03
Chapter

Uninformed Search

2I1AE1: Artificial Intelligence

Régis Clouard, ENSICAEN - GREYC

"Intelligence is what you use
when you do not know what to do."
Jean Piaget

In this chapter

- **Unformed Search Algorithms**
  - Brute force algorithms that do not use information on the problem.
    → This is not AI ! But these are prerequisites for AI algorithms.

1) Breadth-first search
2) Depth-first search
3) Iterative deepening depth-first search
4) Bidirectional search
5) Elimination of state repeats
6) Uniform cost search

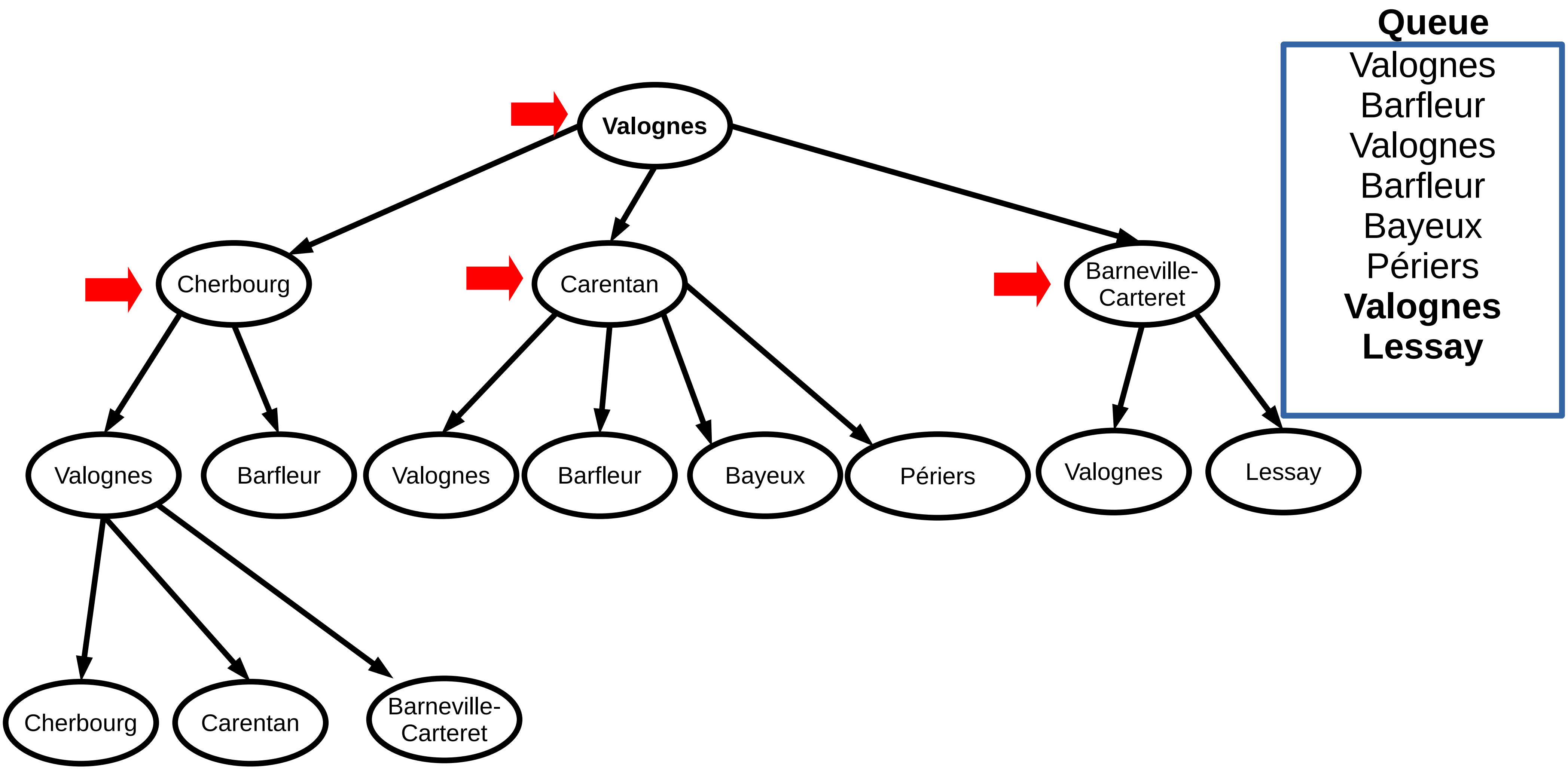
1. Breadth-First Search (BFS)

- Strategy: expand the shallowest node first
- Implementation of the open-list: **FIFO**
  - **ADD-IN-LIST**: add successors to the **end of the list**

```
function GENERAL-SEARCH(problem) returns solution
  var open-list ← MAKE-LIST(MAKE-NODE(INITIAL-STATE[problem]))
  LOOP
     IF EMPTY(open-list) THEN return failure
     node ← REMOVE-FRONT-LIST(open-list)
     IF IS-GOAL(problem, STATE[node]) THEN return the related solution
     open-list ← ADD-IN-LIST(GET-SUCCESSORS(node, problem), open-list)
end
```

Breadth-First Search (BFS)

- Strategy: expand the shallowest node first.



[bfs1.sh; bfs2.sh]

Properties of Breadth-First Search

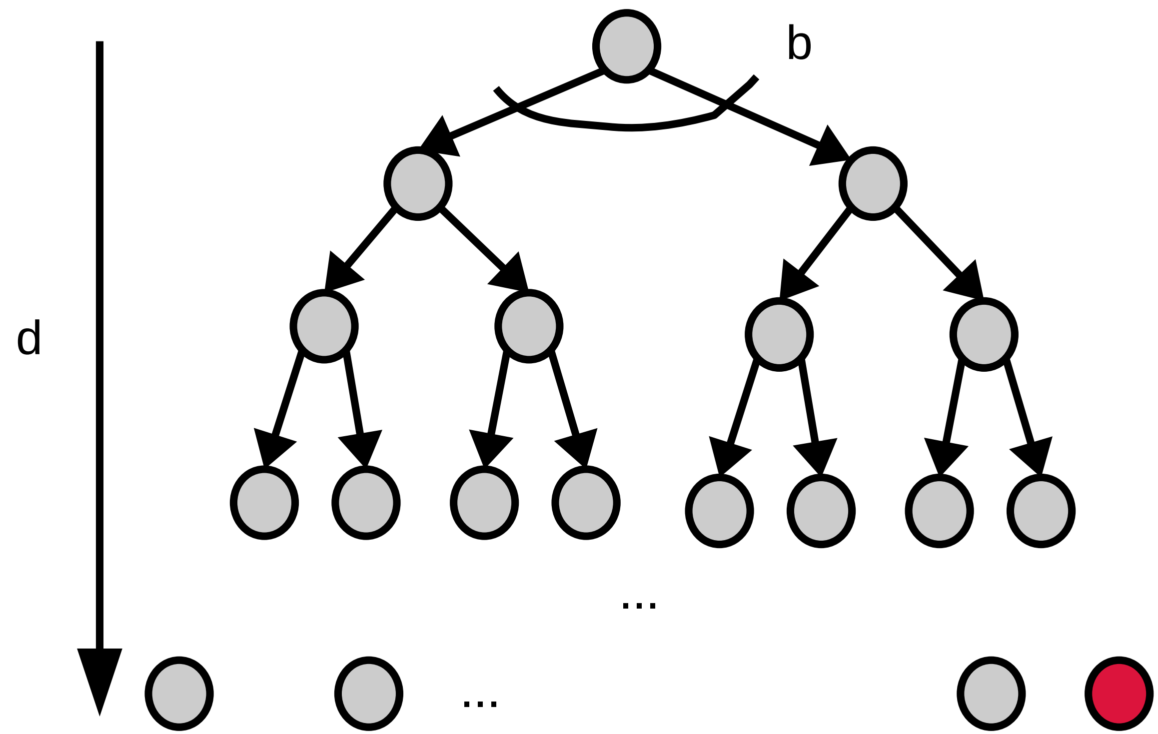
- **Completeness**
  - Yes. All nodes are examined.
- **Optimality**
  - Yes, for the smallest number of nodes.
- **Time complexity**
  - Proportional to the number of examined nodes.
- **Space complexity**
  - Proportional to the number of nodes stored at a time.

Assume:

- *b – maximum **b**ranching factor.*
- *d – **d**epth of the optimal solution.*
- *m – **m**aximum depth of the search tree.*

BFS. Time Complexity (Max)

- Proportional to the number of examined nodes.



Depth	Number of nodes ***(**case b = 2)*
0	1
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
d	2^d (b^d)

Total nodes $= \sum_{i=0}^{d} b^i$

$$= \frac{1-b^{d+1}}{1-b}$$

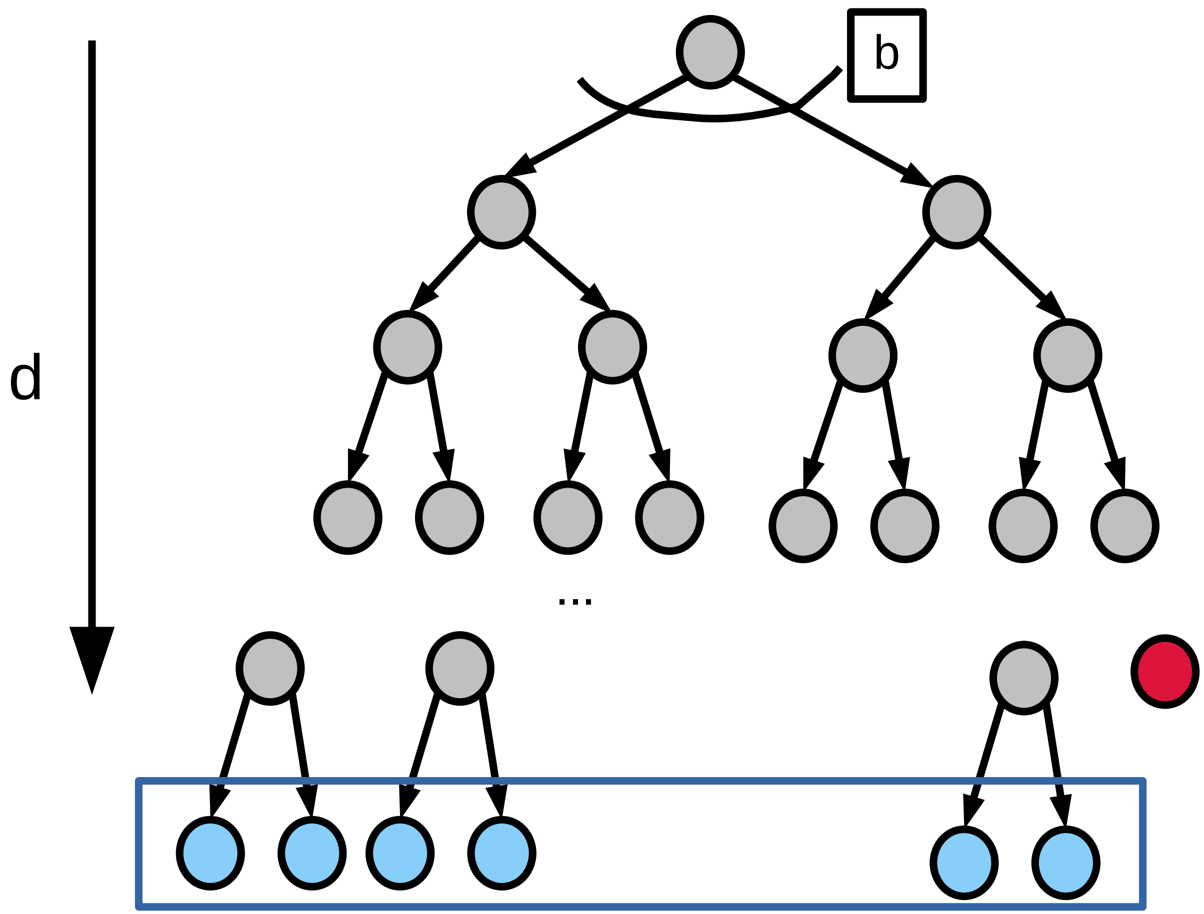
Total examined nodes: $O(b^d)$

Properties of Breadth-First Search

- **Completeness**
  - Yes. All nodes are examined.
- **Optimality**
  - Yes, for the smallest number of nodes.
- **Time complexity**
  - Worst-case $O(b^d)$
  - Exponential in the depth of the solution.
- **Space complexity**
  - ?

BFS. Space Complexity (Max)

- Count nodes kept in the tree structure or in the queue.



Depth	Number of nodes (*Case b = 2*)
0	1
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
d	2^d (b^d)
d+1	2^{d+1} (b^{d+1})

Total stored nodes : $O(b^{d+1})$

Properties of Breadth-First Search

- **Completeness**
  - Yes. All nodes are examined.
- **Optimality**
  - Yes, for the smallest number of nodes.
- **Time complexity**
  - Worst-case $O(b^d)$
  - Exponential in the depth of the solution.
- **Space complexity**
  - Worst-case $O(b^{d+1})$
  - Exponential with the number of nodes kept in the memory.

Properties of Breadth-First Search

- The costs are very high.
- Example: assuming the machine performances.
  - b=10; 100,000 nodes/second; 1000 bytes/node.

Depth	Nodes	Time	Space
2	111	1.1 milliseconds	107 kilobytes
4	11,111	111 milliseconds	10.6 megabytes
6	10^6	11 seconds	1 gigabytes (10^9)
8	10^8	19 minutes	103 gigabytes
10	10^{10}	31 hours	10 terabytes (10^{12})
12	10^{12}	129 days	1 petabytes (10^{15})
14	10^{14}	35 years	99 petabytes
16	10^{16}	3,523 years	10 exabytes (10^{19})

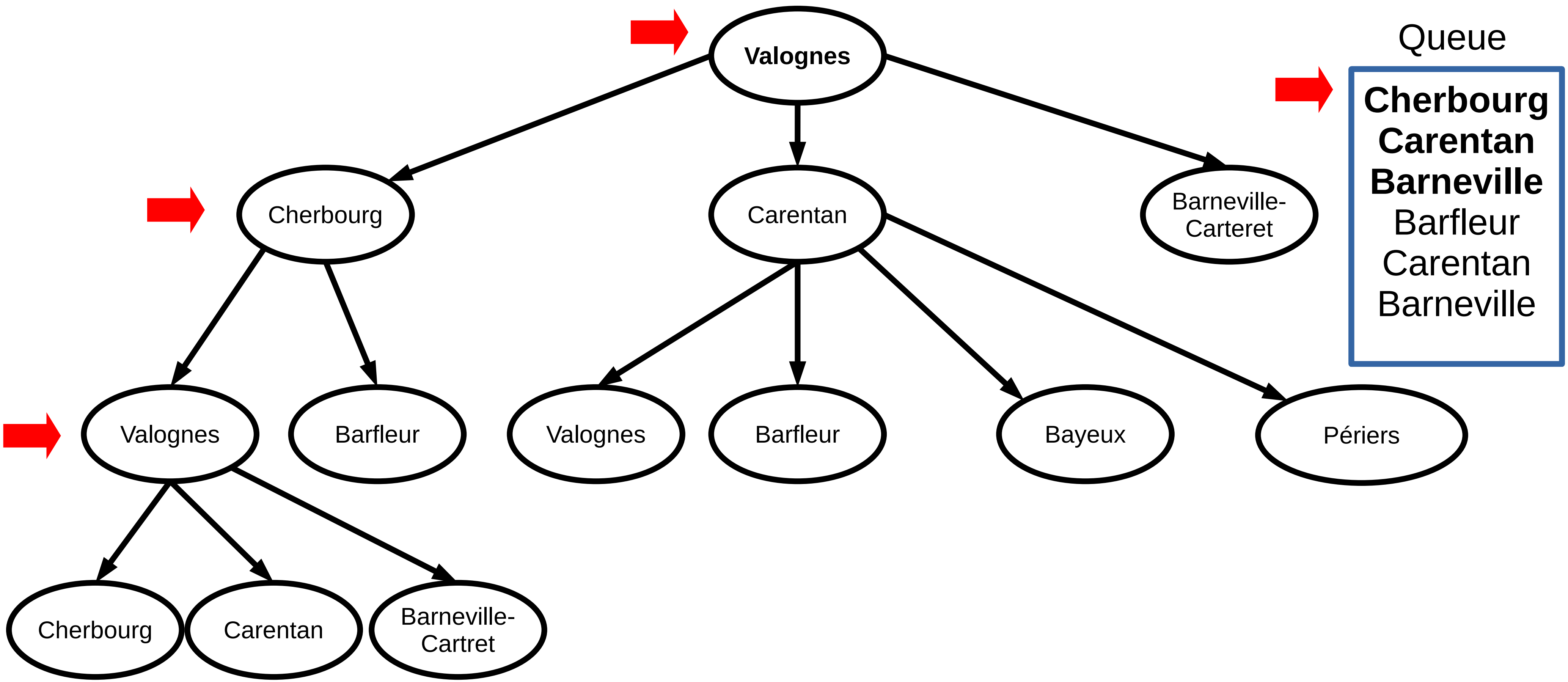
2. Depth-First Search (DFS)

- Strategy: expand the deepest node first.
  - **Backtrack** when the path cannot be further expanded.
- Implementation of the open-list: **LIFO**
  - **ADD-IN-LIST**: add successors to the **beginning of the list.**

```
function GENERAL-SEARCH(problem) returns solution
  var open-list ← MAKE-LIST(MAKE-NODE(INITIAL-STATE[problem]))
  LOOP
     IF EMPTY(open-list) THEN return failure
     node ← REMOVE-FRONT-LIST(open-list)
     IF IS-GOAL(problem, STATE[node]) THEN return the related solution
     open-list ← ADD-IN-LIST(GET-SUCCESSORS(node, problem), open-list)
end
```

Depth-First Search (DFS)

- Strategy: expand the deepest node first.
  - Backtrack when the path cannot be further expanded.



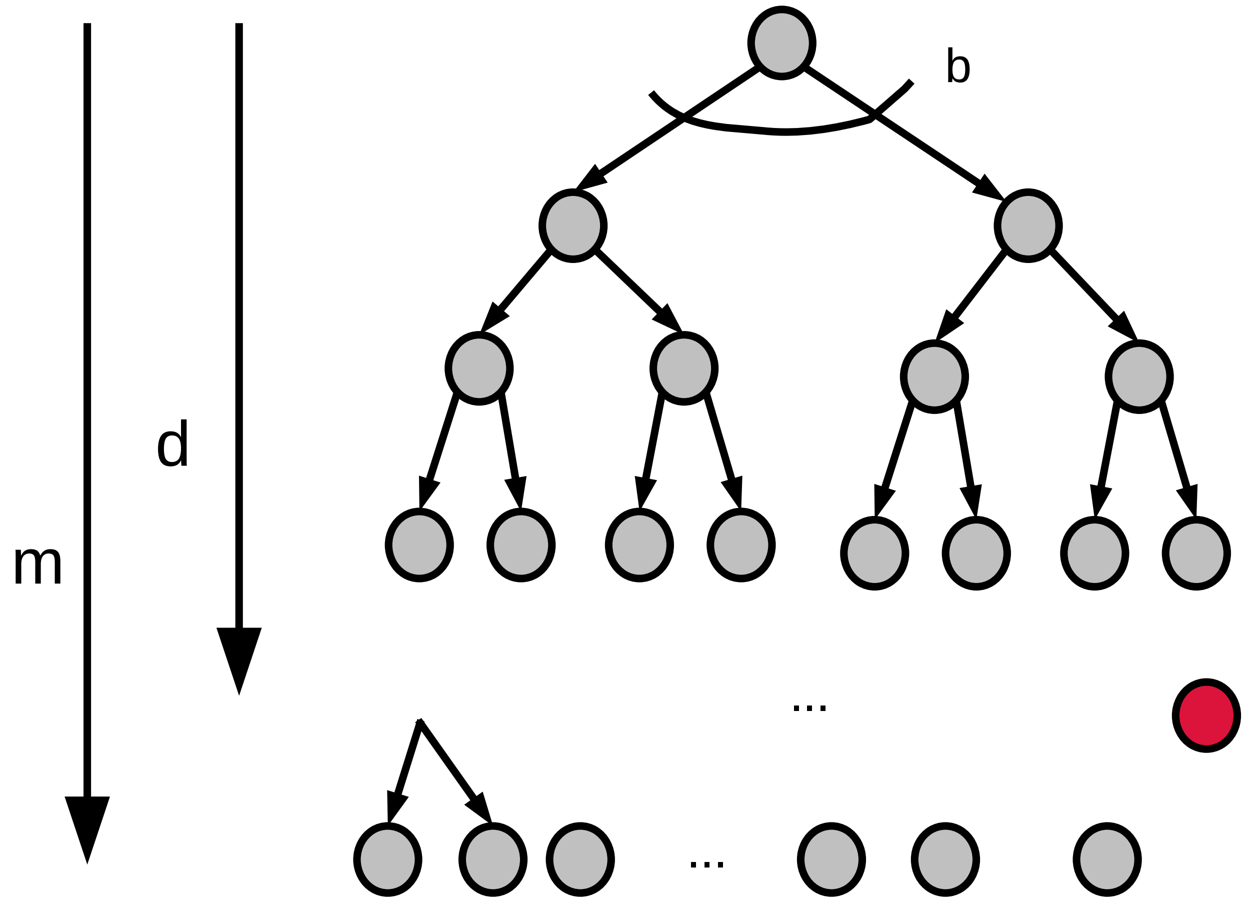
[dfs1.sh; dfs2.sh]

Properties of the Depth-First Search

- **Completeness**
  - No. For example Knuth's conjecture problem ("one can start at 3 and reach any integer by iterating factorial, sqrt, and floor.", eg. $\lfloor\sqrt{\sqrt{(3!)!}}\rfloor=5$) → infinite depth
- **Optimality**
  - No. Solution found first may not be the shortest.
- **Time complexity**
  - ?
- **Space complexity**
  - ?

DFS. Time Complexity

- Proportional to the number of examined nodes.



Depth	Number of nodes
0	1
1	$2^1 = 2$
2	$2^2 = 4$
3	$2^3 = 8$
d	2^d
m	2^m

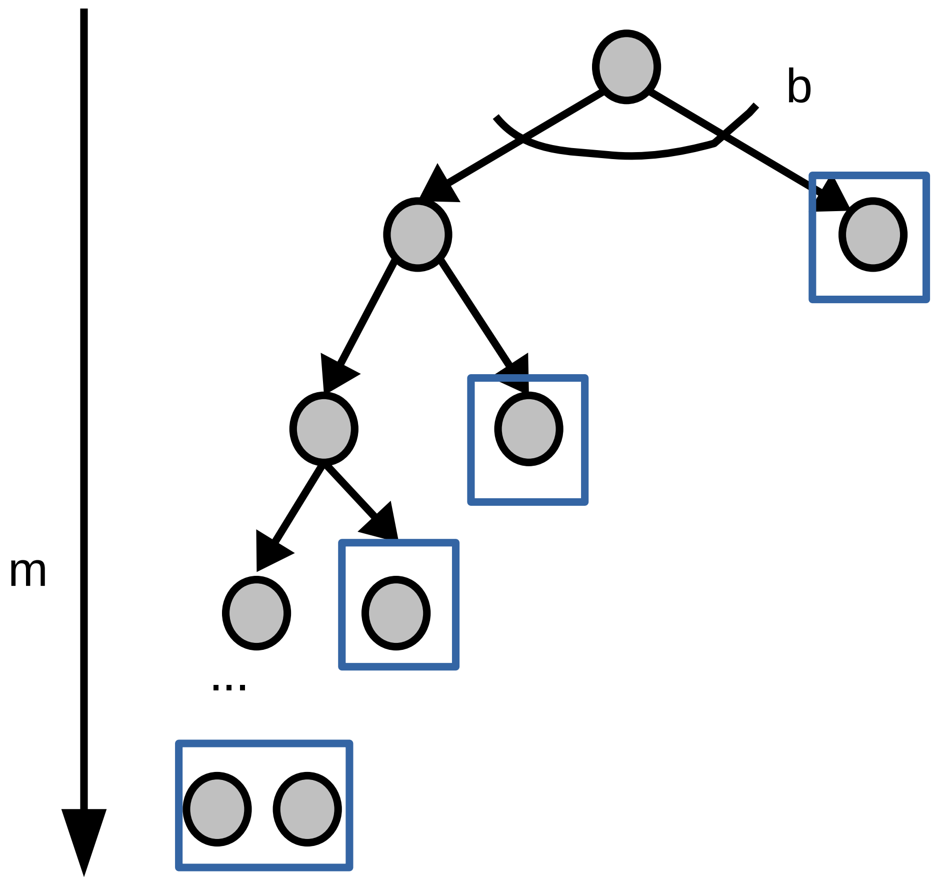
Total examined nodes: $O(b^m)$

$$\text{Total nodes} = \sum_{i=0}^{m} b^i = \frac{1-b^{m+1}}{1-b}$$

Properties of the Depth-First Search

- **Completeness**
  - No. Infinite loops can occur.
- **Optimality**
  - No. Solution found first may not be the shortest.
- **Time complexity**
  - *Worst-case* $O(b^m)$
  - Exponential in the maximum depth of the search tree.
  - Terrible if m is much larger than d.
- **Space complexity**
  - ?

DFS. Space Complexity



Depth	Number of nodes kept
0	1
1	1 = (b-1)
2	1 = (b-1)
3	1 = (b-1)
m	2 = b

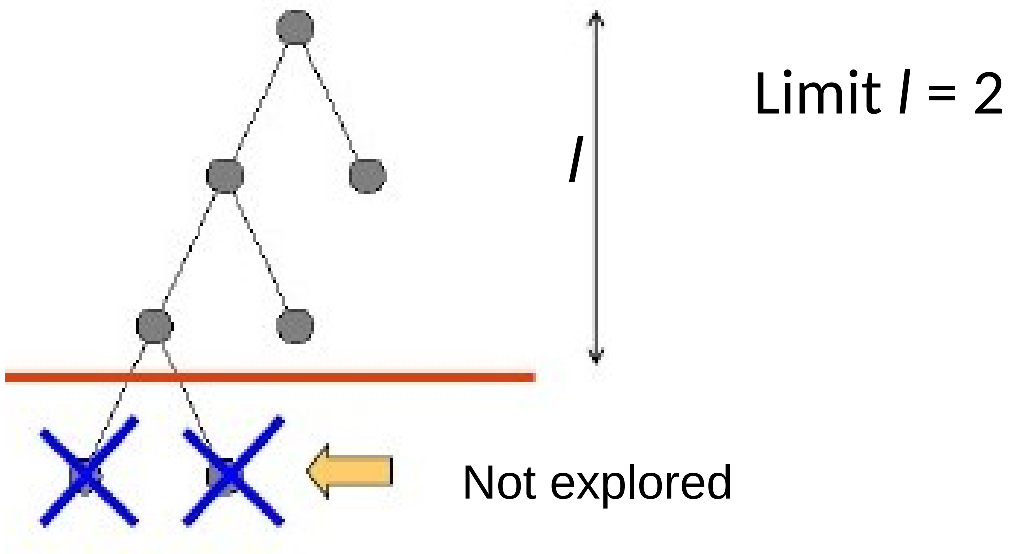
Complexity: *O(b.m)*

Properties of the Depth-First Search

- **Completeness**
  - No. Infinite loops can occur.
- **Optimality**
  - No. Solution found first may not be the shortest.
- **Time complexity**
  - *Worst-case $O(b^m)$*
  - Exponential in the maximum depth of the search tree.
  - Terrible if *m* is much larger than *d*.
- **Space complexity**
  - Worst-case *$O(b.m)$*
  - Linear in the maximum depth of the search tree.
  - Example: assuming the machine performances.
    - b=10; 1000 bytes/node
    - Depth 16 → 160 x 10^3 bytes (vs 10^{19} bytes for BFS)

Limited-Depth Depth-First Search

- How to eliminate infinite depth-first exploration?
- Put a limit *l* on the depth of the depth-first exploration.



- **Completeness**
  - yes
- **Optimality**
  - no
- **Time complexity**
  - Worst-case $O(b^l)$
- **Space complexity**
  - Worst-case $O(b.l)$

Limited-Depth Depth-First Search

- Problem: How to pick the maximum depth?

- Example: Assume we have a traveler problem with 20 cities.
  - How to pick the maximum tree depth?
  - Trivial: we need to consider only paths of length <= 20.

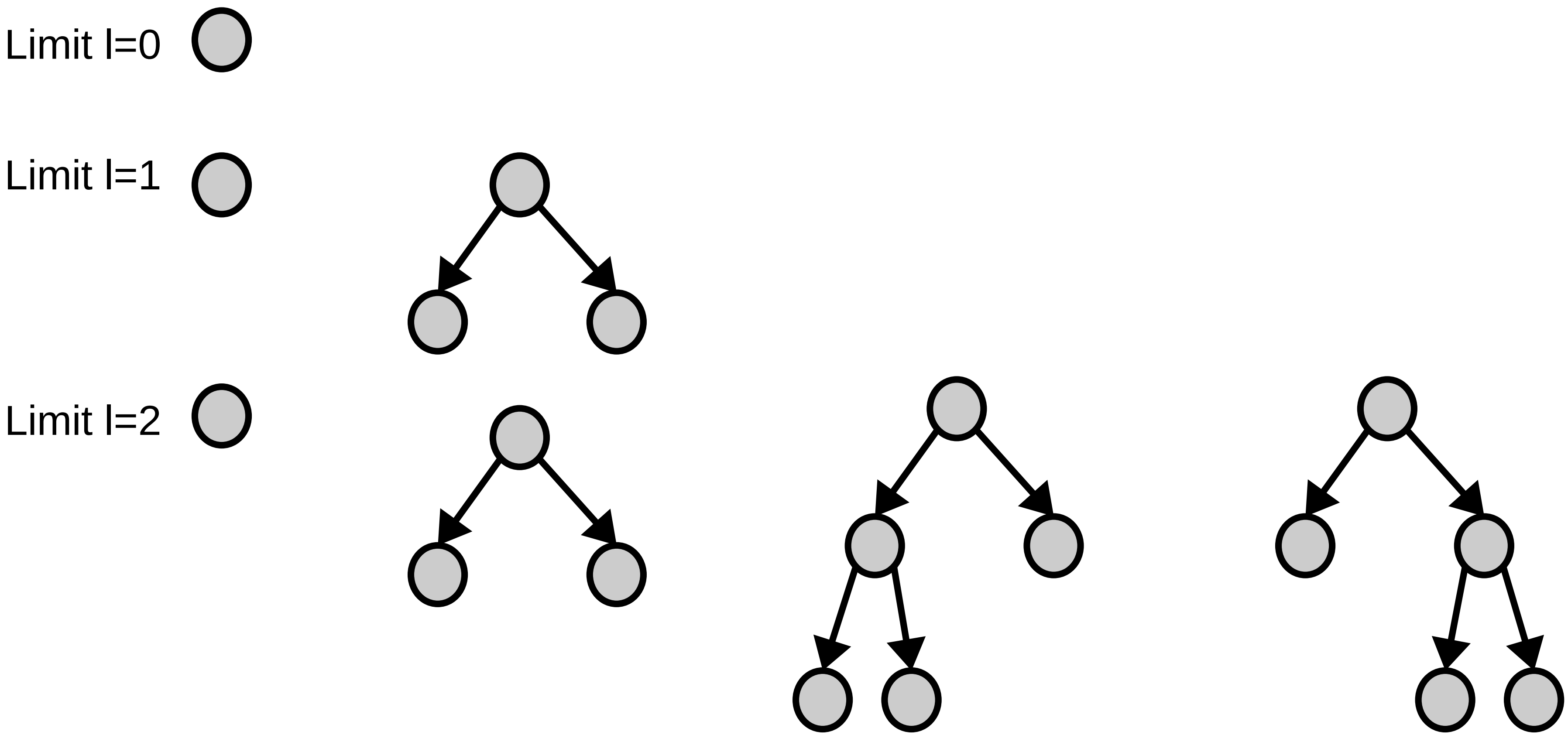
    ⇒ Limited-depth DFS with $l = 20$.
    - **Time complexity** (worst-case): $O(b^l)$
    - **Space complexity** (worst-case): $O(bl)$
- But most of the time, it is impossible to predict the maximum depth.

3. Iterative Deepening Search (IDS)

- Based on the idea of the limited-depth search, but it resolves the difficulty of knowing the depth limit ahead of time.
- Idea:
  - Try all depth limits in an increasing order.
  - That is, search first with the depth limit l=1, then l=2, l=3.., and so on until the solution is reached.
- Iterative deepening combines advantages of the depth-first and breadth-first search with only moderate computational overhead.

IDS

- Progressively increases the limit of the limited-depth depth-first search.

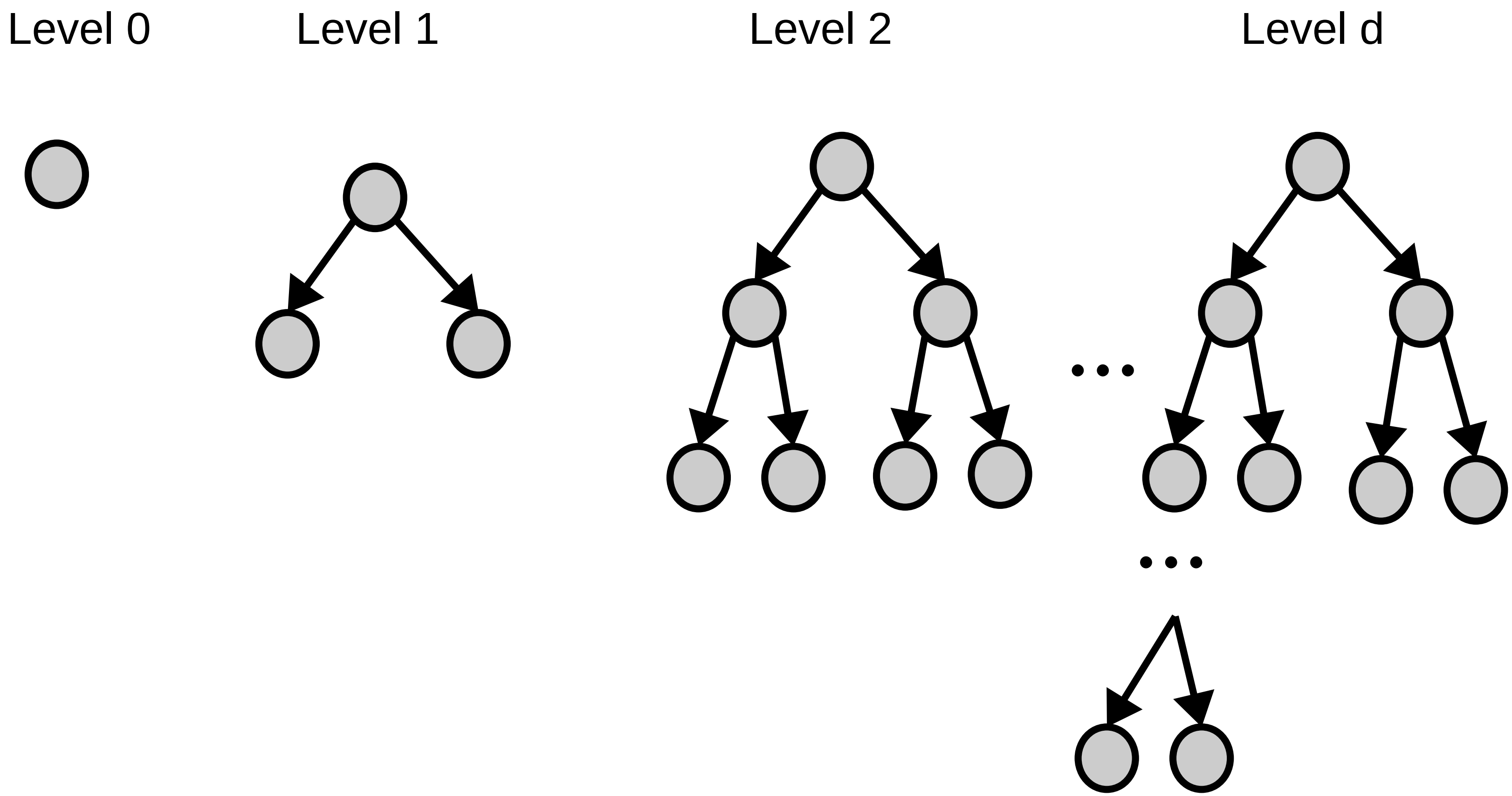


[ids1.sh]

Properties of IDS

- **Completeness**
  - Yes. The solution is reached if it exists (when the limit is always increased by 1).
- **Optimality**
  - Yes, for the smallest number of nodes.
- **Time complexity**
  - ?
- **Space complexity**
  - ?

IDS. Time Complexity



$= 1 + d.b + (d-1).b^2 + \ldots + (1)b^d$

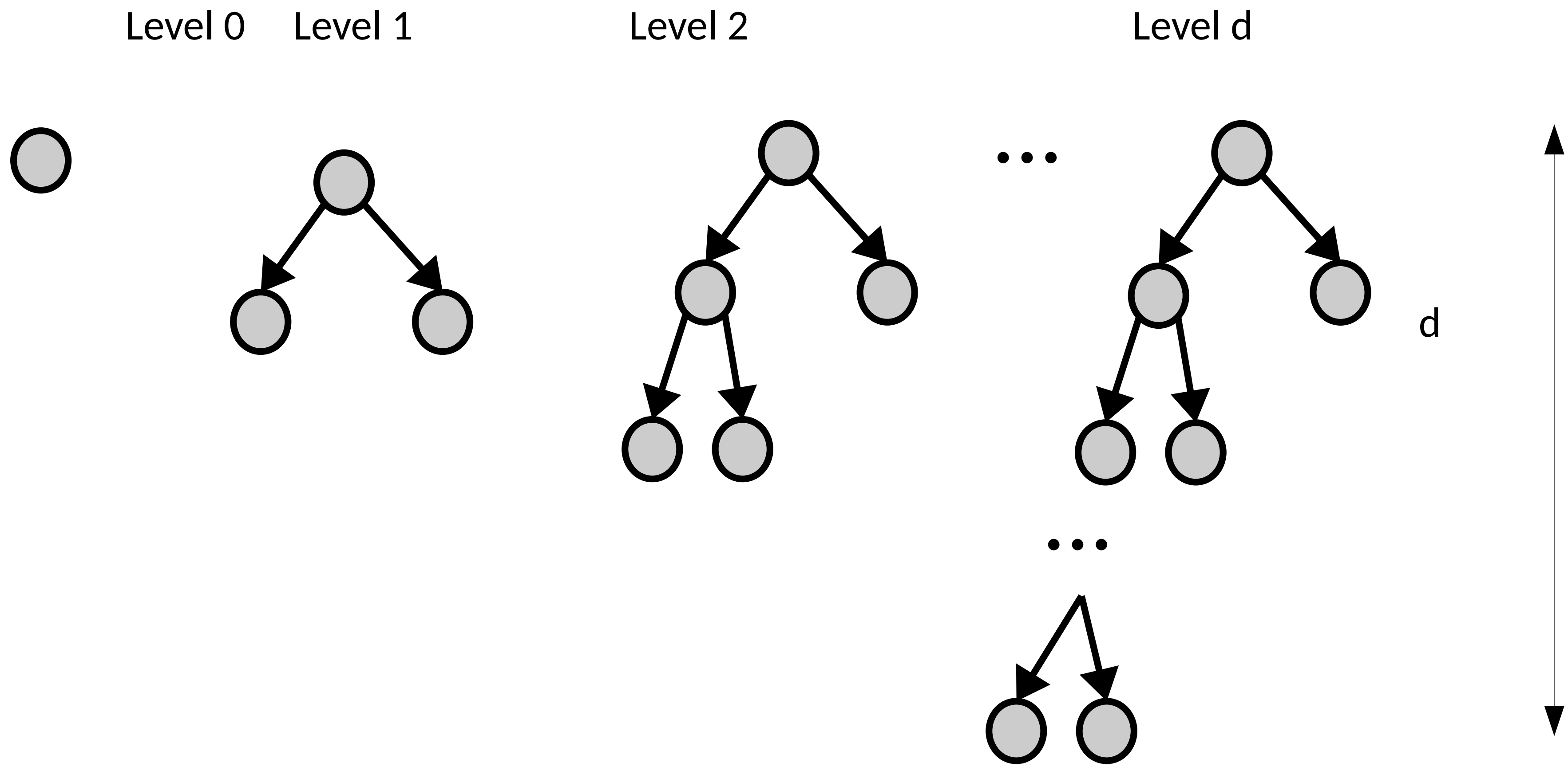
$= [b^{d+2} + d(b-1) + 1)] / [b – 1]^2$

$= O(b^d)$

Properties of IDS

- **Completeness**
  - Yes. The solution is reached if it exists (when the limit is always increased by 1).
- **Optimality**
  - Yes, for the smallest number of nodes.
- **Time complexity**
  - Worst-case $O(b^d)$
  - Exponential in the depth of the solution.
- **Space complexity**
  - ?

IDS. Space Complexity



$$\underbrace{\mathit{max}\,(O(1)\,,\quad O(b)\,,\quad O(2b)\,,\ \ldots\quad,\ O(db))}_{O(d.b)}$$

Properties of IDS

- **Completeness**
  - Yes. The solution is reached if it exists (when the limit is always increased by 1).
- **Optimality**
  - Yes, for the smallest number of nodes (and path cost is a non-decreasing function of depth).
- **Time complexity**
  - *Worst-case $O(b^d)$*
  - Exponential in the depth of the solution.
- **Space complexity**
  - *Worst-case $O(db)$* much better than BFS.

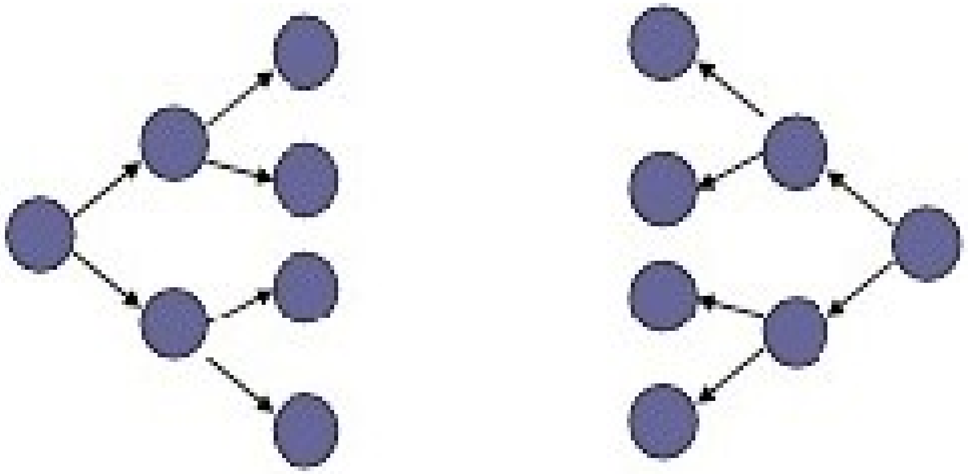
Compare IDS and BFS

- IDS and BFS are complete and optimal.
- Time overhead
  - Time complexity IDS is worse than BFS, but asymptotically the same since the most part of the nodes is at the last level.
    - Previous levels are explored multiple times
      - $= 1 + d.b + (d-1).b^2 + \ldots + (2)b^{d-1}$
        $= [b^{d+1} + d(b-1) + 1)] / [b - 1]^2 =$ $\mathbf{O(b^{d-1})}$
    - Last level: $\mathbf{O(b^d)}$, which is explored once.
    - So, the last level have more nodes to explore than all the previous levels even if they are explored several times.
  - Example with (b=10 and d=5)
    - $N(IDS) = d.b + (d-1).b^2 + .. + b^d = 123{,}540$ nodes expanded.
    - $N(BFS) = b + b^2 + .. + b^d = 111{,}110$ nodes expanded.
    - Difference is about 10%.
  - The majority of nodes are at the last level and they are examined once.
- Space complexity of IDS is linear, BFS is exponential.

4. Bidirectional Search

- Bi-directional search idea:

Initial state

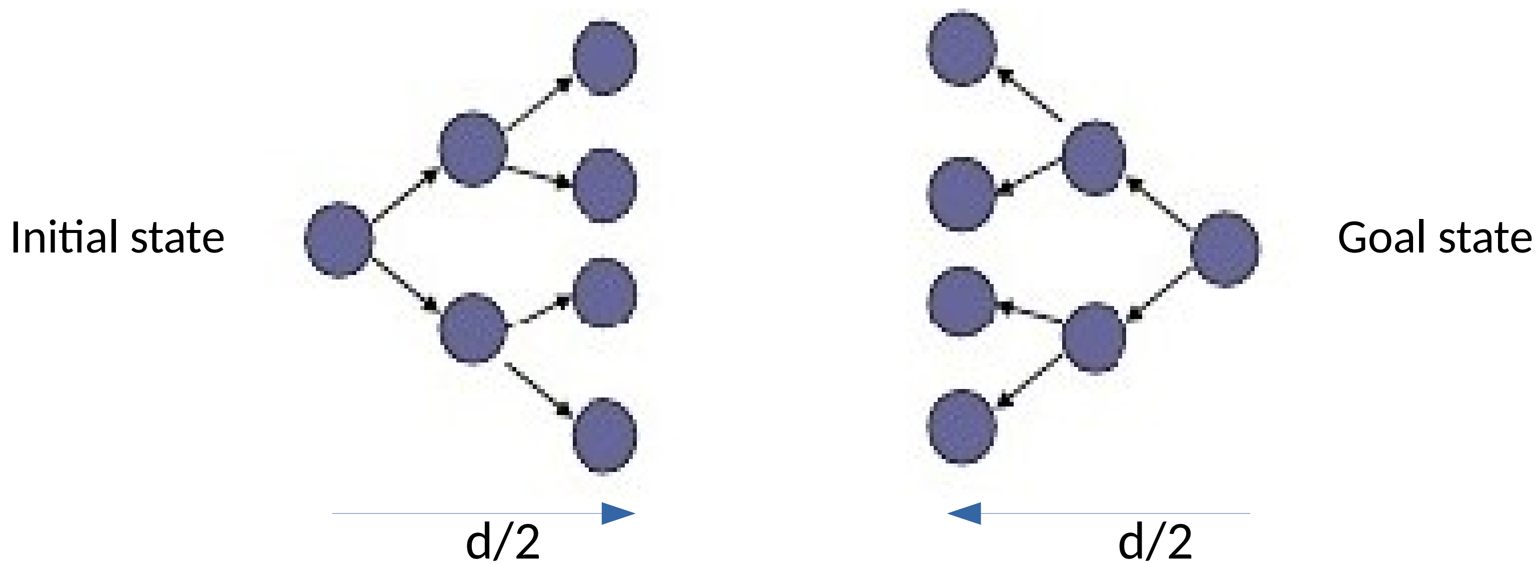


Goal state

- Search both from the initial state and the goal state.
  - Adaptable for BSF, DFS with limited depth and IDS.
- Use **inverse operators** for the goal-initiated search.
  - Not all problem.

Bidirectional Search

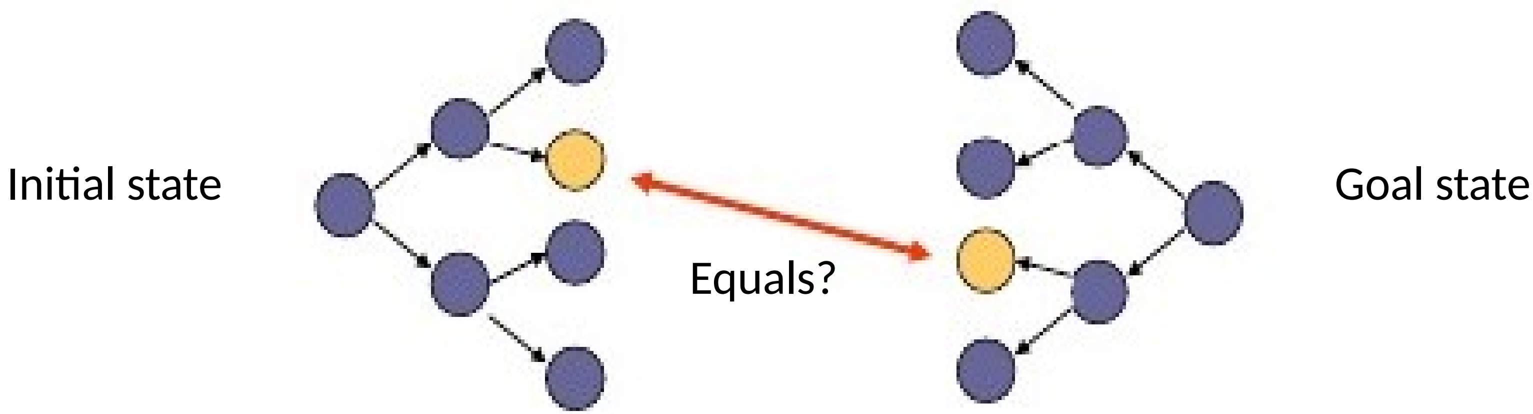
- Why bidirectional search? What is the benefit? Assume BFS.
  - Cut the depth of the search space by half.



  - $O(b^{d/2})$ for time and space complexity.

Bidirectional Search

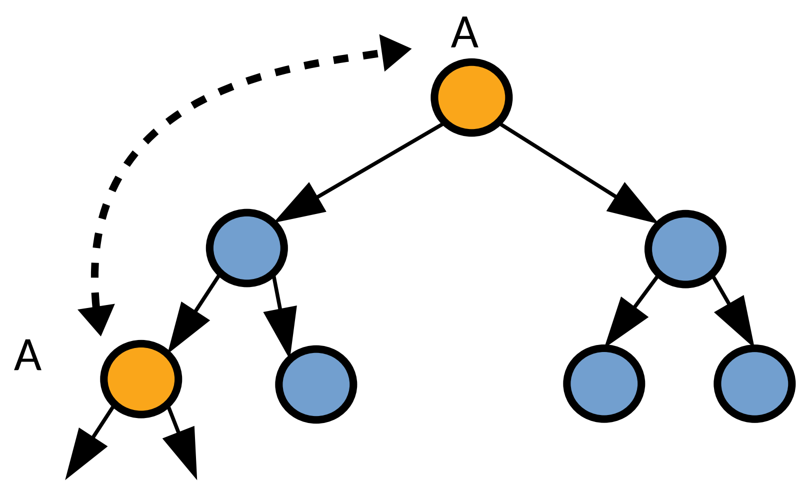
- What is necessary?
  - Merge the solutions



- How?
  - A hash table
    - The hash structure remembers the side of the tree the state was expanded first time. If the same state is reached from other side we have a solution.

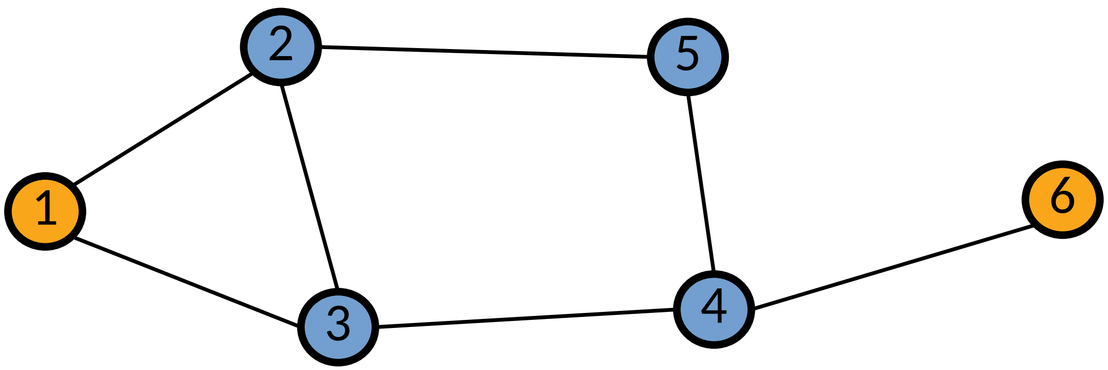
5. Elimination of State Repeats

- While searching the state space for the solution we can encounter the same state many times.
- Failure to detect repeated states can cause exponentially more work. Why?
  - The search space is a no more a search tree but a graph.



Elimination of State Repeats: BFS

- In BFS, we can safely eliminate all repeats of the same state.
  - Can this wreck completeness? No: we proceed iteratively on depth
  - Can this wreck optimality? No: depth(1-2-3) > depth(1-3) always



Elimination of State Repeats: BFS

- Implementation: very simple fix: never expand a state twice.
  - Store the explored list (aka. **closed list**) as a separated list.
  - In Python, prefer a Set over a List for efficiency.

```
function GENERAL-SEARCH(problem) returns solution
  var open-list ← MAKE-LIST(MAKE-NODE(INITIAL-STATE[problem]))
  LOOP
     IF EMPTY(open-list) THEN return failure
     node ← REMOVE-FRONT-LIST(open-list)
     IF IS-GOAL(problem, STATE[node]) THEN return the related solution
     open-list ← ADD-IN-LIST(GET-SUCCESSORS(node, problem), open-list)
end
```

Elimination of State Repeats: BFS

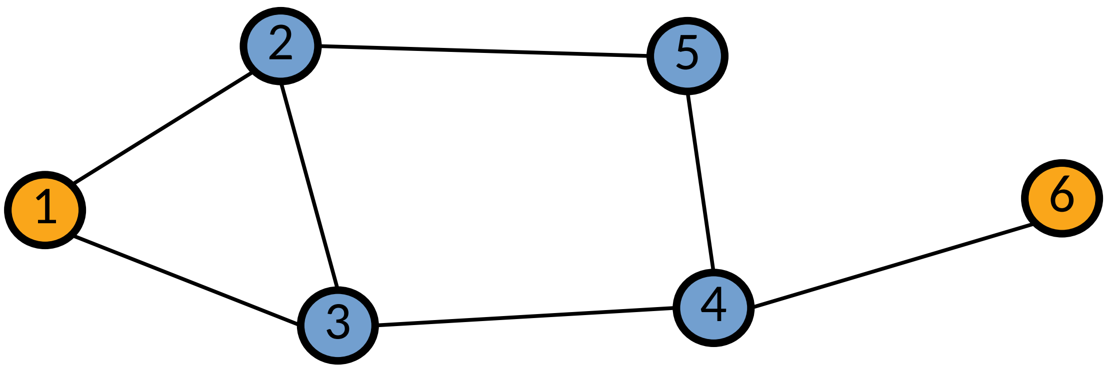
- Implementation: very simple fix: never expand a state twice.
  - Store the explored list (aka. **closed list**) as a separated list.
  - In Python, prefer a Set over a List for efficiency.

```
function GRAPH-SEARCH(problem) returns solution
  var open-list ← MAKE-LIST(MAKE-NODE(INITIAL-STATE[problem]))
  var closed-list ← MAKE-SET(MAKE-NODE(INITIAL-STATE[problem])
  LOOP
     IF EMPTY(open-list) THEN return failure
     node ← REMOVE-FRONT(open-list)
      closed-list.add(STATE[node])
     IF IS-GOAL(problem, STATE[node]) THEN return the solution
     neighbors ← GET-SUCCESSORS(node, problem)
     FOR neighbor in neighbors DO
        IF STATE[neighbor] is not in closed-list THEN
          open-list ← ADD-IN-LIST(neighbor, open-list)
end
```

[bfsa.sh; bfsb.sh]

Elimination of State Repeats: DFS

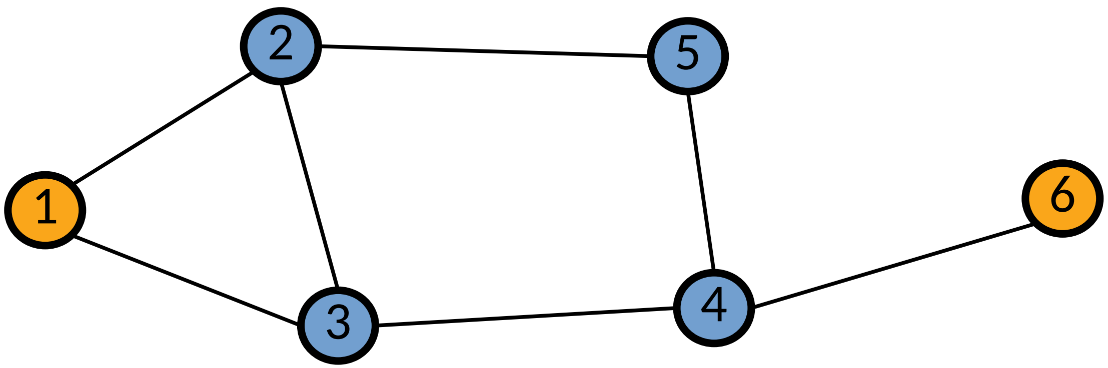
- In DFS, we can also safely eliminate all repeats of the same state. Why?
  - Can this wreck completeness? DFS is not complete anyway
  - Can this wreck optimality? 1-2-3 > 1-3 always



- Use same fix than BFS: a set of explored nodes (closed-list).

Elimination of State Repeats: IDS

- In IDS, we cannot eliminate all repeats of the same state as in the previous algorithms. Why?



Depth of the solution = 3

If path 1-2-3-4 is examined first, it prevents path 1-3-4-6, so it wrecks optimality

Elimination of State Repeats: IDS

- Use of closed-list is not possible, however:
  - We could eliminate loops.
    - No need to use an extra list, use the current branch.
  - We could eliminate state explored at a higher depth than the previous visit.
    - Use a hashmap (dictionary in Python).

```
closed-list[node] = depth
neighbors ← GET-SUCCESSORS(node, problem)
FOR neighbor in neighbors DO
   loop ← node in current_path
   isAlreadyVisited←closed-list[node] <= depth
   IF not loop and not isAlreadyVisited THEN
      open-list ← ADD-IN-LIST(neighbor, open-list)
```

Elimination of State Repeats : Complexity

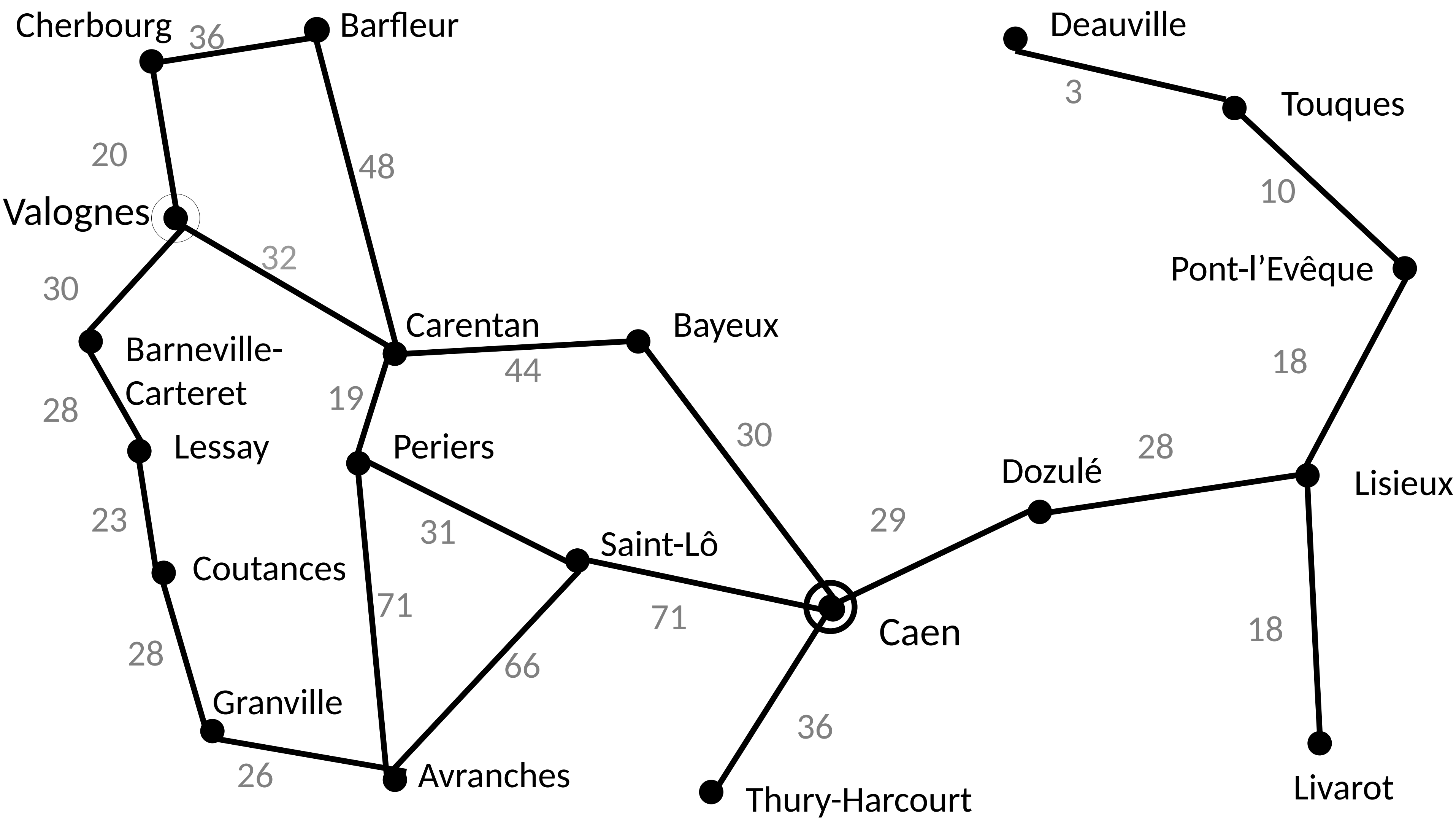
- How the space complexity is affected by using a closed list?
  - BFS
    - The explored list size: $O(b^d)$
    - So, the space complexity remains exponential $O(b^{d+1})$
  - DFS
    - The explored list size: $O(b^m)$
    - So, the space complexity becomes exponential! $O(b^m)$
  - Depth-Limited DFS
    - The explored list size: $O(b^d)$
    - So, the space complexity becomes exponential! $O(b^d)$
  - IDS
    - If we use a dictionary of explored nodes: $O(b^d)$
    - So, the space complexity becomes exponential! $O(b^d)$
- Note: In practice, the closed list reduces the number of explored nodes, therefore the average space complexity.
- Note: Not all problem needs closed-list (e.g, puzzle-8).

6. Case of Minimum Cost Path-Search Problem

- **New problem statement**
  - Adds weights or costs to operators (links).
    - e.g., distance between two neighboring cities.
  - Path cost function *g(n)*
    - Path cost from the initial state to node n.

Searching for the Minimum Cost Path

- Traveler example with distances [km].



- Optimal path: the shortest distance path.

3- Uniform Cost Search (aka Dijkstra algorithm)

- The basic algorithm for finding the minimum cost path:
  - Dijkstra's shortest path (only with non-negative edge costs).
  - In AI, the algorithm goes under the name: **Uniform Cost Search**.
- Strategy
  - For each node *n*, keep the cost from the start: g(*n*)
  - At each search step, expand the cheapest node (minimum *g*(*n*)).
- Note
  - When operator costs are all equal to 1 it is equivalent to BFS. It finds the shortest path in terms of number of visited nodes.

Uniform Cost Search (UCS)

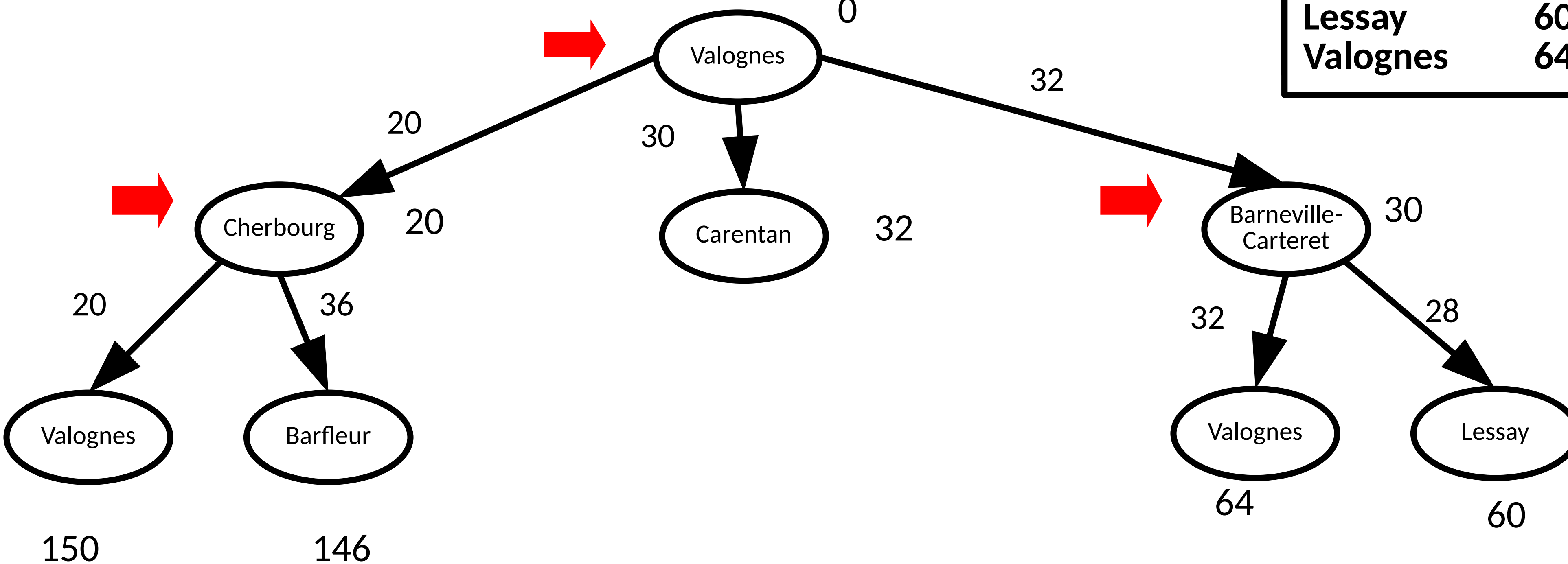
- Strategy: expand the shallowest node first
- Implementation of the open-list: **Priority queue** ordered by g(n).
  - **ADD-IN-LIST**: Ordered nodes by **current path cost.**

```
function GENERAL-SEARCH(problem) returns solution
  var open-list ← MAKE-LIST(MAKE-NODE(INITIAL-STATE[problem]))
  LOOP
     IF EMPTY(open-list) THEN return failure
     node ← REMOVE-FRONT-LIST(open-list)
     IF IS-GOAL(problem, STATE[node]) THEN return the related solution
     open-list ← ADD-IN-LIST(GET-SUCCESSORS(node, problem), open-list)
end
```

Uniform Cost Search

- Implementation (same general algorithm).
  - The open list is: **Priority queue** ordered by g(n).
  - ADD-IN-LIST: Ordered nodes by current path cost.

	g(n)
Carentan	32
Valognes	40
Barfleur	56
Lessay	**60**
Valognes	**64**

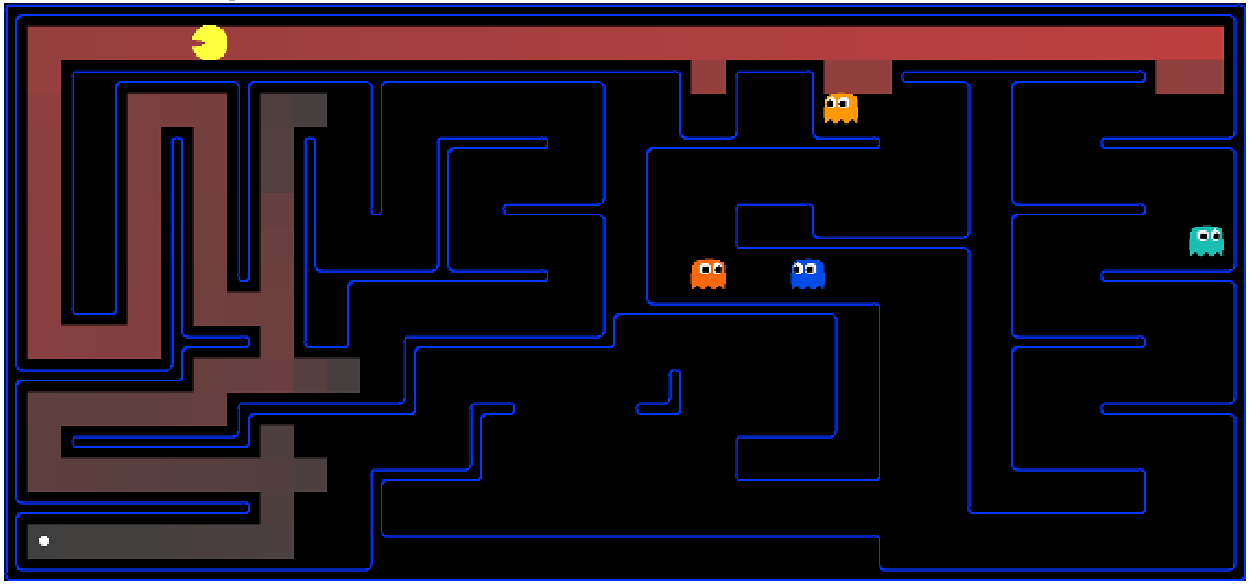


Properties of the Uniform Cost Path

- **Completeness**
  - Yes, assuming that operator costs are non-negative (the cost of path never decreases).
    - $g(n) \leq g(successor(n))$
  - In the worst case, all node will be examined.
- **Optimality**
  - Yes. Returns the least-cost path.
  - At each search step, we follow the cheapest route.
- **Time complexity**
  - Worst case $O(b^d)$
  - In practice: proportional to the number of nodes for paths with $g(n)$ < optimal cost.
- **Space complexity**
  - Worst case $O(b^{d+1})$
  - In practice: proportional to the number of nodes for paths with

Action Cost

- Cost is the **only way** to express constraint on the problem. Cost can favor or penalize paths to the solution without prohibiting them.
- Example 1: By changing the cost function, we can encourage Pacman to find different paths.



  - For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.
- Example 2: project Formula one
  - Penalize sand routes.

[ucs.sh]

Summary

- Uniformed algorithms

Algorithm	Completeness	Optimality	**Time complexity** (Worst-case)	**Space complexity** (Worst-case)
BFS	YES	YES	$O(b^d)$	$O(b^{d+1})$
DFS	NO	NO	$O(b^d)$	$O(b.m)$
Limited-Depth DFS	YES (if l≥d)	NO	$O(b^d)$	$O(b.l)$
IDS	YES	YES	$O(b^d)$	$O(b.d)$
UCS	YES	YES	$O(b^d)$	$O(b^{d+1})$