Reinforcement Learning

Prof. Christian Bauckhage

lecture 03
problem solving as search

in this lecture, we study *problem solving* as *(tree) search* and will see that

tree search is a very general concept

it applies to any problem with *discrete state space* structure for which we need to find a *path* from an *initial state* to some *goal state*

$\Leftrightarrow$ *tree search algorithms* apply whenever the *state space* of a problem can be modeled as a *graph*

for the fun of it, we will then have a closer look at tree search algorithms for spatial *path planning* and finally tie this back to *policy learning*

outline

recap / introduction

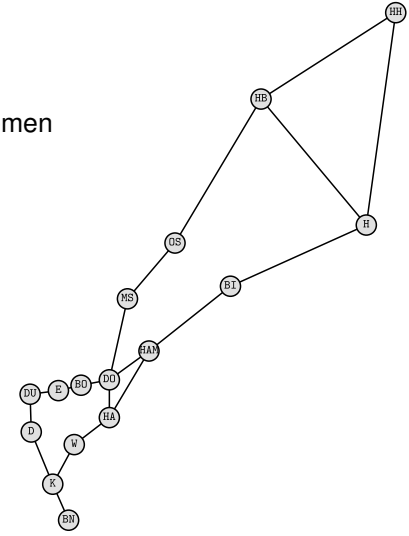
previously, we looked at *tree search algorithms* for decision making in games ...

next, we widen our perspective and consider tree search in more general settings

problem solving as tree search

motivating example

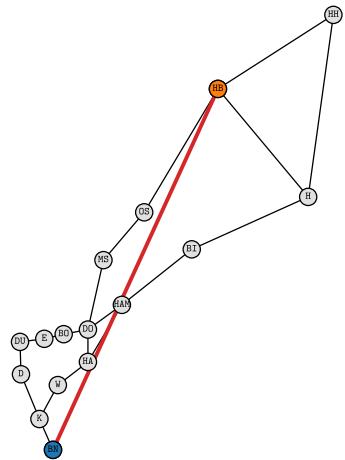
plan a time and cost *efficient* trip from Bonn to Bremen



“high speed” railroad net
in northwestern Germany

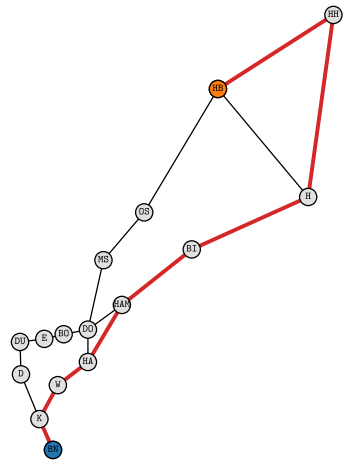
1st solution

charter a helicopter and fly (not cost efficient)



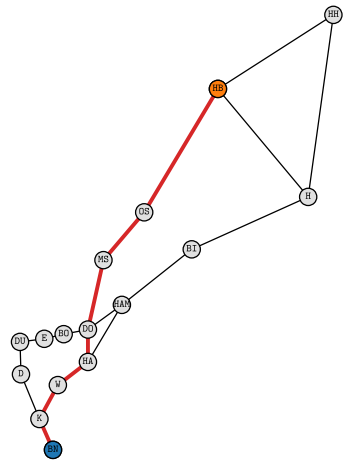
2nd solution

travel by train via Hannover / Hamburg (likely not time efficient)



3rd solution

travel by train via Münster / Osnabrück (reasonably efficient)

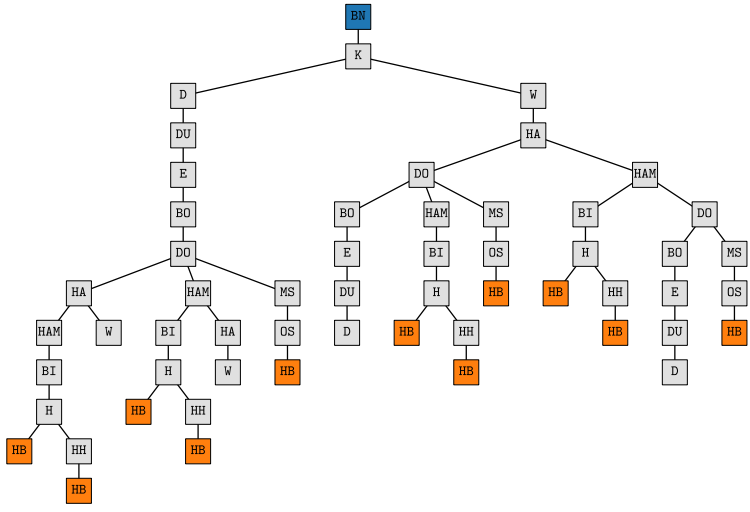


question

but where is the *search tree* ?

answer

here

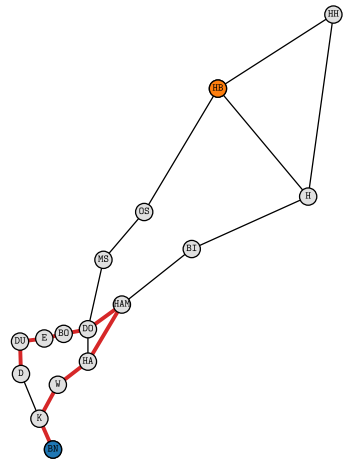

BN
K
D
W
DU
HA
E
DO
HAM
BO
BO
HAM
MS
BI
DO
DO
E
BI
OS
H
BO
MS
HA
HAM
MS
DU
H
HB
HB
HH
E
OS
HAM
W
BI
HA
OS
D
HB
HH
HB
DU
HB
BI
H
W
HB
HB
D
H
HB
HH
HB
HB
HB
HH
HB

question

why are there *dead ends* in Düsseldorf and Wuppertal ?

answer

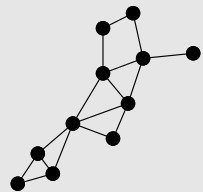
because our state space has *cycles* so that paths may loop back onto themselves . . .



terms and definitions

(well defined) problem

$\Leftrightarrow$ consists of these components

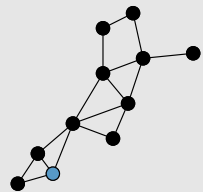


(well defined) problem

⇔ consists of these components

initial state

⇔ state an agent starts in



(well defined) problem

⇔ consists of these components

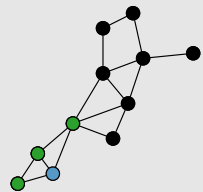
initial state

⇔ state an agent starts in

successor function

⇔ determines possible *moves* in a state

initial state and successor function jointly define a *search tree* over the *state space*



(well defined) problem

$\Leftrightarrow$ consists of these components

initial state

$\Leftrightarrow$ state an agent starts in

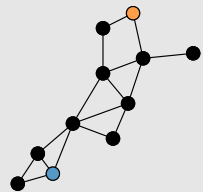
successor function

$\Leftrightarrow$ determines possible *moves* in a state

initial state and successor function jointly define a *search tree* over the *state space*

goal test function

$\Leftrightarrow$ tests if *current state* s is a *goal state*



(well defined) problem

$\Leftrightarrow$ consists of these components

initial state

$\Leftrightarrow$ state an agent starts in

successor function

$\Leftrightarrow$ determines possible *moves* in a state

initial state and successor function jointly define a *search tree* over the *state space*

goal test function

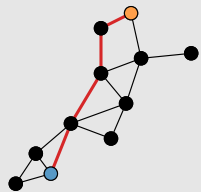
$\Leftrightarrow$ tests if *current state* s is a *goal state*

path cost function

$\Leftrightarrow$ assigns *numeric costs* to *paths*

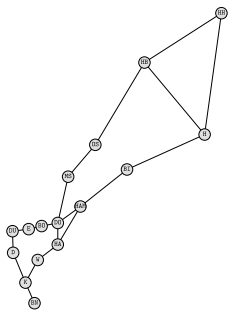
we say that *paths* $\Leftrightarrow$ *sequences of states* connected by *moves*

we say that *path costs* $\Leftrightarrow$ sum of *step costs* $\Leftrightarrow$ costs of moves

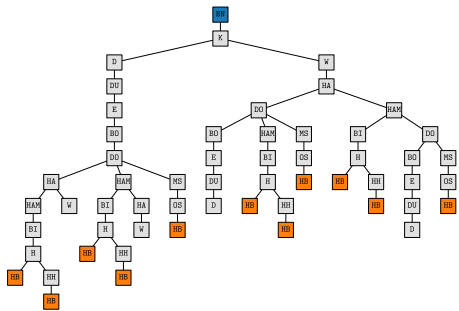


examples

for route planning, we have

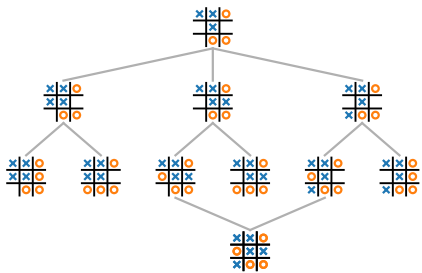


state space

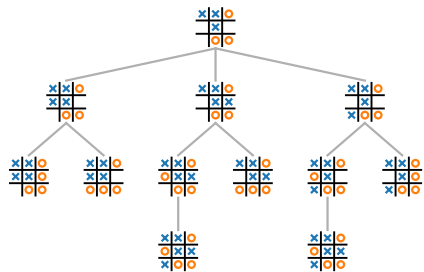


(exemplary) search tree

for decision making in ***tic tac toe*** , we have



state space



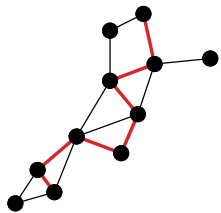
search tree

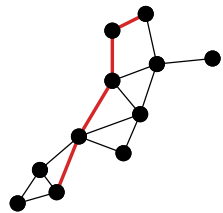
solution

$\Leftrightarrow$ path from the initial state to *a* or *the* goal state

optimal solution

$\Leftrightarrow$ solution of minimum path costs





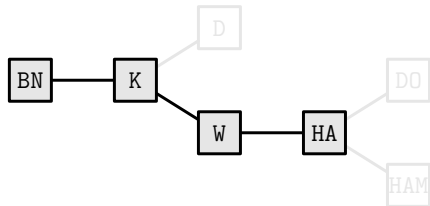
now . . .

search tree expansion

problem solving / planning

$\Leftrightarrow$ *expansion of a search tree*

$\Leftrightarrow$ starting in the initial state s_{src}, *explore* the state space by *generating* and *examining* successors of s_{src}
continue recursively until *a* or *the* goal state is reached



search strategy

$\Leftrightarrow$ determines which generated *tree nodes* are to be examined

fringe

$\Leftrightarrow$ set of nodes that have been generated but not yet examined

naïve search tree expansion

function *TreeSearch*(*problem*, *strategy*, *fringe* $= \{s_{src}\}$)
  while *fringe* $\neq \varnothing$
    use *strategy* to choose $s \in$ *fringe* for examination

    if *GoalTest*(s, *problem*) yields `true`
      return *success*
    else
      fringe $\leftarrow$ *fringe* $\setminus \{ s \}$
      fringe $\leftarrow$ *fringe* $\cup$ *Succ*(s)

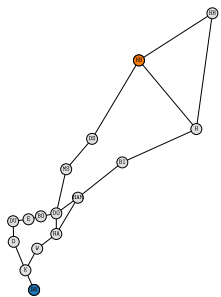
  return *failure*

question

is there a “problem” with this algorithm ?

answer

consider this: if we search for a path from Bonn to Bremen, we have . . .



$$
\begin{aligned}
& \mathit{fringe} = \{\texttt{BN}\} && \text{not much to choose from fringe} \\
\Rightarrow \quad & s = \texttt{BN}, \mathit{Succ}(s) = \{\texttt{K}\} \\
\Rightarrow \quad & \mathit{fringe} = \{\texttt{K}\} && \text{not much to choose from fringe} \\
\Rightarrow \quad & s = \texttt{K}, \mathit{Succ}(s) = \{\texttt{BN, D, W}\} \\
\Rightarrow \quad & \mathit{fringe} = \{\texttt{BN, D, W}\} && \text{but we already were in Bonn ???} \\
& \vdots
\end{aligned}
$$

improved search tree expansion

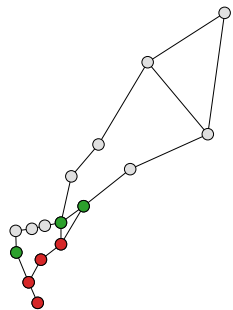
function *TreeSearch*(*problem*, *strategy*, *fringe* $= \{s_{src}\}$, *closed* $= \varnothing$)
  while *fringe* $\neq \varnothing$
    use *strategy* to choose $s \in$ *fringe* for expansion

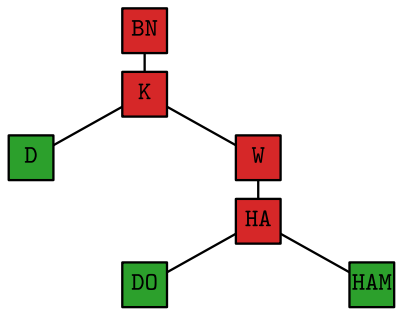
    if *GoalTest*(*s*, *problem*) yields `true`
      return *success*
    else
      $closed \leftarrow closed \cup \{ s \}$
      $fringe \leftarrow fringe \setminus \{ s \}$
      $fringe \leftarrow fringe \cup \left\{ s' \in Succ(s) \;\middle|\; s' \notin closed \right\}$

  return *failure*

other common names for the *fringe* are frontier or open list

each element of the *fringe* is a *leaf* of the current search tree





$closed = \{\texttt{BN}, \texttt{K}, \texttt{W}, \texttt{HA}\}$ $fringe = \{\texttt{D}, \texttt{DO}, \texttt{HAM}\}$

question

why *search strategies* ?

answer

avoid infinite expansions

quickly determine solutions

question

what kind of *search strategies* are there ?

answer

there basically exist *two* major families . . .

uninformed / blind search

$\Leftrightarrow$ considers *only* problem definition / state space structure

informed / heuristic search

$\Leftrightarrow$ considers problem definition *and* additional knowledge

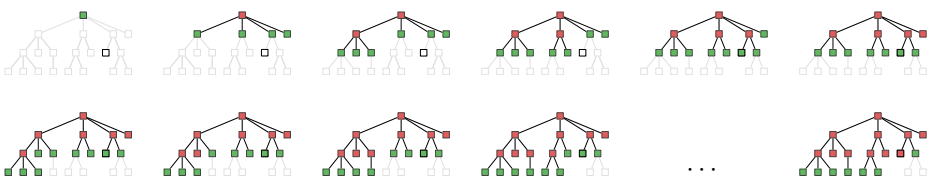
(usually given in form of *attributes* or *features* of states)

uninformed search strategies

breadth-first search

expand the *root*, then its successors, then their successors, ...

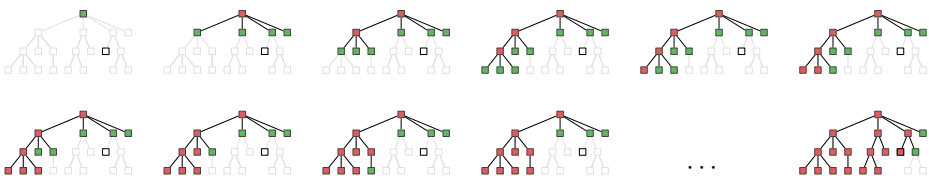
$\Leftrightarrow$ expand every search tree node in level l before expanding any node in level $l + 1$



depth-first search

always expand a or the deepest node in the current *fringe*

when this reaches a *terminal state* that is *not* a *goal state*,
back track to nodes with unexplored successors



remarks

for *finite* branching b, breadth-first search is complete

$\Leftrightarrow$ if *a* or *the* shallowest goal node resides on level d, it will eventually be found once levels 0 through $d-1$ have been searched exhaustively

in general, the *shallowest solution* may not be *optimal*

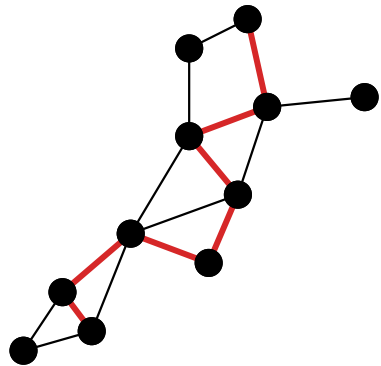
however, breadth-first search is optimal, if *path costs* monotonically increase with search tree level l

for *infinite* depth D, depth-first search is incomplete

$\Leftrightarrow$ it may forever descend down a “wrong” subtree and never find a goal state

depth-first search is not optimal

$\Leftrightarrow$ it may return a solution found in a “wrong” subtree



note

worst case time- and memory complexity of *breadth-first search* are

$$O\left(b^d\right)$$

if goal state s_{tgt} is the “last node” in level d, the algorithm expands

$$b^{d+1}/b = O\left(b^d\right)$$

new nodes in level $d+1$
before s_{tgt} gets examined

$\Leftrightarrow$ even for a “moderate” b, naïve breadth-first search is usually infeasible
(mainly due to exploding memory consumption)

note

memory complexity of *depth-first search* is $O(bD)$

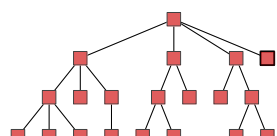
worst case time complexity of *depth-first search* is

$$O(b^D)$$

if goal state s_tgt is the “last node” in level d, the algorithm expands

$$b^0 + b^1 + \ldots + b^D - \left(b^0 + b^1 + \ldots + b^{D-d}\right) = O(b^D)$$

nodes before s_{tgt} is examined



$\Leftrightarrow$ even for a “moderate” b, naïve depth-first search us usually infeasible
(mainly due to exploding computation times)

note

for *bf-search*, the *fringe* should be implemented as a **FIFO queue**

for *df-search*, the *fringe* should be implemented as a **LIFO queue**

simple *python* code (`fringe` and `node2succs[node]` are lists)

```
### breadth-first node selection and fringe update
node   = fringe.pop(0)
fringe = fringe + node2succs[node]

### depth-first node selection and fringe update
node   = fringe.pop(0)
fringe = node2succs[node] + fringe
```

further uninformed search strategies

uniform cost search

$\Leftrightarrow$ instead of expanding the shallowest node, expand a node of currently *lowest path cost*

if all step costs are equal, this is *breadth-first search*

depth limited search

$\Leftrightarrow$ expand deepest node only if $l < \lambda$

if $\lambda = \infty$, this is *depth-first search*

if $\lambda < \infty$, this avoids problems due to unbounded m

if $\lambda < d$, depth-limited search will always be incomplete

iterative deepening depth-first search

$\Leftrightarrow$ depth limited search with gradually increasing depth limit

combines benefits of *breadth-first* and *depth-first search* (completeness and low memory costs)

usually considered the “best uniformed search strategy” when b and m are large and d is unknown

beam search

$\Leftrightarrow$ expand only the k most promising nodes in level l

memory efficient but only works for *known goals*

bidirectional search

run two searches

forward from the *initial state*

and simultaneously

backward from the *goal state*

stop searching, once the two *fringes* meet / intersect

memory and time efficient but only works for *known goals*

remarks

for further details on *uninformed search strategies*, consult chapter 3 in

S. Russell and P. Norvig, *Artificial Intelligence*, 4th edition, **2021**

we shall mostly ignore those, because —after all— *uninformed search* is just *uninformed search*

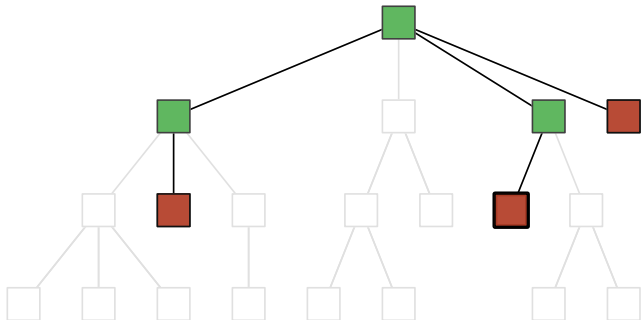
but there is one more thing . . .

random search

randomly expand the fringe one node at a time

for *small* b and D, this may work surprisingly well

for *large* b and D, this usually goes horribly wrong



however . . .

the above example suggests two directions for where to go from here

however . . .

the above example suggests two directions for where to go from here

Monte Carlo tree search

$\Leftrightarrow$ use information *systematically* gathered over many *random* expansions

informed search strategies

$\Leftrightarrow$ replace *mechanical* or *random* selection of node n by *informed decisions*

note: the former is incredibly important for this course and will be studied next time
the latter is incredibly important for path planning and will be studied now . . .

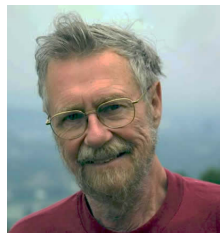
path planning as tree search

Dijkstra's algorithm

Dijkstra's algorithm

$\Leftrightarrow$ an *informed tree search algorithm* for computing the *lengths* of all shortest paths emanating from a source vertex src in a finite graph $G = (V, E)$ where

- G is connected
- G is either directed or undirected
- G is weighted such that $w(v_i, v_j) = w_{ij} > 0$



E.W. Dijkstra
($*1930$, $\dagger 2002$)
Turing Award 1972

Dijkstra's algorithm – main idea

informed selection of vertex v to be expanded

node selection considers *costs* $dist(v)$ where

$dist(v) \equiv$ path length from src to vertex v

observe that $dist(v)$ is a *numerical feature* of v

Dijkstra's algorithm – in words

initialize "dictionary" of distances

$$dist[v] = \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$$

initialize *fringe* and *closed list*

$$fringe = V$$
$$closed = \varnothing$$

while $fringe \neq \varnothing$

select $fringe$ node $v \leftarrow \text{argmin}_u\, dist[u]$
remove v from $fringe$ and add it to $closed$
examine all neighbors v' of v not in $closed$
if $dist[v'] > dist[v] + w_{vv'}$, update $dist[v'] \leftarrow dist[v] + w_{vv'}$

$\Rightarrow$ dictionary of *geodesic distances* between src and every v in G

note

Dijkstra's algorithm computes (step) distances between vertices

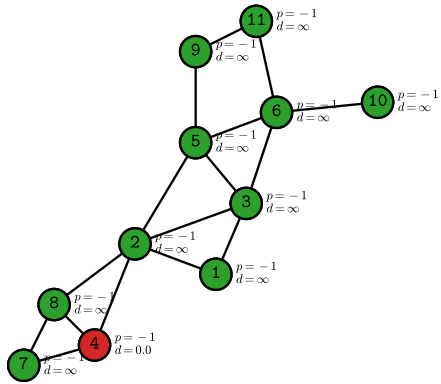
but we can modify it to become a shortest path finding algorithm . . .

shortest path from source src to target tgt

function *Dijkstra*(G, src, tgt)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return $dist$, $pred$



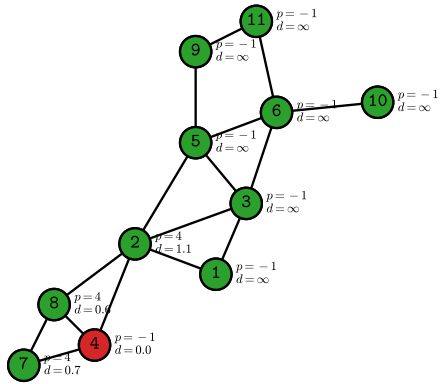
finding the shortest path from v_4 to v_{11}

shortest path from source src to target tgt

function *Dijkstra*(G, src, tgt)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return $dist$, $pred$



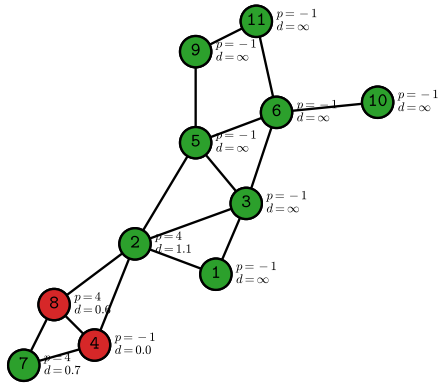
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(G, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



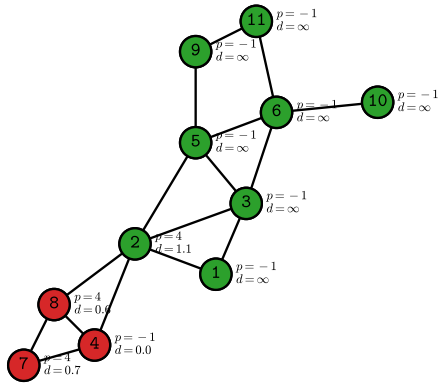
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(*G*, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



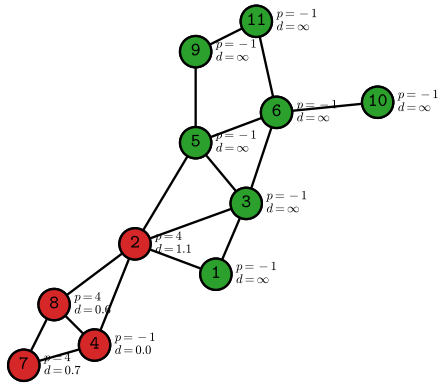
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(*G*, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



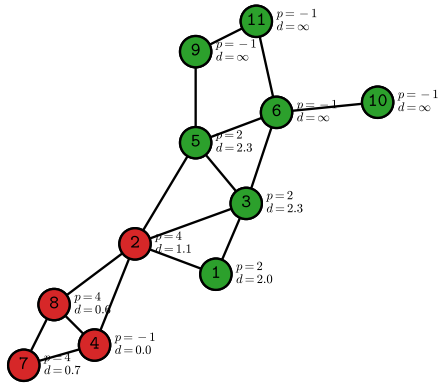
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(*G*, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



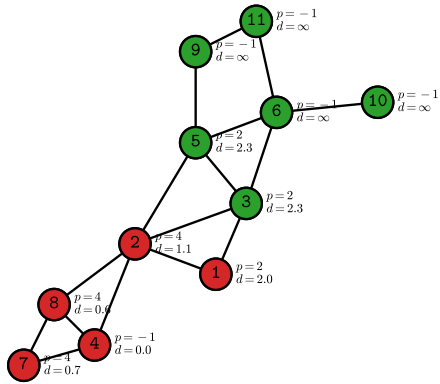
finding the shortest path from v_4 to v_{11}

shortest path from source src to target tgt

function *Dijkstra*(G, src, tgt)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return $dist$, $pred$



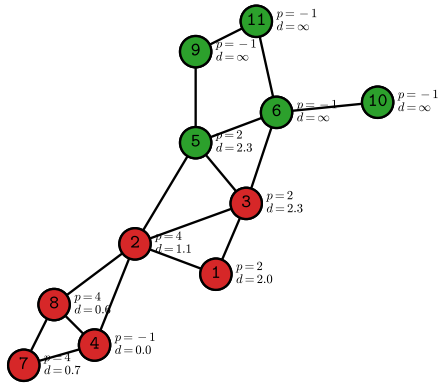
finding the shortest path from v_4 to v_{11}

shortest path from source src to target tgt

function *Dijkstra*(G, src, tgt)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return $dist$, $pred$



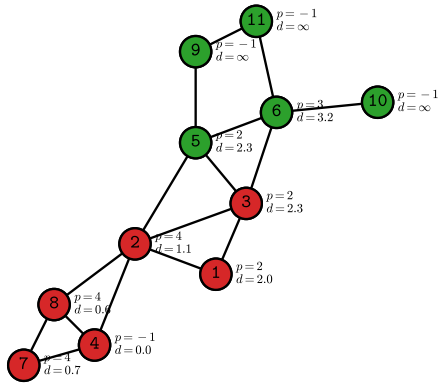
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(G, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



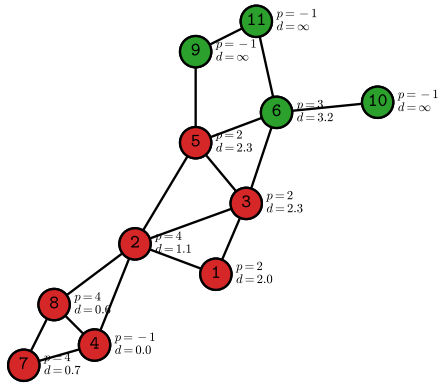
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(*G*, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



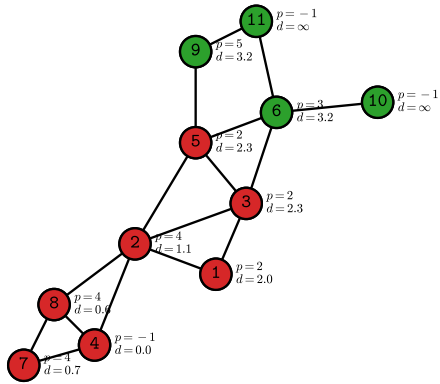
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(*G*, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



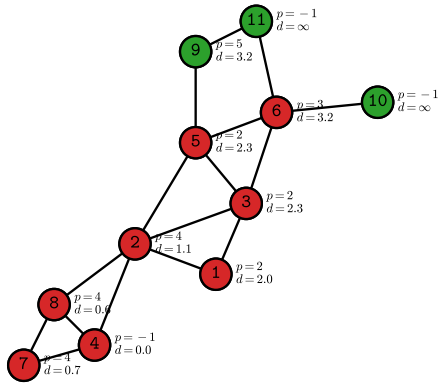
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(*G*, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



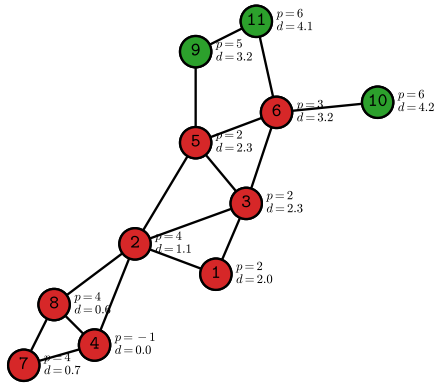
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(*G*, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



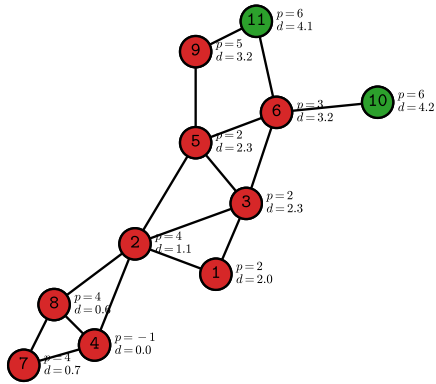
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(G, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



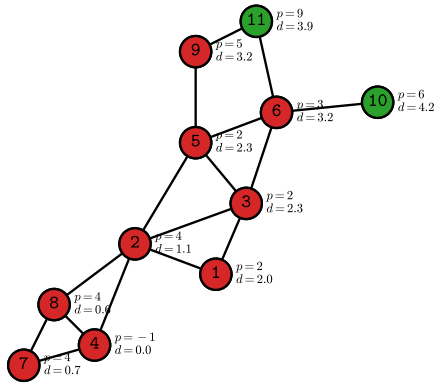
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(*G*, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



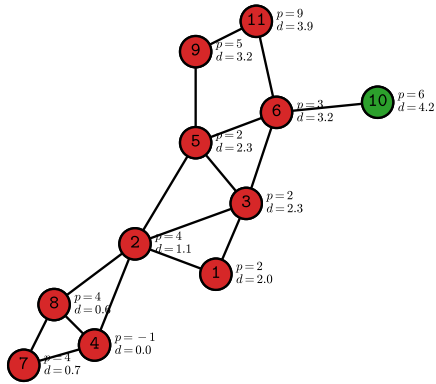
finding the shortest path from v_4 to v_{11}

shortest path from source *src* to target *tgt*

function *Dijkstra*(G, *src*, *tgt*)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases}$
    $pred[v] \leftarrow -1$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow V$

  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}_u \{dist[u] \mid u \in fringe\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      if $dist[v'] > dist[v] + w_{vv'}$
        $dist[v'] \leftarrow dist[v] + w_{vv'}$
        $pred[v'] \leftarrow v$

  return *dist*, *pred*



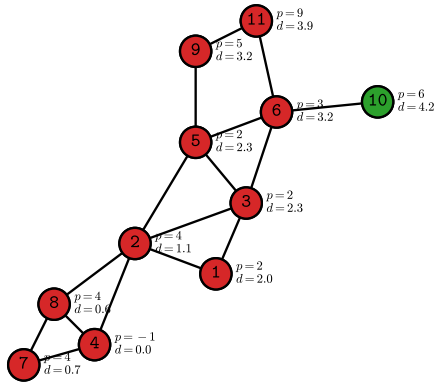
finding the shortest path from v_4 to v_{11}

reverse iteration to retrieve path from source to target

function *ShortestPath*(*pred*, *src*, *tgt*)
    // initialize an empty sequence
    $path \leftarrow \epsilon$

    while $pred[tgt] \neq -1$
        $path \leftarrow tgt : path$
        $tgt \leftarrow pred[tgt]$

    return $src : path$



$$path = [4, 2, 5, 9, 11]$$

remarks

Dijkstra's original algorithm (1956) runs in $O\big(|V|^2\big)$

Fredman and Tarjan (1984) introduce *Fibonacci heaps* and achieve $O\big(|E| + |V| \log |V|\big)$

$\Rightarrow$ very fast on directed graphs

further improvements w.r.t. finding shortest path from src to tgt

try to avoid examining nodes that lie in the "wrong direction" (such as nodes 7 and 8 in the above example)

$\vdots$

the A^* algorithm

the A^* algorithm

$\Leftrightarrow$ an *informed tree search* algorithm for computing the shortest path from a source vertex src to a target vertex tgt in a finite graph $G = (V, E)$ where

- G is connected
- G is either directed or undirected
- G is weighted such that $w(v_i, v_j) = w_{ij} > 0$

P.E. Hart, N.J. Nilsson, and B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Trans. Systems Science and Cybernetics, 4(2), **1968**

the A^* algorithm – main idea

informed selection of vertex v to be expanded

selection considers heuristic cost $f(v)$ where

$f(v) = g(v) + h(v)$

$g(v) \equiv$ path length (step costs) from src to v

$h(v) \equiv$ estimate of the future path length from v to tgt

$\Leftrightarrow$ we have

$f(v) =$ *estimated* cost of cheapest solution *through* v

observe that $f(v)$, $g(v)$, and $h(v)$ are *numerical features* of v

the A^* algorithm – in words

initialize "dictionaries" of distances (g) and *heuristic costs* (f)

$$g[v] = \begin{cases} 0 & \text{if } v = src \\ \infty & \text{otherwise} \end{cases} \qquad f[src] = g[src] + h(src)$$

initialize *fringe* and *closed list*

$$fringe = \{src\} \qquad closed = \varnothing$$

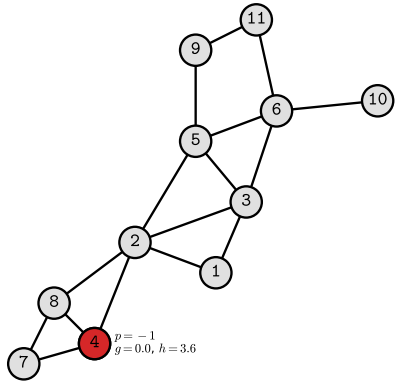
while $fringe \neq \varnothing$

select $fringe$ node u with smallest $f[u]$ and stop, if $u = tgt$
remove u from $fringe$ and add it to $closed$
examine all neighbors v of u not in $closed$
if $v \notin fringe \vee g[v] > g[u] + w_{uv}$, set $g[v] = g[u] + w_{uv}$, $f[v] = g[v] + h(v)$, and $p[v] = u$
if $v \notin fringe$, add v to $fringe$

$\Rightarrow$ *shortest path* / *geodesic* $src \xrightarrow{*} tgt$, if *heuristic* $h(\cdot)$ is admissible

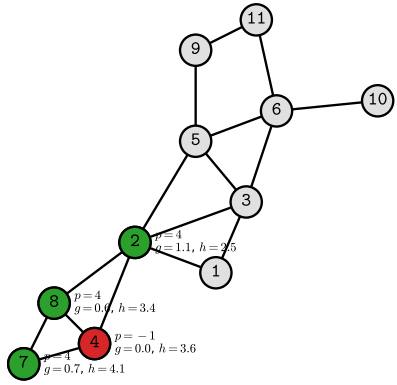
shortest path from source src to target tgt

```
function A*(G, src, tgt)
    closed ← ∅
    fringe ← {src}
    g[src] ← 0
    f[src] ← g[src] + h(src)
    while fringe ≠ ∅
        v ← argmin {f[u] | u ∈ fringe}
        if v = tgt
            break
        closed ← closed ∪ {v}
        fringe ← fringe \ {v}
        for v' ∈ Neib(v) \ closed
            g' ← g[v] + w_vv'
            if v' ∉ fringe ∨ g[v'] > g'
                g[v'] ← g'
                f[v'] ← g[v'] + h(v')
                pred[v'] ← v
                if v' ∉ fringe
                    fringe ← fringe ∪ {v'}
    return g, pred
```



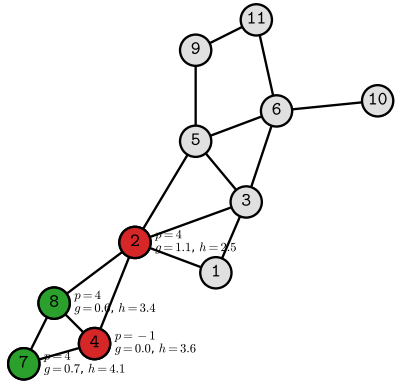
shortest path from source src to target tgt

function $A^*(G, src, tgt)$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow \{src\}$
  $g[src] \leftarrow 0$
  $f[src] \leftarrow g[src] + h(src)$
  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}\left\{f[u] \middle| u \in fringe\right\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      $g' \leftarrow g[v] + w_{vv'}$
      if $v' \notin fringe \bigvee g[v'] > g'$
        $g[v'] \leftarrow g'$
        $f[v'] \leftarrow g[v'] + h(v')$
        $pred[v'] \leftarrow v$
        if $v' \notin fringe$
          $fringe \leftarrow fringe \cup \{v'\}$
  return $g, pred$



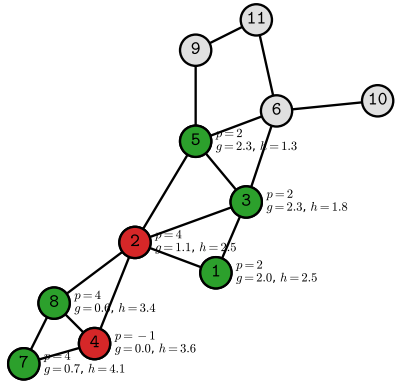
shortest path from source src to target tgt

function $A^*(G, src, tgt)$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow \{src\}$
  $g[src] \leftarrow 0$
  $f[src] \leftarrow g[src] + h(src)$
  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}\left\{f[u] \middle| u \in fringe\right\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      $g' \leftarrow g[v] + w_{vv'}$
      if $v' \notin fringe \bigvee g[v'] > g'$
        $g[v'] \leftarrow g'$
        $f[v'] \leftarrow g[v'] + h(v')$
        $pred[v'] \leftarrow v$
        if $v' \notin fringe$
          $fringe \leftarrow fringe \cup \{v'\}$
  return $g, pred$



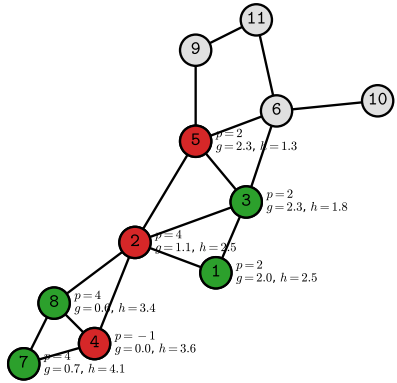
shortest path from source src to target tgt

function $A^*(G, src, tgt)$
- $closed \leftarrow \varnothing$
- $fringe \leftarrow \{src\}$
- $g[src] \leftarrow 0$
- $f[src] \leftarrow g[src] + h(src)$
- **while** $fringe \neq \varnothing$
  - $v \leftarrow \text{argmin}\left\{f[u] \,\middle|\, u \in fringe\right\}$
  - **if** $v = tgt$
    - **break**
  - $closed \leftarrow closed \cup \{v\}$
  - $fringe \leftarrow fringe \setminus \{v\}$
  - **for** $v' \in Neib(v) \setminus closed$
    - $g' \leftarrow g[v] + w_{vv'}$
    - **if** $v' \notin fringe \vee g[v'] > g'$
      - $g[v'] \leftarrow g'$
      - $f[v'] \leftarrow g[v'] + h(v')$
      - $pred[v'] \leftarrow v$
      - **if** $v' \notin fringe$
        - $fringe \leftarrow fringe \cup \{v'\}$
- **return** $g, pred$



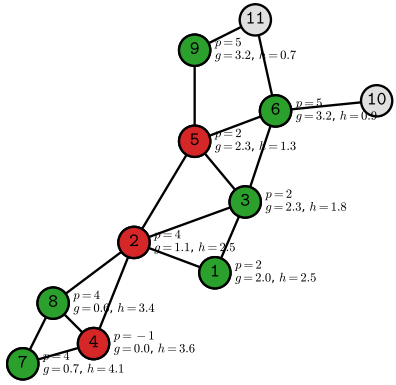
shortest path from source src to target tgt

function $A^*(G, src, tgt)$
  $closed \leftarrow \varnothing$
  $fringe \leftarrow \{src\}$
  $g[src] \leftarrow 0$
  $f[src] \leftarrow g[src] + h(src)$
  while $fringe \neq \varnothing$
    $v \leftarrow \text{argmin}\left\{f[u] \middle| u \in fringe\right\}$
    if $v = tgt$
      break
    $closed \leftarrow closed \cup \{v\}$
    $fringe \leftarrow fringe \setminus \{v\}$
    for $v' \in Neib(v) \setminus closed$
      $g' \leftarrow g[v] + w_{vv'}$
      if $v' \notin fringe \bigvee g[v'] > g'$
        $g[v'] \leftarrow g'$
        $f[v'] \leftarrow g[v'] + h(v')$
        $pred[v'] \leftarrow v$
        if $v' \notin fringe$
          $fringe \leftarrow fringe \cup \{v'\}$
  return $g, pred$



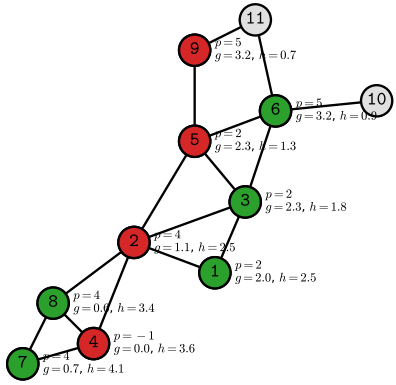
shortest path from source src to target tgt

function $A^*(G, src, tgt)$
- $closed \leftarrow \varnothing$
- $fringe \leftarrow \{src\}$
- $g[src] \leftarrow 0$
- $f[src] \leftarrow g[src] + h(src)$
- **while** $fringe \neq \varnothing$
  - $v \leftarrow \text{argmin}\left\{f[u] \,\middle|\, u \in fringe\right\}$
  - **if** $v = tgt$
    - **break**
  - $closed \leftarrow closed \cup \{v\}$
  - $fringe \leftarrow fringe \setminus \{v\}$
  - **for** $v' \in Neib(v) \setminus closed$
    - $g' \leftarrow g[v] + w_{vv'}$
    - **if** $v' \notin fringe \bigvee g[v'] > g'$
      - $g[v'] \leftarrow g'$
      - $f[v'] \leftarrow g[v'] + h(v')$
      - $pred[v'] \leftarrow v$
      - **if** $v' \notin fringe$
        - $fringe \leftarrow fringe \cup \{v'\}$
- **return** $g, pred$



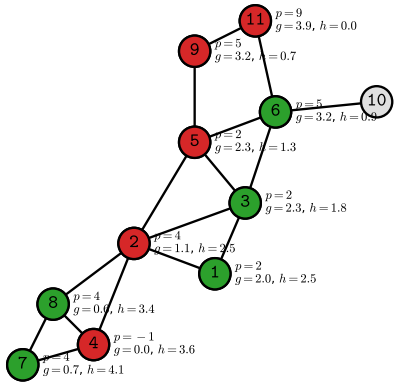
shortest path from source *src* to target *tgt*

function $A^*(G, src, tgt)$
- $closed \leftarrow \varnothing$
- $fringe \leftarrow \{src\}$
- $g[src] \leftarrow 0$
- $f[src] \leftarrow g[src] + h(src)$
- **while** $fringe \neq \varnothing$
  - $v \leftarrow \text{argmin}\left\{f[u] \middle| u \in fringe\right\}$
  - **if** $v = tgt$
    - **break**
  - $closed \leftarrow closed \cup \{v\}$
  - $fringe \leftarrow fringe \setminus \{v\}$
  - **for** $v' \in Neib(v) \setminus closed$
    - $g' \leftarrow g[v] + w_{vv'}$
    - **if** $v' \notin fringe \bigvee g[v'] > g'$
      - $g[v'] \leftarrow g'$
      - $f[v'] \leftarrow g[v'] + h(v')$
      - $pred[v'] \leftarrow v$
      - **if** $v' \notin fringe$
        - $fringe \leftarrow fringe \cup \{v'\}$
- **return** $g, pred$



shortest path from source src to target tgt

function $A^*(G, src, tgt)$
- $closed \leftarrow \varnothing$
- $fringe \leftarrow \{src\}$
- $g[src] \leftarrow 0$
- $f[src] \leftarrow g[src] + h(src)$
- **while** $fringe \neq \varnothing$
  - $v \leftarrow \text{argmin}\left\{f[u] \middle| u \in fringe\right\}$
  - **if** $v = tgt$
    - **break**
  - $closed \leftarrow closed \cup \{v\}$
  - $fringe \leftarrow fringe \setminus \{v\}$
  - **for** $v' \in Neib(v) \setminus closed$
    - $g' \leftarrow g[v] + w_{vv'}$
    - **if** $v' \notin fringe \bigvee g[v'] > g'$
      - $g[v'] \leftarrow g'$
      - $f[v'] \leftarrow g[v'] + h(v')$
      - $pred[v'] \leftarrow v$
      - **if** $v' \notin fringe$
        - $fringe \leftarrow fringe \cup \{v'\}$
- **return** $g, pred$



note

the A^* algorithm is a *best-first* (tree) search algorithm

in spatial path finding, the heuristic $h(v)$ is usually the Euclidean distance

$$h(v) = \|\boldsymbol{x}_v - \boldsymbol{x}_{tgt}\|$$

where $\boldsymbol{x}_v$ and $\boldsymbol{x}_{tgt}$ are spatial coordinates of v and tgt

if $h(v) = c$ for all v, A^* becomes Dijkstra's algorithm

note

a heuristic $h(v)$ is admissible, if it does *not over estimate* future path costs

$\Leftrightarrow$ a heuristic $h(v)$ is admissible, if

$$h(v) \leqslant C\left(v \xrightarrow{*} tgt\right)$$

where $C\left(v \xrightarrow{*} tgt\right)$ denotes the *true* path costs from vertex v to target tgt

note

one can show that, if $h(v)$ is admissible, then the A^* algorithm is *optimal* and *complete*

for spatial path planning, the Euclidean distance $||\boldsymbol{x}_v - \boldsymbol{x}_{tgt}||$ is an admissible heuristic

this makes A^* the *de-facto* algorithm for route planning (on Google maps etc.)
(it is typically run using bidirectional search (starting at the source and at the target))

note

Lemma 1

If h is admissible, then A^* is optimal

Lemma 2

if $b < \infty$ and $w_{ij} > \varepsilon > 0$ for all v_i, v_j, then A^* is complete

proof of Lemma 1 (sketch)

let v_* be an *optimal* goal state with costs

$$f(v_*) = g(v_*) + \underbrace{h(v_*)}_{=0} = g(v_*)$$

let $v_\circ$ be a *suboptimal* goal state such that

$$f(v_*) < f(v_\circ) = g(v_\circ) + \underbrace{h(v_\circ)}_{=0} = g(v_\circ)$$

finally, let v be a fringe state on the optimal path from v_{src} to v_* so that, if h is admissible, $f(v_*) \geqslant f(v)$ taken together, all of this says

$$f(v) \leqslant f(v_*) < f(v_\circ)$$

so that A^* will not expand $v_\circ$, i.e. always finds an optimal solution before considering a subotimal one $\square$

proof of Lemma 2 (sketch)

let v_* be an *optimal* goal state and $f(v_*)$ be the cost of reaching it

A^* cannot reach v_*, whenever

$$\left|\{v \mid f(v) \leqslant f(v_*)\}\right| = \infty.$$

but this can only happen, if

- the search tree has an infinite branching factor
- there is a path of finite costs but infinitely many nodes

however, under the assumptions of the lemma, neither can occur $\square$

remarks

if $h(\cdot)$ is admissible, i.e. *under*-estimates future path costs, then

A^* produces an *optimal solution*

$f(v) = g(v) + h(v)$ is biased towards $g(v)$

$\Rightarrow$ expansion favors nodes far from tgt

$\Leftrightarrow$ slow

if $h(\cdot)$ is inadmissible, i.e. *over*-estimates future path costs, then

A^* is not guaranteed to produce an *optimal solution*

$f(v) = g(v) + h(v)$ is biased towards $h(v)$

$\Rightarrow$ expansion favors nodes close to tgt

$\Leftrightarrow$ either very fast or terribly slow

remarks

let h_i and h_j be two admissible heuristics, h_i dominates h_j if $\forall\, v : h_i(v) \geqslant h_j(v)$

"one can show" that, if h_i dominates h_j, then A^* with heuristic h_i will expand fewer nodes on average than A^* with heuristic h_j

$\Rightarrow$ when facing a choice of admissible heuristics $h_1, \ldots, h_k$, one should opt for

$$h = \max\{h_1, \ldots, h_k\}$$

note

"one can show" that A^* is optimally efficient for any given heuristic function h

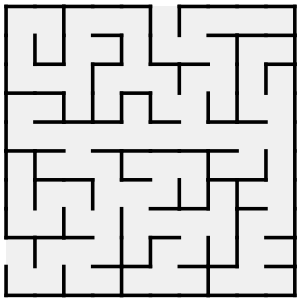
$\Leftrightarrow$ no other optimal algorithm applying heuristic h can be guaranteed to expand fewer nodes than A^* (if we ignore situations which necessitate (random) tie-breaking)

note: we could say much more about A^*, but will leave it at this

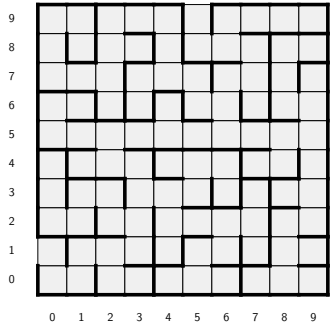
one more thing . . .

well, several actually

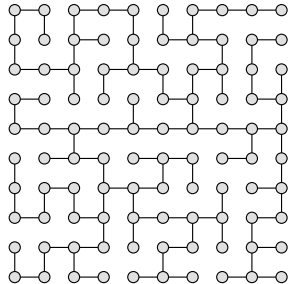
mazes



a 2D maze

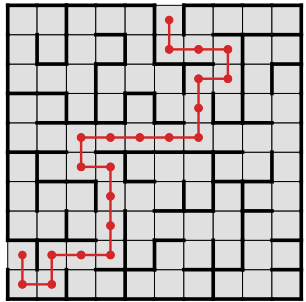


consists of cells $[x, y] \in \mathbb{N}^2$

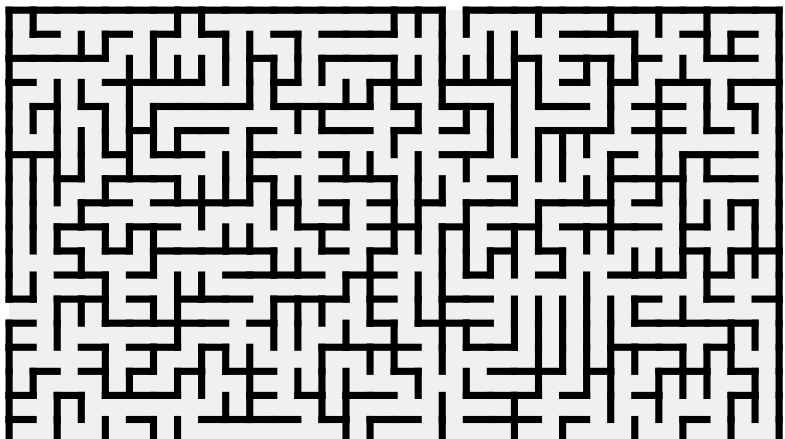


and is also an undirected graph

$\Rightarrow$ if src and tgt are known, then A^* easily finds the shortest path $src \xrightarrow{*} tgt$



if that seemed too easy, we can always up the ante ;-)



note

if we know the goal state of a problem, we typically wield tremendous power over said problem

if we know how to properly utilize *fringe* and *closed list*, then we are the masters of problem solving

for instance . . .

the flood fill algorithm

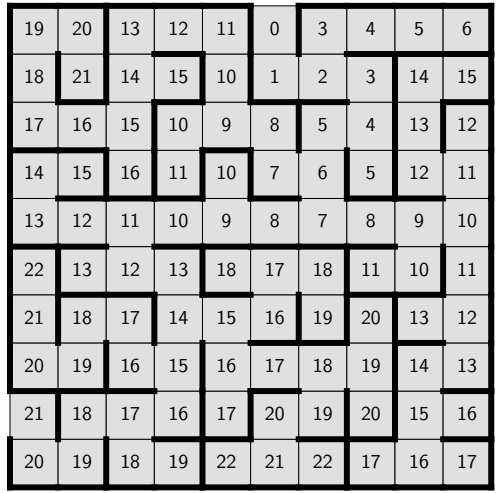
function *FloodFill*(G, tgt)
  for $v \in G$
    $dist[v] \leftarrow \begin{cases} 0 & \text{if } v = tgt \\ \infty & \text{otherwise} \end{cases}$

  $closed \leftarrow \varnothing$
  $fringe \leftarrow \{tgt\}$

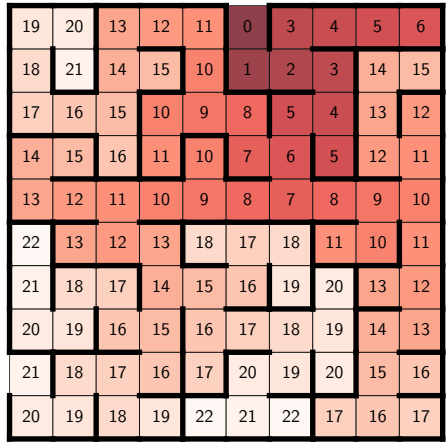
  while $fringe \neq \varnothing$
    $u \leftarrow$ some $n \in fringe$

    $closed \leftarrow closed \cup \{u\}$
    $fringe \leftarrow fringe \setminus \{u\}$

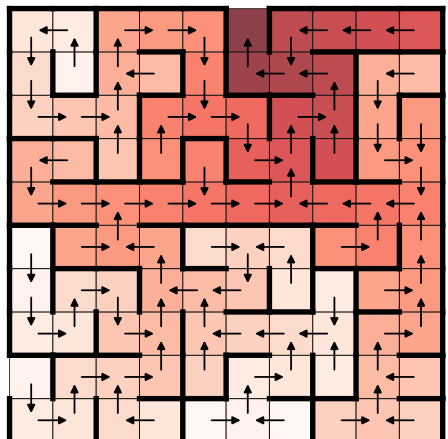
    for $v \in Neib(u) \setminus closed$
      $dist[v] = \min\big(dist[v], dist[u] + 1\big)$
      $fringe \leftarrow fringe \cup \{v\}$



in the language of physics, the algorithm produces a (discrete) potential field over a (discrete) domain



this potential field induces a (discrete) gradient field
over the cells of the maze



note

$\Rightarrow$ **just like that, we have found a *policy* that tells us what action to execute in which state in order to quickly reach a (long term) goal state**

AI / physics		RL
potential field	$\sim$	value function

summary

we now know about

(well defined) problems

tree search for planning / problem solving

(optimal) solutions, fringes, and search strategies

common uninformed tree search algorithms and
their characteristics (strengths and weaknesses)

the most important informed tree search algorithm — **the A^* algorithm**

a peculiar connection between tree search, potential fields, and policies

turn page for final words of wisdom . . .

note

those who know how to work with *fringe* and *closed lists*
are the kings and queens of classical AI

A^* is one of the top 10 most important algorithms presently known to humankind