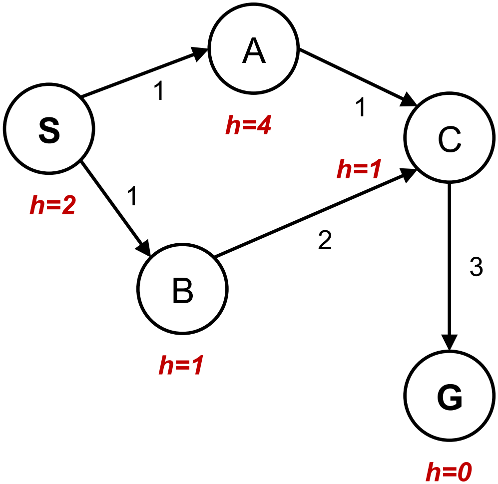


Notes

As it will be explained in the next slides: $h(A) \leq c(A, C) + h(C) = 2$

$h(S) \leq c(S, A) + h(A)$ it means $h(A) \geq h(S) - c(A, S) = 1$

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

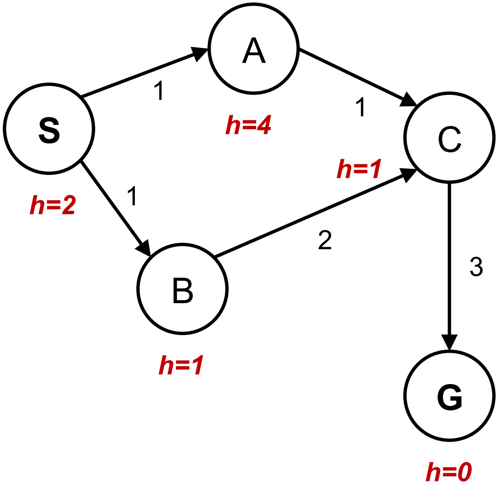
$f(n) = g(n) + h(n)$ along a path never decreases!

Notes

Our heuristic was admissible.
With *tree search* it would have worked. It would have expanded C and found the alternative, cheaper path.
For graph search, the problem is the $A \to C \to G$ subgraph where the *consistent* heuristic condition is violated.
The general condition means we have two constraints for (A) for this particuar graph:
$h(S) - h(A) \leq c(S, A)$
$h(A) - h(C) \leq c(A, C)$

Yes, all consistent heuristics are also admissible. Btw., it is not easy to invent a heuristics that is admissible but not consistent.

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

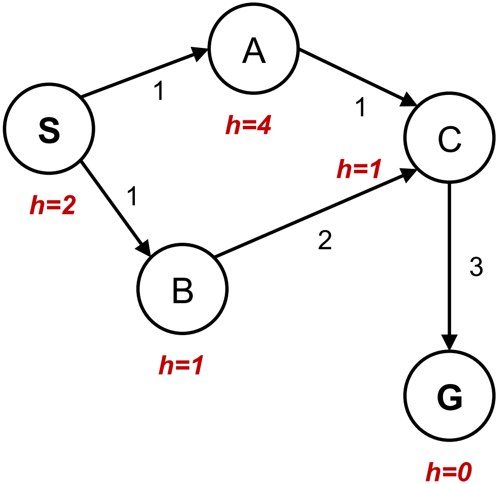
$f(n) = g(n) + h(n)$ along a path never decreases!

Notes

Our heuristic was admissible.
With *tree search* it would have worked. It would have expanded C and found the alternative, cheaper path.
For graph search, the problem is the $A \rightarrow C \rightarrow G$ subgraph where the *consistent* heuristic condition is violated.
The general condition means we have two constraints for (A) for this particuar graph:
$h(S) - h(A) \leq c(S, A)$
$h(A) - h(C) \leq c(A, C)$

Yes, all consistent heuristics are also admissible. Btw., it is not easy to invent a heuristics that is admissible but not consistent.

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

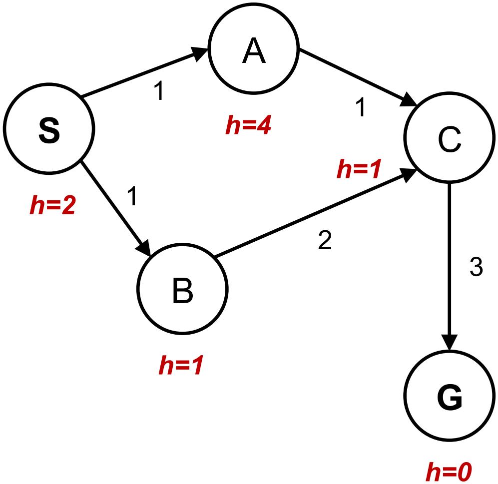
$f(n) = g(n) + h(n)$ along a path never decreases!

Notes

Our heuristic was admissible.
With *tree search* it would have worked. It would have expanded C and found the alternative, cheaper path.
For graph search, the problem is the $A \rightarrow C \rightarrow G$ subgraph where the *consistent* heuristic condition is violated.
The general condition means we have two constraints for (A) for this particuar graph:
$h(S) - h(A) \leq c(S, A)$
$h(A) - h(C) \leq c(A, C)$

Yes, all consistent heuristics are also admissible. Btw., it is not easy to invent a heuristics that is admissible but not consistent.

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!

Notes

Our heuristic was admissible.
With *tree search* it would have worked. It would have expanded C and found the alternative, cheaper path.
For graph search, the problem is the $A \to C \to G$ subgraph where the *consistent* heuristic condition is violated.
The general condition means we have two constraints for (A) for this particuar graph:
$h(S) - h(A) \leq c(S, A)$
$h(A) - h(C) \leq c(A, C)$

Yes, all consistent heuristics are also admissible. Btw., it is not easy to invent a heuristics that is admissible but not consistent.

Summary

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

Notes

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/.

Notes