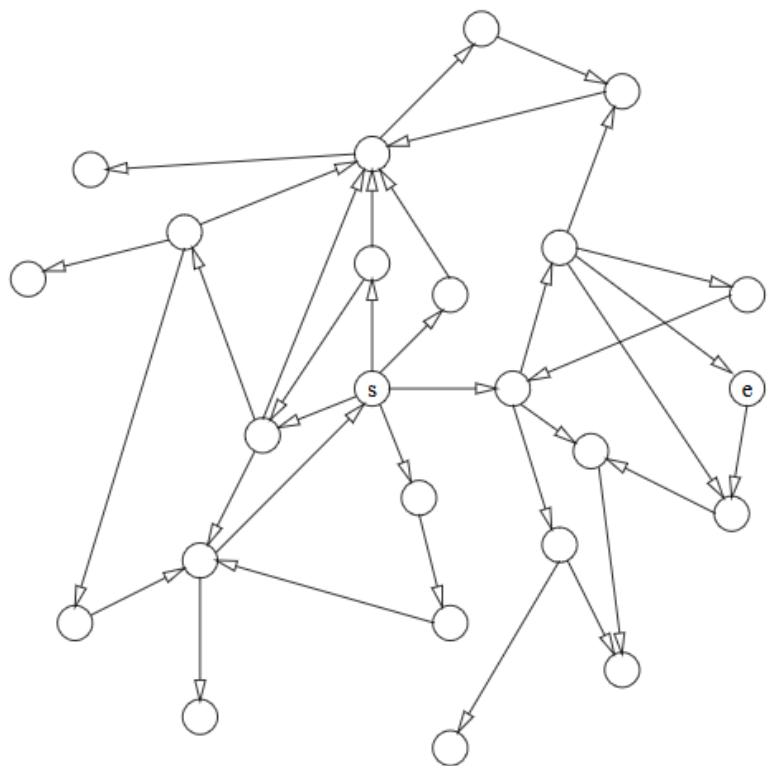


A*

breadth-first search
with heuristics to direct it

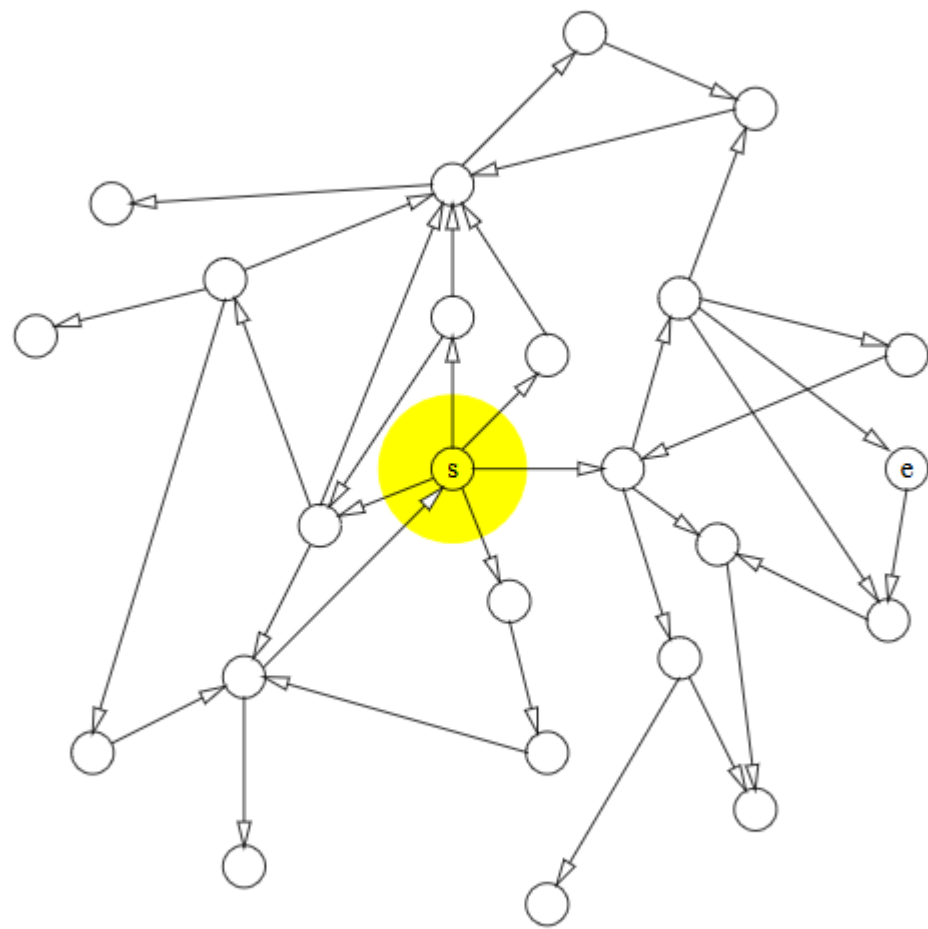
example problem



go from *s* to *e*

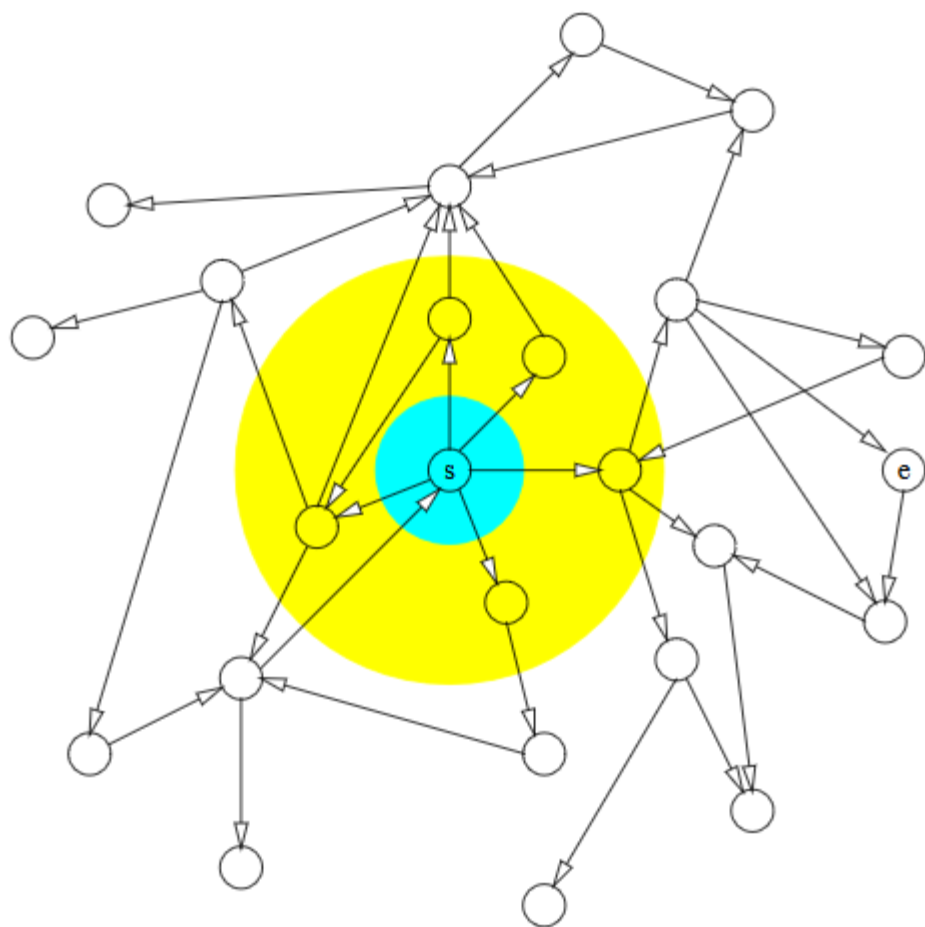
shortest path
Dijkstra's algorithm

dijkstra

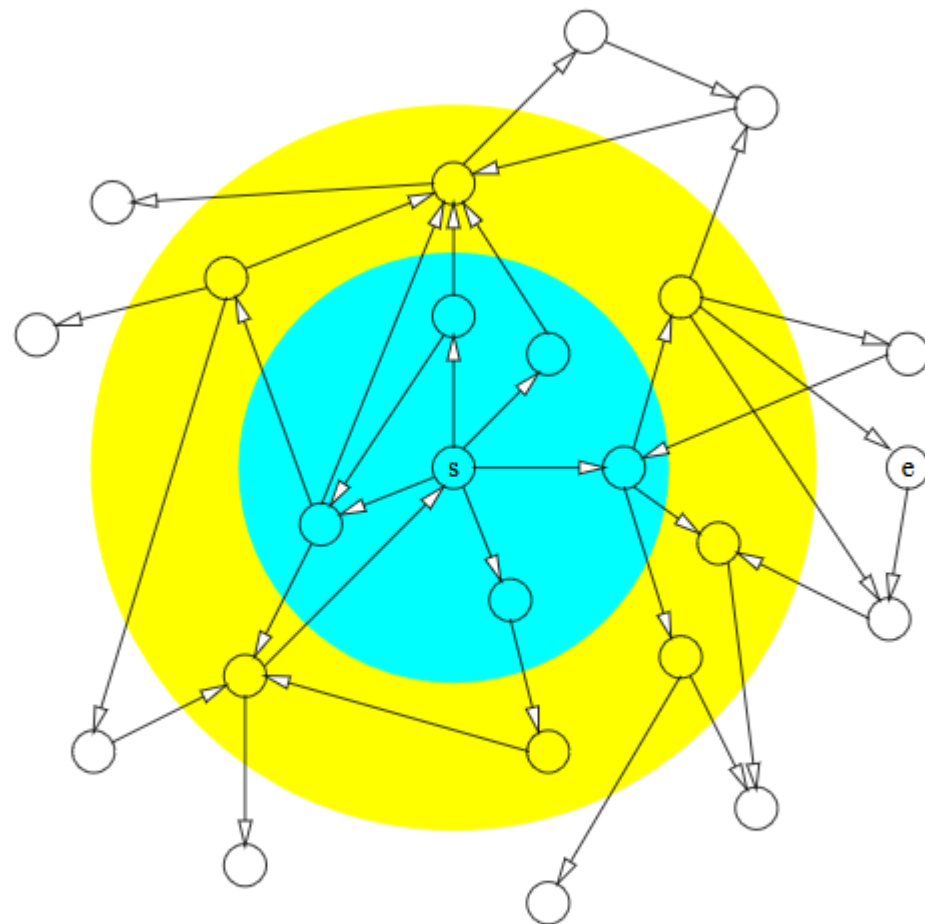


start from the initial state

dijkstra: expand



dijkstra: expand again



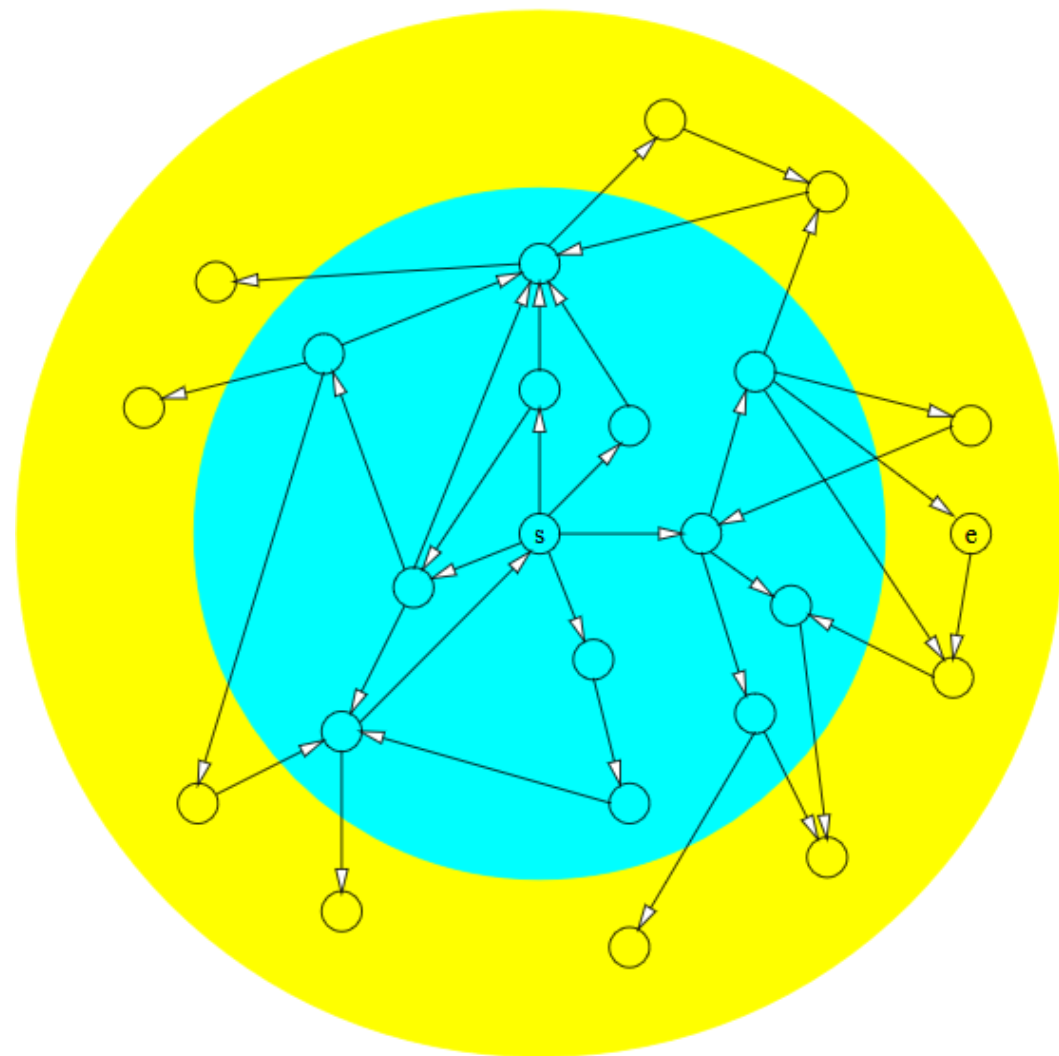
expand by following the arrows in all directions

mark previous node as already considered

expand the yellow area by following arrows

also expand the inner area (already considered nodes)

dijkstra: expand and reach the goal

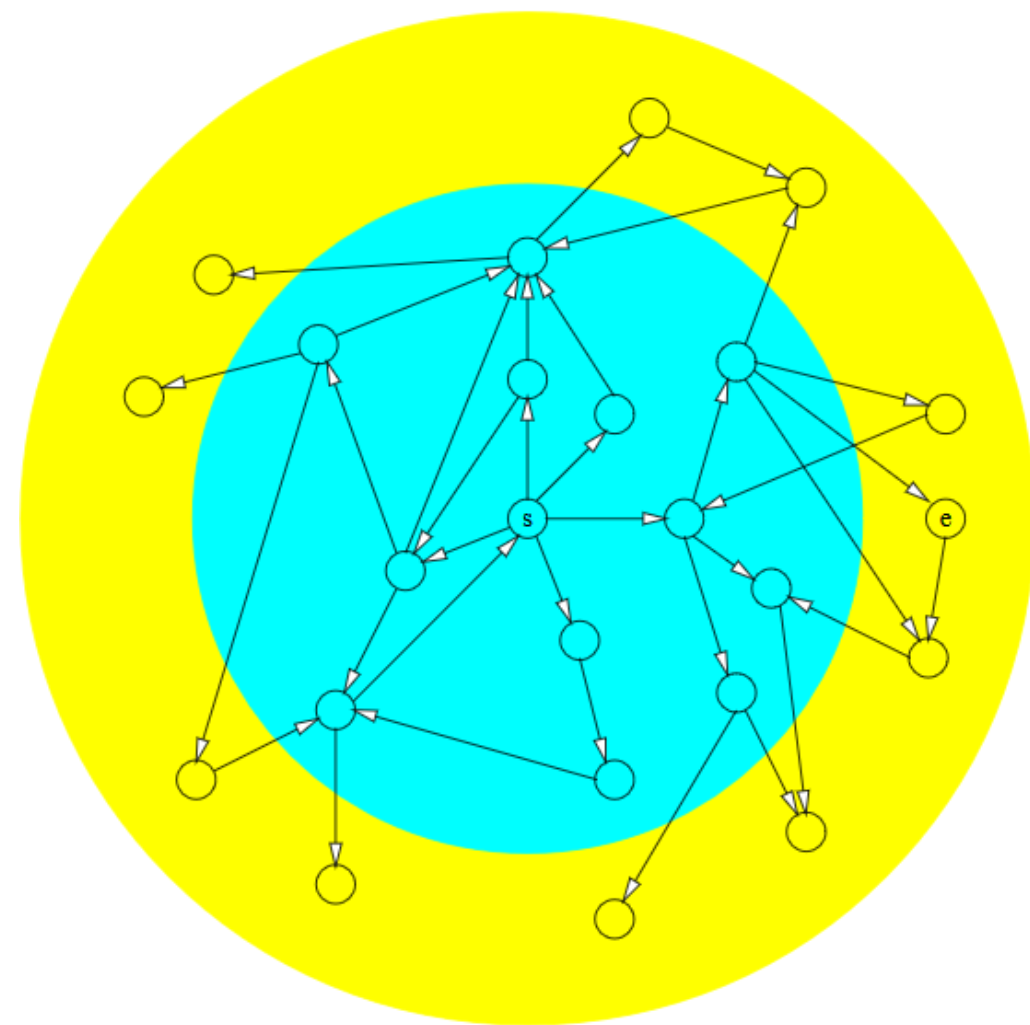


again

node e is reached

actual algorithm also keeps track of arrows followed
to know how e was reached

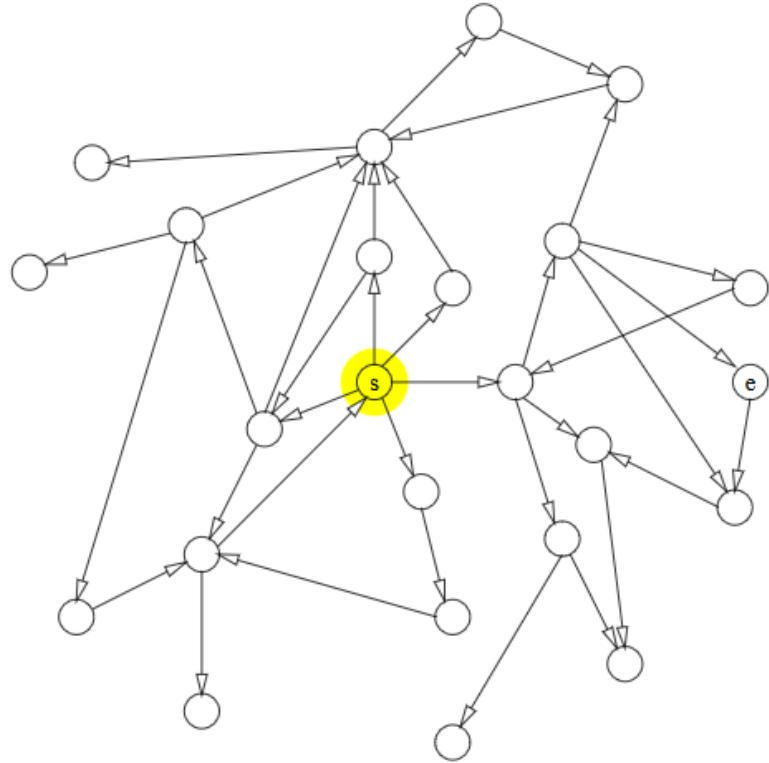
directions



the algorithm starts from s
then expand the circles in every direction

- the number of states may be exponential
PDDL: number of states exponential in the number of predicates
- does not keep into account hints
in this case: the destination e was on the right of s

informed search

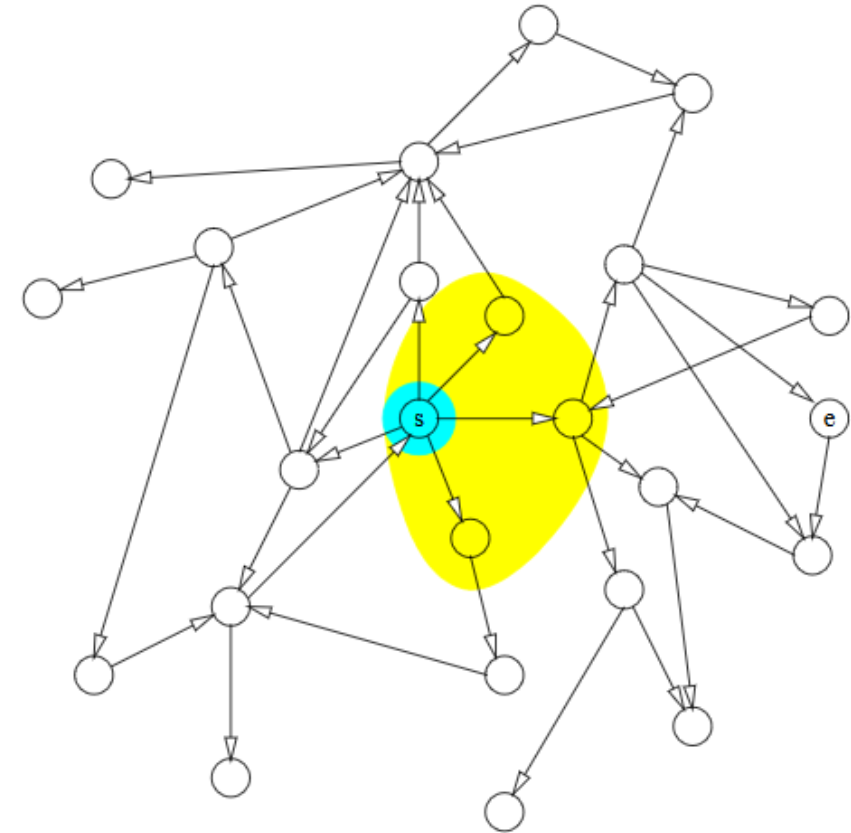


start from the beginning

target is on the right

prefer moving to the right

informed search: expand

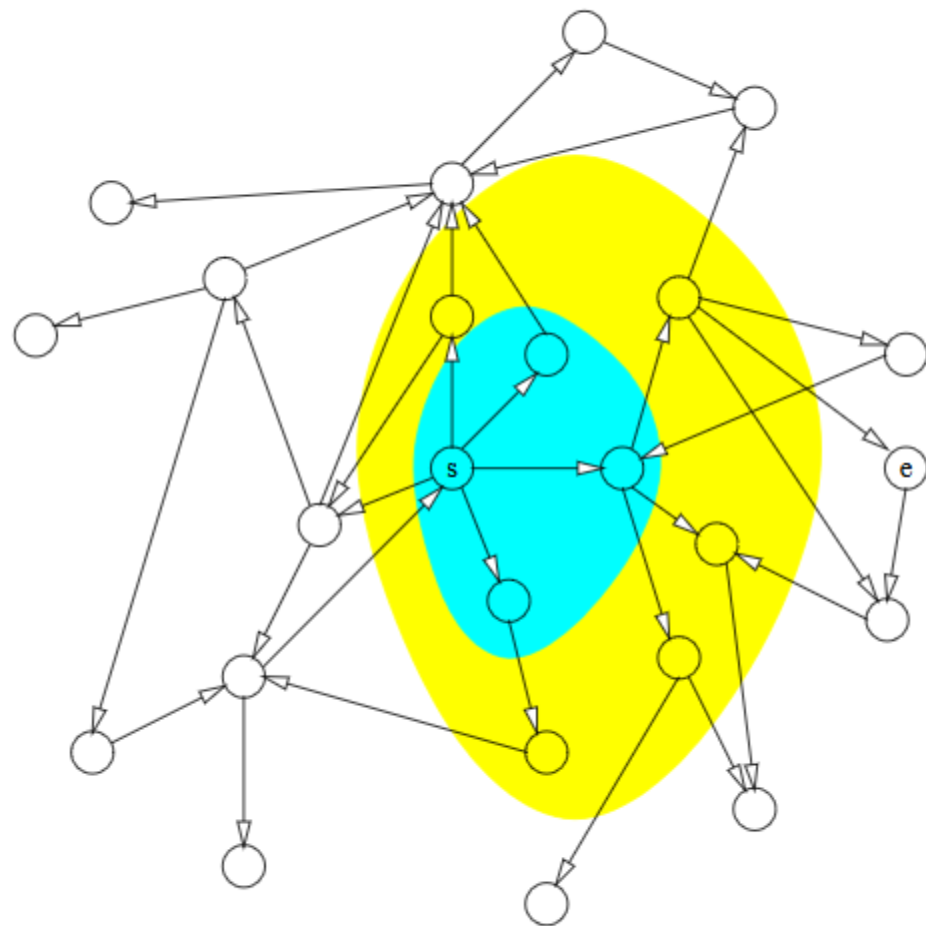


first expansion

unbalanced to the right

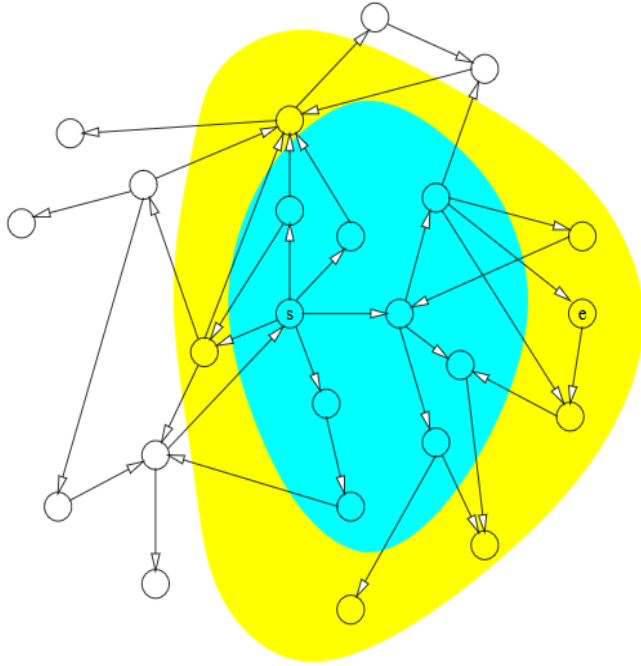
[note] This is not yet A*. Is only an illustration of how the hint “the target is not the right” could be taken into account so to speed up the search and reduce the number of stored nodes.

informed search: expand again



second expansion

informed search: expand and reach the goal



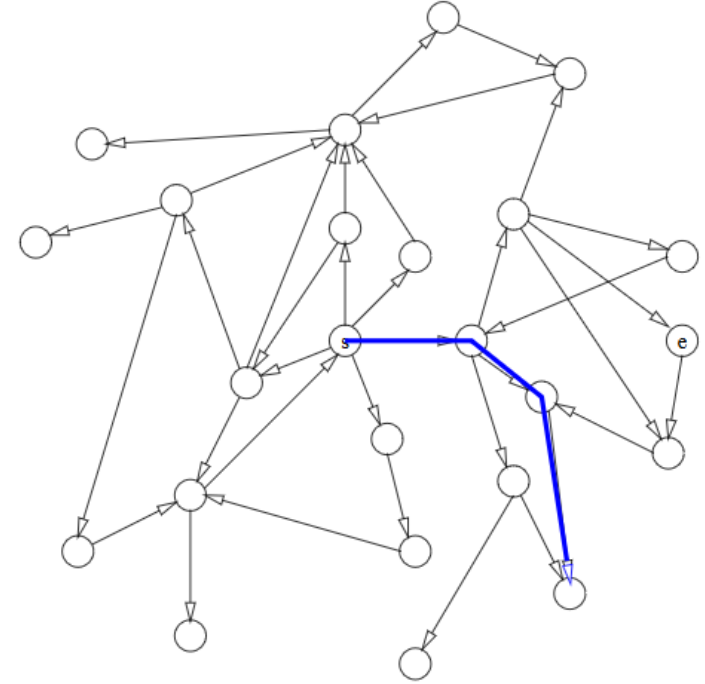
third expansion

node e reached

less nodes considered \Rightarrow less memory used, less time spent to analyze them



just move to the right!



always move to the node furthest on the right

may end up in a dead end

can be done, but needs a way to escape dead ends
(backtracking or learning)

[note] This example is just for illustrating how the general idea of breadth-first informed search works. The progression of the yellow and cyan areas does not mimic that of any specific algorithm explained in the sequel, such as A*.

heuristic search

heuristics: tells the most promising direction

in the example: right is better

search algorithms use it to improve the efficiency of the search

heuristics

domain-dependent

are specific to the problem

like in the example: "right is better"

only works because the ending node was on the right

domain-independent

not specific to a particular problem

example: states are propositional interpretation,

heuristics is to prefer lower Hamming distance to the goal

admissible heuristics: never overestimate the distance to the goal

search algorithms

- A*
- IDA*
- AO*
- LPA*
- D*
- theta*
- ...

[note] Some are for general search (A* and IDA*), others solve related but different problems (AO* is for and-or graphs, theta* is for geometric search)

A*

$d(n)$ distance $start \Rightarrow n$

$h(n)$ estimated distance $n \Rightarrow goal$

$d(n) + h(n)$ = estimated distance $start \Rightarrow n \Rightarrow goal$

estimated distance from $start$ to $goal$ via n

the node n of minimal $d(n) + h(n)$ is the best candidate

algorithm complicated by some details

idea first shown on a preliminary version (preliminary-A*)

[note] Notation: $a \Rightarrow b$ is a path from a to b ; it may comprise multiple steps, such as $a \rightarrow c \rightarrow d \rightarrow b$.

Instead, $a \rightarrow b$ is a direct (one-step) link from a to b . In planning, it means that executing a single action once turns state a into state b .

preliminary-A*

main idea of A*

(but is not A*)

$d(n)$ distance $start \Rightarrow n$

$h(n)$ estimate distance $n \Rightarrow goal$

in a sum, $n \rightarrow m$ stands for the cost of this action

```
frontier={start}
until goal not in frontier
    n = node in frontier with minimal d(n) + h(n)
    frontier -= n
    for all n → m
        frontier += m
        d(m) = d(n) + n → m
```

works on trees

what if some node is reachable by two different paths?

[note] This is not A*, only an implementation of its main idea. The real A* solves its inability to correctly deal with nodes reachable by different paths from the start, as shown next.

multiple paths

do not go to nodes already analyzed:

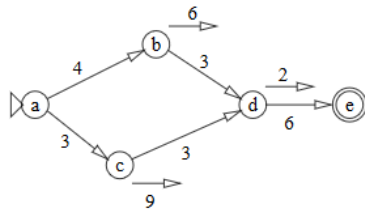
dijkstra

optimal

A*

not optimal

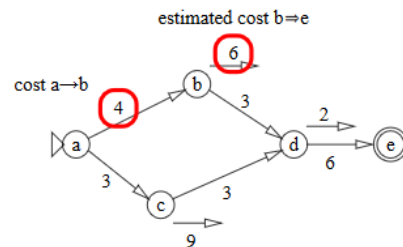
dijkstra vs. preliminary-A*



what dijkstra and preliminary-A* do in this example

in both cases: do not go to nodes already analyzed

meaning of numbers

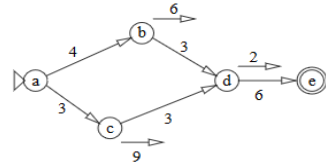


for example:

- $a \rightarrow b = 4$
- $h(b) = 6$

[note] The figures in these slides have numbers next to arrow to represent costs. The numbers on arrows not ending on a node represent the estimate $h(\text{node})$.

dijkstra



ignore heuristic distance

start node is a

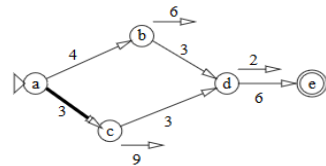
arrows that can be followed from {a}:

a→b and a→c

distance: 4 vs. 3

go to c

dijkstra

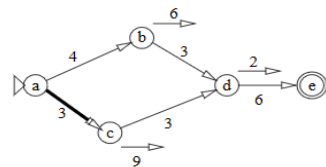


closest node is c

store the arrow a→c

shortest path to c

dijkstra



two choices: follow a→b or c→d

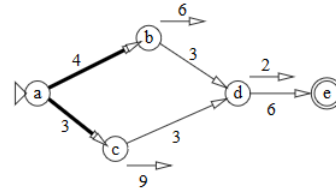
distance a→c→d = 3+3 = 6

distance a→b = 4

go to b



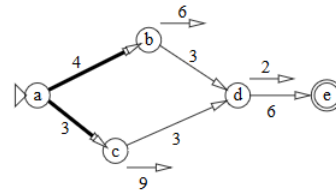
dijkstra



store arrow a→b

shortest path to b

dijkstra

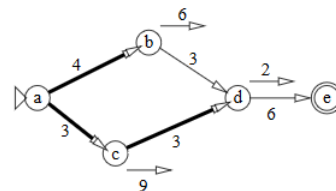


choose between b→d and c→d

distance is a→b→d=4+3=7 VS. a→c→d=3+3=6

arrow c→d is better

dijkstra

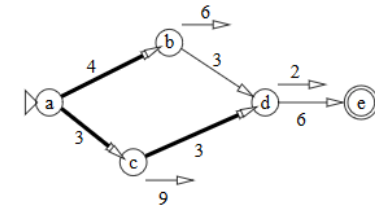


store c→d

note: really the shortest path to d!



dijkstra



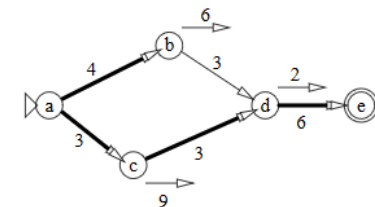
choose between b→d and d→e

a→b→d shorter than a→c→d→e, but a path to d already exists

do not store b→d

choose d→e instead

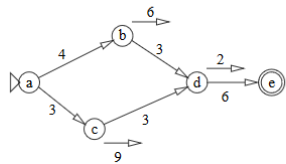
dijkstra



not only the goal is reached
the path is minimal

why: always reach nodes first by a shortest path
the first path reaching d was a→c→d = 3+3 = 6
shorter than the second path reaching it: a→b→d = 4+3 = 7

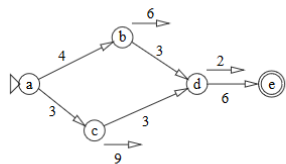
preliminary-A*



again, start from a
frontier is {a}

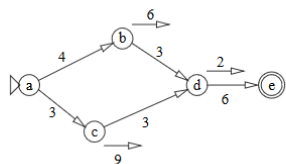
use the heuristics
do not go to nodes already taken out of the frontier

preliminary-A*



initialization
frontier = {a}

preliminary-A*



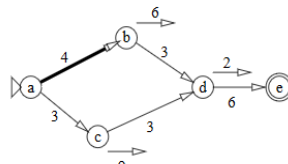
take a out of the frontier
add successors

frontier is now {b, c}
take out one of them: b or c?

$a \rightarrow b + h(b) = 4 + 6 = 10$
 $a \rightarrow c + h(c) = 3 + 9 = 12$

take b out of the frontier and analyze it
“to go b”

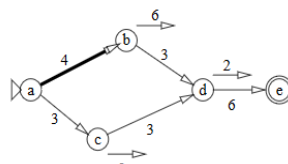
preliminary-A*



take out b from the frontier
add its successors (d) in its place

frontier is now {c, d}

preliminary-A*

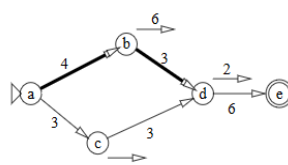


frontier is now {c, d}

 $a \rightarrow c + h(c) = 3 + 9 = 12$
 $a \rightarrow b \rightarrow d + h(d) = 4 + 3 + 2 = 9$

take d out of the frontier and analyze it
“go to d”

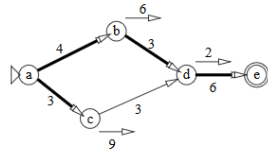
preliminary-A*



only successor of d is e
frontier is {c, e}

goal reached

goal reached, but...



goal reached, but not optimally!

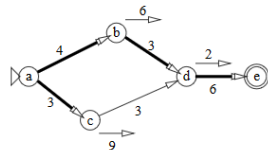
path found: $a \rightarrow b \rightarrow d \rightarrow e$, distance $4 + 3 + 6 = 13$

optimal was: $a \rightarrow c \rightarrow d \rightarrow e$, distance $3 + 3 + 6 = 12$

solution?

better: what was the problem?

the problem



frontier {c, e} contains the goal
but also c

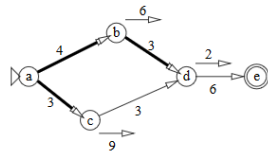
proceed: take c out of the frontier
only successor is d

known path to d has distance $a \rightarrow b \rightarrow d = 7$

distance via c is $a \rightarrow c \rightarrow d = 6$

the new path is shorter

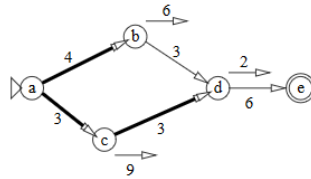
solution: reopen



remove b-d

store c-d instead

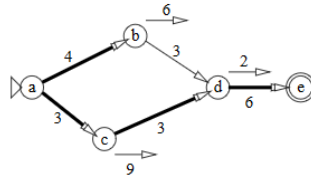
change path



now d is reached from c instead of b

continue from d

optimal path



path is now optimal

A*

$d(n)$ length of minimal *known-so-far* path $\text{start} \Rightarrow n$

$h(n)$ estimate distance $n \Rightarrow \text{goal}$

frontier nodes to be analyzed next

inner nodes already analyzed

```
inner={}
```

```
frontier={start}
```

```
d(m) =  $\infty$  for all m
```

```
d(start) = 0
```

```
until frontier not empty
```

```
    n = node in frontier with minimal  $d(n) + h(n)$ 
```

```
    frontier -= n
```

```
    inner += n
```

```
    for all  $n \rightarrow m$ 
```

```
        if m not in inner
```

```
            frontier += m
```

```
             $d(m) = \min(d(m), d(n) + n \rightarrow m)$ 
```

```
        if m in inner and  $d(n) + n \rightarrow m < d(m)$ 
```

```
            frontier += m
```

```
            inner -= m
```

```
             $d(m) = d(n) + n \rightarrow m$ 
```

when reaching m again,

if the length of the new path is shorter than the old

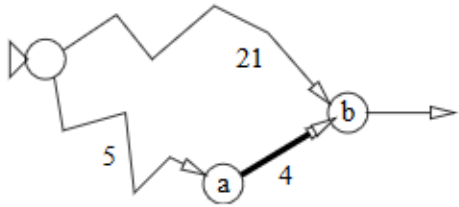
update $d(m)$

also remove m from inner

why?

[note] The algorithm is similar to the ones shown before, but deals with the cases of $n \rightarrow m$ where m is either in a frontier or inner node. Why these cases need special attention is shown next.

alternative roads



a path $\text{start} \Rightarrow b$ already known
node a now analyzed, and $a \rightarrow b$ exists

node b is reached again
shorter path ($5+4 < 21$)

dijkstra: impossible

b is always reached first by the shortest path
in this case, first $\text{start} \Rightarrow a \rightarrow b$, then the other path

A*: possible

happens if the heuristics misguided the algorithm first to path $\text{start} \Rightarrow b$ of cost 21
even if a shorter path existed

[note] A* may first reach a node by a path of some length and then by a shorter path. This is of course only a possibility, and the opposite case is possible as well.

Taking into account the case of nodes reached again is not always easy to fix, as shown next.

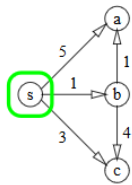
reaching a node again: the cases

when considering $n \rightarrow m$:

- m neither an inner nor a frontier node
(obvious: m has never been considered)
- m is in the frontier
(easy: m next to be considered)
- m is an inner node
(hard)

[note] If m is neither a frontier nor an inner node, then it has not been reached before. This case needs no special care. The other two are considered next.

reach again a node in the frontier



$$h(s) = h(a) = h(b) = h(c) = 1$$

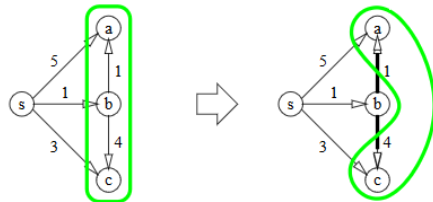
initially: $\text{frontier} = \{s\}$

best node in the frontier is s

take s out of the frontier

add its successors a , b and c

reach again a node in the frontier: the problem



$\text{frontier} = \{a, b, c\}$

best node b

take b out of the frontier

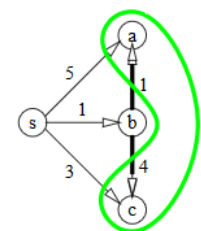
consider $b \rightarrow a$ and $b \rightarrow c$

update $d(a)$ and $d(c)$

$$d(a) = d(b) + b \rightarrow a = 1 + 1 \text{ (correct)}$$

$$d(c) = d(b) + b \rightarrow c = 1 + 4 \text{ (wrong because of } s \rightarrow c = 3)$$

reach again a node in the frontier: the fix



$d(b) + b \rightarrow c = 1 + 4$
but $d(c)$ was previously known to be 3

easy to solve:

- initially $d(c) = \infty$
- rather than $d(c) = d(b) + b \rightarrow c$
update by $d(c) = \min(d(c), d(b) + b \rightarrow c)$

easy because c not yet analyzed
only a candidate for being analyzed later
now: hard problem

[note] The situation where a node already in the frontier is reached again is easy to solve, since that node has not been analyzed yet, but is only a candidate for the subsequent analysis. All that it takes is to update its distance-from-start if it is lower than its currently know value.

reach again an inner node

m inner implies that a path from $start$ to m was already found

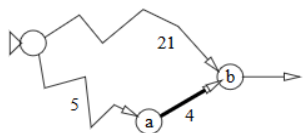
example:

- known distance so far $d(m) = 21$
- when analyzing node n :
 $d(n) = 5, n \rightarrow m = 4$

just updating $d(m) = 9$ is not enough

m is an inner node = its successors already went in the frontier
with a suboptimal value of $d(.)$

new path



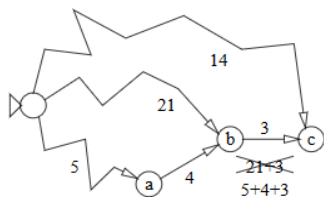
take a out of the frontier
arrow $a \rightarrow b$ exists: update $d(b)$

previously $d(b) = 21$
new path of length $5 + 4$ found

set $d(b) = 9$

but: b has successors!

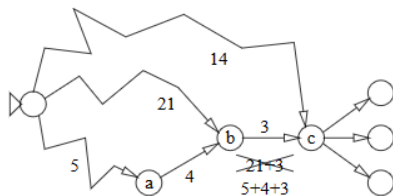
successor of updated node



previously known shortest path to c : length 14

new path: length $9 + 3 = 12$

update the successors?

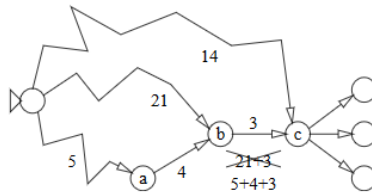


c may in turn have other successors

updating $d(c) = 9$ is not enough

[note] Unlike the case where the node reached again is in the frontier, when an inner node like b is reached again, updating it is not enough. It may have successors that have already been analyzed with the previous value of $d(b)$.

solution: reopen



when reaching b from a new path:

do like if b was just reached

put it back in the frontier
as a node yet to be analyzed

correct: is to be analyzed with the new value of $d(b)$

this triggers the update of its successor c
and the successors of c , etc.

reopening: cost

the same node b may be reached several times

each time, it has to be put back in the frontier

also its successors, etc.

some implementations do no reopen
give up optimality

[note] This closes the description of reopenings. The next slide is about a different issue, another consequence of the possibility that a node is first reached by a non-optimal path.

goal reached?

a node may be reached first by a non-optimal path
(unlike dijkstra)

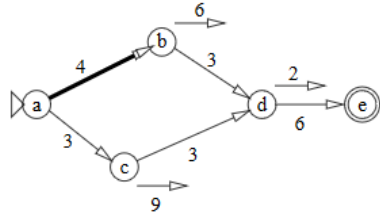
also the goal

do not stop when reaching the goal,
only when the frontier is empty

improvement:
if $d(n) \geq d(e)$, where e is the goal
remove n from the frontier

[note] Initially $d(e) = \infty$, and this value is only changed when e is reached for the first time. Until that, the condition $d(n) \geq d(e)$ is always false for a node n in the frontier.

what caused the problem?



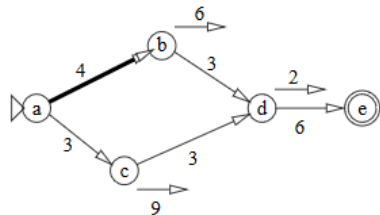
heuristics grossly underestimated the distance to the goal from d
 $h(d) = 2$ vs. really 6

but was correct in c
 $c \rightarrow d \rightarrow e = 3 + 6 = 9 = h(c)$

this made $b \rightarrow d$ preferred over $a \rightarrow c$

a better heuristics would have avoided the problem
or maybe just one that is always wrong in the same way?

inconsistent heuristics

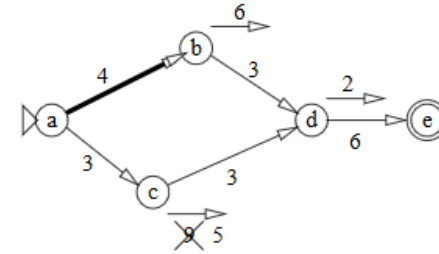


$h(c) = 9$
but
 $c \rightarrow d + h(d) = 3 + 2 = 5$

heuristics was *inconsistent* with itself:
estimates c to be at distance 9 to the goal
but estimates $c \rightarrow d$ plus distance from d to the goal to 5

consistent heuristics: $h(c) \leq c \rightarrow d + h(d)$
in this case, at most $h(c)$ could be at most $3 + 2 = 5$

consistent heuristics



if the heuristics were consistent,
 $h(c)$ could be at most 5
instead of 9

with this change, $a \rightarrow b \rightarrow d + h(d) = 4 + 3 + 2 = 9$
while $a \rightarrow c + h(c) = 3 + 5 = 8$

go to c
then to d via c

solutions

reopen

reconsider nodes reached again
costly: same node may be considered many times

use a consistent heuristics

nodes always reached first from their shortest paths
an accurate consistent heuristics may not be available

fix the heuristic, make it consistent


LRTA* does this, but also proceeds in a different way

otherwise: accept non-optimal solution
some A* implementations do this

domain-dependent heuristics

example: solve the 15-puzzle

8	7	10	13
3	15	14	2
6	9	5	12
1		4	11



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

state = position

change only by sliding a tile to the empty space


heuristics?

15-puzzle, heuristics 1

number of tiles not in the correct position


15-puzzle, heuristics 2

8	7	10	13
3	15	14	2
6	9	5	12
1		4	11



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

	15		




		15	

Manhattan distance from each tile to its goal position

sum over all tiles


15-puzzle, heuristics 3

8	7	10	13
3	15	14	2
6	9	5	12
1		4	11



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

-	-	-	13
3	-	-	2
-	9	5	-
1		4	-



1	2	3	4
5	-	-	-
9	-	-	-
13	-	-	-

consider only the tiles that go to the top and left borders
make the others blank (erase the number on them)

calculates moves to get them and the empty square in goal position

number of moves is heuristics for the original configuration

heap

store the frontier as an heap

key of node n is $d(n) + h(n)$

- get node of minimal key: logarithmic
- insert node: logarithmic
- remove node: logarithmic

reopening: may require updating non-minimal nodes

direct link from parent useful

[note] Heaps are data structures that store objects associated to values, and are optimized for finding and extracting an object of minimal value. The value is called the key.

If node m is in the heap (= is in the frontier) with key $d(m) + h(m)$ and a new path $start \Rightarrow n \rightarrow m$ with lower $d(n) + (n \rightarrow m) + h(m)$ is found, m has to be changed its key even if it is not the currently minimal node in the heap.

A*: terminology

- inner = closed nodes
- frontier = open nodes
- $d(.) = g(.)$ or $gScore$

[note] The commonly used terms used for the elements in the A* algorithm.