

# PROBLEM SOLVING AND SEARCH

## CHAPTER 3

## Example: Apulia

On holiday in Apulia.

Currently in Bitetto.

Bus leaves tomorrow from Trani

Formulate goal:

be in Trani

Formulate problem:

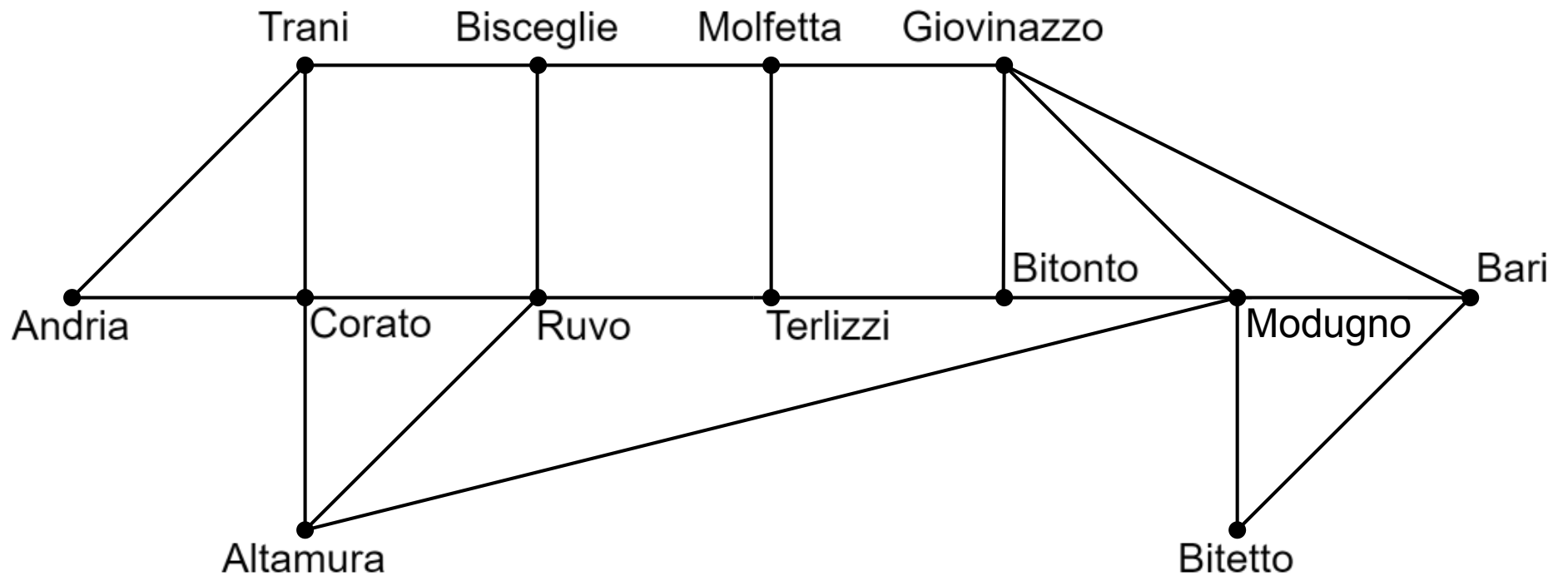
states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g., Modugno, Altamura, Corato, Trani

## Example: Apulia



## Single-state problem formulation

A **problem** is defined by four items:

**initial state** e.g., "at Bitetto"

**successor function**  $S(x)$  = set of **action-state pairs**

e.g.,  $S(\text{Bitetto}) = \{ \underbrace{\langle \text{Bitetto} \rightarrow \text{Modugno} \rangle}_{\text{Action}}, \underbrace{\text{Modugno} \rangle}_{\text{State}}, \dots \}$

**goal test**, can be

**explicit**, e.g.,  $x = \text{"at Trani"}$  **implicit**, e.g.,  
 $\text{NoDirt}(x)$

**path cost** (additive)

e.g., sum of distances, number of actions executed, etc.

$c(x, a, y)$  is the **step cost**, assumed to be  $\geq 0$

A **solution** is a sequence of actions

leading from the initial state to a goal state

# Tree search algorithms

Basic idea:

offline, simulated exploration of state space  
by generating successors of already-explored states  
(a.k.a. **expanding** states)

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

## Implementation: states vs. nodes

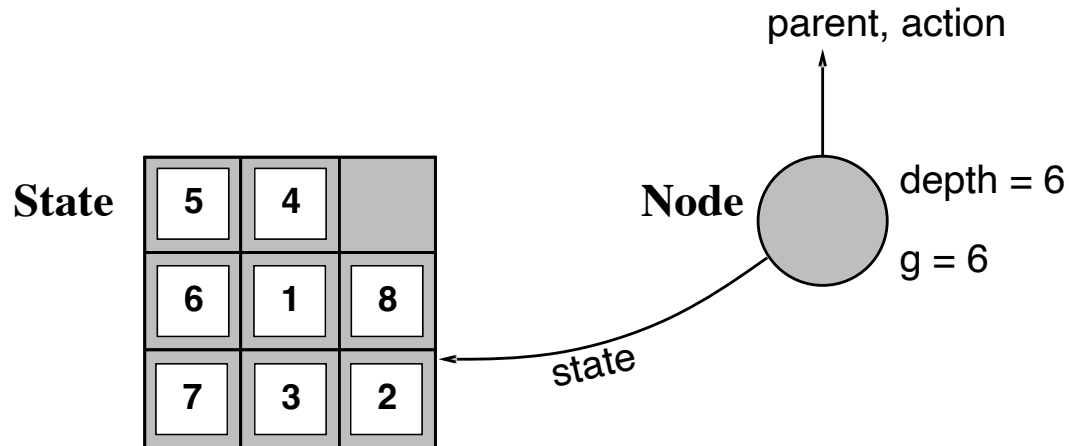
in the search tree

A **state** is a (representation of) a physical configuration

A **node** is a data structure constituting part of a search tree

includes **parent**, **children**, **depth**, **path cost**  $g(x)$

**States do not have parents, children, depth, or path cost!**



The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem to create the corresponding states.

## Implementation: general tree search

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return node
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

---

```
function EXPAND(node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
    s ← a new NODE
    PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
```

build a  
successor  
node

## Graph search

**function** GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

*closed* ← an empty set

*fringe* ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)

**loop do**

**if** *fringe* is empty **then return** failure

*node* ← REMOVE-FRONT(*fringe*)

**if** GOAL-TEST(*problem*, STATE[*node*]) **then return** *node*

**if** STATE[*node*] is not in *closed* **then**

        add STATE[*node*] to *closed*

*fringe* ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

**end**



## Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

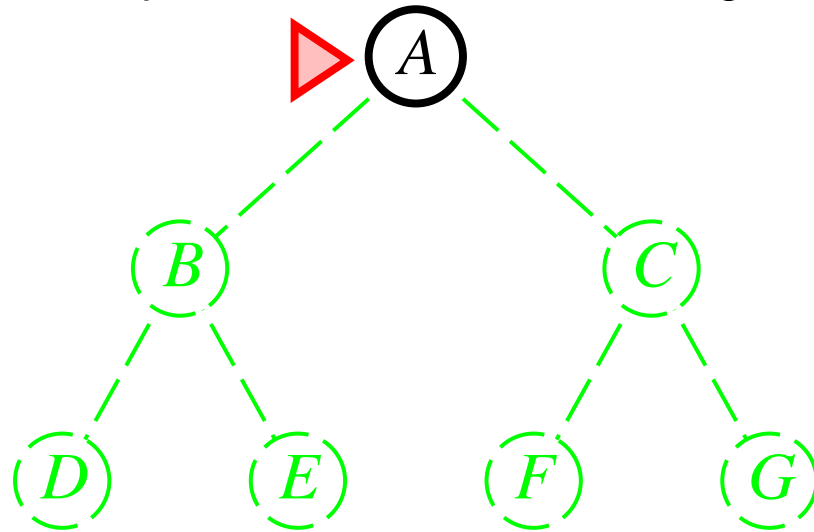
Iterative deepening search

## Breadth-first search

Expand shallowest unexpanded node

**Implementation:**

*fringe* is a FIFO queue, i.e., new successors go at end



## Uniform-cost search

Expand least-cost unexpanded node

### Implementation:

*fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost  $\geq \epsilon$  if cost == 0 and NoOp -> infinite LOOP

Time?? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$   
where  $C^*$  is the cost of the optimal solution

Space?? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$

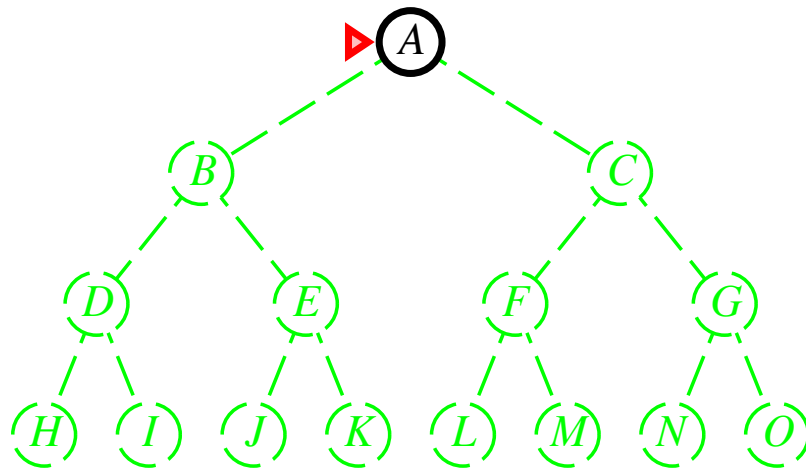
Optimal?? Yes—nodes expanded in increasing order of  $g(n)$

# Depth-first search

Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front



## Depth-limited search

= depth-first search with depth limit  $l$ ,  
i.e., nodes at depth  $l$  have no successors

### Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred?  $\leftarrow$  false
    if GOAL-TEST(problem, STATE[node]) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred?  $\leftarrow$  true
        else if result  $\neq$  failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

## Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
  inputs: problem, a problem
  for depth  $\leftarrow$  0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result
end
```

## Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??: 1 per move

[Note: optimal solution of  $n$ -Puzzle family is NP-hard]