Solving Problems by Searching in Python

Gloria Beraldo (gloria.beraldo@unipd.it)
Department of Information Engineering, University of Padova

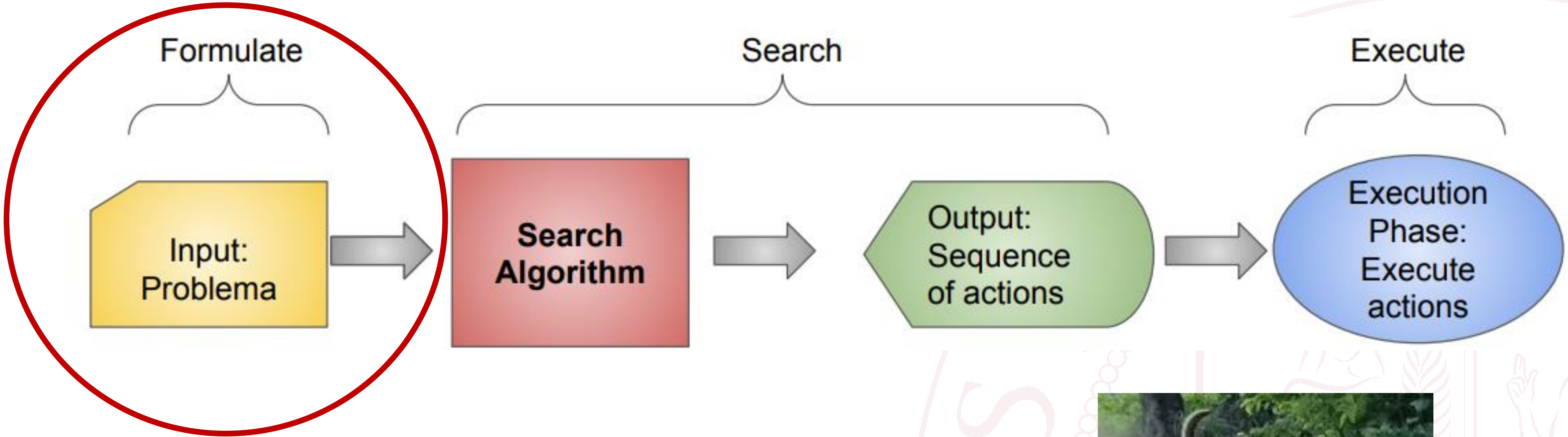
Topics:

- Problem Formulation
- Search Tree
- Node
- Queues
- Best-First Search
- Greedy Best-First Search, A*, Weighted A*, Uniform-cost
- Breadth-First Search
- Iterative-Deepening Search

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Introduction to Search

Search can be defined as the steps to find a sequence of actions aiming the goal state



Problem Formulation

- **Initial state**: the state of the agent starts in the beginning.
- **Actions**: set of actions applicable in the current state.

  ACTIONS(s): return possible actions applicable in state s.

- **Transition model**: define the consequence of the each action application over the state.

  RESULT(s, a): returns the state achieve after applying action a over state s.

- **State space:**
  a) It is defined by the initial state, actions, and transition model.
  b) It has all states which can be obtained by applying a sequence of actions.

Problem Formulation

- **Goal test:** verify when a goal state is reached.

  IS_GOAL(s): return True when s is equal to the goal

- **Path cost function:** measures the solution quality by evaluating each path cost.

The path is the sequence of states within the state space provided by a sequence of actions
c(s,a,s'): step cost to reach state s' from state s by applying action a.

ACTION_COST(s,a,s')

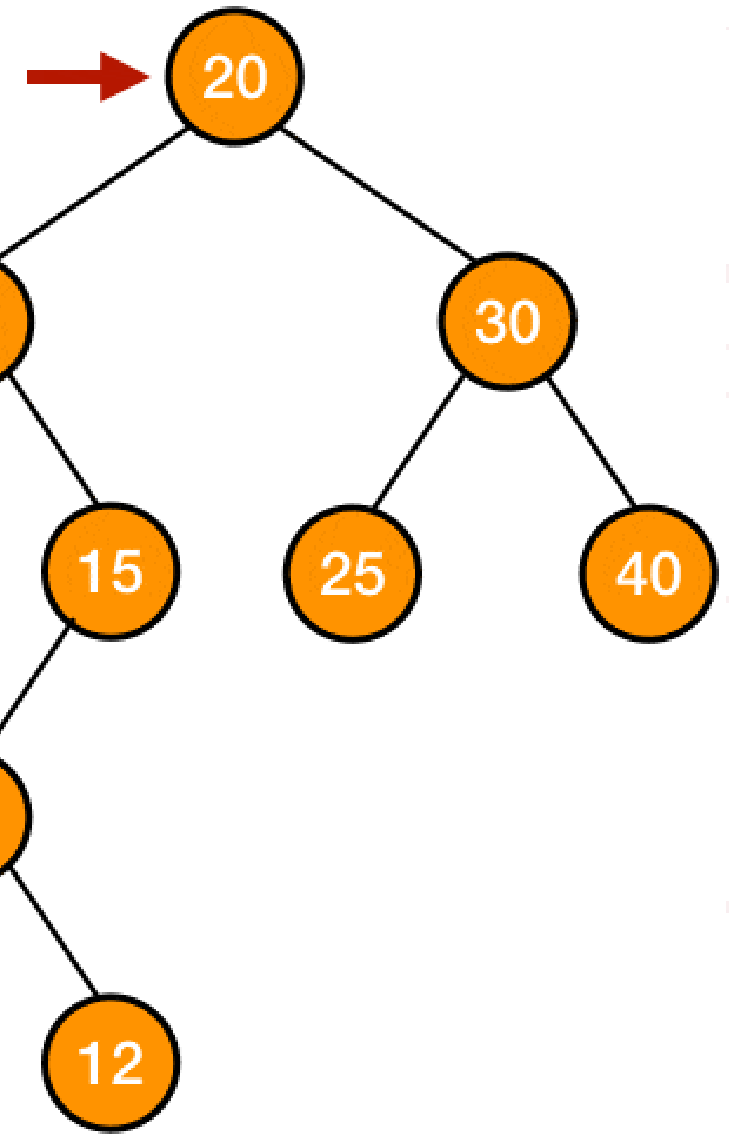
The optimal solution will have the lowest path cost.

class: Problem

```python
class Problem(object):
    """The abstract class for a formal problem. A new domain subclasses this,
    overriding `actions` and `results`, and perhaps other methods.
    The default heuristic is 0 and the default action cost is 1 for all states.
    When you create an instance of a subclass, specify `initial`, and `goal` states
    (or give an `is_goal` method) and perhaps other keyword args for the subclass."""

    def __init__(self, initial=None, goal=None, **kwds):
        self.__dict__.update(initial=initial, goal=goal, **kwds)

    def actions(self, state):        raise NotImplementedError
    def result(self, state, action): raise NotImplementedError
    def is_goal(self, state):        return state == self.goal
    def action_cost(self, s, a, s1): return 1
    def h(self, node):               return 0

    def __str__(self):
        return '{}({!r}, {!r})'.format(
            type(self).__name__, self.initial, self.goal)
```

Searching via Search Tree

we consider algorithms that superimpose a **search tree** onto the state space graph, forming various **paths starting from the initial state** and trying to **find one that reaches a target state**.



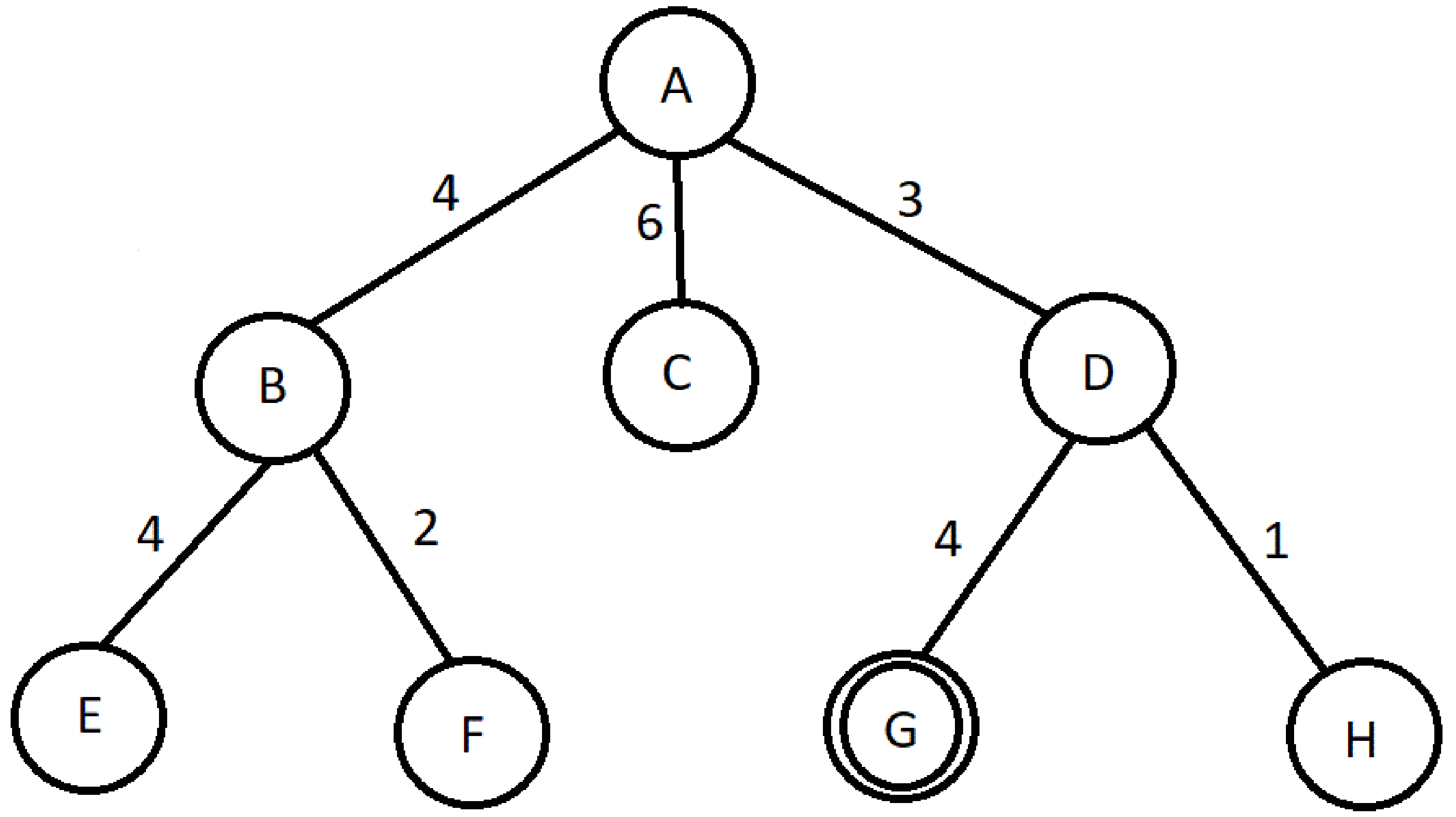
Each node in the search tree corresponds to a **state in the state space**,

and **branches** in the search tree correspond to **actions**.

The **root of the tree** corresponds to the **initial state** of the problem.

class: Node

```python
class Node:
    "A Node in a search tree."
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.__dict__.update(state=state, parent=parent, action=action, path_cost=path_cost)

    def __repr__(self): return '<{}>'.format(self.state)
    def __len__(self): return 0 if self.parent is None else (1 + len(self.parent))
    def __lt__(self, other): return self.path_cost < other.path_cost
```



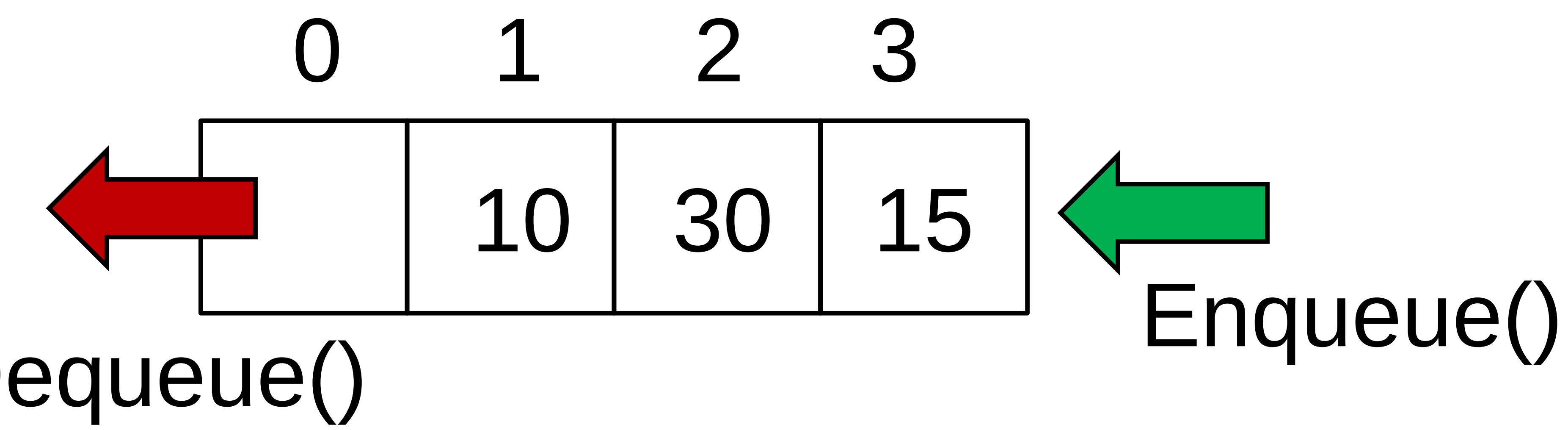
class: Node

```python
failure = Node('failure', path_cost=math.inf) # Indicates an algorithm couldn't find a solution.
cutoff  = Node('cutoff',  path_cost=math.inf) # Indicates iterative deepening search was cut off.


def expand(problem, node):
    "Expand a node, generating the children nodes."
    s = node.state
    for action in problem.actions(s):
        s1 = problem.result(s, action)
        cost = node.path_cost + problem.action_cost(s, action, s1)
        yield Node(s1, node, action, cost)


def path_actions(node):
    "The sequence of actions to get to this node."
    if node.parent is None:
        return []
    return path_actions(node.parent) + [node.action]


def path_states(node):
    "The sequence of states to get to this node."
    if node in (cutoff, failure, None):
        return []
    return path_states(node.parent) + [node.state]
```

Queues for Search Algorithms Implementation

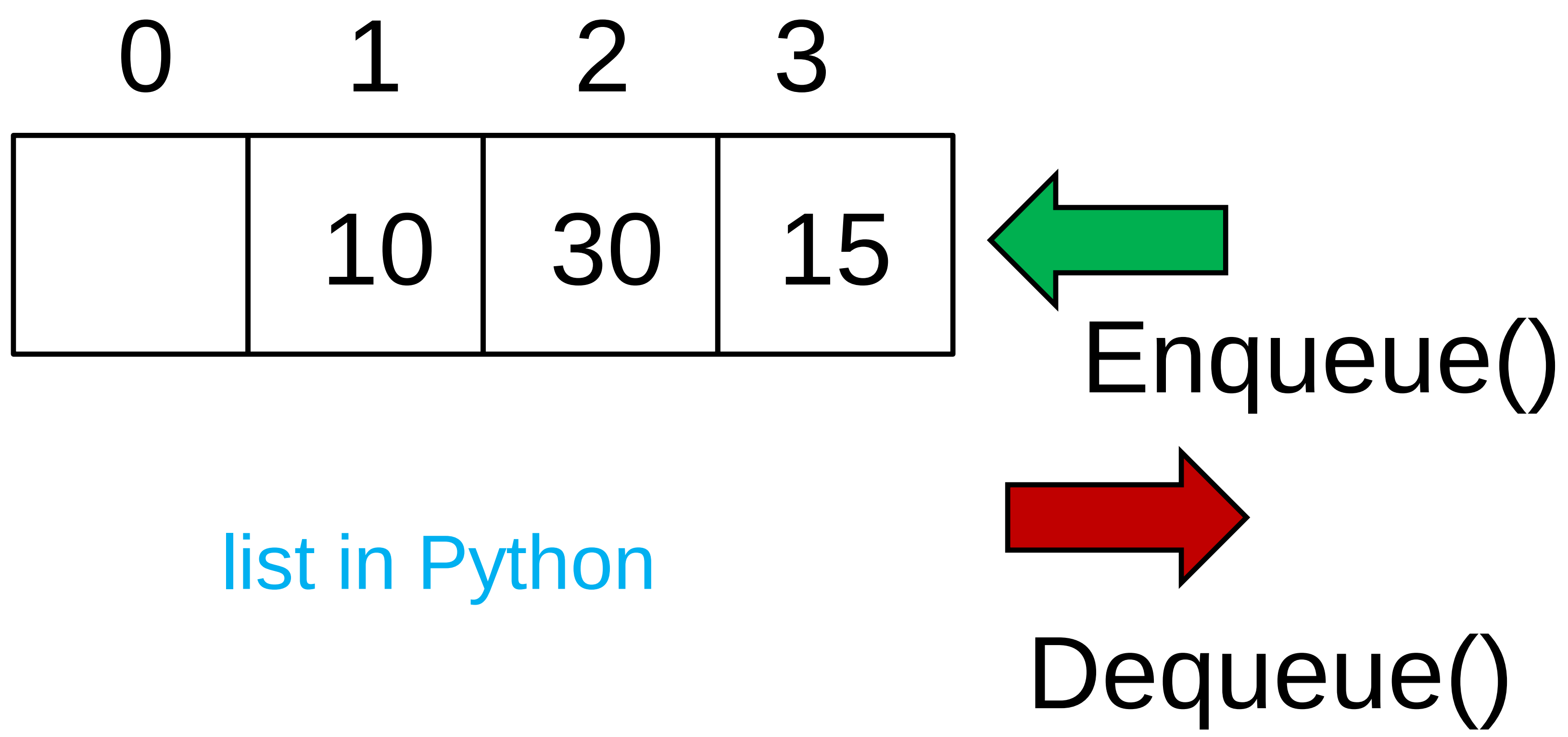
In the search algorithms, we use three kinds of queues:

First-In, First-Out (FIFO)



deque in Python

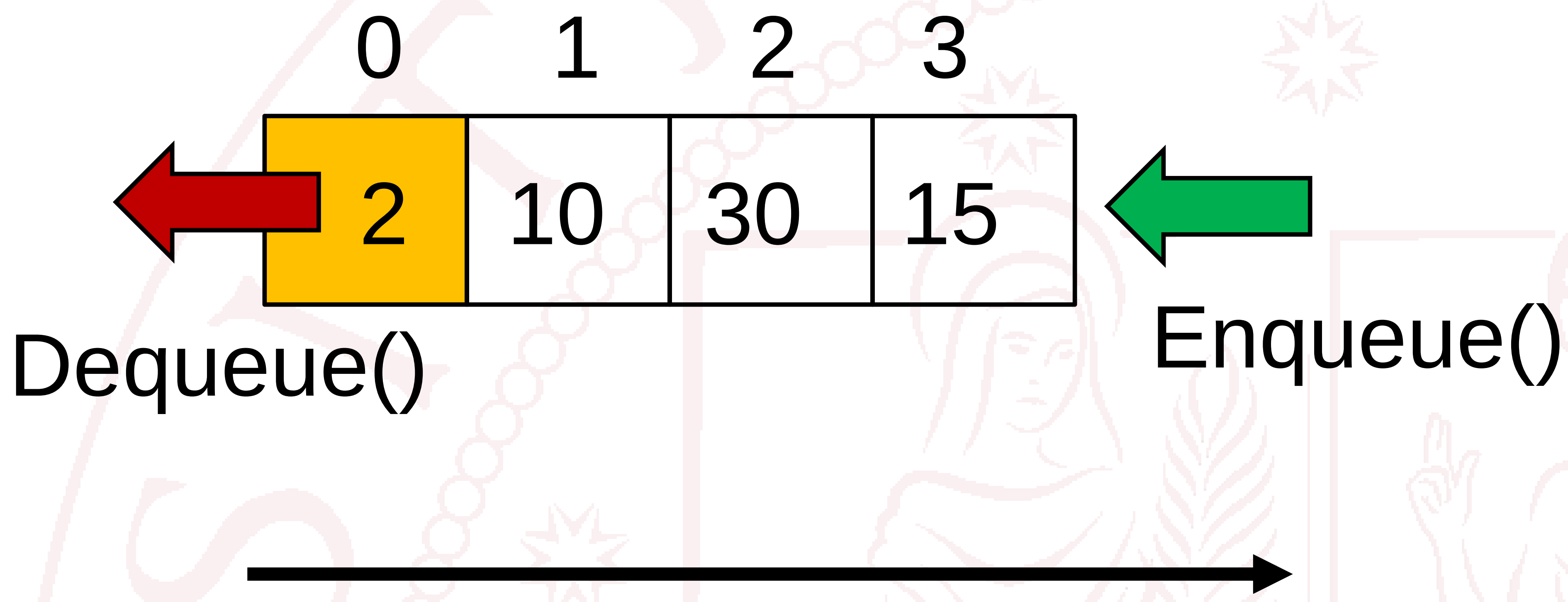
Last-In, First-Out (LIFO)



list in Python

Priority Queue

Element with the highest priority



DECREASING PRIORITY ORDER

PriorityQueue class based on heap in Python

Best-First Search (BFS): PseudoCode

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Best-First Search in Python

```
def best_first_search(problem, f):
    "Search nodes with minimum f(node) value first."
    node = Node(problem.initial)
    frontier = PriorityQueue([node], key=f)
    reached = {problem.initial: node}
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        for child in expand(problem, node):
            s = child.state
            if s not in reached or child.path_cost < reached[s].path_cost:
                reached[s] = child
                frontier.add(child)
    return failure
```

each child node is added to the frontier if it has not been reached previously or is added again if it has now been reached with a lower cost path than the previous ones

Best-First Search in Python

```python
def best_first_search(problem, f):
    "Search nodes with minimum f(node) value first."
    node = Node(problem.initial)
    frontier = PriorityQueue([node], key=f)
    reached = {problem.initial: node}
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        for child in expand(problem, node):
            s = child.state
            if s not in reached or child.path_cost < reached[s].path_cost:
                reached[s] = child
                frontier.add(child)
    return failure
```

From best_first_search to other search algorithms

```
def best_first_search(problem, f):
    "Search nodes with minimum f(node) value first."
    node = Node(problem.initial)
    frontier = PriorityQueue([node], key=f)
    reached = {problem.initial: node}
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        for child in expand(problem, node):
            s = child.state
            if s not in reached or child.path_cost < reached[s].path_cost:
                reached[s] = child
                frontier.add(child)
    return failure
```

We will modify the evaluation function f(n)

Breadth-First-BFS & Depth-First-BFS

```python
def breadth_first_bfs(problem):
    "Search shallowest nodes in the search tree first; using best-first."
    return best_first_search(problem, f=len)
```

```python
def depth_first_bfs(problem):
    "Search deepest nodes in the search tree first; using best-first."
    return best_first_search(problem, f=lambda n: -len(n))
```

Check how len is computed in the Node class

Greedy best-first search in Python

It is possible to apply the Best-First Search algorithm

The evaluation function is f(n) = h(n)

h(n) = estimated cost to goal from n (i.e., the closest to reach to the goal)

```
def greedy_bfs(problem, h=None):
    """Search nodes with minimum h(n)."""
    h = h or problem.h
    return best_first_search(problem, f=h)
```

A* Search in Python

It is possible to apply the Best-First Search algorithm

The evaluation function is f(n) = **g(n)** + h(n):

g(n) = cost so far to reach n

h(n) = estimated cost to goal from n

```python
def g(n): return n.path_cost
```

```python
class Node:
    "A Node in a search tree."
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.__dict__.update(state=state, parent=parent, action=action, path_cost=path_cost)

    def __repr__(self): return '<{}>'.format(self.state)
    def __len__(self): return 0 if self.parent is None else (1 + len(self.parent))
    def __lt__(self, other): return self.path_cost < other.path_cost
```

A* Search in Python

It is possible to apply the Best-First Search algorithm

The evaluation function is $f(n) = g(n) + h(n)$:

$g(n)$ = cost so far to reach n

$h(n)$ = estimated cost to goal from n

```python
def astar_search(problem, h=None):
    """Search nodes with minimum f(n) = g(n) + h(n)."""
    h = h or problem.h
    return best_first_search(problem, f=lambda n: g(n) + h(n))
```

Weighted A* Search in Python

It is possible to apply the Best-First Search algorithm

The evaluation function is $f(n) = g(n) + weight * h(n)$:

$g(n)$ = cost so far to reach n

$h(n)$ = estimated cost to goal from n

```
def weighted_astar_search(problem, h=None, weight=1.4):
    """Search nodes with minimum f(n) = g(n) + weight * h(n)."""
    h = h or problem.h
    return best_first_search(problem, f=lambda n: g(n) + weight * h(n))
```

Uniform-cost Search in Python

It is possible to apply the Best-First Search algorithm

The evaluation function is f(n) = g(n)

g(n) = cost so far to reach n

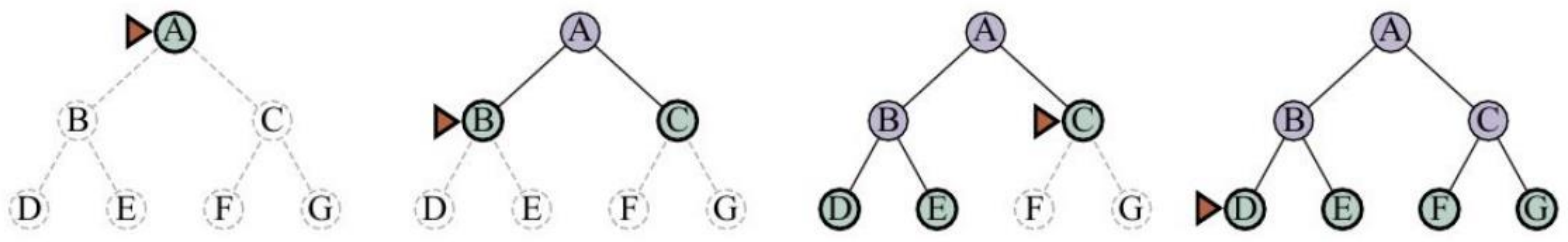
```
def uniform_cost_search(problem):
    "Search nodes with minimum path cost first."
    return best_first_search(problem, f=g)
```

Breadth-First Search: PseudoCode

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

NB: You add
to the left

- Frontier = {**A**} → Frontier = {**B**, C} → Frontier = {D, E, **C**} → Frontier = {**D**, E, F, G}



Breadth-First Search in Python

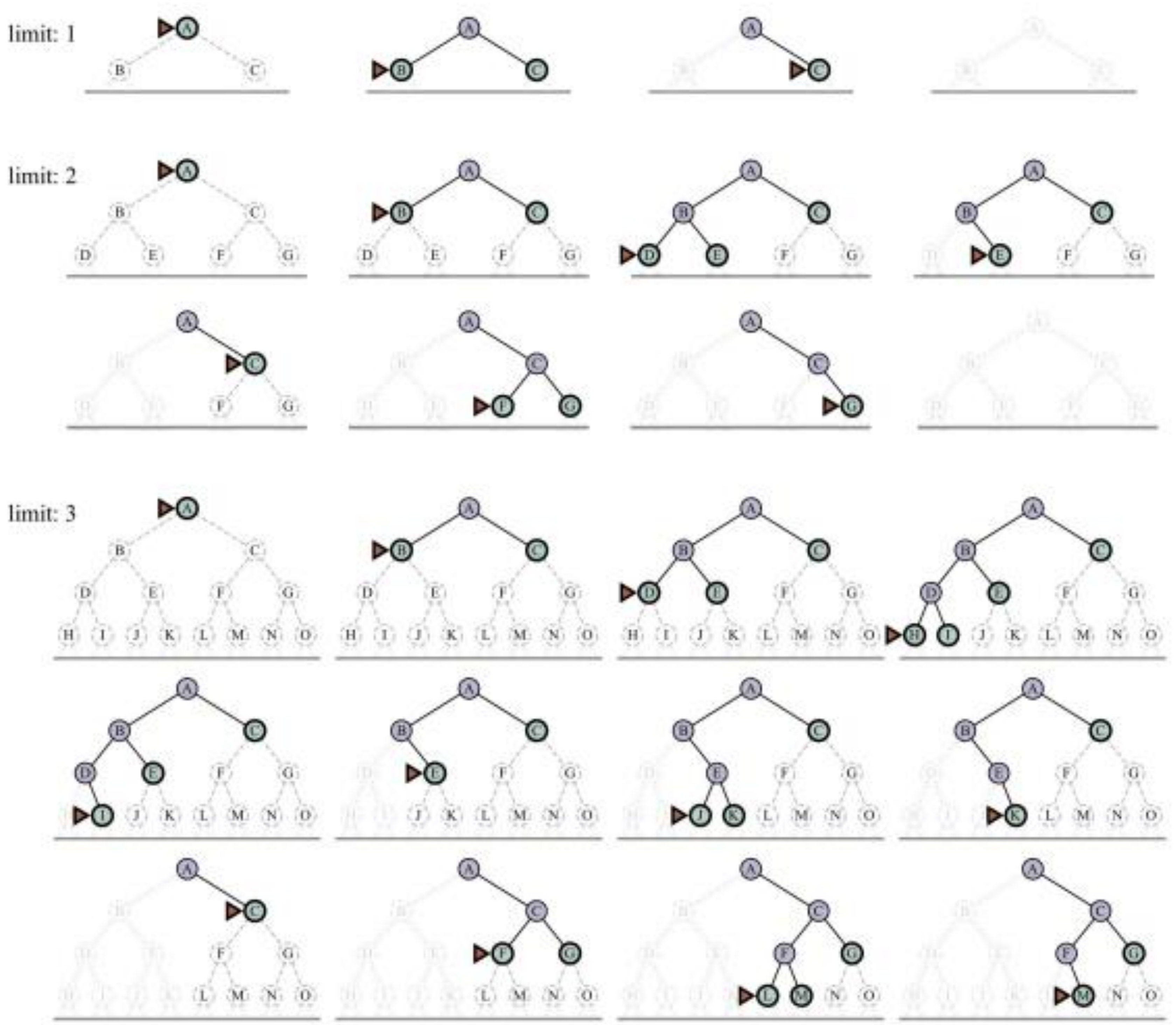
```python
def breadth_first_search(problem):
    "Search shallowest nodes in the search tree first."
    node = Node(problem.initial)
    if problem.is_goal(problem.initial):
        return node
    frontier = FIFOQueue([node])
    reached = {problem.initial}
    while frontier:
        node = frontier.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s not in reached:
                reached.add(s)
                frontier.appendleft(child)
    return failure
```

```
deque:  deque([1, 2, 3])

The deque after appending at right is :
deque([1, 2, 3, 4])

The deque after appending at left is :
deque([6, 1, 2, 3, 4])
```

Iterative-Deepening Search: PseudoCode

iteratively increasing the limit

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) >  ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```



Iterative-Deepening Search in Python

```python
def iterative_deepening_search(problem):
    "Do depth-limited search with increasing depth limits."
    for limit in range(1, sys.maxsize):
        result = depth_limited_search(problem, limit)
        if result != cutoff:
            return result


def depth_limited_search(problem, limit=10):
    "Search deepest nodes in the search tree first."
    frontier = LIFOQueue([Node(problem.initial)])
    result = failure
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        elif len(node) >= limit:
            result = cutoff
        elif not is_cycle(node):
            for child in expand(problem, node):
                frontier.append(child)
    return result
```

Questions