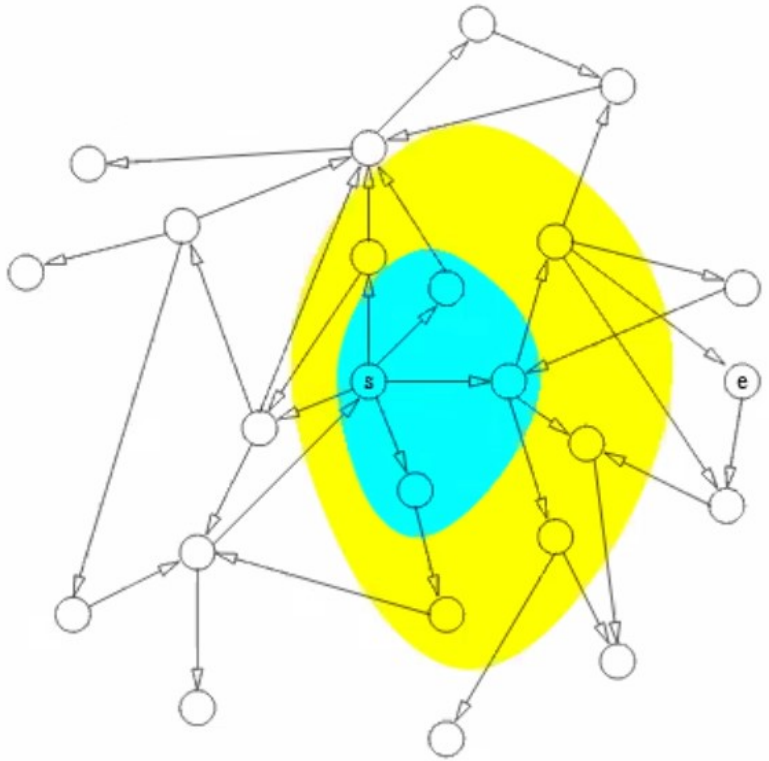
iterative deepening



problem with A*:

- tries to proceed toward the goal
- still, frontier may be large

state variables = exponential number of states

time vs. space

large time
    may be allowed
    example: planning moving clearing an area of the ocean
    execution takes days, large time for planning allowed
large space
    not enough memory = no solution

trade time for memory

solution:

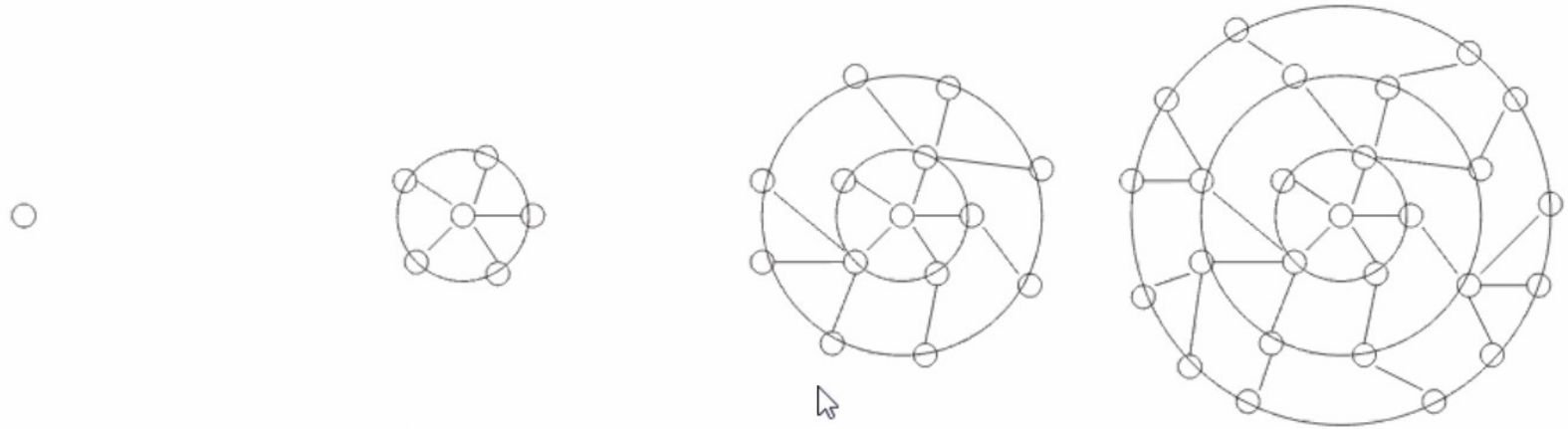
- spend more time
- save memory

iterative deepening

- start from initial state
- reach all states at distance one
- reach all states at distance two
- reach all states at distance three
- …

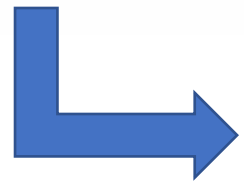
this is iterative deepening in general
applied to A*: later

iterative deepening, graphically



distance 0 distance 1 distance 2 distance 3

features?
drawbacks?



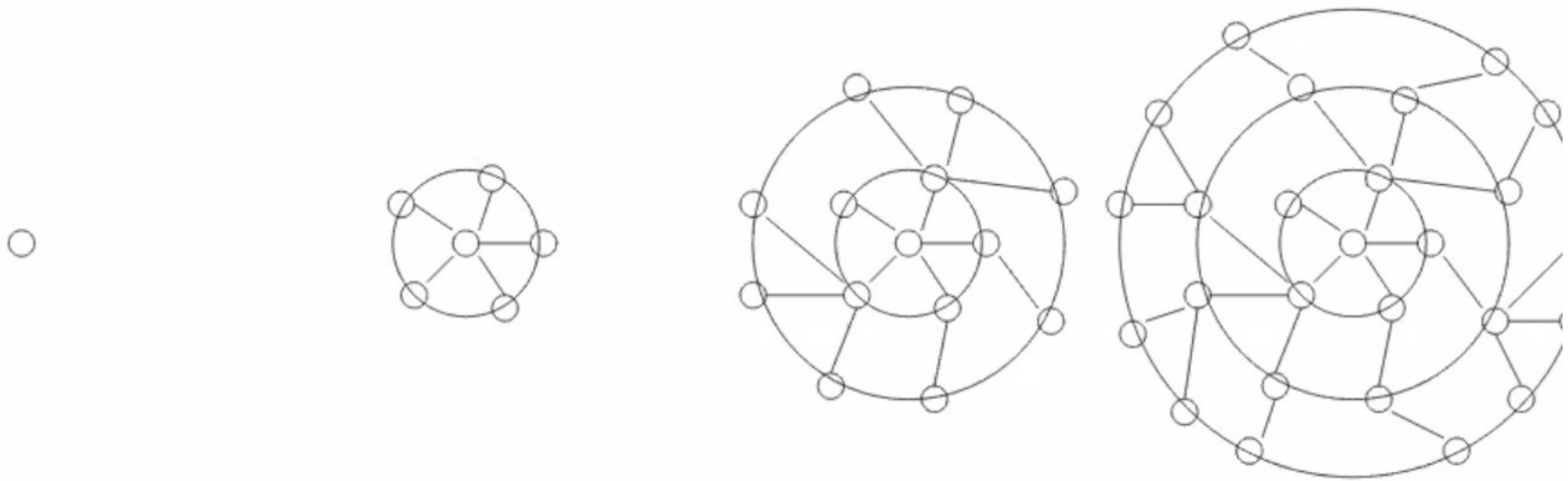
required memory

```
iterative_deepening() {
        bound = 0
        while (true) {
                search(root, bound);
                bound++;
        }
}

search(node, bound) {
        if (bound = 0)
                return;
        for each successor s of node
                search(s, bound - 1)
}
```

for each bound: depth-first visit
required memory: up to `bound` nodes

repetitions



bound = 0 bound = 1 bound = 2 bound = 3

nodes visited when `bound=1`
visited again for `bound=2`, `bound=3`, etc.

even worse with cycles!
(but still works)

loops

do not return to a node already visited in the *current* search

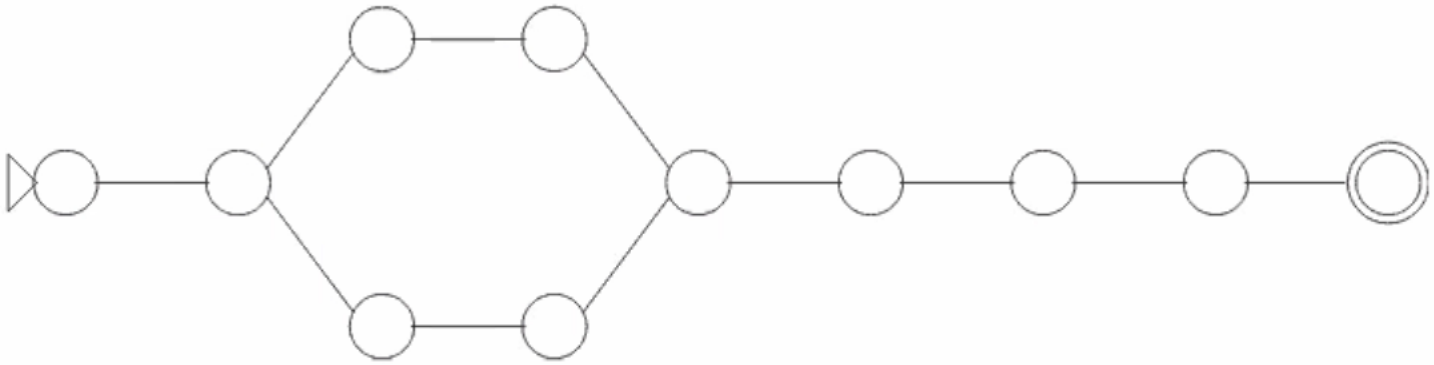
```
iterative_deepening() {
        bound = 0
        while (true) {
                search(root, ∅, bound);
                bound++;
        }
}

search(node, visited, bound) {
        if (bound = 0 or s ∈ visited)
                return;
        for each successor s of node
                search(s, visited ∪ {s}, bound - 1)
}
```

do not proceed if `s ∈ visited`

`visited`: nodes in the path from the root to `s`

repetitions and loops

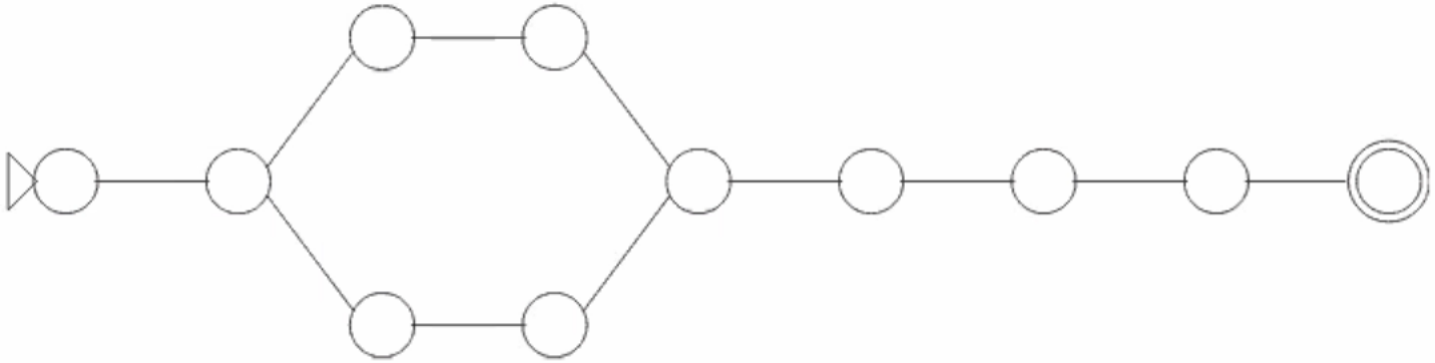


an example problem
(with a loop)

in the following slides:

- half line = direction left to explore
  (because search went in the other way)
- red line = blocked because node already visited

example order of the successors

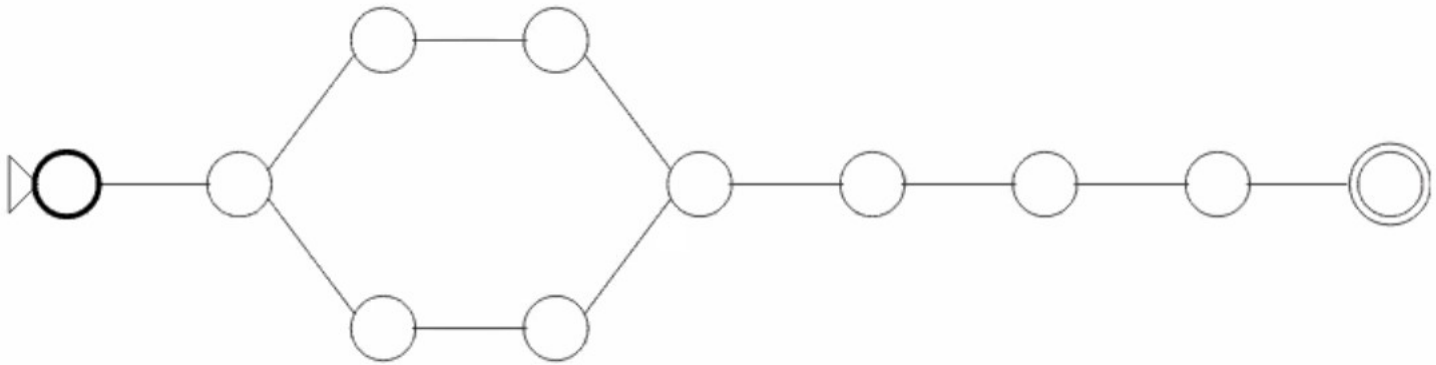


most nodes in this example have only one successors

two nodes with two successors: where to go first?

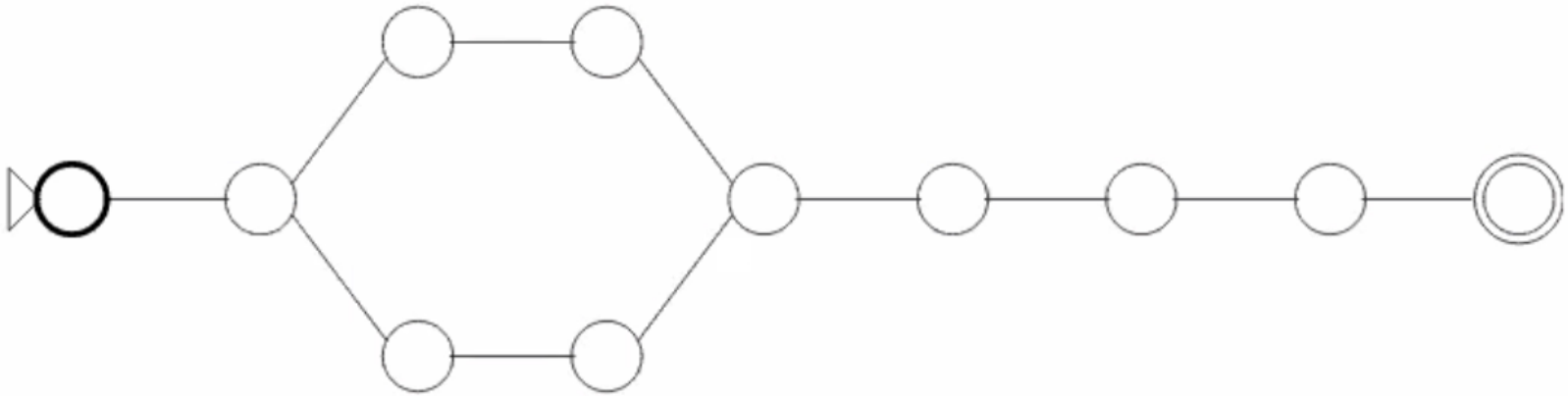
for the sake of this example:
first successors on the left if possible, otherwise down

bound = 0



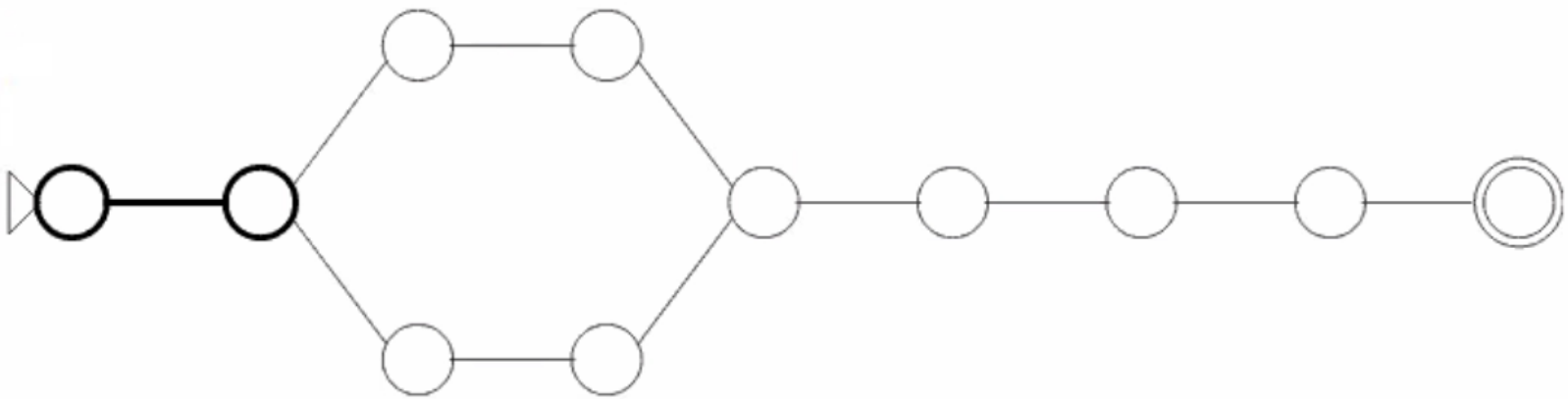
the call to `search()` returns immediately

bound = 1



`search()` called on the starting node

calls itself on its successor

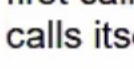


stops because of the bound

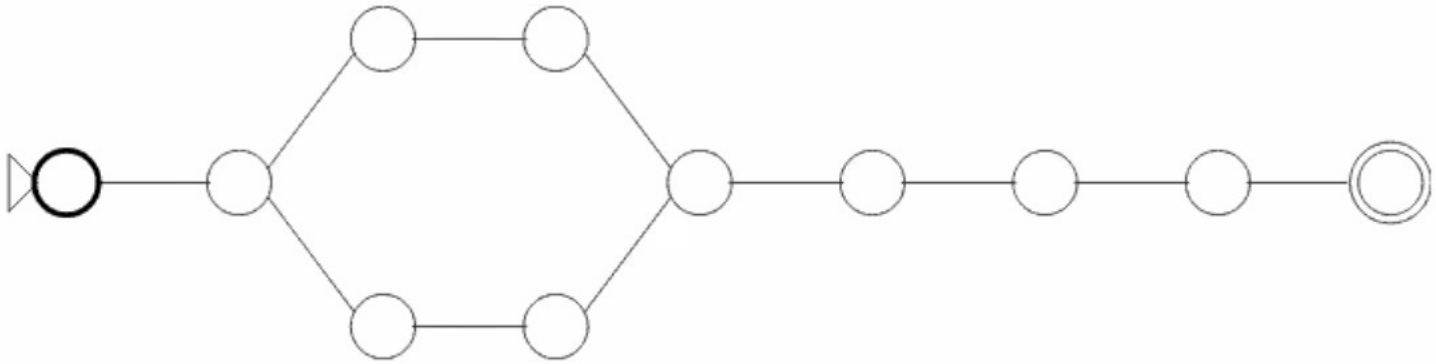
bound = 2

this is longer
four recursive calls to `search()`

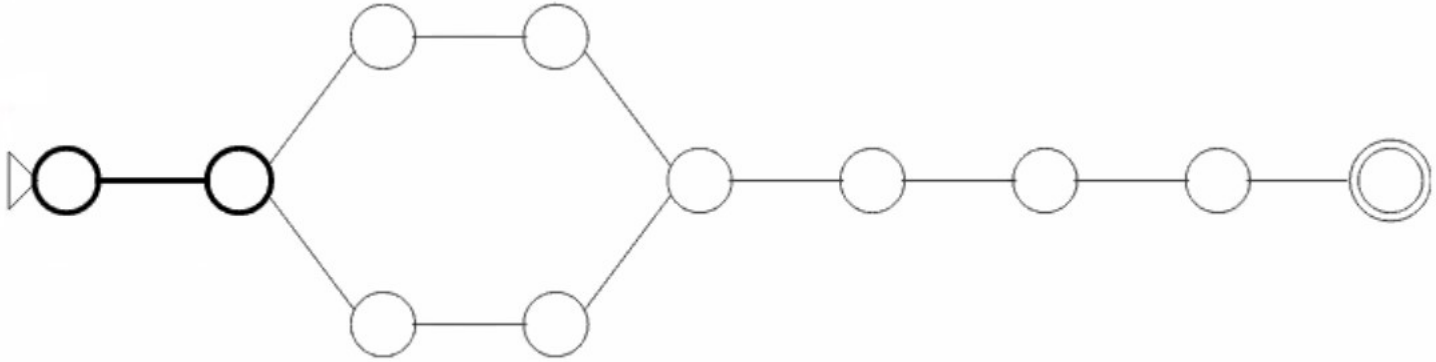
two slides



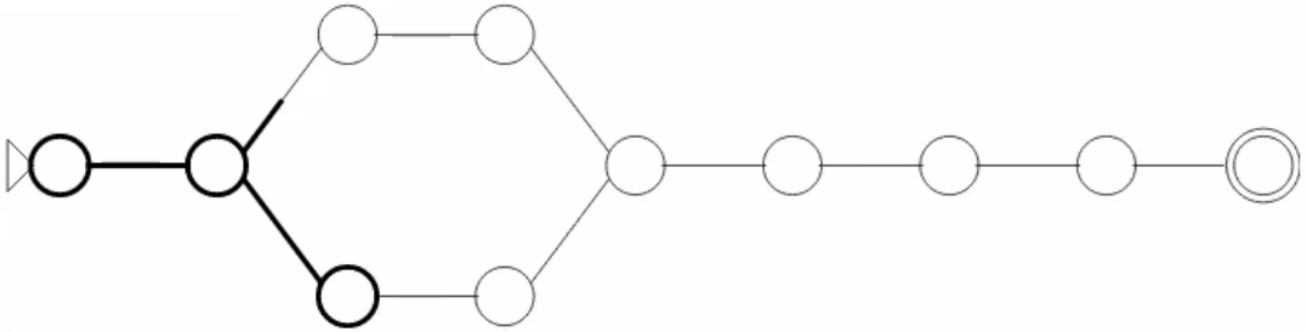
bound = 2, first part



first call to `search()`
calls itself on the successor



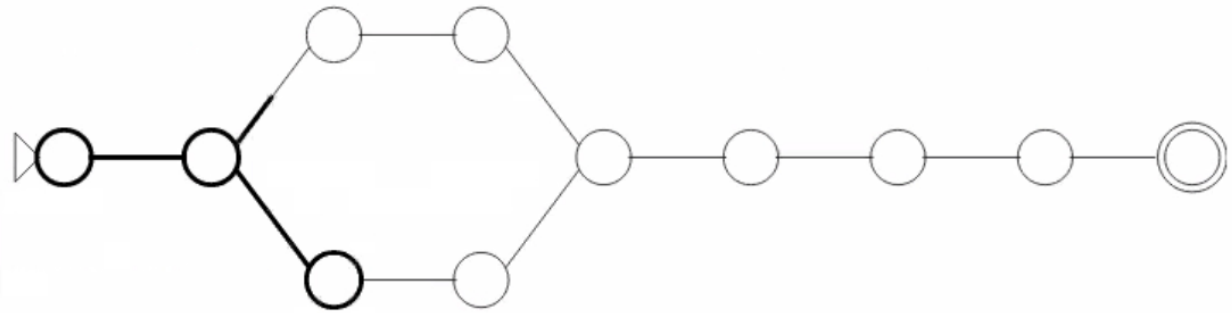
recursive call to `search()`
two successors: first go down, then up



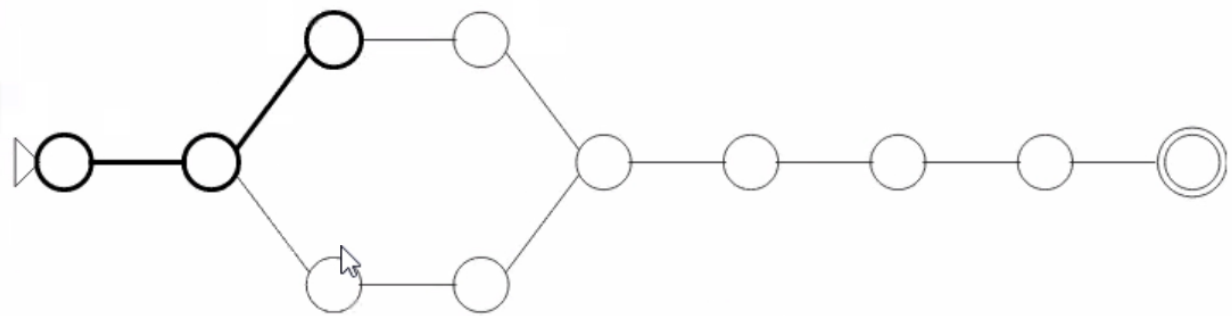
go down, leave up direction left to explore when done

stop because of the bound
go back

bound = 2, second part



go back, take last road not taken before



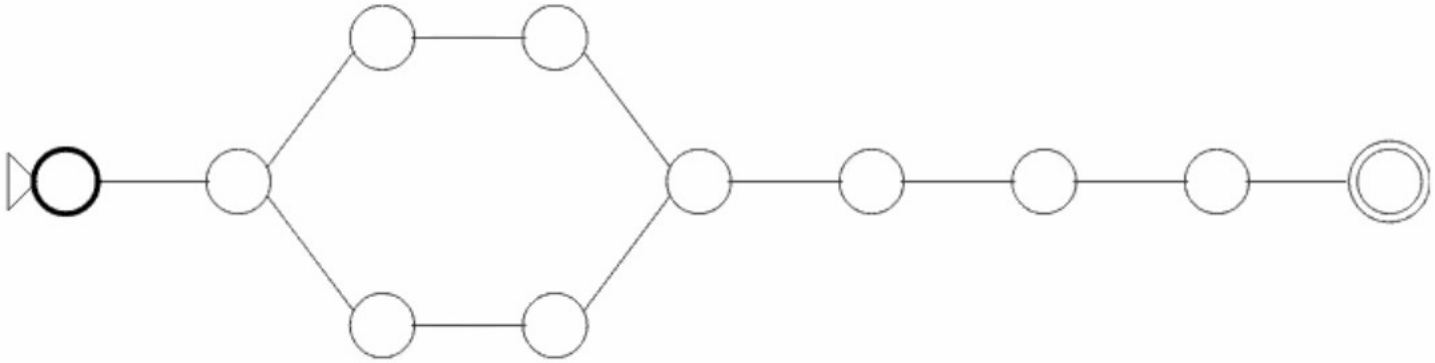
again, stop because of the bound

fast forward to `search = 7`

`search()` called with `bound = 3,…6`

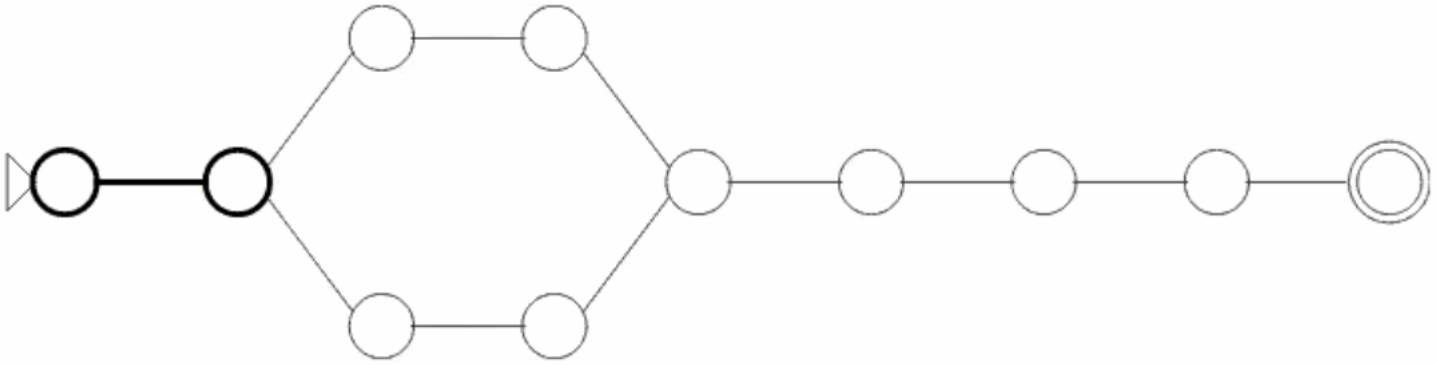
omitted here

`bound = 7`

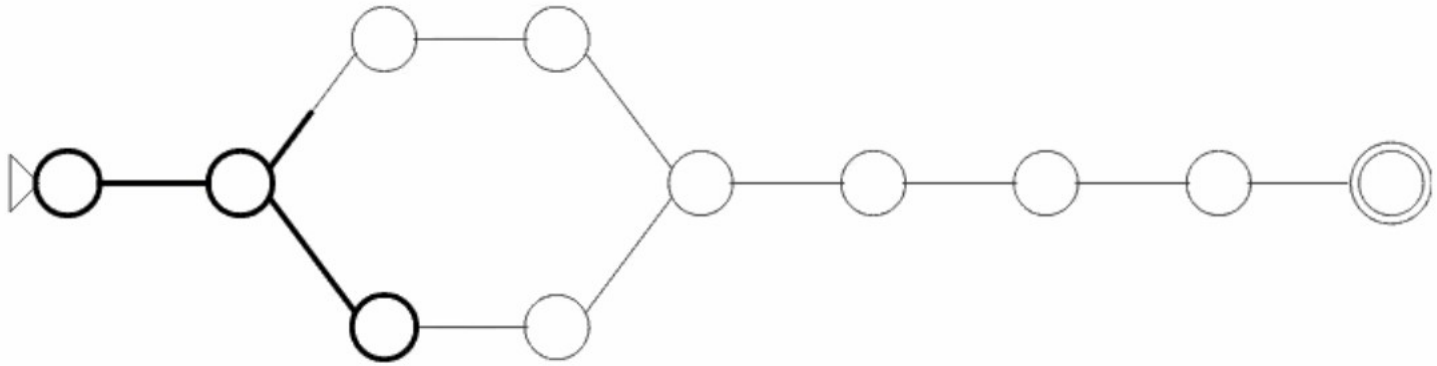


start again from the beginning

`bound = 7`

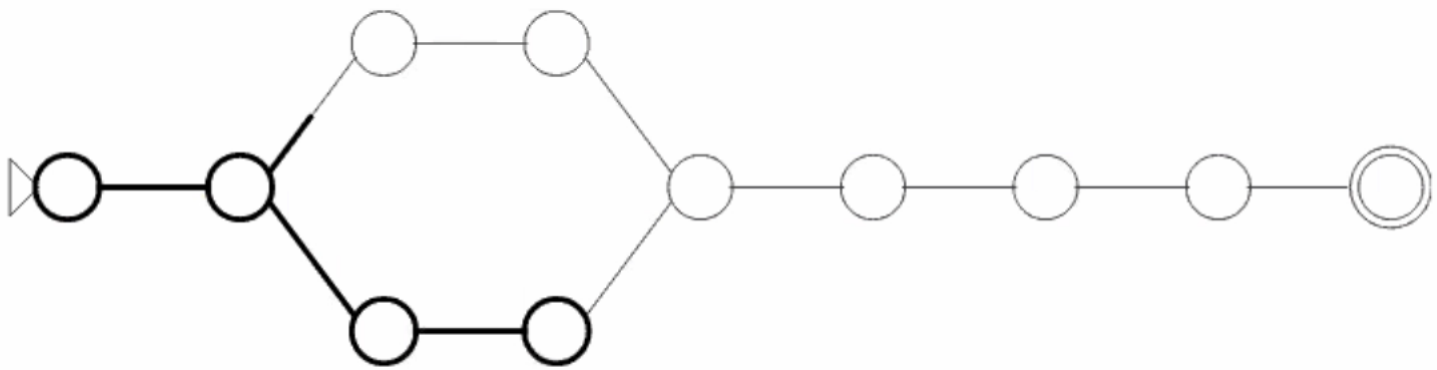


bound = 7

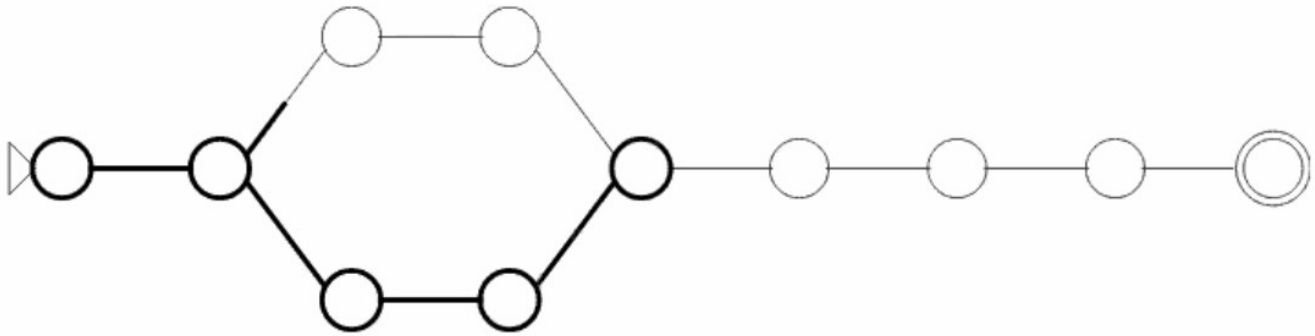


first visited successor
other will be visited later

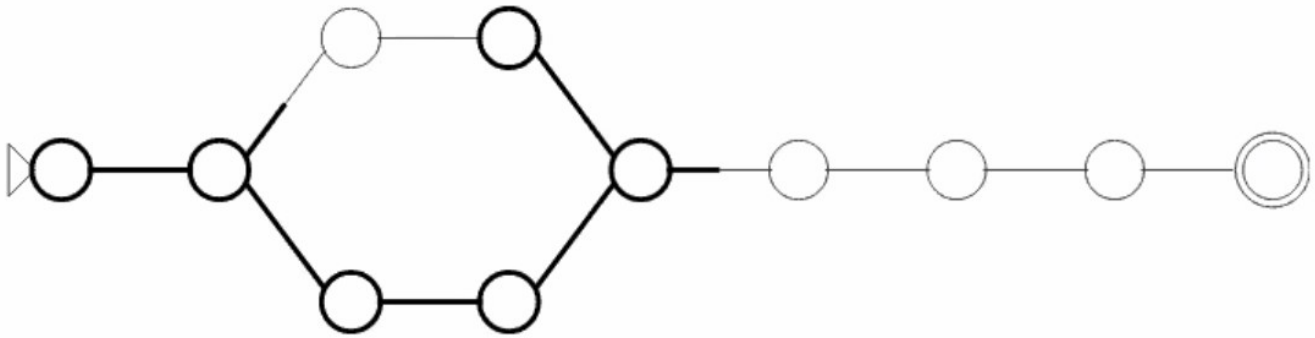
bound = 7



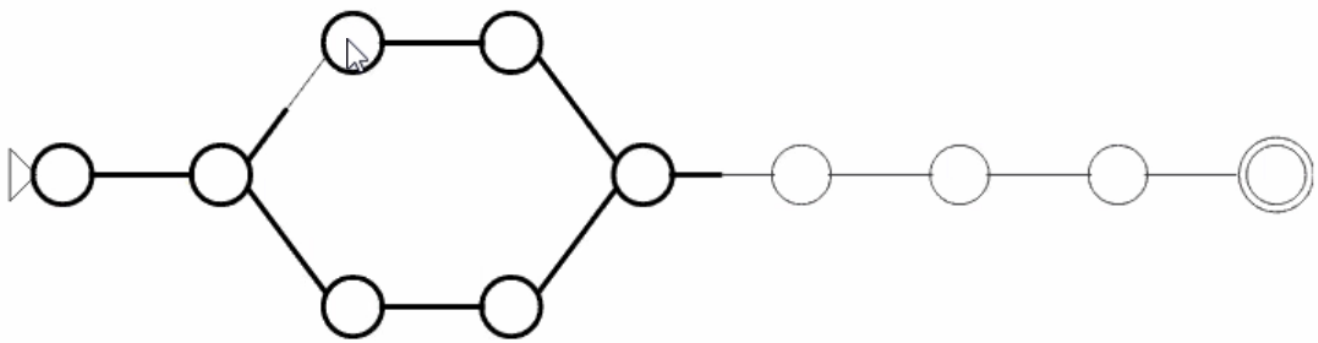
bound = 7



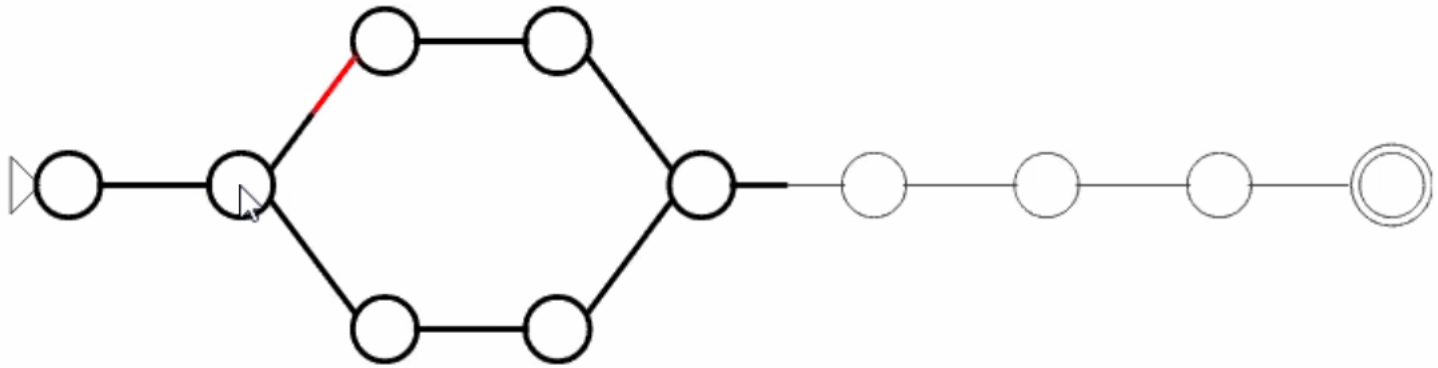
bound = 7



bound = 7



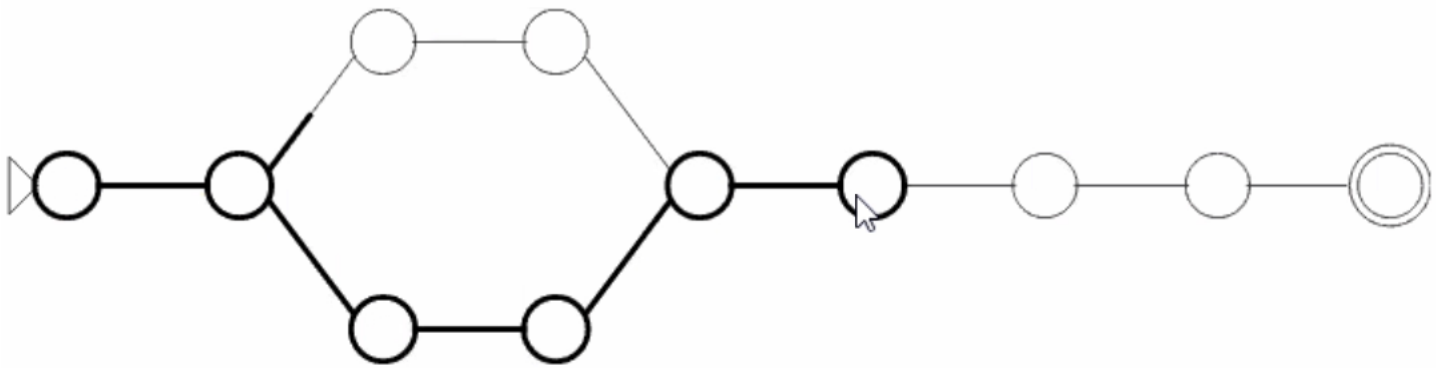
bound = 7



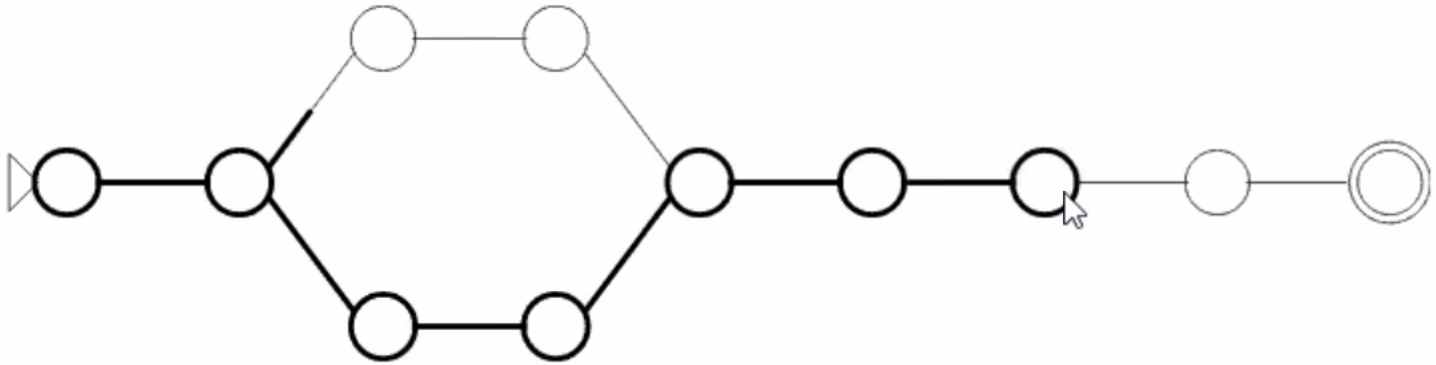
node already visited
do not go there!

backtrack instead
follow the last suspended direction
(black half-line on the right)

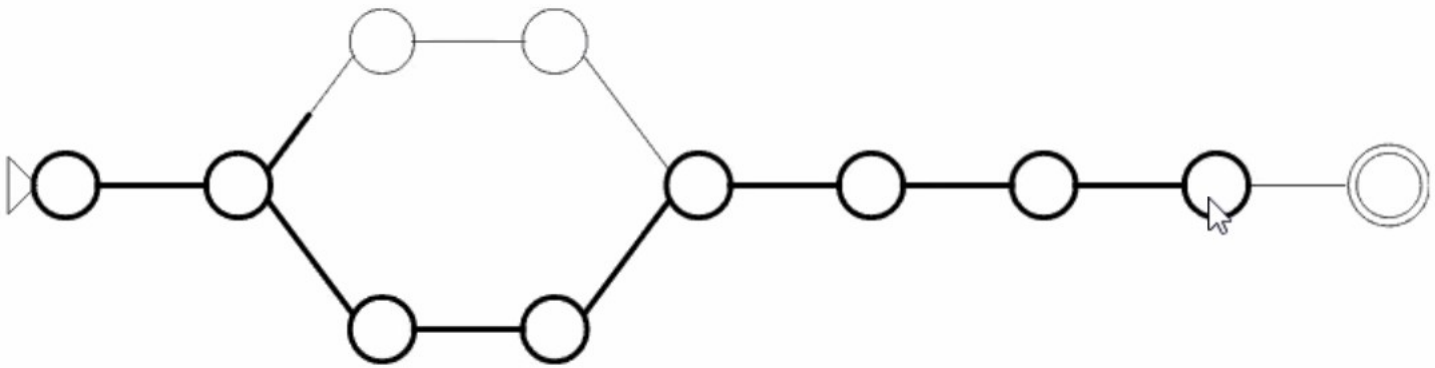
bound = 7



bound = 7



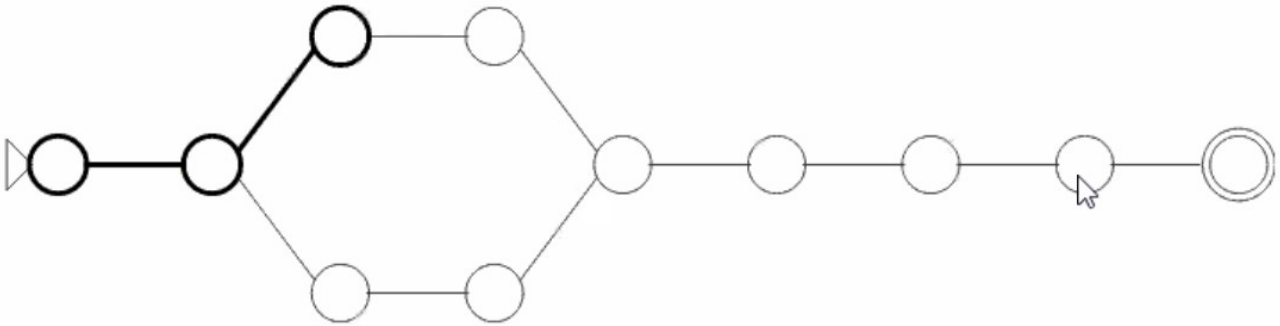
bound = 7



stop because of the bound

backtrack
still one direction unexplored
go back and take it

bound = 7

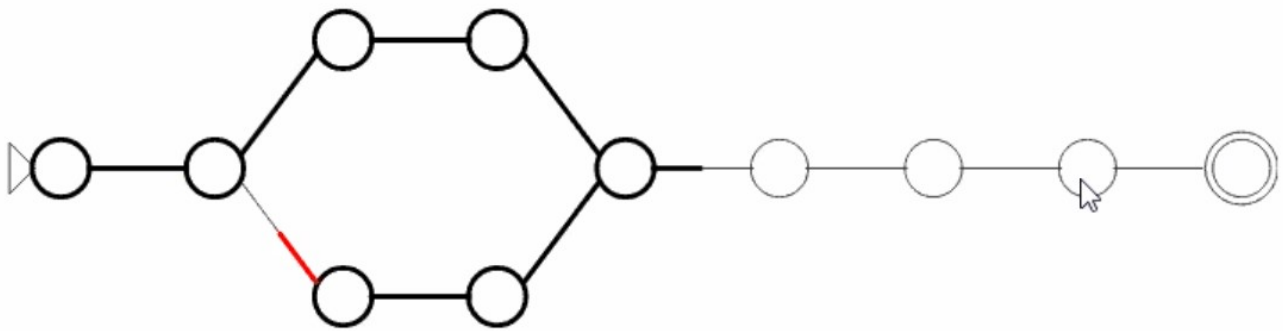


go back all the way

until the last untried choice

then go forward

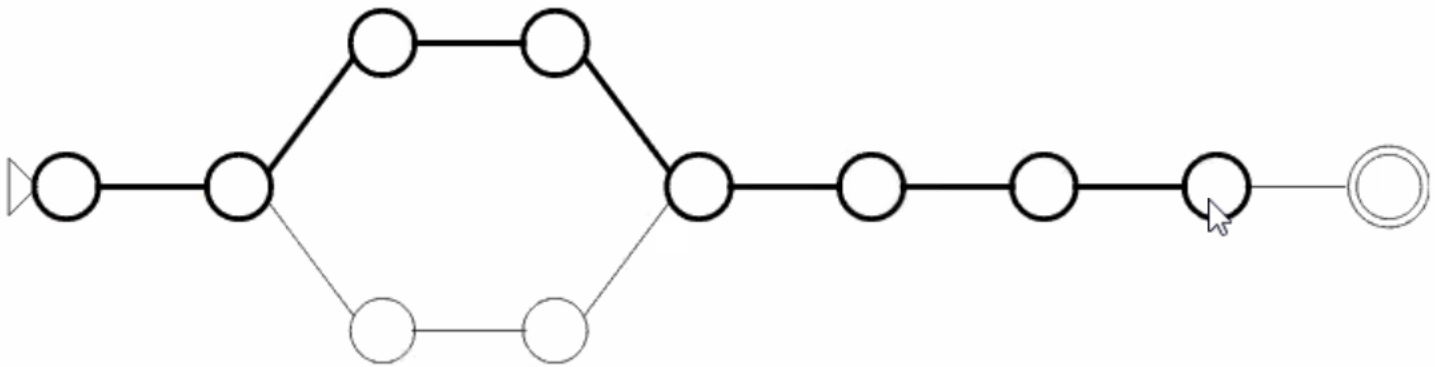
bound = 7



like previously:
search stopped because node already visited

go the last untried choice

bound = 7



stopped because of the bound

no suspended direction

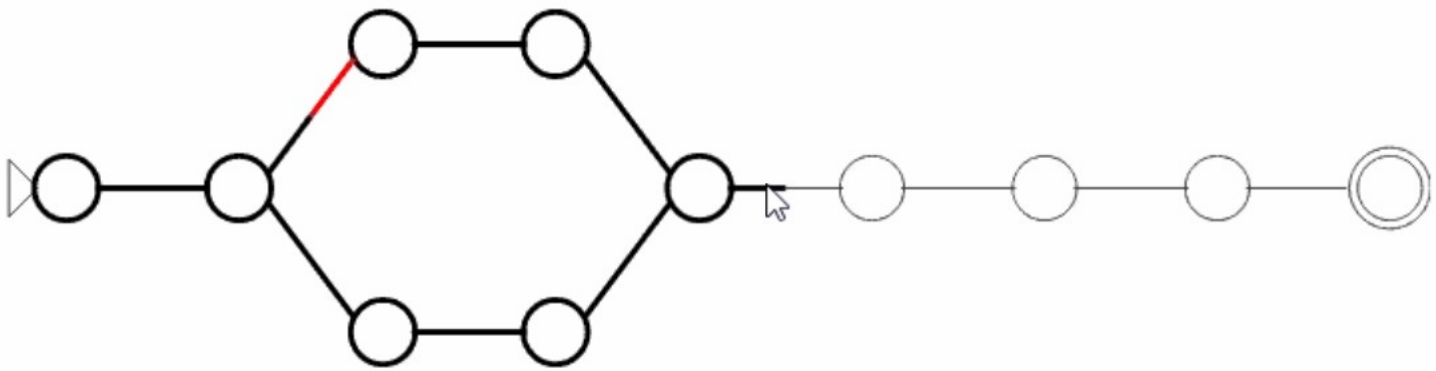
backtrack all the way

bound = 8

next iterationis with `bound = 8`

first part as before

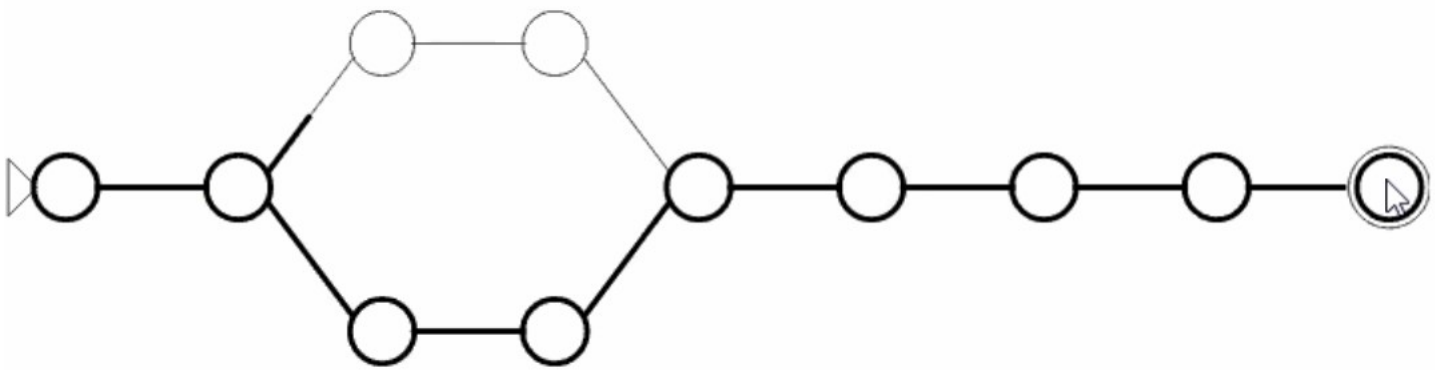
bound = 8



stop because node already visited

try last suspended way

bound = 8



target reached

comments on the examples

node reached for `bound = x`
is reached again for `bound = x+1, x+2, x+3, …`

loops: nodes reached twice for the same `bound`
in general: multiple times

efficiency of iterative deepening

bredth-first = each node only reached once
iterative deepening = nodes reached multiple times

bredth-first more time-efficient than iterative deepening

but: requires storing the frontier
may become large

iterative deepening only stores the *path* to the current node
requires less space

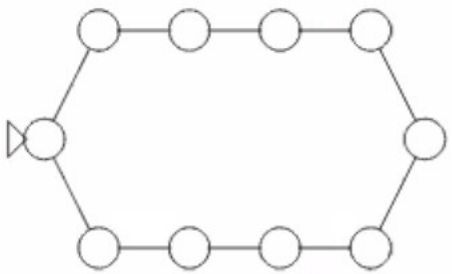
efficiency of iterative deepening

in comparison with breadth-first search:

1. does not store frontier, only path from starting node
2. nodes visited multiple times

how bad point 2 is?

comparison: large loops



iterative deepening ends up storing all nodes in the path

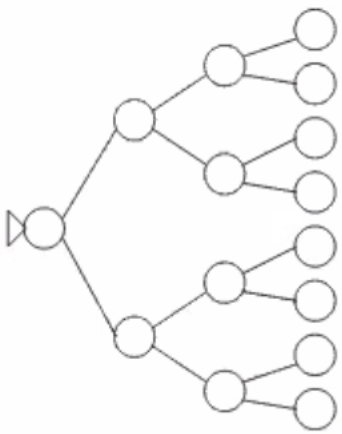
breadth-first does not: frontier never larger than two nodes

comparison: long lines

breadth-first: linear

iterative deepening: quadratic

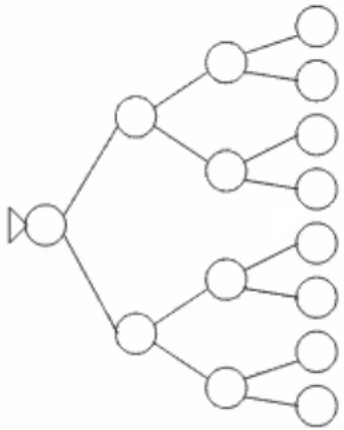
comparison: complete trees



breadth-first: size of frontier doubles at each levels

iterative deepening: path is logarithmic with nodes
time?

comparison: complete trees, time



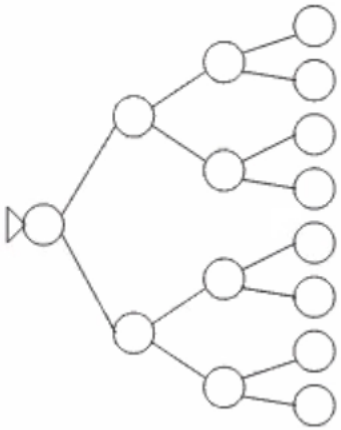
breadth-first: number of nodes

iterative deepening:

- `bound = 1`, cost 1
- `bound = 2`, cost 2
- …
- last iteration: number of nodes

really so bad?

comparison: complete trees, time in reverse order



breadth-first: number of nodes (16)

iterative deepening
(sum from last iteration to first):

- number of nodes (16)
- half the number of nodes (8)
- half of that (4)
- half of that (2)
- half of that (1)

total: `16 + (8 + 4 + 2 +1) = 16 + 15`
only twice the cost of breadth-first

the lilypad and the pond

a lilypad in a pond doubles its covered area each day

on day 30, it finishes covering the whole pond

which day did it cover half of it?

lilypad = ninfea
pond = stagno

unexpected properties of exponential functions

pond covered on day 30

double area each day

previous day (29): half of the pond

iterative deepening vs. breadth-first

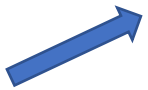
iterative deepening wins when:

- nodes have many successors
- loops are small

cost of actions

previous example: all edges have cost 1

if costs differ: one issue arises



iterative deepening with costs

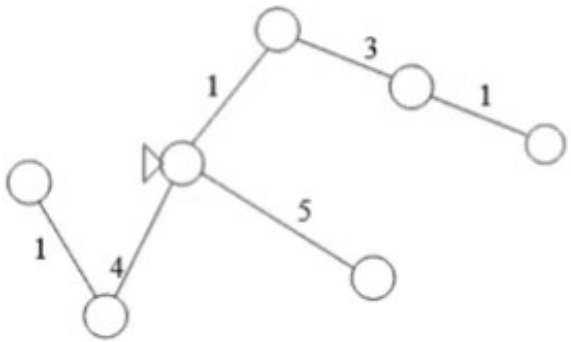
```
iterative_deepening() {
        bound = 0
        while (true) {
                search(root, ∅, bound);
                bound++;
        }
}

search(node, visited, bound) {
        if (bound ≤ 0 or s ∈ visited)
                return;
        for each successor s of node
                search(s, visited ∪ {s}, bound - node→s)
}
```

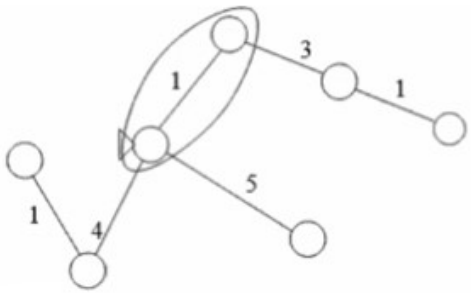
previously: `node→s = 1`
now: some value

issue = ?

useless iterations

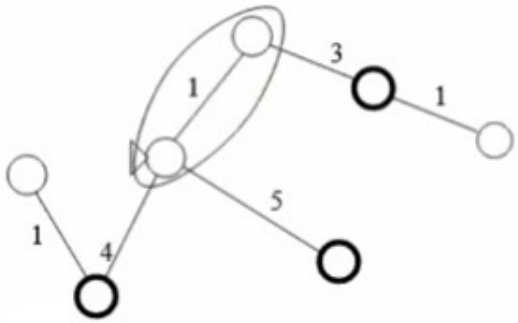


bound = 1



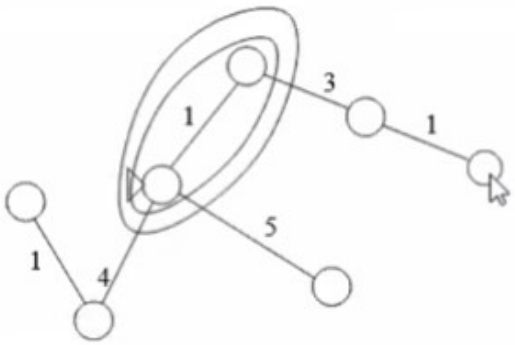
nodes at cost-distance 4 and 5 not reached
since both 4 and 5 are greater than `bound = 1`

bound = 2



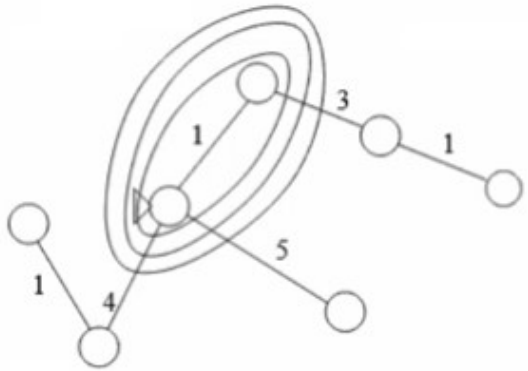
again!

bound = 3



and again!

bound = 4



finally, some new nodes

useless iterations: the problem

the cause of the problem:

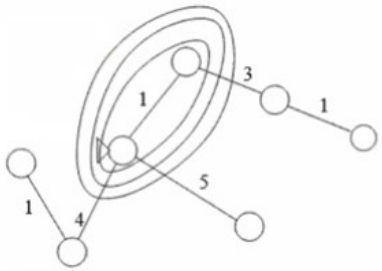
```
iterative_deepening() {
        bound = 0
        while (true) {
                search(root, ∅, bound);
                bound++;                          ⇐ HERE!
        }
}
```

not a problem if costs are all one

in the example:
no node is at cost-distance between `1` and `2`

in general:
no node between `bound` and `bound+1`

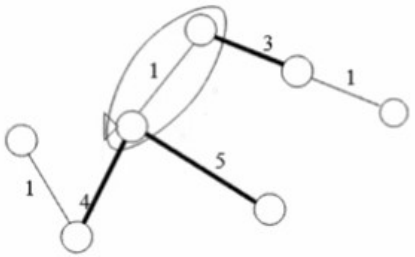
missing distances: solution



go from `bound = 1` straight to `bound=4`
skip `2` and `3`

how to tell the bounds to skip without actually searching?

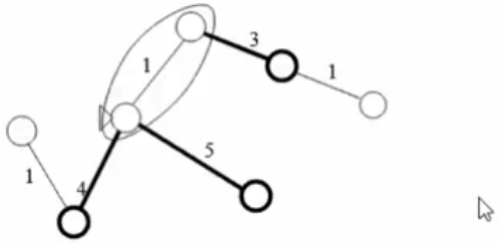
first search: gather additional information



bold: lines that cross the border

nodes next to be visited
but were not because of the bound

first search: next node



nodes about to be visited
but were not because of the bound

their distance: `4`, `5`, `1+3`

with `bound = 4` at least a new node is reached

detect and skip useless bounds

during the search:

- when stopping the search because of the bound
- calculate distance of nodes about to be visited
- maintain the minimum

next iteration: `bound = minimum`

iterative deepening A*

instead of the distance `start⇒n`

use `start⇒n + h(n)`