3010

ARTIFICIAL INTELLIGENCE

Lecture 2 Dijkstra’s shortest path algorithm

Masashi Shimbo

2019-05-13

TODAY'S AGENDA

- Uniform-cost search (to consider edge weights) for trees
- Dijkstra's algorithm (for non-tree graphs)

Uniform-cost Search for Trees

BREADTH-/DEPTH-FIRST SEARCH ALGORITHMS

Breadth-first search expands shallowest nodes first
Depth-first search expands deepest nodes first

In both algorithms, edge (=action) costs are ignored

➥ they are not admissible if edge costs vary

“UNIFORM-COST TREE SEARCH” EXPANDS NODES WITH THE CHEAPEST COST FIRST

To make this possible,

1. For each node v, maintain the path cost from the initial node

$$g[v] = \text{path cost from the initial state to } v$$

2. Use a **priority queue** to implement the OPEN list
— to select the node with the minimum g-value from OPEN

PRIORITY QUEUE

- Has an associated function (**priority function**) g that determines the priority $g[v]$ of each item v in the list

  g-value = low $\implies$ priority = high

- Allows retrieval of an item with the **minimum** g-value

PRIORITY QUEUE

Two functions for manipulating priority queue P_g:

$\mathrm{Insert}_g(P_g, v)$

Put item v in P_g

$\mathrm{DeleteMin}_g(P_g)$

Remove and return an item with the minimum g-value from P_g

➥ returned item v is such that

$$v = \underset{u \in P_g}{\text{argmin}}\, g[u]$$

before its removal

UNIFORM-COST SEARCH FOR TREES

$\mathrm{OPEN} \leftarrow$ **new** ~~List~~ $\mathrm{PriorityQueue}_g$

$g[s] \leftarrow 0$

~~Insert~~ $\mathrm{Insert}_g(\mathrm{OPEN}, s)$

loop do

    if $\mathrm{IsEmpty}(\mathrm{OPEN})$ **then return** "failure"

    $v \leftarrow$ ~~RemoveOne~~ $\mathrm{DeleteMin}_g(\mathrm{OPEN})$

    if $\mathrm{IsGoal}(v)$ **then return** $\mathrm{Solution}(v, s)$

    $\mathrm{Expand}(v)$

N.B. OPEN, g, and Parent are global variables

procedure $\mathrm{Expand}(v)$

now maintains the cost g from the initial state

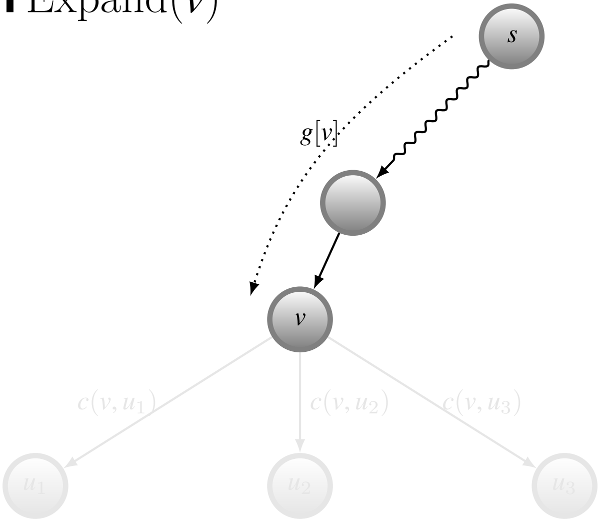
Input : v: node to expand
Output : set of successors of v

---

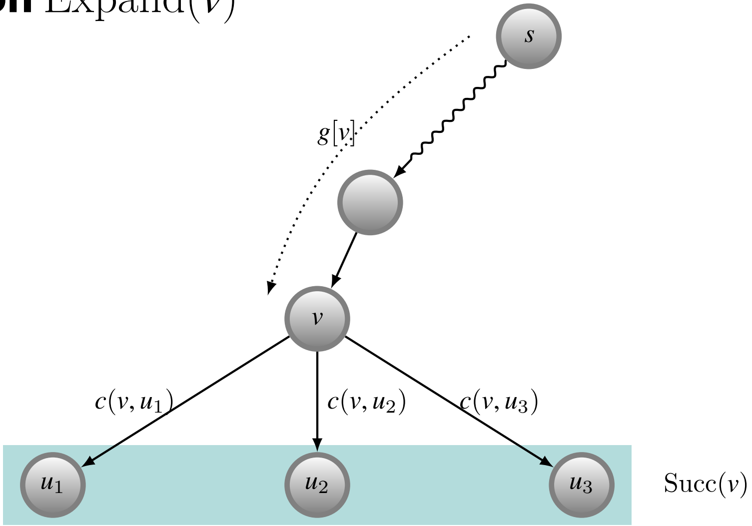
foreach node $u \in \mathrm{Succ}(v)$ **do**
    Reserve memory for node u
    $\mathrm{Parent}[u] \leftarrow v$
    $g[u] \leftarrow g[v] + c(v, u)$
    $\cancel{\mathrm{Insert}}\ \mathrm{Insert}_g(\mathrm{OPEN}, u)$

N.B. OPEN, g, and Parent are global variables

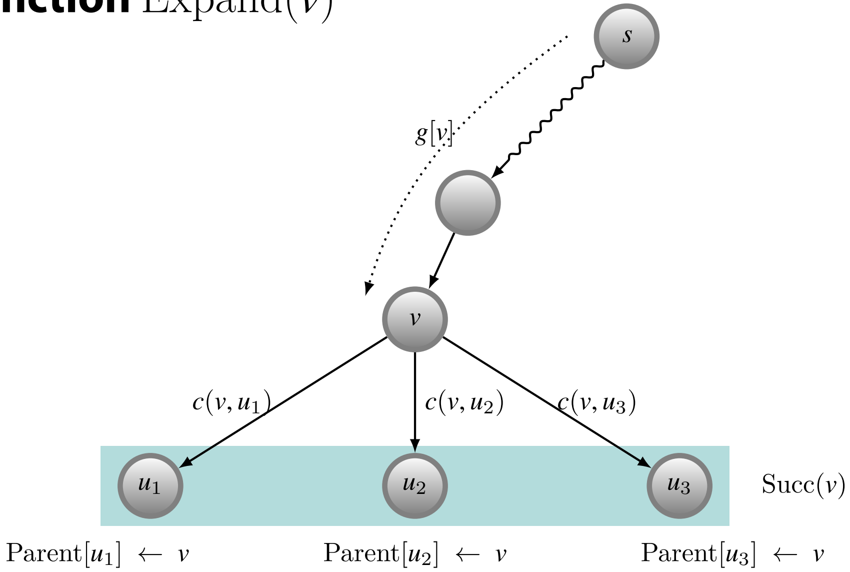
function $\mathrm{Expand}(v)$



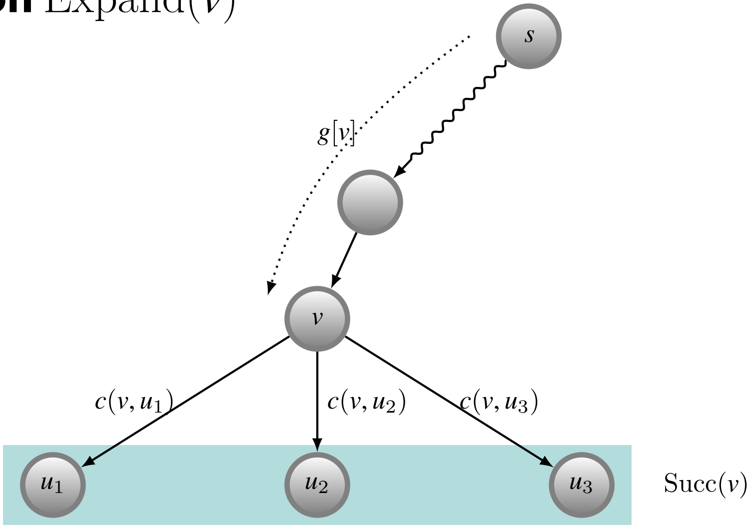
function $\mathrm{Expand}(v)$



function $\mathrm{Expand}(v)$



function $\text{Expand}(v)$



$\text{Parent}[u_1] \leftarrow v$ $\quad$ $\text{Parent}[u_2] \leftarrow v$ $\quad$ $\text{Parent}[u_3] \leftarrow v$

$g[u_1] \leftarrow g[v] + c(v, u_1)$ $\quad$ $g[u_2] \leftarrow g[v] + c(v, u_2)$ $\quad$ $g[u_3] \leftarrow g[v] + c(v, u_3)$

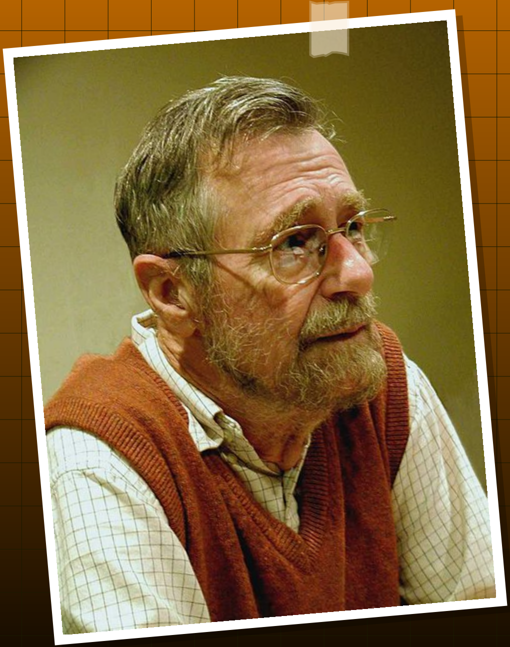
UNIFORM-COST SEARCH ALGORITHM FOR TREES

function $\mathrm{UnformCostSearch}(s)$

$\mathrm{OPEN} \leftarrow$ **new** $\mathrm{PriorityQueue}_g$
$g[s] \leftarrow 0$
$\mathrm{Insert}_g(\mathrm{OPEN}, s)$
loop do
  if $\mathrm{IsEmpty}\ (\mathrm{OPEN})$ **then**
    return "failure"
  $v \leftarrow \mathrm{DeleteMin}_g(\mathrm{OPEN})$
  if $\mathrm{IsGoal}(v)$ **then**
    return $\mathrm{Solution}(v, s)$
  $\mathrm{Expand}(v)$

procedure $\mathrm{Expand}(v)$

foreach $u \in \mathrm{Succ}(v)$ **do**
  Reserve memory for u
  $\mathrm{Parent}[u] \leftarrow v$
  $g[u] \leftarrow g[v] + c(v, u)$
  $\mathrm{Insert}_g(\mathrm{OPEN}, u)$

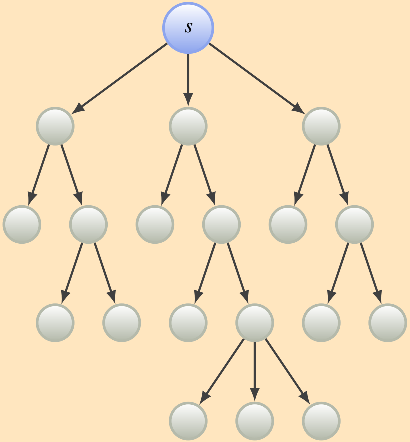


Dijkstra's
shortest-path algorithm

ダイクストラの最短経路アルゴリズム

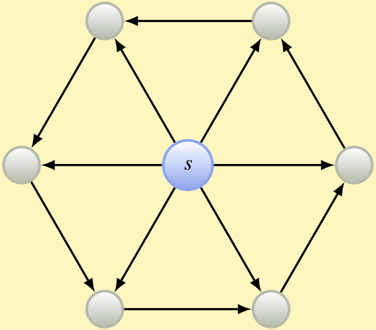
TREES

There exists exactly one path from the initial node to each node



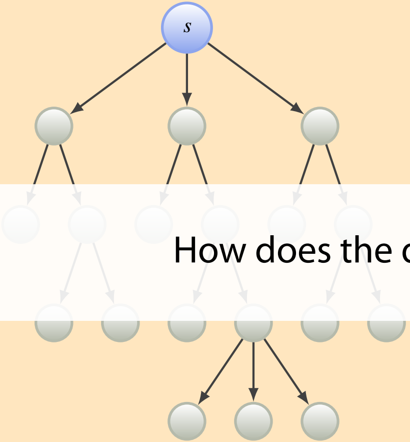
GRAPHS

There can be many paths from the initial node to a node



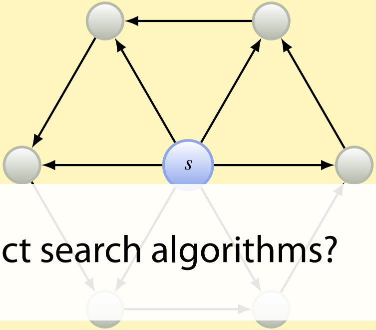
TREES

There exists exactly one path from the initial node to each node



GRAPHS

There can be many paths from the initial node to a node



How does the difference affect search algorithms?

SHORTEST-PATH SEARCH IN A TREE

For every node, there is only one path from the initial node s

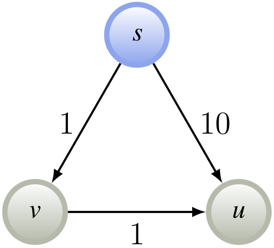
➥ The first-found path to a node is the only path to the node

➥ of course, it is also the shortest path to the node

These do not apply to general (= non-tree) graphs

SHORTEST-PATH SEARCH IN A GENERAL GRAPH

The first-found path to a node may not be the shortest

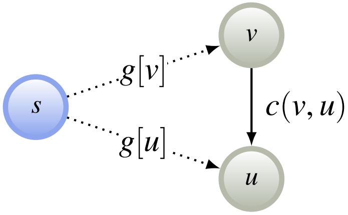


After s is expanded and u is generated,

$$g[u] = c(s, u) = 10$$

But path $s \to u$ is not the shortest path! ($s \to v \to u$ is)

GRAPH SEARCH: NODE EXPANSION



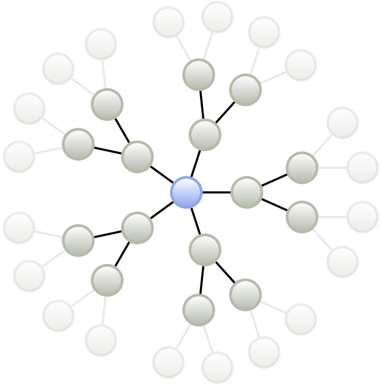
When a function call $\mathrm{Expand}(v, \mathrm{OPEN})$ tries to generate a successor u of v, three cases are possible:

1. u has never been generated
2. u has been generated but not yet expanded
3. u has been generated and expanded

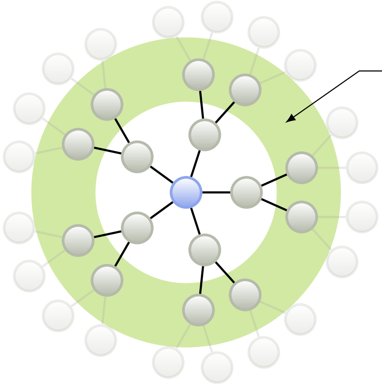
➥ We need to distinguish these cases

Do we have sufficient information to do so? No

OPEN AND EXPANDED NODES

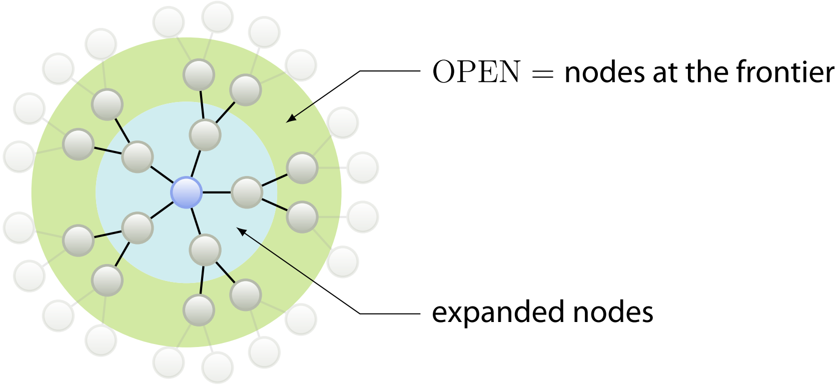


OPEN AND EXPANDED NODES



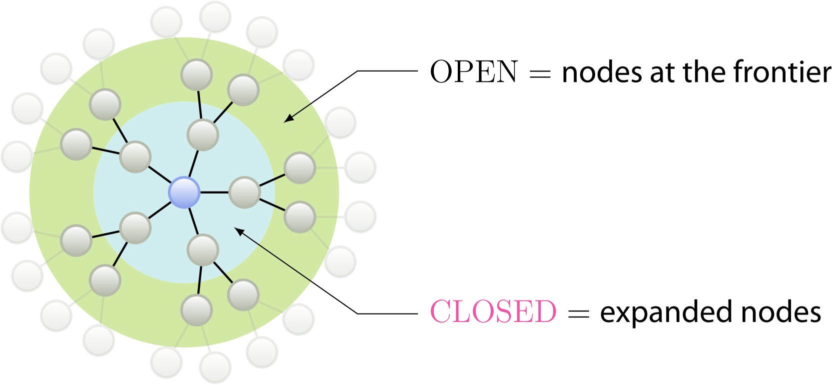
OPEN = nodes at the frontier

OPEN AND EXPANDED NODES



So far, we have not kept record of expanded nodes at all
— they were simply taken out of OPEN after expansion

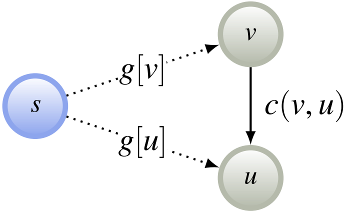
OPEN AND EXPANDED NODES



So far, we have not kept record of expanded nodes at all
— they were simply taken out of OPEN after expansion

➥ Let's keep these nodes in a set called CLOSED

GRAPH SEARCH: NODE EXPANSION



If expanded nodes are kept in CLOSED, the three cases can be restated:

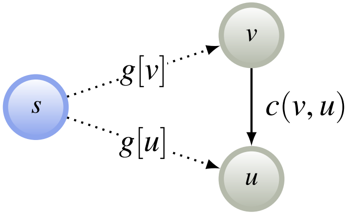
1. u has never been generated

   ➥ $u \notin \mathrm{OPEN} \cup \mathrm{CLOSED}$

2. u has been generated but not yet expanded

3. u has been generated and expanded

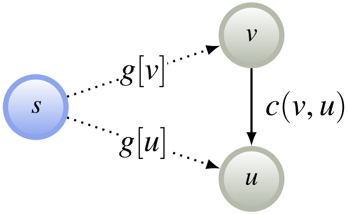
GRAPH SEARCH: NODE EXPANSION



If expanded nodes are kept in CLOSED, the three cases can be restated:

1. u has never been generated
   - ➥ $u \notin \text{OPEN} \cup \text{CLOSED}$
2. u has been generated but not yet expanded
   - ➥ $u \in \text{OPEN}$
3. u has been generated and expanded

GRAPH SEARCH: NODE EXPANSION

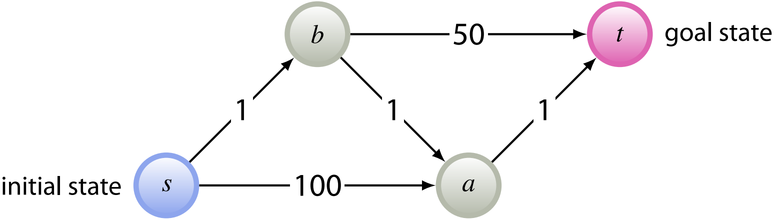


If expanded nodes are kept in CLOSED, the three cases can be restated:

1. u has never been generated
   ➥ $u \notin \mathrm{OPEN} \cup \mathrm{CLOSED}$
2. u has been generated but not yet expanded
   ➥ $u \in \mathrm{OPEN}$
3. u has been generated and expanded
   ➥ $u \in \mathrm{CLOSED}$

EXAMPLE

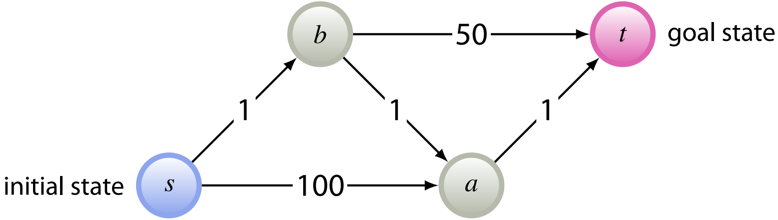
What happens if we run uniform-cost tree search in a graph?



- 3 paths from s to t:
  - $s \to a \to t$
  - $s \to b \to t$
  - $s \to b \to a \to t$
- Shortest path is $s \to b \to a \to t$ with cost$=3$

EXAMPLE

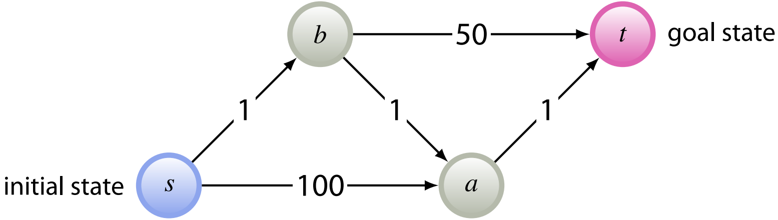
What happens if uniform-cost tree search is run in a non-tree graph?



Step	Expanded	OPEN	$g[a]$	$g[b]$	$g[t]$
0		$\{s\}$	—	—	—

EXAMPLE

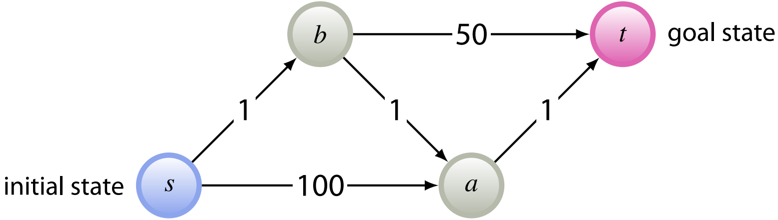
What happens if uniform-cost tree search is run in a non-tree graph?



Step	Expanded	OPEN	$g[a]$	$g[b]$	$g[t]$
0		$\{s\}$	–	–	–
1	s	$\{a,\ b\}$	100	1	–

EXAMPLE

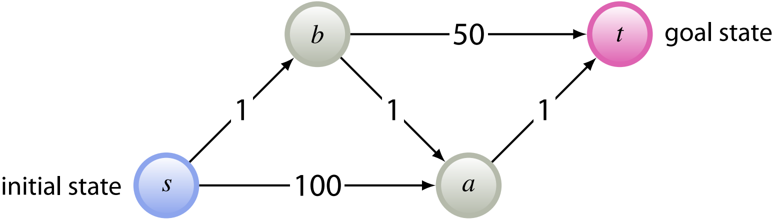
What happens if uniform-cost tree search is run in a non-tree graph?



Step	Expanded	OPEN	$g[a]$	$g[b]$	$g[t]$
0		$\{s\}$	–	–	–
1	s	$\{a,\ b\}$	100	1	–
2	b	$\{a,\ t\}$	?	1	51

Suppose we kept the value of $g[a]$ intact. . .

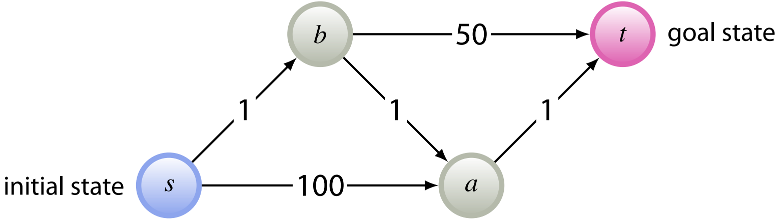
仮に $g[a]$ を更新しなかったとすると...



Step	Expanded	OPEN	$g[a]$	$g[b]$	$g[t]$
0		$\{s\}$	–	–	–
1	s	$\{a,\ b\}$	100	1	–
2	b	$\{a,\ t\}$	100	1	51

Suppose we kept the value of $g[a]$ intact...

仮に $g[a]$ を更新しなかったとすると...

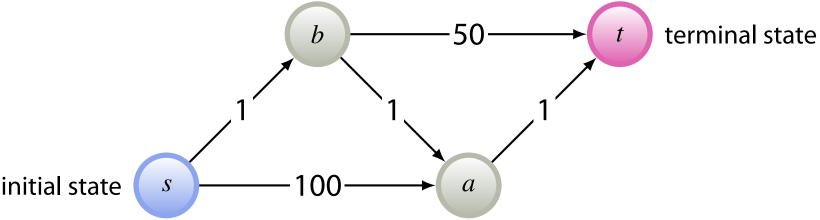


Step	Expanded	OPEN	$g[a]$	$g[b]$	$g[t]$
0		$\{s\}$	–	–	–
1	s	$\{a, b\}$	100	1	–
2	b	$\{a, t\}$	100	1	51
3	t	$\{a\}$	100	1	51

Solution $s \to b \to t$: cost=51 (not the shortest path)

If we let $g[a] \leftarrow \min\{g[a], g[b] + c(b, a)\}$

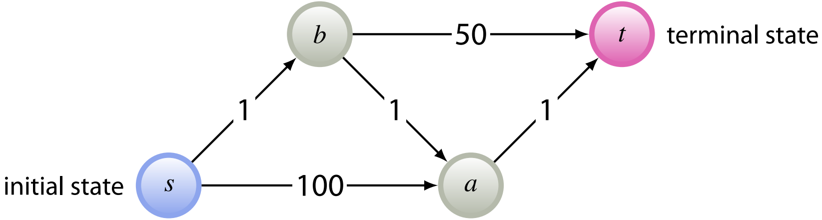
より短い方のコストを $g[a]$ に保持すれば…



Step	Expanded	OPEN	$g[a]$	$g[b]$	$g[t]$
0		$\{s\}$	–	–	–
1	s	$\{a,\ b\}$	100	1	–
2	b	$\{a,\ t\}$	2	1	51

If we let $g[a] \leftarrow \min\{g[a], g[b] + c(b,a)\}$

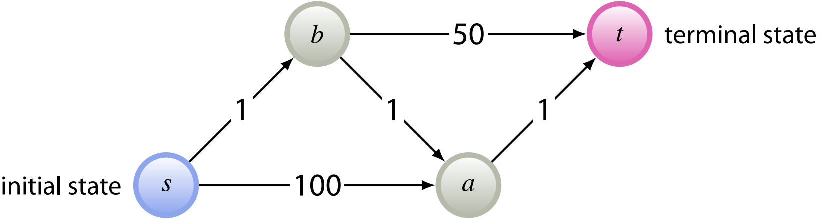
より短い方のコストを $g[a]$ に保持すれば…



Step	Expanded	OPEN	$g[a]$	$g[b]$	$g[t]$
0		$\{s\}$	–	–	–
1	s	$\{a,\ b\}$	100	1	–
2	b	$\{a,\ t\}$	2	1	51
3	a	$\{t\}$	2	1	3

If we let $g[a] \leftarrow \min\{g[a], g[b] + c(b,a)\}$

より短い方のコストを $g[a]$ に保持すれば…

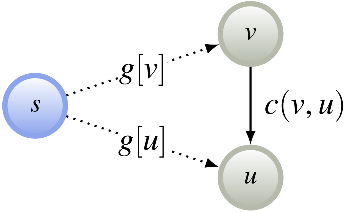


Step	Expanded	OPEN	$g[a]$	$g[b]$	$g[t]$
0		$\{s\}$	–	–	–
1	s	$\{a,\ b\}$	100	1	–
2	b	$\{a,\ t\}$	2	1	51
3	a	$\{t\}$	2	1	3
4	t	$\emptyset$	2	1	3

Solution $s \rightarrow b \rightarrow a \rightarrow t$: cost=3 (shortest path)

DISJKSTRA'S ALGORITHM: GRAPH SEARCH STRATEGY

for each of the three possible cases

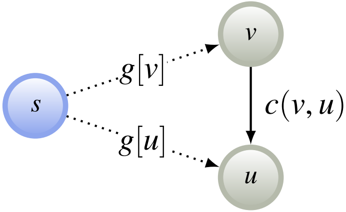


Recall the three cases when a successor u of a node v is encountered

1. u has never been generated $\equiv$ $u \notin \mathrm{OPEN} \cup \mathrm{CLOSED}$
2. u has been generated but not yet expanded $\equiv$ $u \in \mathrm{OPEN}$
3. u has been generated and expanded $\equiv$ $u \in \mathrm{CLOSED}$

CASE 1. NODE $u \notin \mathrm{OPEN} \cup \mathrm{CLOSED}$

The successor u has never been generated

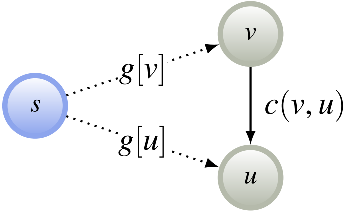


A brand-new node

➥ Proceed in the same way as tree search

CASE 2. $u \in \mathrm{OPEN}$

The successor u has been generated but not expanded



- If $g[v] + c(v, u) < g[u]$, we have found a better path to u (via v)

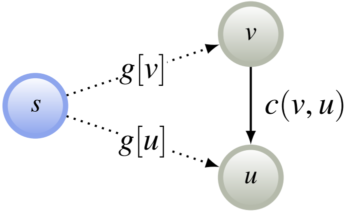
  ➥ Update $g[u] \leftarrow g[v] + c(v, u)$ (also update $\mathrm{Parent}[u]$ to v)

- Otherwise, do nothing

Note: This operation is called **relaxation** of edge (v, u)

CASE 3. $u \in \mathrm{CLOSED}$

The successor u has been expanded



In this case, (as we will prove later) it always holds that

$$g[u] \leq g[v] + c(v, u)$$

- In fact, we already have a shortest path to u, if $u \in \mathrm{CLOSED}$
  (and $g[u] =$ the cost of that shortest path)
- No more processing is necessary—just skip u.

DIJKSTRA'S SHORTEST PATH ALGORITHM

$\text{CLOSED} \leftarrow \emptyset$ # CLOSED: set of expanded states
$\text{OPEN} \leftarrow$ **new** PriorityQueue_g
$g[s] \leftarrow 0$
$\text{Insert}_g(\text{OPEN}, s)$ # OPEN: set of states generated but not expanded
loop do
    if $\text{IsEmpty}(\text{OPEN})$ **then return** "failure"
    $v \leftarrow \text{DeleteMin}_g(\text{OPEN})$ # choose a node with the smallest g
    $\text{CLOSED} \leftarrow \text{CLOSED} \cup \{v\}$ # put v into CLOSED
    if $\text{IsGoal}(v)$ **then return** $\text{Solution}(v, s)$
    $\text{Expand}(v)$

N.B. OPEN, Parent, and g are global variables (accessible from Expand and Solution)

procedure $\mathrm{Expand}(v)$ FOR DIJKSTRA'S ALGORITHM


```
foreach u ∈ Succ(v) do
    if u ∉ OPEN ∪ CLOSED then
        Reserve memory for u
        Parent[u] ← v
        g[u] ← g[v] + c(v, u)
        Insert_g(OPEN, u)
    else if u ∈ OPEN then
        if g[v] + c(v, u) < g[u] then
            Parent[u] ← v
            g[u] ← g[v] + c(v, u)
```

N.B. OPEN, Parent, and g are global variables

procedure $\mathrm{Expand}(v)$ FOR DIJKSTRA'S ALGORITHM

foreach $u \in \mathrm{Succ}(v)$ **do**
- **if** $u \notin \mathrm{OPEN} \cup \mathrm{CLOSED}$ **then**
  - Reserve memory for u
  - $\mathrm{Parent}[u] \leftarrow v$
  - $g[u] \leftarrow g[v] + c(v, u)$
  - $\mathrm{Insert}_g(\mathrm{OPEN}, u)$
- **else if** $u \in \mathrm{OPEN}$ **then**
  - **if** $g[v] + c(v, u) < g[u]$ **then**
    - $\mathrm{Parent}[u] \leftarrow v$
    - $g[u] \leftarrow g[v] + c(v, u)$

Cf. $\mathrm{Expand}(v)$ for tree search

foreach $u \in \mathrm{Succ}(v)$ **do**
- Reserve memory for u
- $\mathrm{Parent}[u] \leftarrow v$
- $g[u] \leftarrow g[v] + c(v, u)$
- $\mathrm{Insert}_g(\mathrm{OPEN}, u)$

N.B. OPEN, Parent, and g are global variables

PROPERTIES OF DIJKSTRA'S ALGORITHM

In any state space graph (which may or may not be a tree), Dijkstra's algorithm enjoys

Completeness

Dijkstra's algorithm never fails to find a solution

Admissibility

The solution found by Dijkstra's algorithm is optimal