CHAPTER II

Problem-Solving & Search Algorithms

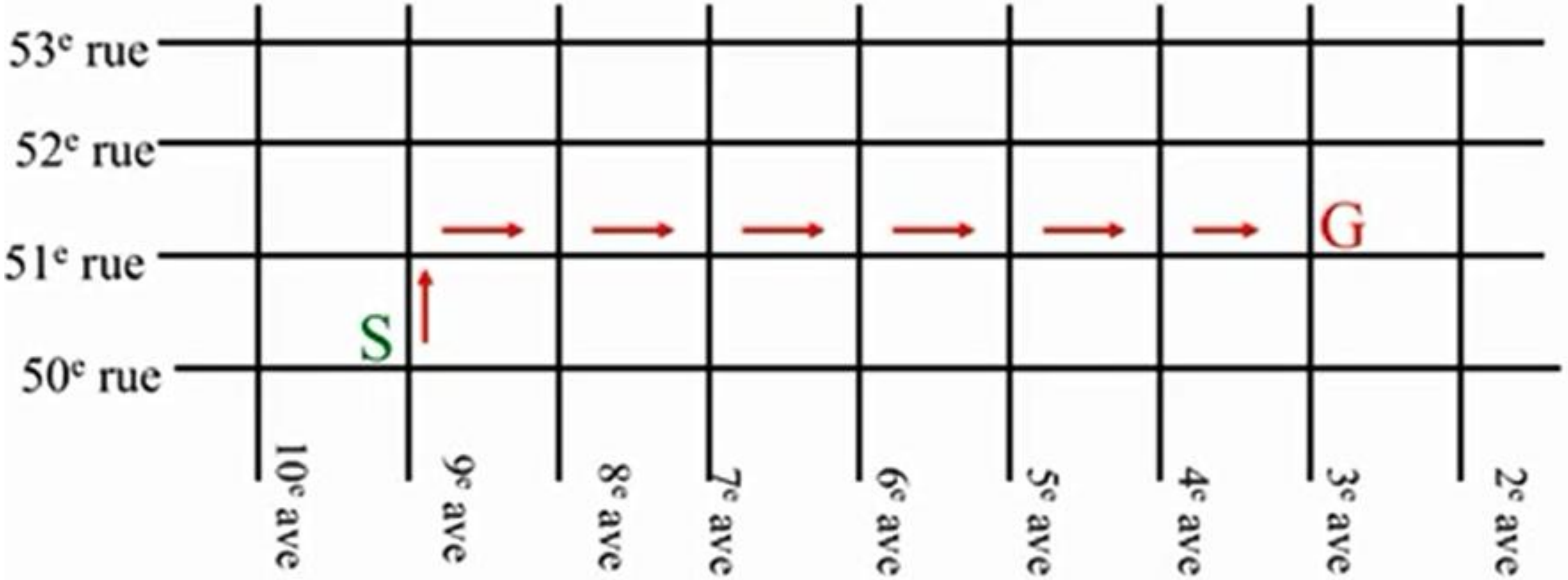
Solving a problem

- **Intuitive steps by a human**
  - Model the current situation
  - List possible solutions
  - Evaluate the value of each solution
  - Select the best option satisfying the goal
- **How to efficiently browse the list of solutions ?**
- **Several problems can be solved by searching in a graph :**
  - Each node represents a state of the environment
  - Each path through a graph represents a sequence of actions
  - The solution: simply look for the path that best satisfies our performance measurement

Problem-Solving

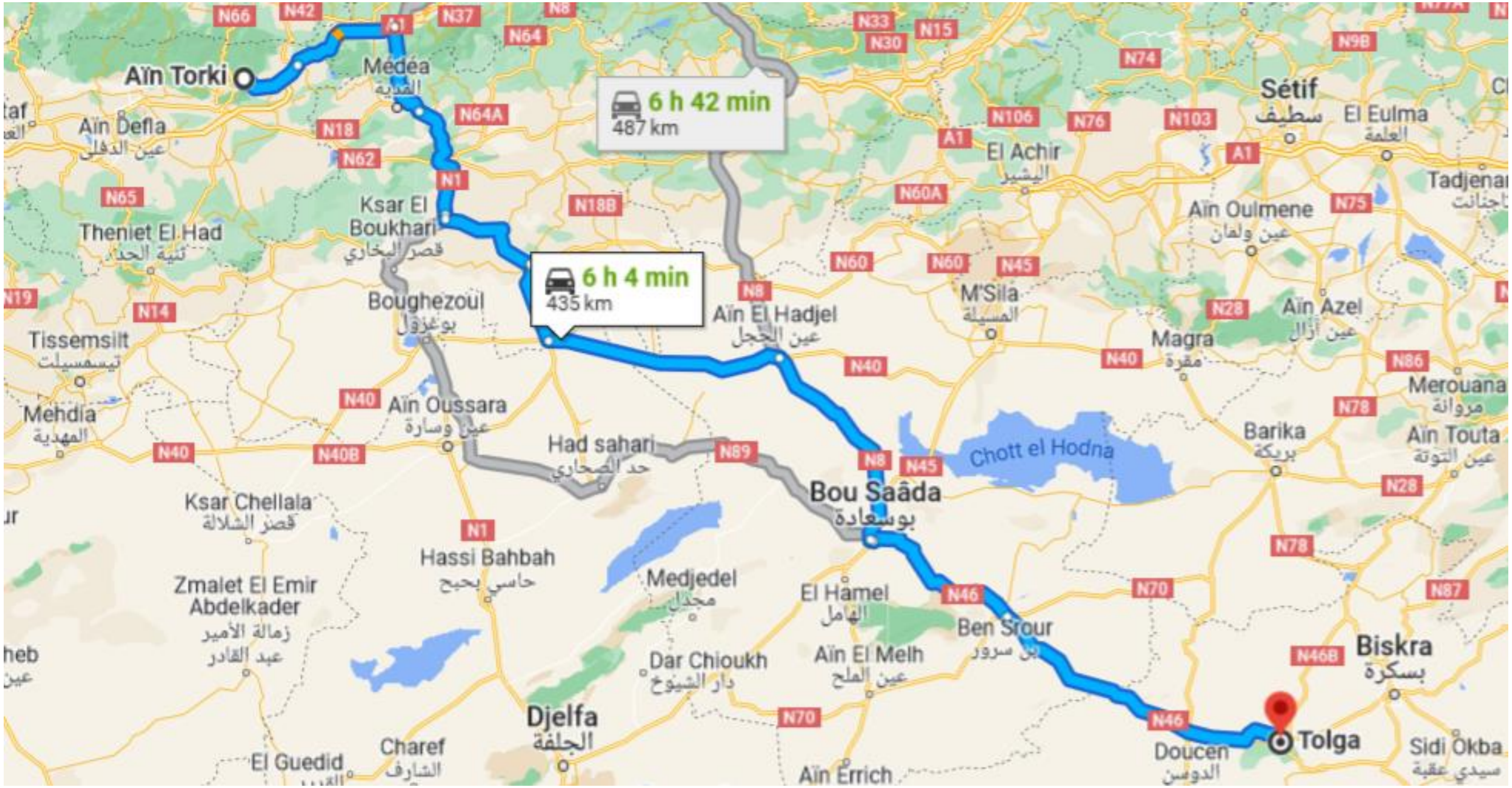
Example: Path-finding in a city

Find the best path between the 9th ave – 50th street to the 3rd ave -51st street



Problem-Solving

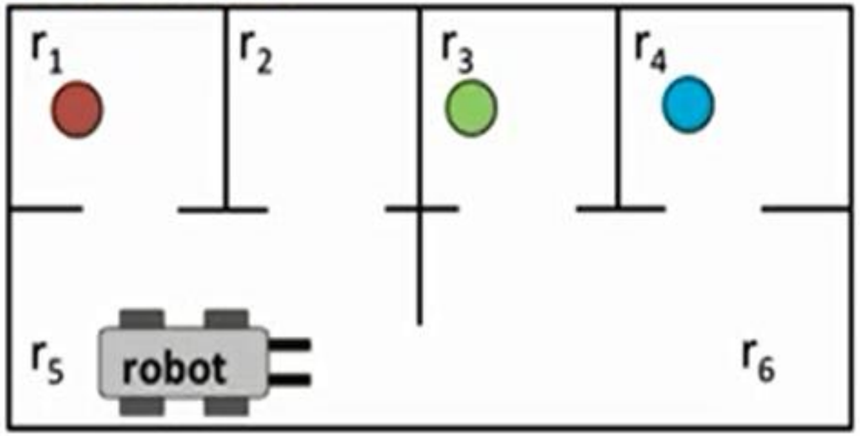
Example: Google Maps



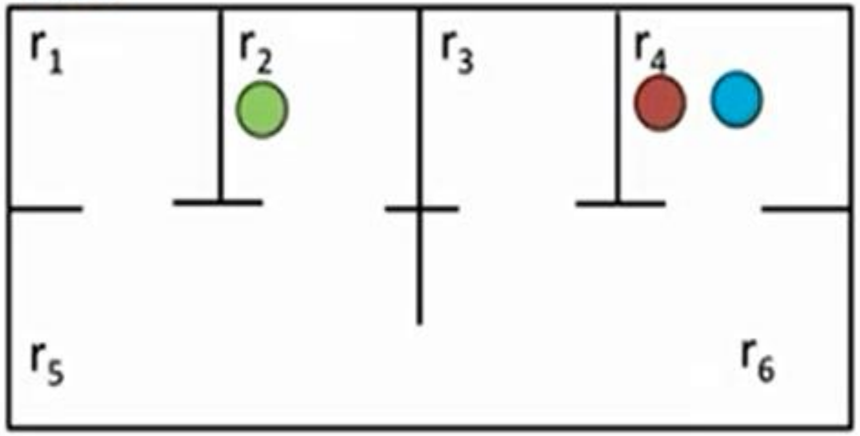
Problem-Solving

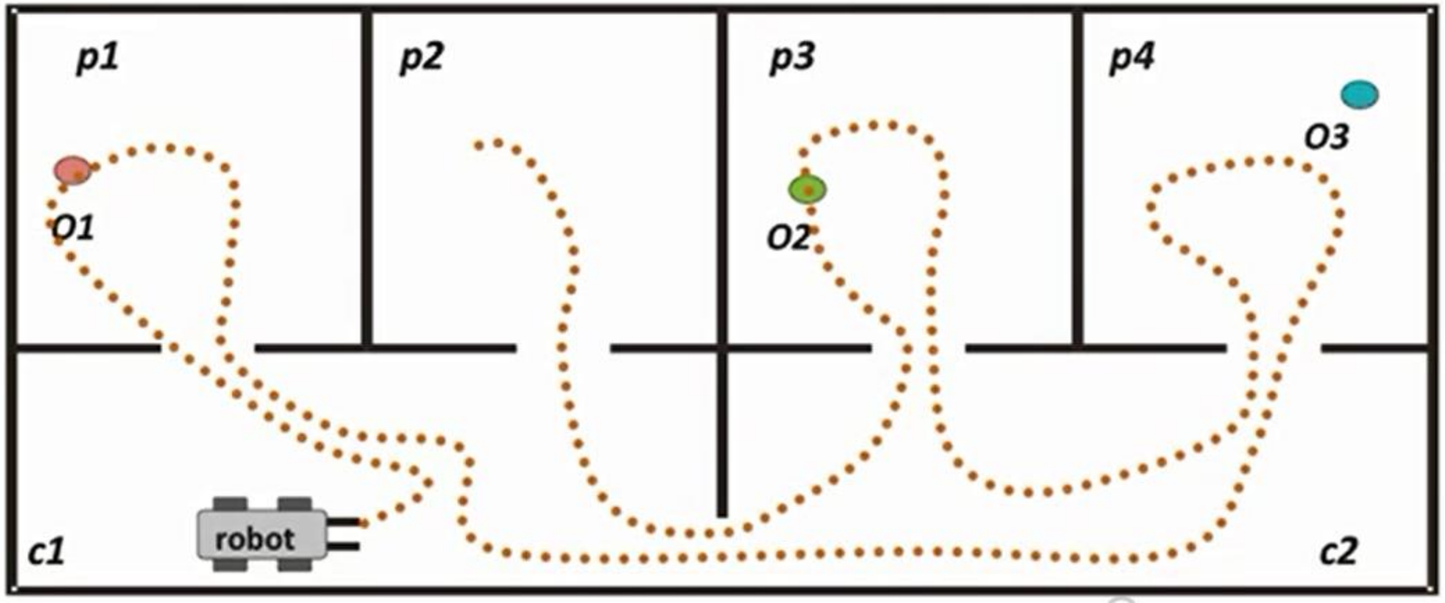
Example: Package delivery

Initial state



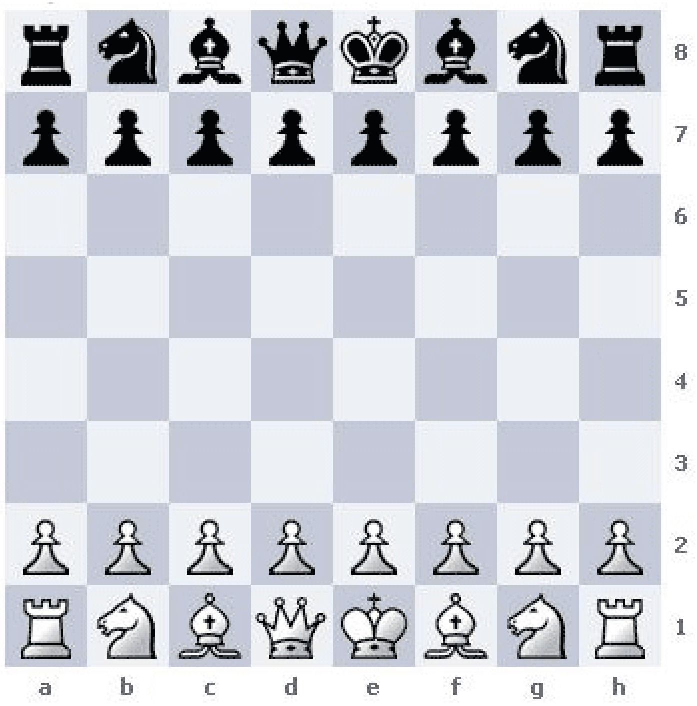
Goal



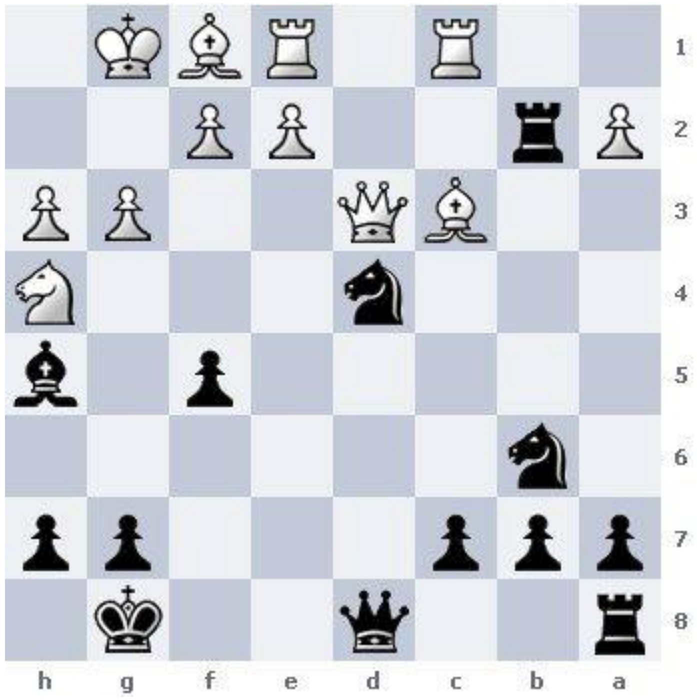


Example: Chess game

Initial state



Goal



Problem-Solving

Example: N-Puzzle

Initial state

Goal

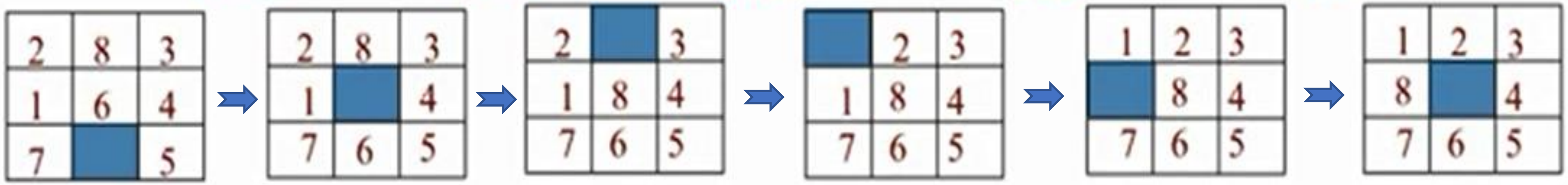
Up

Up

Left

Down

Right



Problem-Solving

Graph search problem

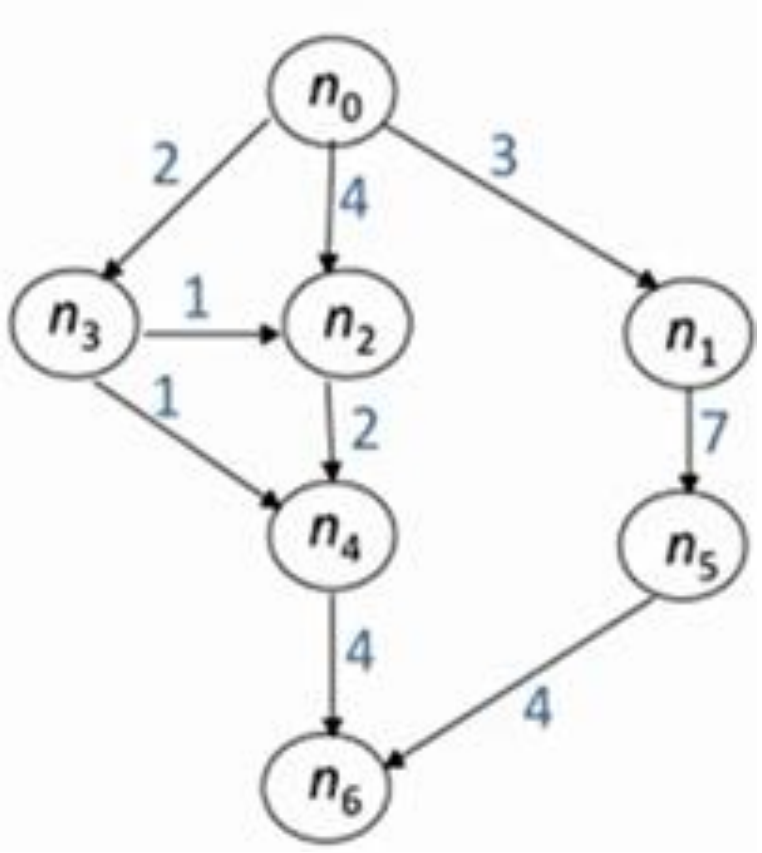
- **Input:**
  - Initial node
  - Goal function **Goal(n)** which returns **True** if the goal is achieved
  - Transition function **Transition(n)** which returns the successor nodes of n
  - Cost function c(n,n’) strictly positive, which returns the cost of going from n to n’

- **Output:**
  - A path in the graph (nodes and edges)
    - The path cost is the sum of all the edges cost in the graph
    - There can be several goal nodes

- **Challenges:**
  - Find a solution path
  - Find an optimal path
  - Quickly find a path (in this case the optimality is not important)

Problem-Solving

A real world example: Find a path between two cities



- Cities: **Nodes**
- Paths between two cities: **Edges**
- Starting city: Initial node $\mathbf{n_0}$
- Roads between cities: $\mathbf{Transition(n_0) = (n_3, n_2, n_1)}$
- Distance between cities: $\mathbf{c(n_0,n_2) = 4}$
- Destination city: **Goal(n) = True** if $n = n_6$ (n_6 is the destination city)

Search Algorithms

Any search problem is characterized by a **starting situation** and a **goal** to achieve and a **search space**:

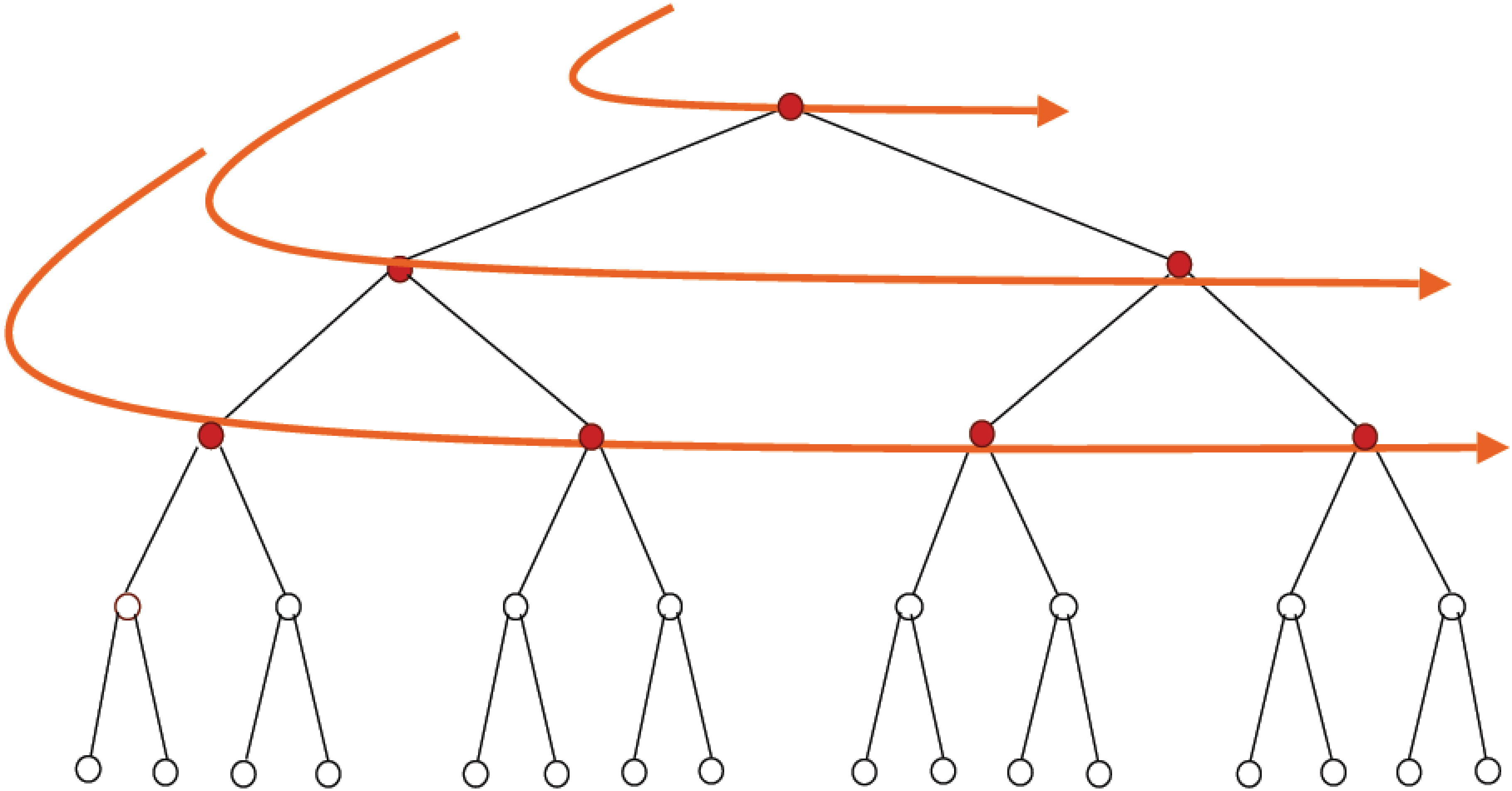
- The search space is composed of the set of all possible states.
- To determine possible operations to move from one state to another.
- To determine a search strategy.

There are different strategies:

- **Uninformed (Blind):**
  - Breadth-first search
  - Depth-first search (and its variations)
  - Uniform cost
- **Informed:**
  - Best-first search,
    - Greedy best-first search,
    - A* algorithm

Breadth-First Search (BFS)

- For a given node, explore the sibling nodes before exploring their children.



Breadth-First Search

Breadth-First Algorithm:

1- Put the **initial state** node in a **FIFO** queue: **OPEN**

2- If **n** corresponds to the **final state** then **Success**

3- If **OPEN** is **empty** then **Failure**

4- Remove **n** from **OPEN**

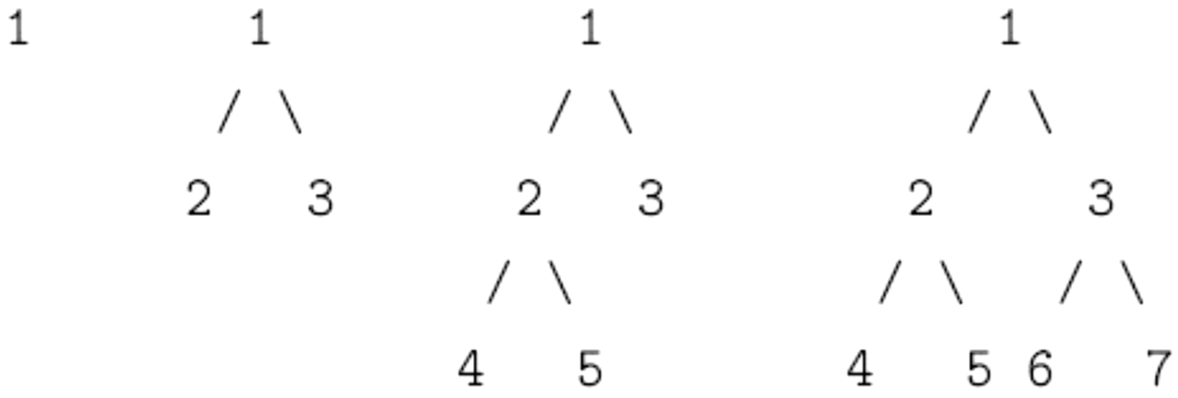
5- If **n** has **no successors** then go to **3,** otherwise:

- Develop the successors of **n**
- Insert them into **OPEN**
- Establish chaining
- Insert **n** into **Closed** (a queue containing the nodes already explored)

6- If among the **successors** there are **final states** then **Success,** otherwise go to **3**

Breadth-First Search

- **Tree traversal**



1.Put 1 in the list. We get [1].

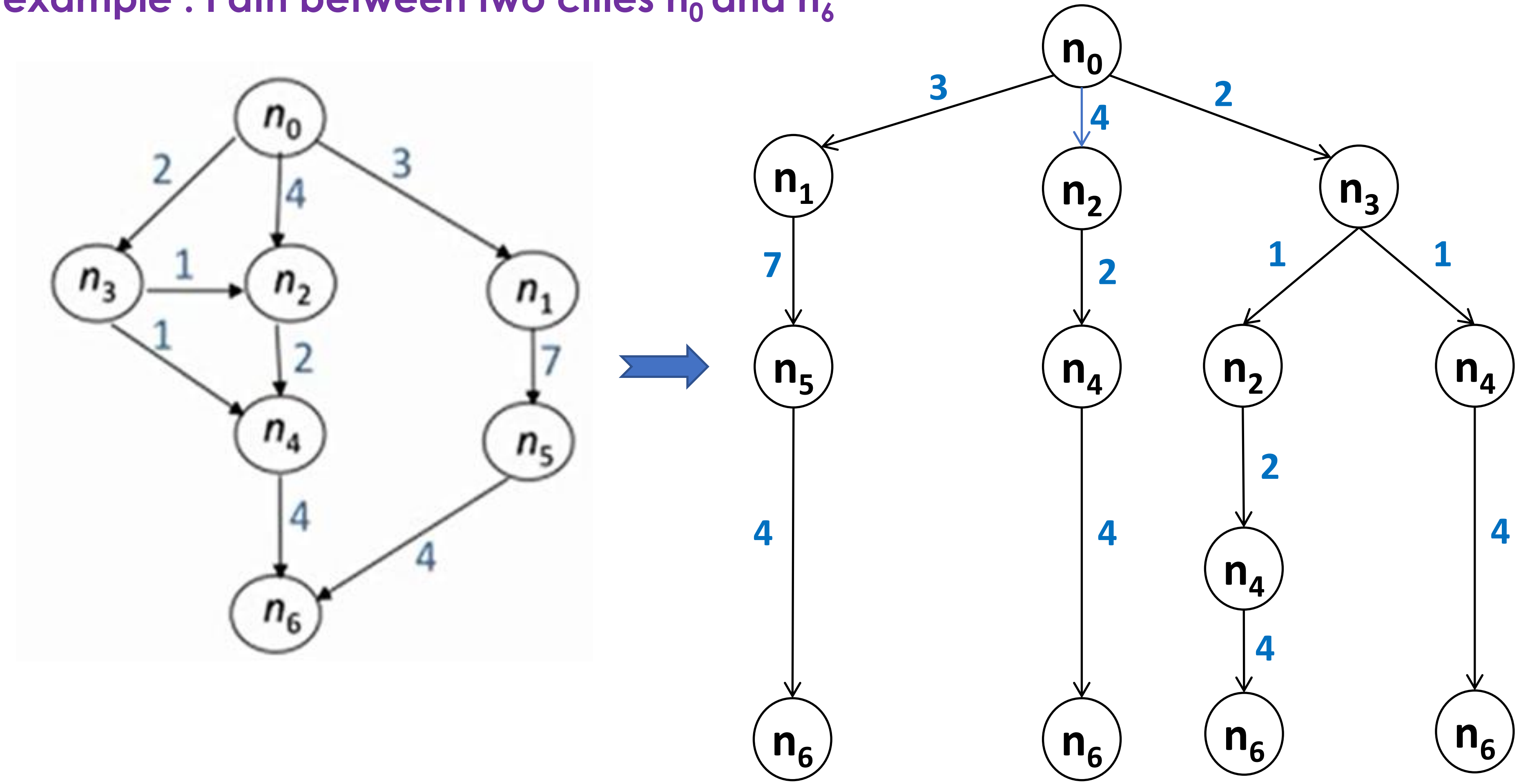
2.Remove the first element of the list (the 1) and add its successors 2, 3. We get [2,3].

3.Remove the first element of the list (the 2) and add its successors 4, 5. We get [3,4,5].

4.Remove the first element of the list (the 3) and add its successors 6, 7. We get [4,5,6,7].

Breadth-First Search

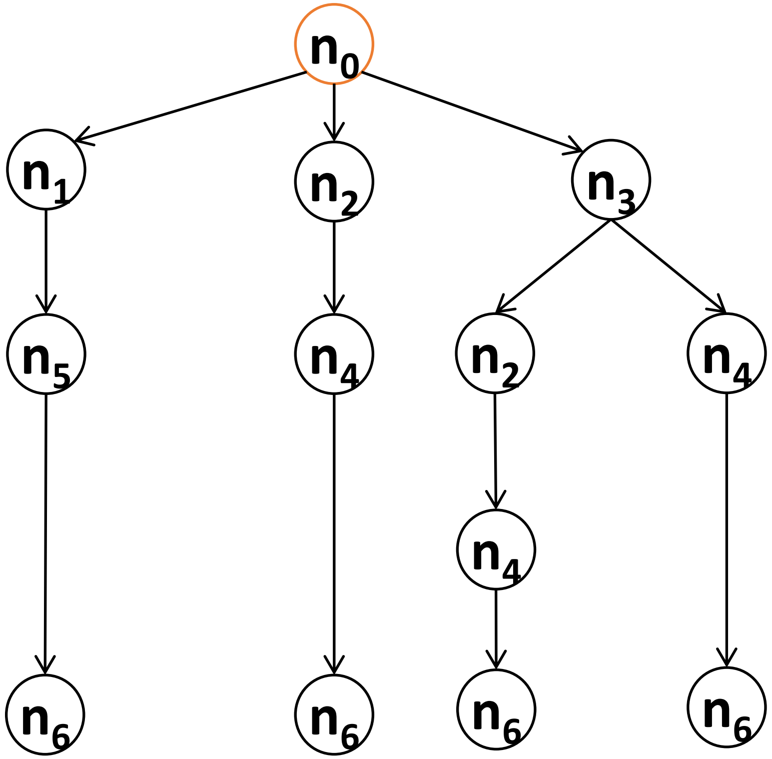
Illustrative example : Path between two cities n_0 and n_6



Breadth-First Search

- **Illustration:**

1. Put **n_0** in the Open list. We get **[n_0]**.

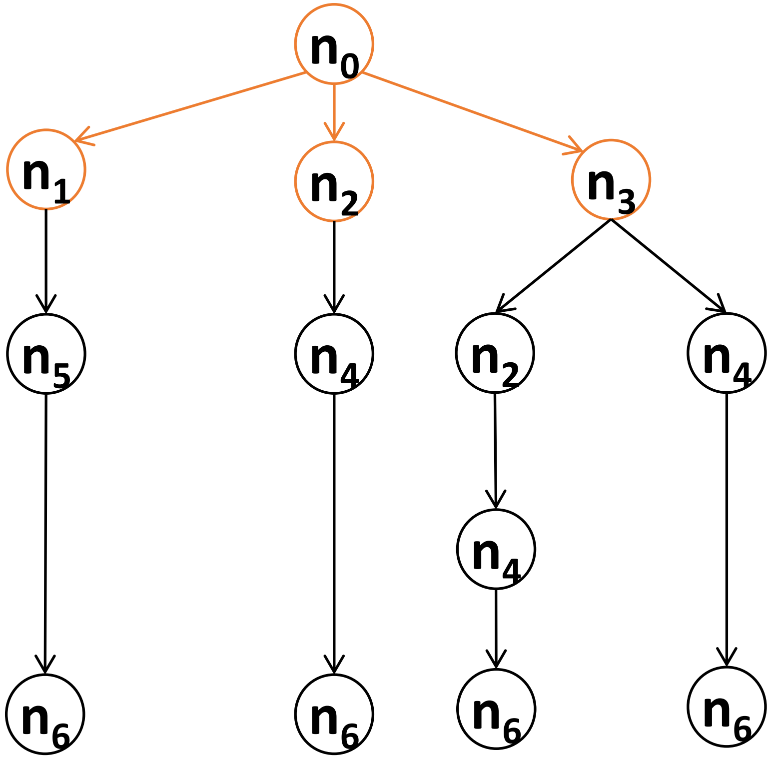


Breadth-First Search

- **Illustration:**

1. Put $\mathbf{n_0}$ in the Open list. We get $\mathbf{[n_0]}$.

2. Remove the first element of the list (the $\mathbf{n_0}$) and add its successors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. We get $[\mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}]$.



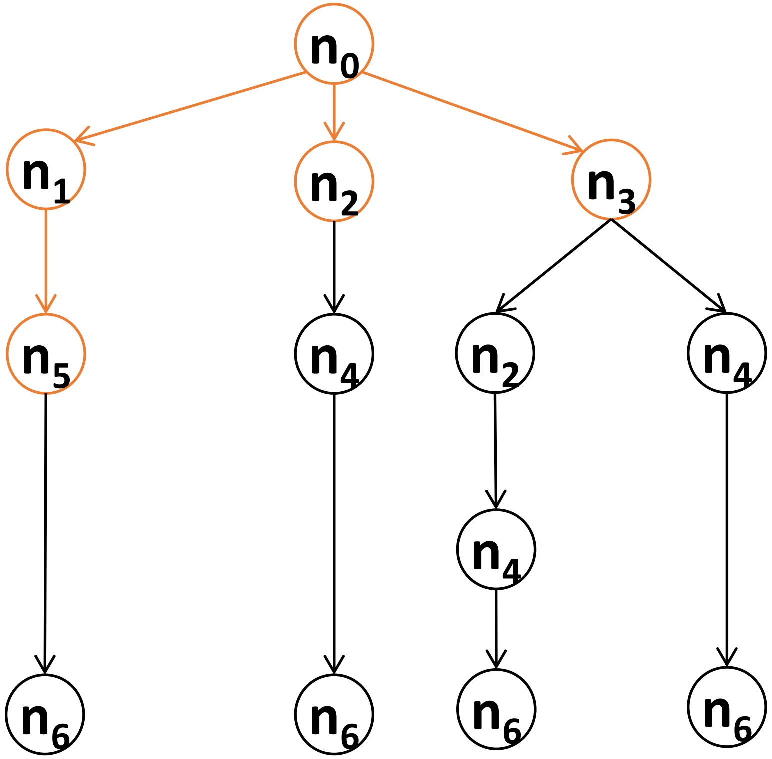
Breadth-First Search

- **Illustration:**

1. Put $\mathbf{n_0}$ in the Open list. We get $[\mathbf{n_0}]$.

2. Remove the first element of the list (the $\mathbf{n_0}$) and add its successors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. We get $[\mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}]$.

3. Remove the first element of the list (the $\mathbf{n_1}$) and add its successors $\mathbf{n_5}$. We get $[\mathbf{n_2}, \mathbf{n_3}, \mathbf{n_5}]$.



Breadth-First Search

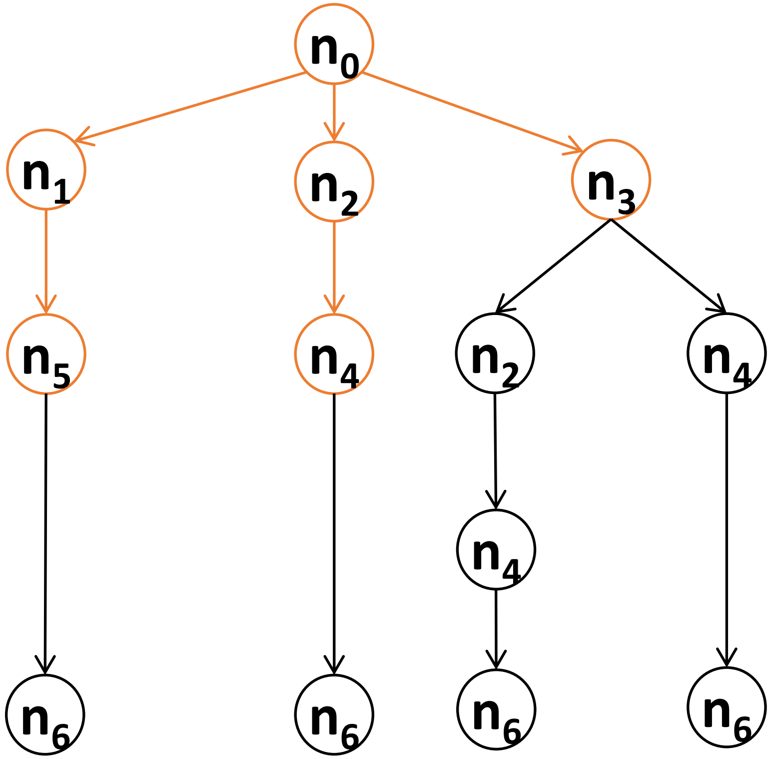
- **Illustration:**

1. Put $\mathbf{n_0}$ in the Open list. We get $\mathbf{[n_0]}$.

2. Remove the first element of the list (the $\mathbf{n_0}$) and add its successors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. We get [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].

3. Remove the first element of the list (the $\mathbf{n_1}$) and add its successors $\mathbf{n_5}$. We get [$\mathbf{n_2}$, $\mathbf{n_3}$, $\mathbf{n_5}$].

4. Remove the first element of the list (the $\mathbf{n_2}$) and add its successors $\mathbf{n_4}$. We get [$\mathbf{n_3}$, $\mathbf{n_5}$, $\mathbf{n_4}$].



Breadth-First Search

- **Illustration:**

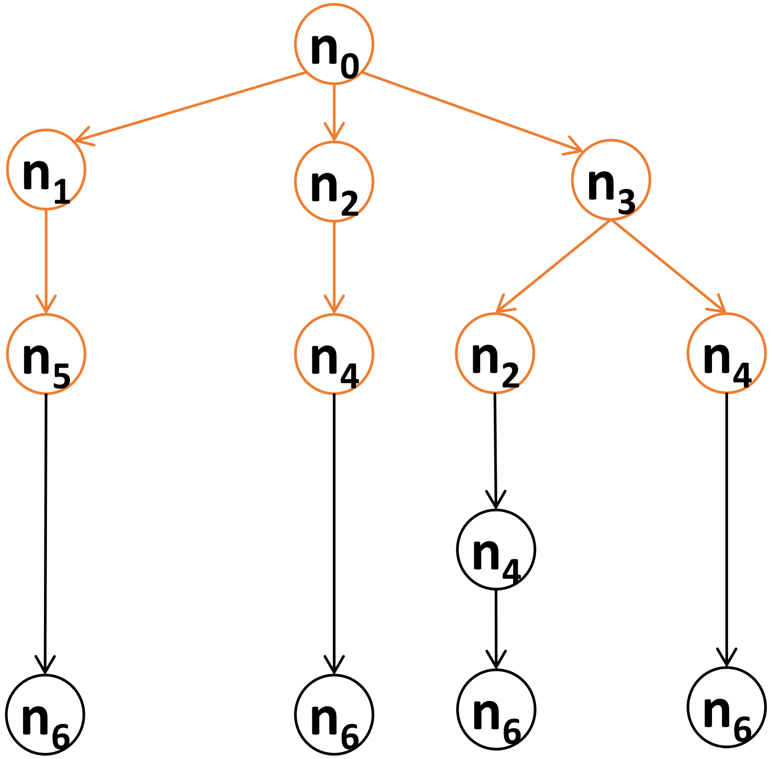
1. Put $\mathbf{n_0}$ in the Open list. We get $\mathbf{[n_0]}$.

2. Remove the first element of the list (the $\mathbf{n_0}$) and add its successors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. We get $[\mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}]$.

3. Remove the first element of the list (the $\mathbf{n_1}$) and add its successors $\mathbf{n_5}$. We get $[\mathbf{n_2}, \mathbf{n_3}, \mathbf{n_5}]$.

4. Remove the first element of the list (the $\mathbf{n_2}$) and add its successors $\mathbf{n_4}$. We get $[\mathbf{n_3}, \mathbf{n_5}, \mathbf{n_4}]$.

5. Remove the first element of the list (the $\mathbf{n_3}$) and add its successors $\mathbf{n_2}$, $\mathbf{n_4}$. We get $[\mathbf{n_5}, \mathbf{n_4}, \mathbf{n_{2(n3)}}, \mathbf{n_{4(n3)}}]$.



Breadth-First Search

- **Illustration:**

1. Put $\mathbf{n_0}$ in the Open list. We get $[\mathbf{n_0}]$.

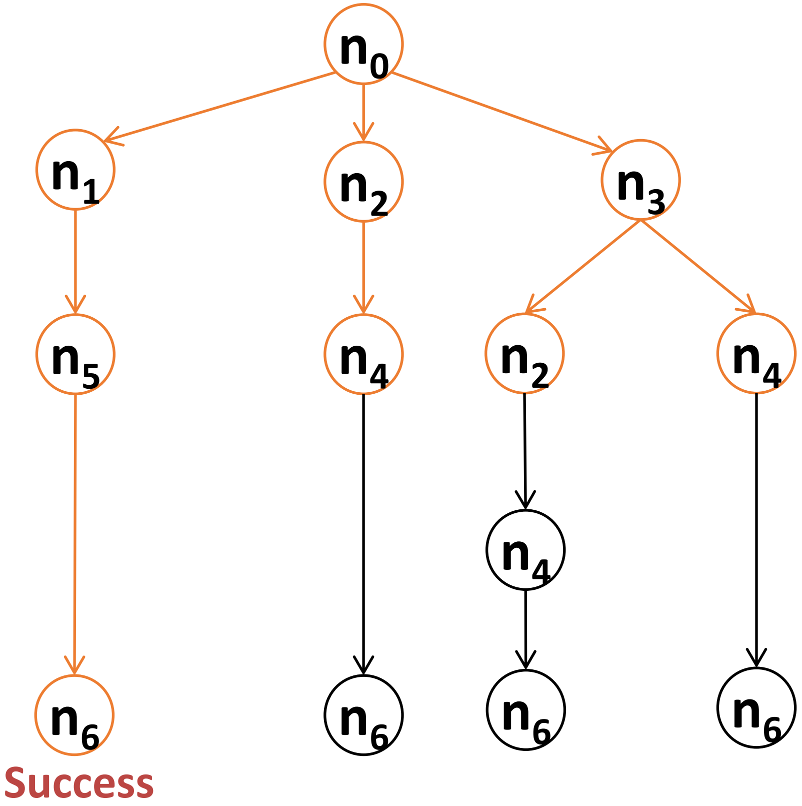
2. Remove the first element of the list (the $\mathbf{n_0}$) and add its successors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. We get $[\mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}]$.

3. Remove the first element of the list (the $\mathbf{n_1}$) and add its successors $\mathbf{n_5}$. We get $[\mathbf{n_2}, \mathbf{n_3}, \mathbf{n_5}]$.

4. Remove the first element of the list (the $\mathbf{n_2}$) and add its successors $\mathbf{n_4}$. We get $[\mathbf{n_3}, \mathbf{n_5}, \mathbf{n_4}]$.

5. Remove the first element of the list (the $\mathbf{n_3}$) and add its successors $\mathbf{n_2}$, $\mathbf{n_4}$. We get $[\mathbf{n_5}, \mathbf{n_4}, \mathbf{n_{2(n3)}}, \mathbf{n_{4(n3)}}]$.

6. Remove the first element of the list (the $\mathbf{n_5}$) and add its successors $\mathbf{n_6}$. We get $[\mathbf{n_4}, \mathbf{n_{2(n3)}}, \mathbf{n_{4(n3)}}, \mathbf{n_6}]$.

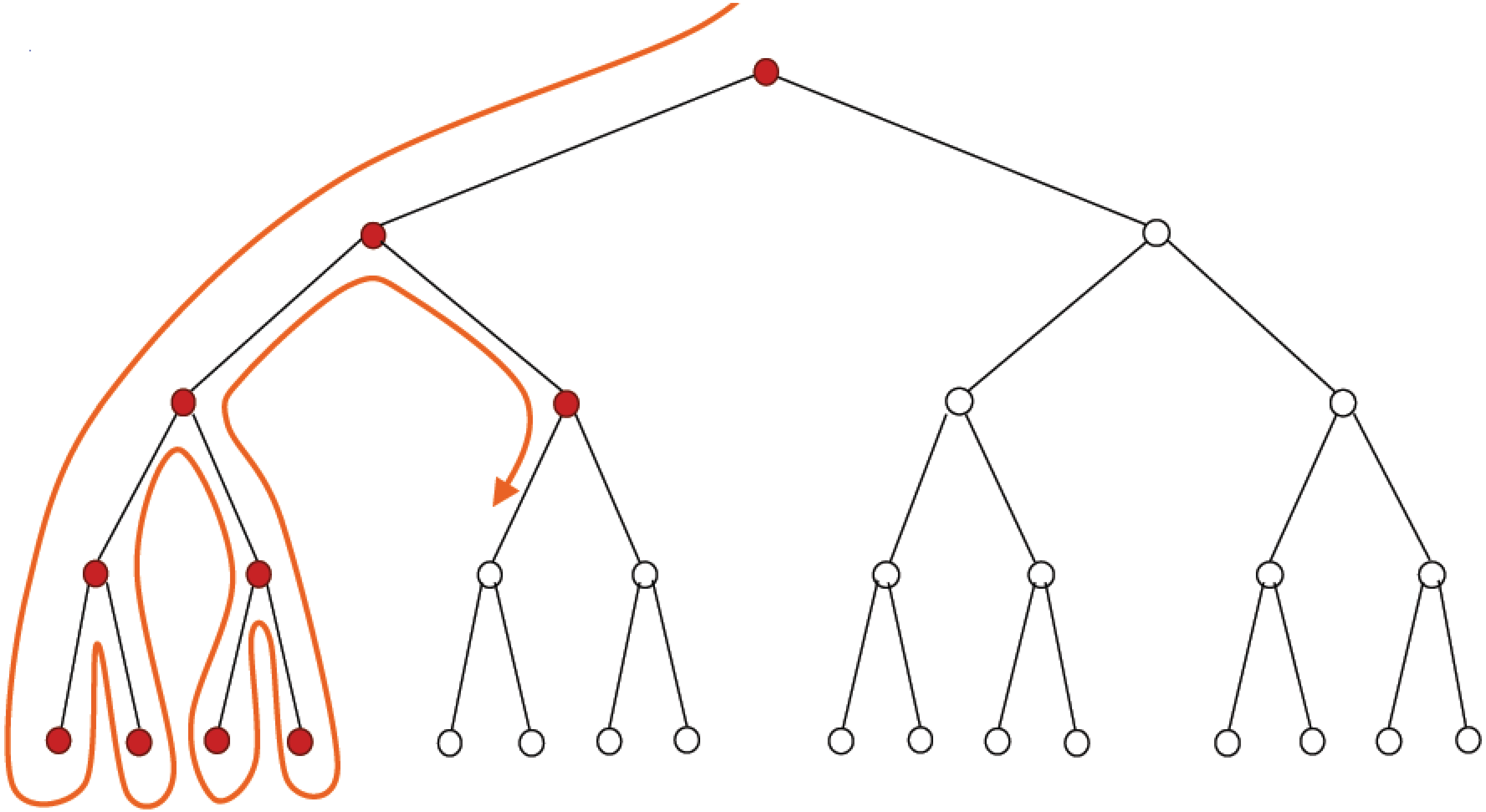


$\mathbf{n_6}$ Appears in OPEN

then Stop

Depth-First Search (DFS)

- For a given node, explore the first child node before exploring the sibling nodes.



Depth-First Search

Depth-First Algorithm:

1- Put the **initial state** node in a **LIFO** stack: **OPEN**

2- If the stack is **empty** then **Failure**

3- Pop **n**

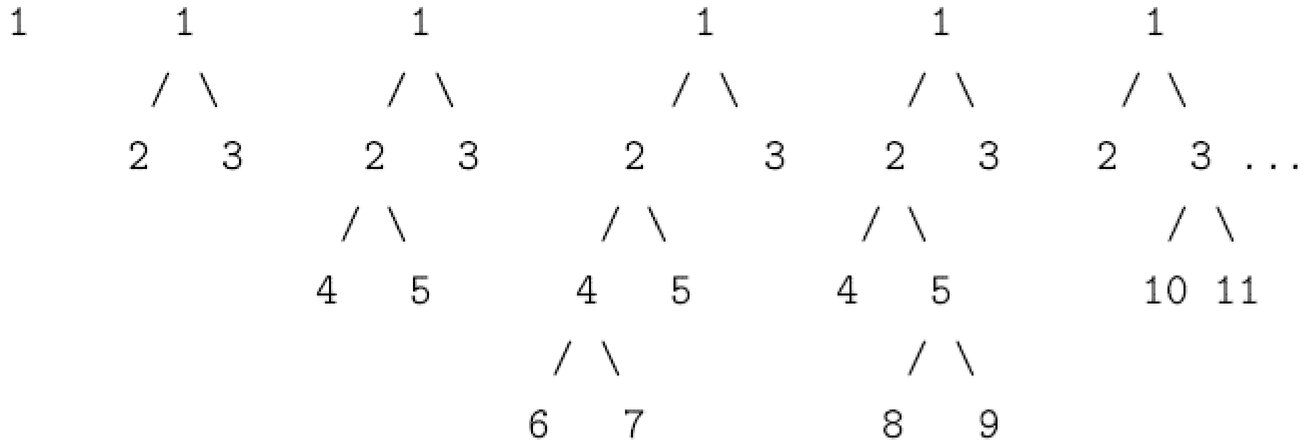
4- Develop the **successors** of **n**:

- If **successors** exist then
  - Push the successors
  - Establish the successor chaining
  - Put **n** in **Closed**

6- If among the **successors** there are **final states** then **Success,** otherwise go to **2**

Depth-First Search

- **Tree traversal**



We are using a stack data structure:

1.Put 1 into the stack. We get [1].

2.Remove the first element from the stack (1) and add its successors 2, 3. We get [2,3].

3.Remove the first element from the stack (2) and add its successors 4, 5. We get [4,5,3].

4.Remove the first element from the stack (4) and add its successors 6, 7. We get [6,7,5,3].

5.Remove the first element from the stack (6) which has no successors. We get [7,5,3].

6.Remove the first element from the stack (7) which has no successors. We get [5,3].

7.Remove the first element from the stack (5) and add its successors 8, 9. We get [8,9,3].

8.Remove the first element from the stack (8) which has no successors. We get [9,3].

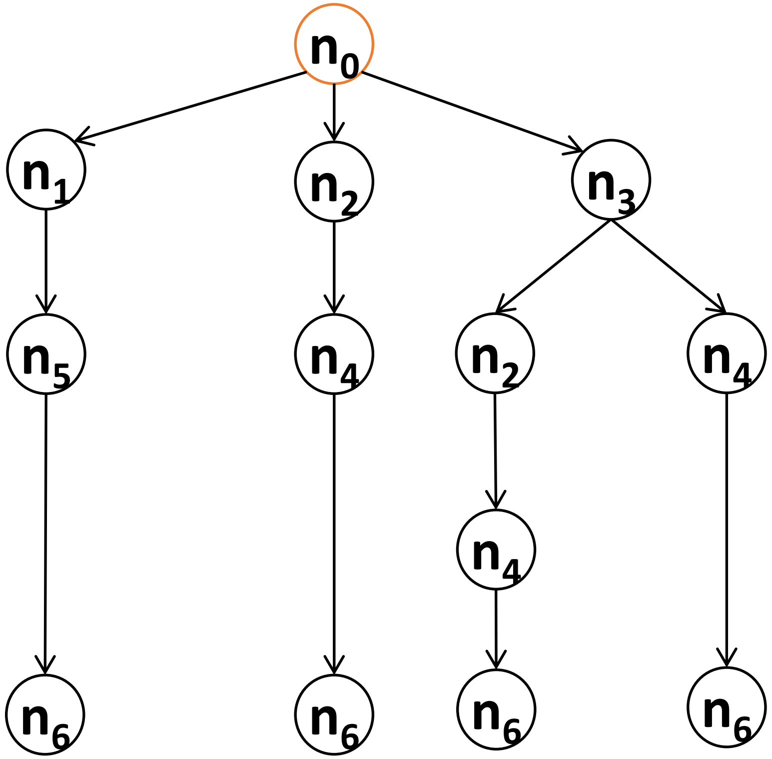
9.Remove the first element from the stack (9) which has no successors. We get [3].

10.Remove the first element from the stack (3) and add its successors 10, 11. We get [10,11]......

Depth-First Search

- **Illustration:**

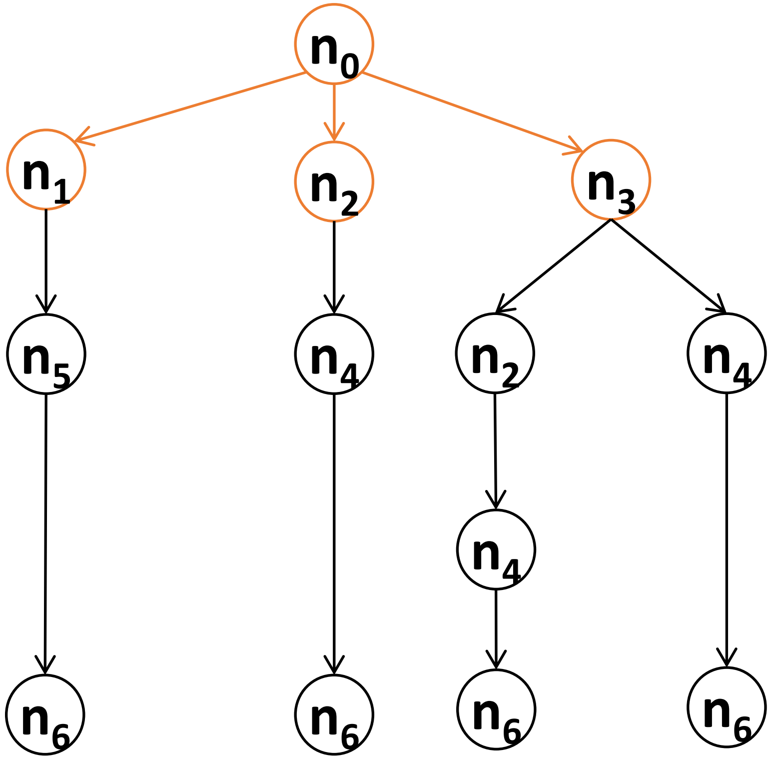
1. Put n_0 in **Open**. We get **[n_0]**.



Depth-First Search

- **Illustration:**

1. Put $\mathbf{n_0}$ in **Open**. We get **[$\mathbf{n_0}$]**.
2. Remove the first element of the stack (the $\mathbf{n_0}$) and add its successors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$. We get [$\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$].



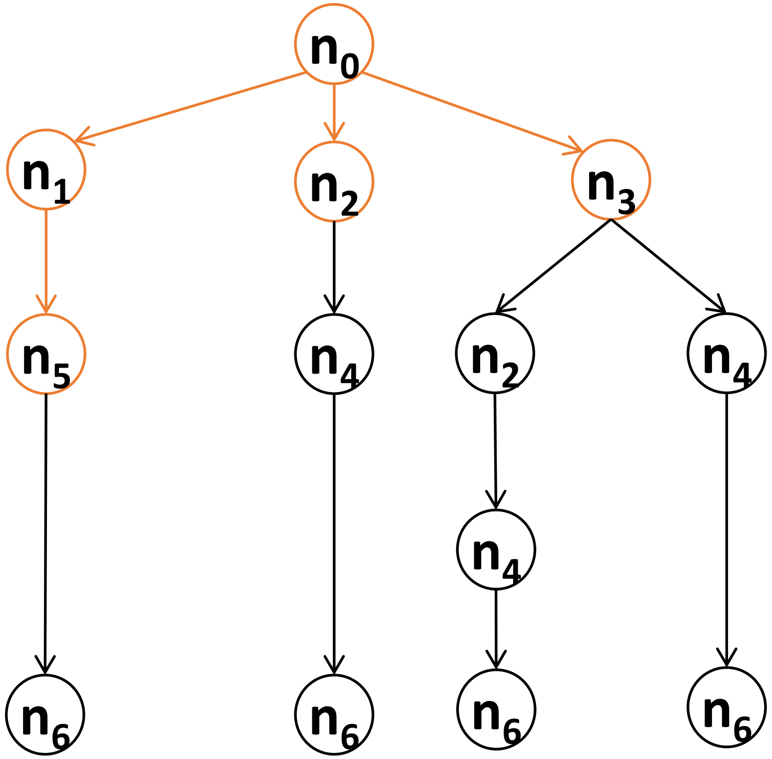
Depth-First Search

- **Illustration:**

1. Put **n_0** in **Open**. We get **[n_0]**.

2. Remove the first element of the stack (the **n_0**) and add its successors **n_1, n_2, n_3**. We get [**n_1, n_2, n_3**].

3. Remove the first element of the stack (the **n_1**) and add its successors **n_5**. We get [**n_5, n_2, n_3**].



Depth-First Search

- **Illustration:**

1. Put $\mathbf{n_0}$ in **Open**. We get $\mathbf{[n_0]}$.

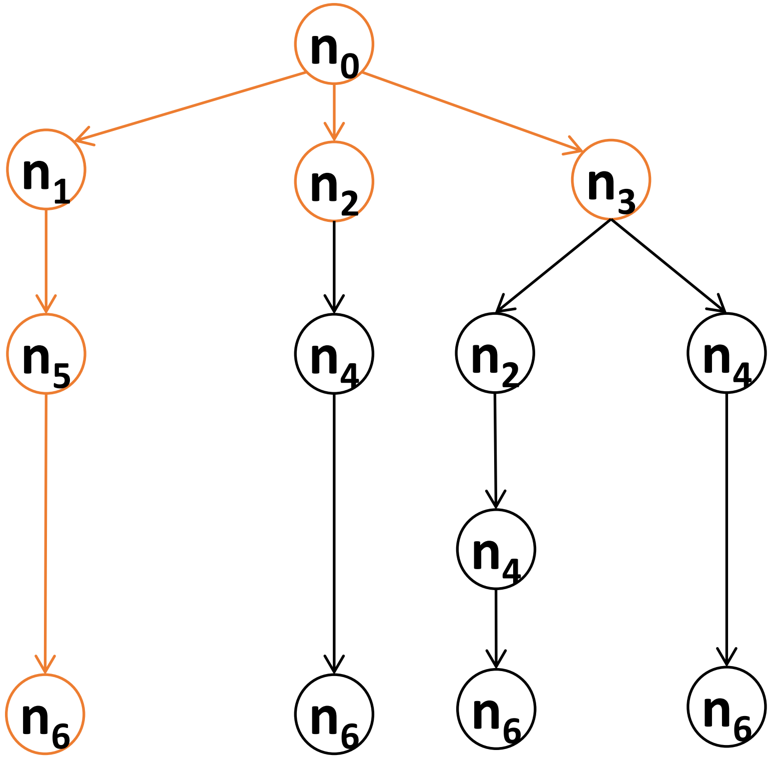
2. Remove the first element of the stack (the $\mathbf{n_0}$) and add its successors $\mathbf{n_1, n_2, n_3}$. We get $[\mathbf{n_1, n_2, n_3}]$.

3. Remove the first element of the stack (the $\mathbf{n_1}$) and add its successors $\mathbf{n_5}$. We get $[\mathbf{n_5, n_2, n_3}]$.

4. Remove the first element of the stack (the $\mathbf{n_5}$) and add its successors $\mathbf{n_6}$. We get $[\mathbf{n_6, n_2, n_3}]$.

$\mathbf{n_6}$ Appears in Open

then Stop



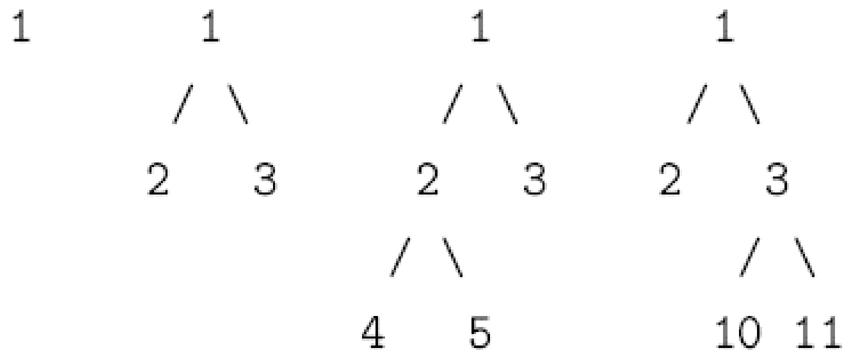
Succès

Limited Depth-First Search (LDFS)

It is a depth-first search where states are explored to a limited depth

Example with **Limit = 2**

We add the successors of a state only if we have not exceeded the limit of 2.



1.Put 1 in the stack with its depth. We get [(1,0)].
2.Remove the first element from the stack (1, 0) and add its successors 2, 3 if we have not exceeded the limit. We get [(2,1),(3,1)].
3.Remove the first element from the stack (2, 1) and add its successors 4, 5 if we have not exceeded the limit. We get [(4,2),(5,2),(3,1)].
4.Remove the first element from the stack (4, 2) and add nothing. We get [(5,2),(3,1)].
5.Remove the first element from the stack (5, 2) and add nothing. We get [(3,1)].
6.Remove the first element from the stack (3, 1) and add its successors 10, 11 if we have not exceeded the limit. We get [(10,2),(11,2)].
7.Remove the first element from the stack (10, 2) and add nothing. We get [(11,2)].
8.Remove the first element from the stack (11, 2) and add nothing. We get [].

Iterative Depth-First Search (IDFS)

It is an iteration of the limited depth-first search

```
For p = 0 to infinite do
{
Limited_DFS(p)
}
```

BFS vs DFS

- ✓ The breadth-first search strategy is interesting because if there exists a path to the goal in the early levels, it finds the shortest path.
- ✓ The depth-first search strategy is also interesting because it returns the best path if the goal state is found in the early branches of the search tree.
- However, these two strategies are considered blind because they do not take into account the path leading to the goal state

Uniform-Cost Search Algorithm

Principle:

- Each arc of the graph is associated with a traversal cost.
- This algorithm provides an optimal cost solution.

  - $C(n_i, n_j)$ is the cost of an arc from n_i to n_j.

  - The cost function of a node **ns**, which is the successor node of **n**, is calculated as follows:

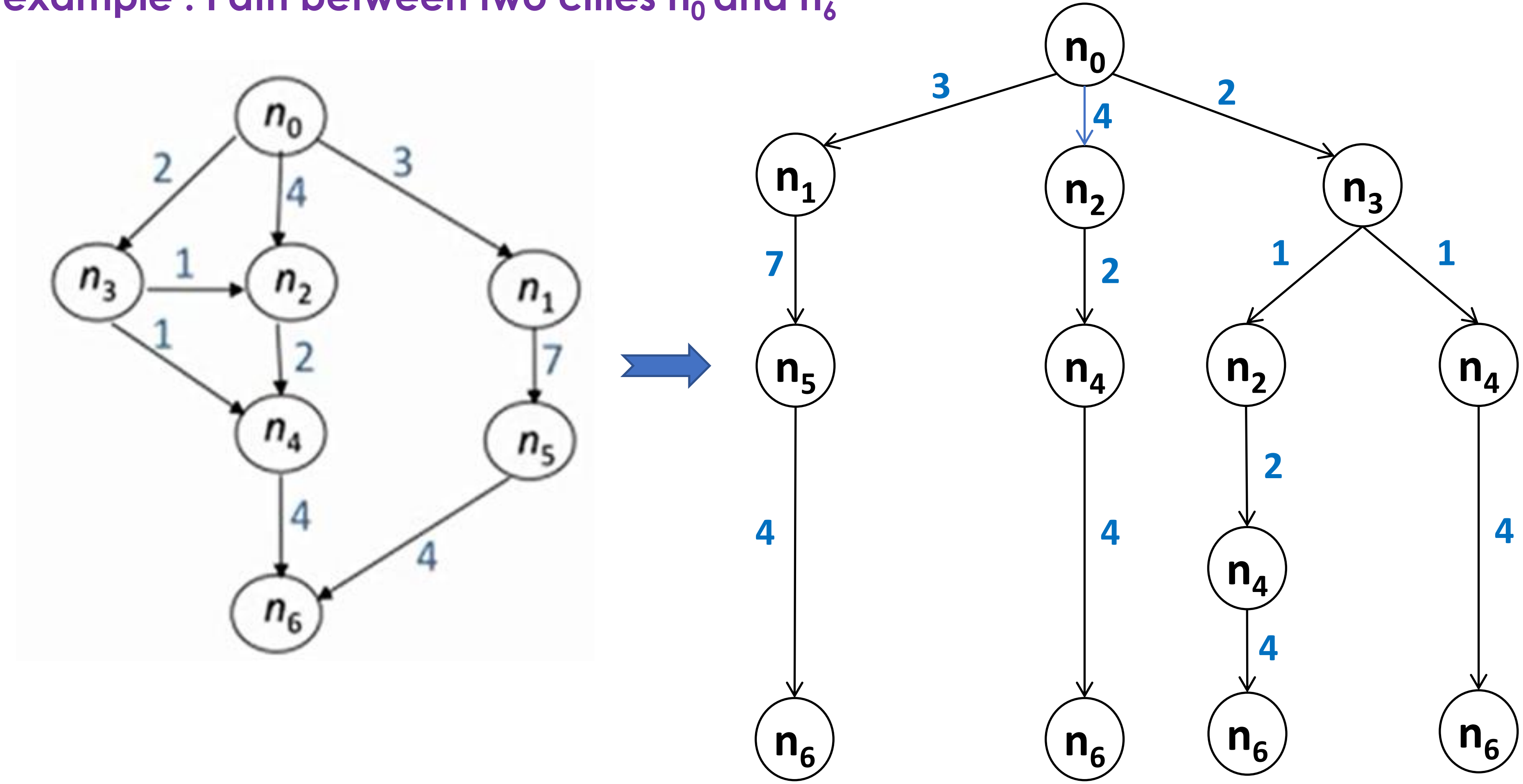
$$g(ns) = g(n) + c(n, ns)$$

Where **g(n)** is the cost up to node **n**

The open nodes of the graph are ordered in ascending order.

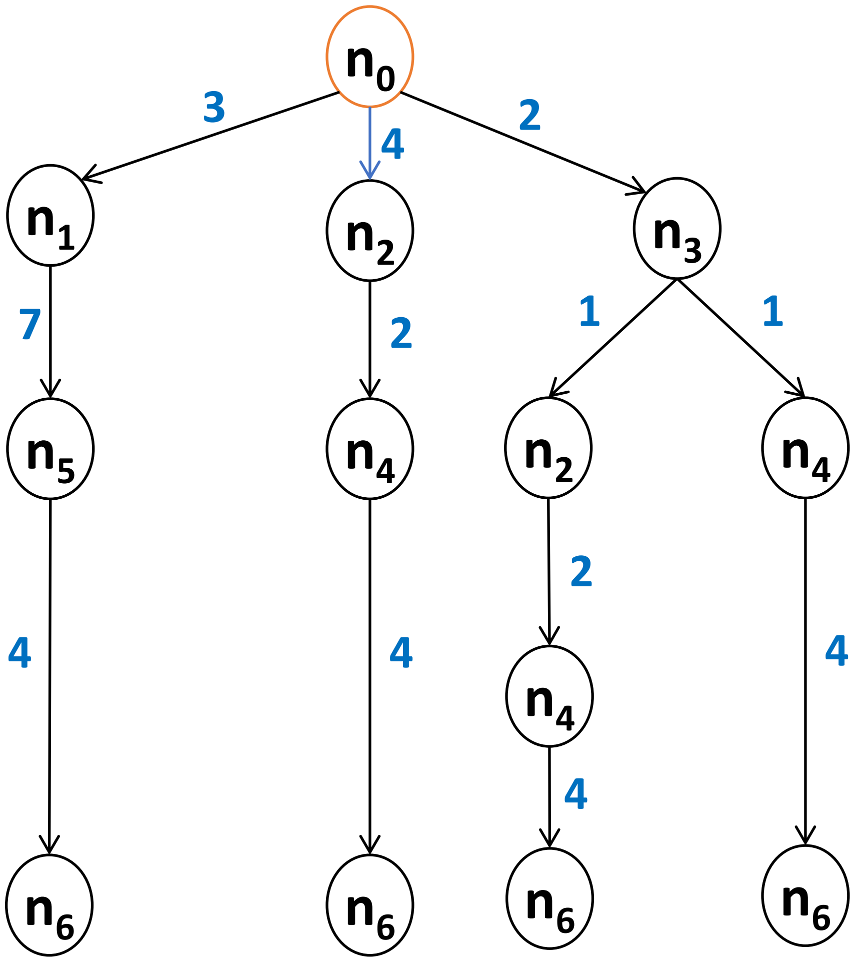
Uniform-Cost Search Algorithm

Illustrative example : Path between two cities n_0 and n_6



Uniform-Cost Search Algorithm

1. Put $\mathbf{n_0}$ in Open with its initial cost. We get **[(n_0,0)]**.



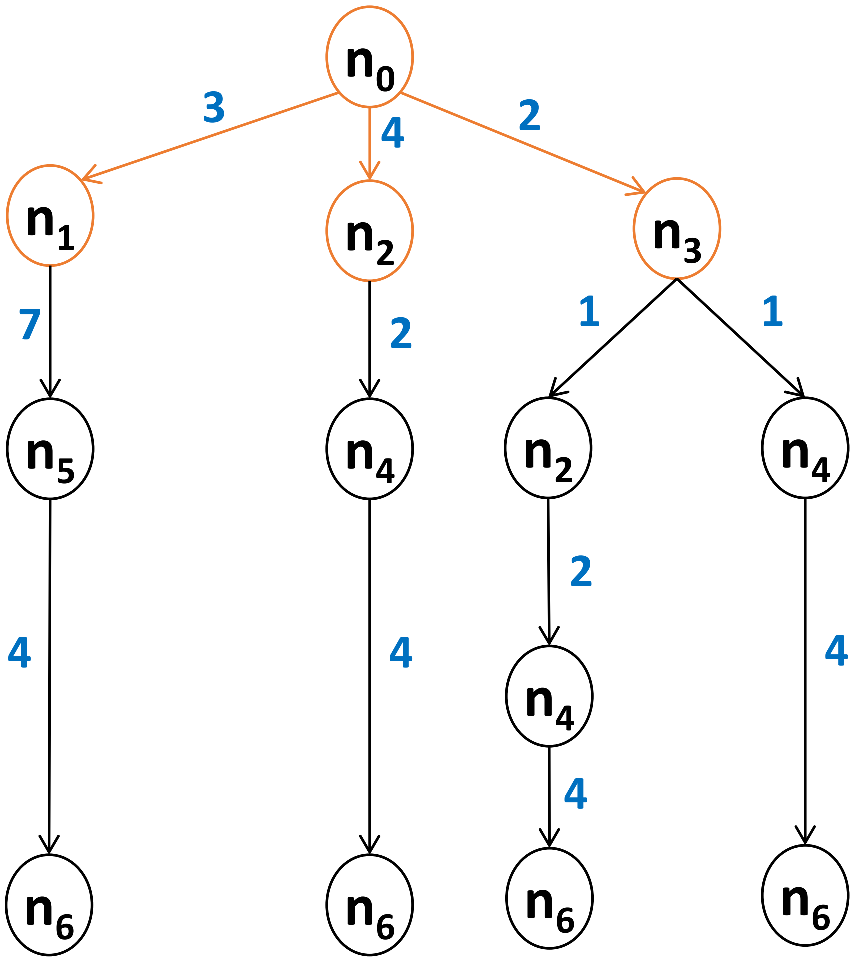
Uniform-Cost Search Algorithm

1. Put $\mathbf{n_0}$ in Open with its initial cost. We get **[(n_0,0)]**.

2. Remove the first element of the list **(n_0,0)** and add its successors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$ to the list of states respecting the ascending order. To achieve this, we will calculate the total cost of each successor : $g(ns) = g(n) + c(n, ns)$

$g(\mathbf{n_1}) = g(\mathbf{n_0}) + c(\mathbf{n_0}, \mathbf{n_1}) = 3$

We get: **[(n_3,2, n_0), (n_1,3, n_0), (n_2,4, n_0)]**



Uniform-Cost Search Algorithm

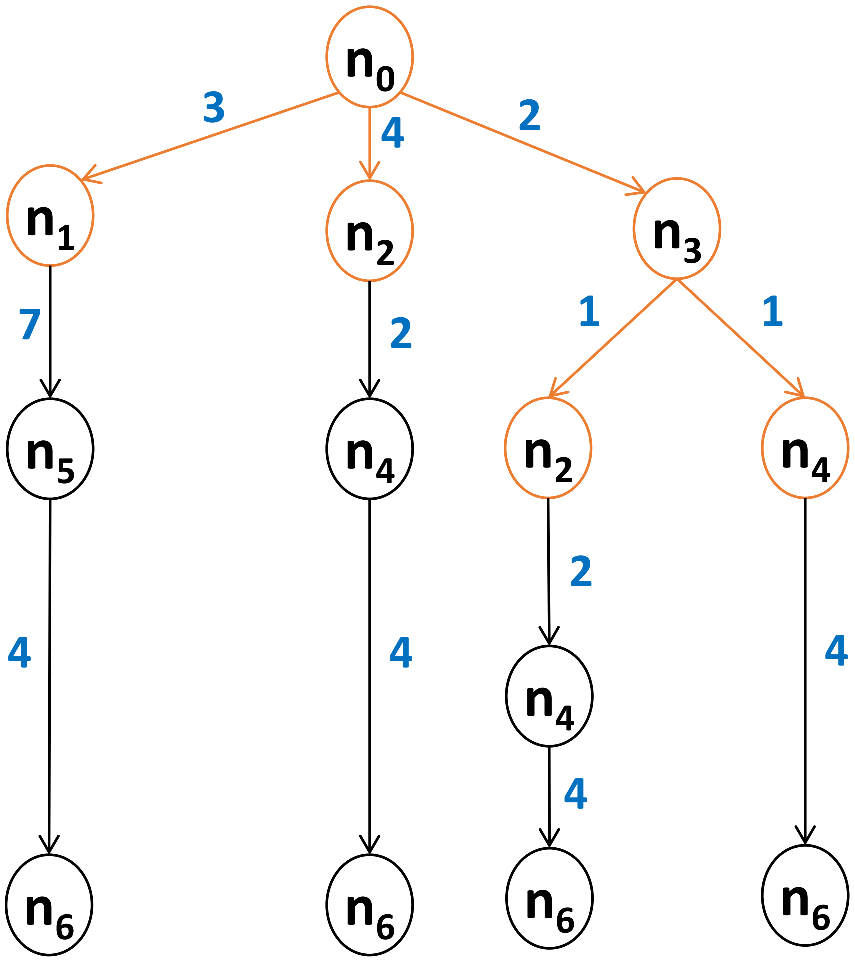
1. Put $\mathbf{n_0}$ in Open with its initial cost. We get **$[(n_0,0)]$**.

2. Remove the first element of the list **$(n_0,0)$** and add its successors $\mathbf{n_1, n_2, n_3}$ to the list of states respecting the ascending order. To achieve this, we will calculate the total cost of each successor : $g(ns) = g(n) + c(n, ns)$

$g(n_1) = g(n_0) + c(n_0, n_1) = 3$

We get: **$[(n_3,2, n_0), (n_1,3, n_0), (n_2,4, n_0)]$**

3. Remove the first element of the list **$(n_3,2,n_0)$** and add its successors $\mathbf{n_2, n_4}$ to the list of states respecting the ascending order. We get: **$[(n_1,3, n_0), (n_2,3, n_3),(n_4,3, n_3), (n_2,4, n_0)]$**



Uniform-Cost Search Algorithm

1. Put $\mathbf{n_0}$ in Open with its initial cost. We get **[(n_0,0)]**.

2. Remove the first element of the list **(n_0,0)** and add its successors $\mathbf{n_1}$, $\mathbf{n_2}$, $\mathbf{n_3}$ to the list of states respecting the ascending order. To achieve this, we will calculate the total cost of each successor : *g(ns) = g(n) + c(n, ns)*

g($\mathbf{n_1}$) = *g*($\mathbf{n_0}$) + *c*($\mathbf{n_0}$, $\mathbf{n_1}$) = 3

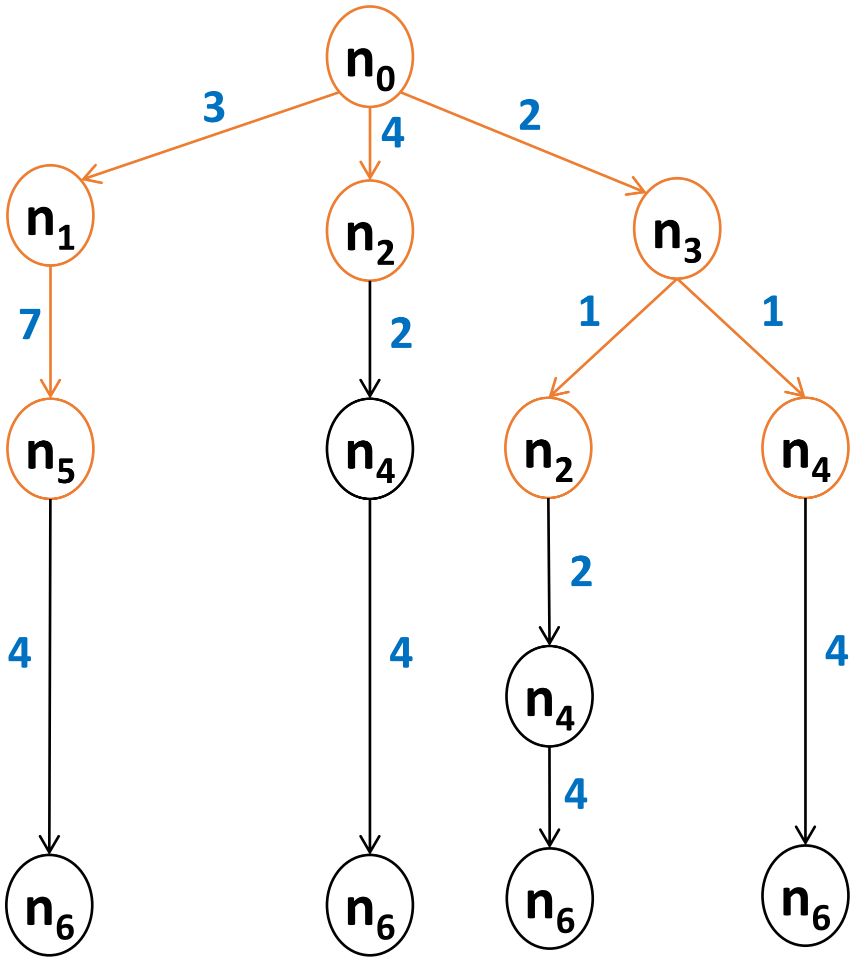
We get: **[(n_3,2, n_0), (n_1,3, n_0), (n_2,4, n_0)]**

3. Remove the first element of the list **(n_3,2,n_0)** and add its successors $\mathbf{n_2}$, $\mathbf{n_4}$ to the list of states respecting the ascending order.

We get: **[(n_1,3, n_0), (n_2,3, n_3),(n_4,3, n_3), (n_2,4, n_0)]**

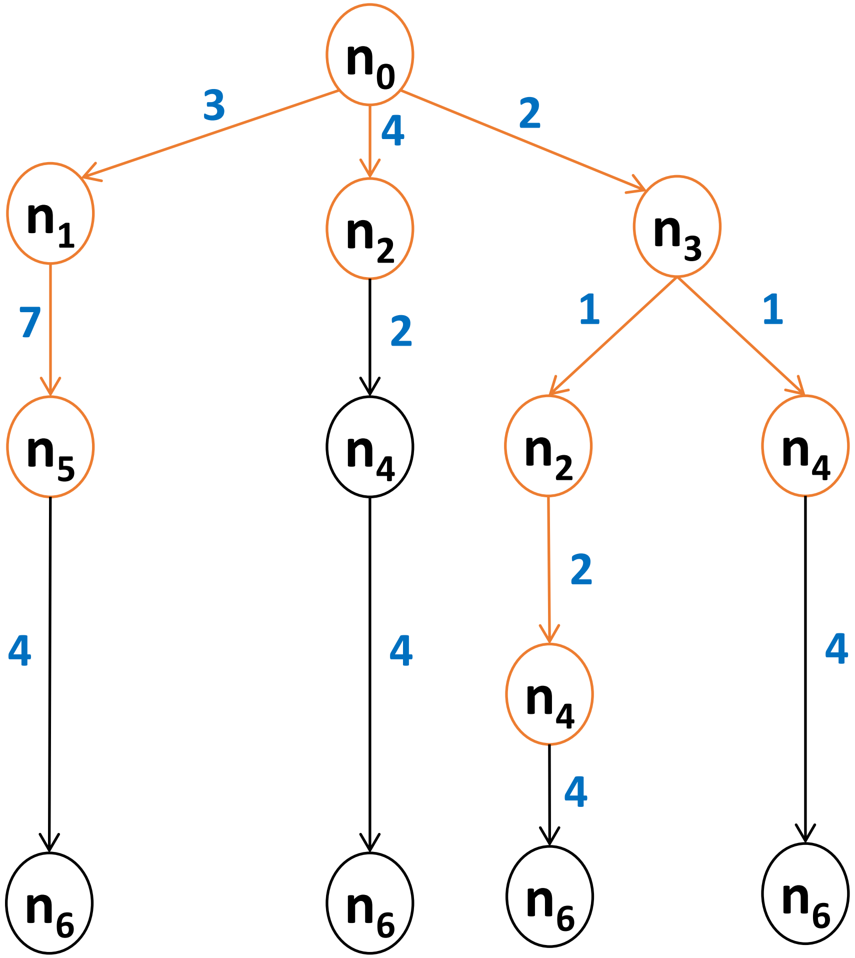
4. Remove the first element of the list **(n_1,3,n_0)** and add its successors $\mathbf{n_5}$ to the list of states respecting the ascending order.

We get: **[(n_2,3, n_3),(n_4,3, n_3), (n_2,4, n_0),(n_5,10, n_1)]**



Uniform-Cost Search Algorithm

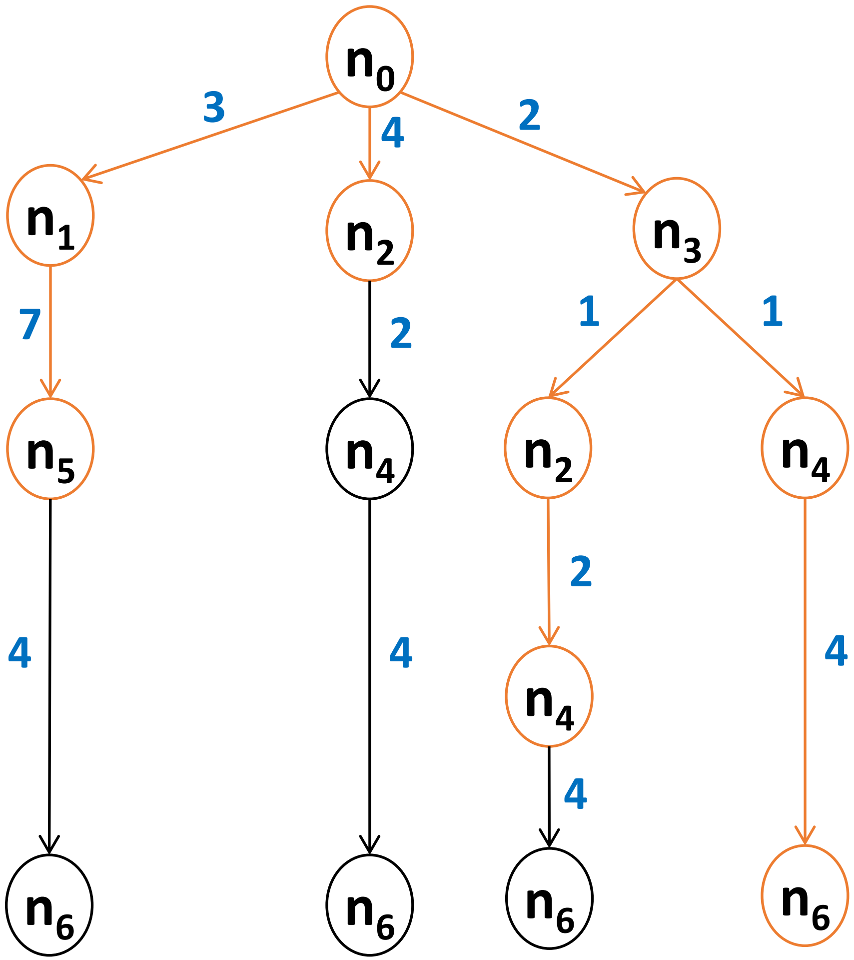
5. Remove the first element of the list **(n_2,3,n_3)** and add its successors **n_4** to the list of states respecting the ascending order. We get: **[(n_4,3,n_3), (n_2,4, n_0), (n_4,5,n_2),(n_5,10,n_1)]**



Uniform-Cost Search Algorithm

5. Remove the first element of the list **(n_2,3,n_3)** and add its successors **n_4** to the list of states respecting the ascending order. We get: **[(n_4,3,n_3), (n_2,4, n_0), (n_4,5,n_2),(n_5,10,n_1)]**

6. Remove the first element of the list **(n_4,3,n_3)** and add its successors **n_6** to the list of states respecting the ascending order. We get: **[(n_2,4, n_0), (n_4,5,n_2),(n_6,7,n_4),(n_5,10,n_1)]**

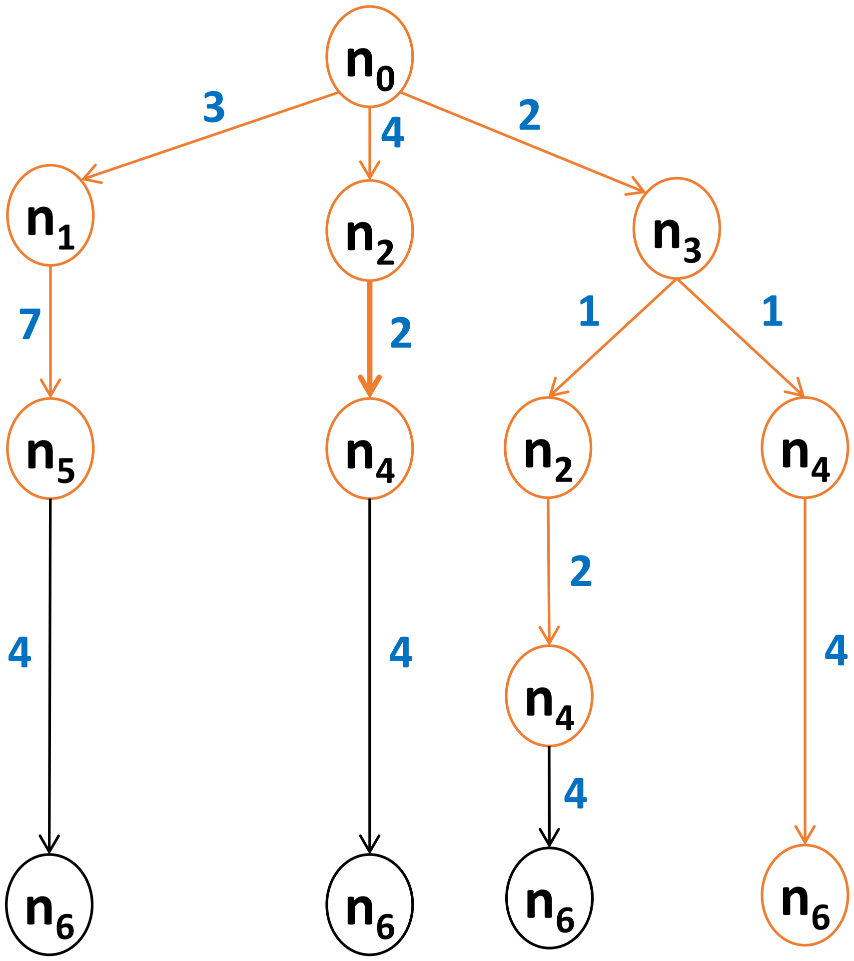


Uniform-Cost Search Algorithm

5. Remove the first element of the list **(n_2,3,n_3)** and add its successors **n_4** to the list of states respecting the ascending order. We get: **[(n_4,3,n_3), (n_2,4, n_0), (n_4,5,n_2),(n_5,10,n_1)]**

6. Remove the first element of the list **(n_4,3,n_3)** and add its successors **n_6** to the list of states respecting the ascending order. We get: **[(n_2,4, n_0), (n_4,5,n_2),(n_6,7,n_4),(n_5,10,n_1)]**

7. Remove the first element of the list **(n_2,4, n_0)** and add its successors **n_4** to the list of states respecting the ascending order. We get: **[(n_4,5,$n_{2\,(n3)}$), (n_4,6,$n_{2\,(n0)}$),(n_6,7,n_4),(n_5,10,n_1)]**



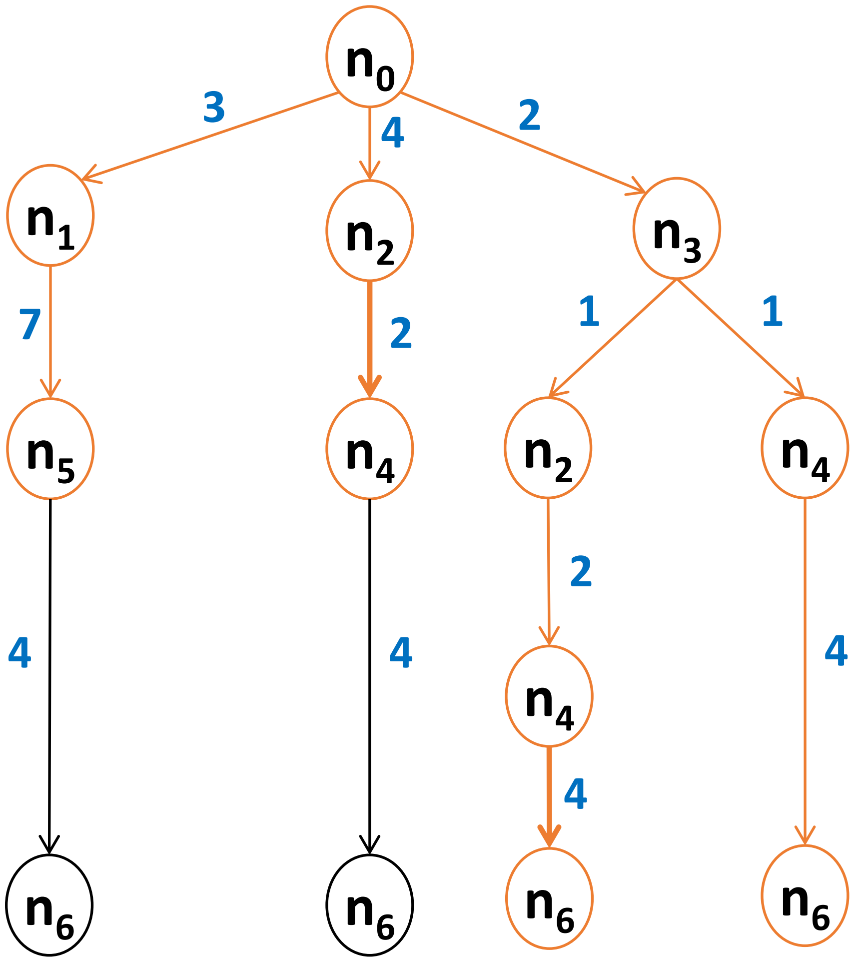
Uniform-Cost Search Algorithm

5. Remove the first element of the list **$(n_2,3,n_3)$** and add its successors **n_4** to the list of states respecting the ascending order.
We get: **$[(n_4,3,n_3), (n_2,4, n_0), (n_4,5,n_2),(n_5,10,n_1)]$**

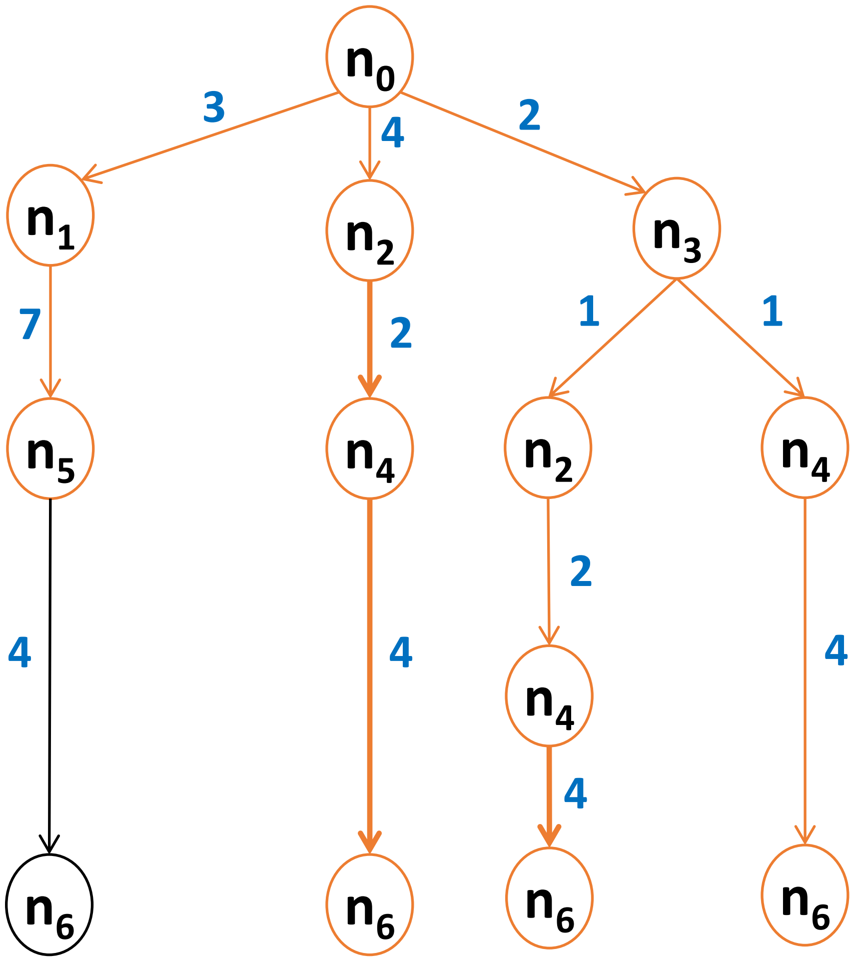
6. Remove the first element of the list **$(n_4,3,n_3)$** and add its successors **n_6** to the list of states respecting the ascending order.
We get: **$[(n_2,4, n_0), (n_4,5,n_2),(n_6,7,n_4),(n_5,10,n_1)]$**

7. Remove the first element of the list **$(n_2,4, n_0)$** and add its successors **n_4** to the list of states respecting the ascending order.
We get: **$[(n_4,5,n_{2\ (n3)}), (n_4,6,n_{2\ (n0)}),(n_6,7,n_4),(n_5,10,n_1)]$**

8. Remove the first element of the list **$(n_4,5,n_{2\ (n3)})$** and add its successors **n_6** to the list of states respecting the ascending order.
We get: **$[(n_4,6,n_{2\ (n0)}),(n_6,7,n_{4\ (n3)}), (n_6,9,n_{4\ (n2)}), (n_5,10,n_1)]$**



Uniform-Cost Search Algorithm



5. Remove the first element of the list **$(n_2,3,n_3)$** and add its successors **n_4** to the list of states respecting the ascending order.
We get: **$[(n_4,3,n_3), (n_2,4, n_0), (n_4,5,n_2),(n_5,10,n_1)]$**

6. Remove the first element of the list **$(n_4,3,n_3)$** and add its successors **n_6** to the list of states respecting the ascending order.
We get: **$[(n_2,4, n_0), (n_4,5,n_2),(n_6,7,n_4),(n_5,10,n_1)]$**

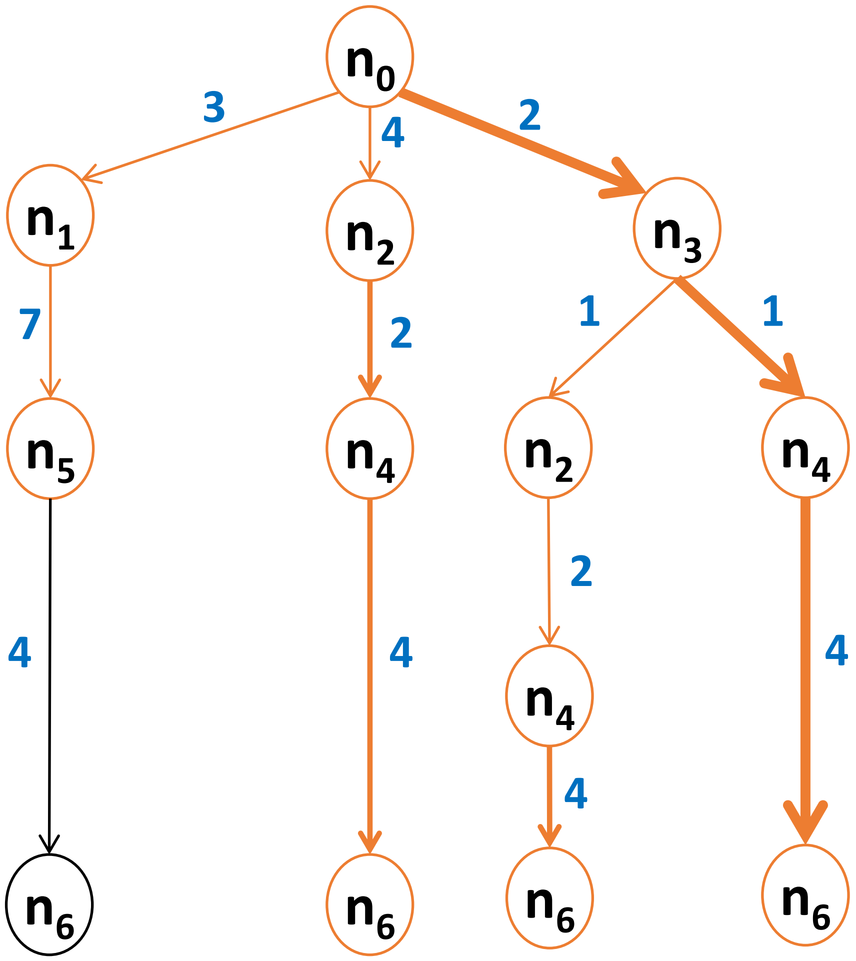
7. Remove the first element of the list **$(n_2,4, n_0)$** and add its successors **n_4** to the list of states respecting the ascending order.
We get: **$[(n_4,5,n_{2\,(n3)}), (n_4,6,n_{2\,(n0)}),(n_6,7,n_4),(n_5,10,n_1)]$**

8. Remove the first element of the list **$(n_4,5,n_{2\,(n3)})$** and add its successors **n_6** to the list of states respecting the ascending order.
We get: **$[(n_4,6,n_{2\,(n0)}),(n_6,7,n_{4\,(n3)}), (n_6,9,n_{4\,(n2)}), (n_5,10,n_1)]$**

9. Remove the first element of the list **$(n_4,6,n_{2\,(n0)})$** and add its successors **n_6** to the list of states respecting the ascending order.
We get: **$[(n_6,7,n_{4\,(n3)}), (n_6,9,n_{4\,(n2,n3)}), (n_6,10,n_{4\,(n2,n0)}), (n_5,10,n_1)]$**

Uniform-Cost Search Algorithm

9. Remove the first element of the list **$(n_4,6,n_{2\,(n0)})$** and add its successors **n_6** to the list of states respecting the ascending order.
We get: **[$(n_6,7,n_{4\,(n3)})$, ~~$(n_6,9,n_{4\,(n2,n3)})$~~, ~~$(n_6,10,n_{4\,(n2,n0)})$~~, $(n_5,10,n_1)$]**



n_6 Appears in the head of

Open then stop the search

The path with optimal cost: n_0,n_3,n_4, n_6

If the cost of **each arc = 1**, then **Uniform-Cost = BFS**

Heuristic-Based Search Algorithms

Best-First Search

1. Start the search by a **List** containing the starting state (**initial node**) of the problem

2. If **List** not empty:

   - Select a state **n** with **minimal** measure to expand

   - If **n** is a final state (**Goal node**) then return Success

   - Else, add all **n successor nodes** to the List with respect of ascending order according to the utility measure.

   - Restart at point 2.

3. Else return Failure.

Heuristic-Based Search Algorithms

Greedy Best-First Search

- The utility measure is given by an estimation function **h**.

- For each state **n, h(n)** represents the **estimated cost** from **n** to a **final state**.

For example, in the problem of the shortest path between two cities,

we can take **h(n) = direct distance** between **n** and the **destination city**.

- Greedy search will choose the state that seems **closest** to a final state

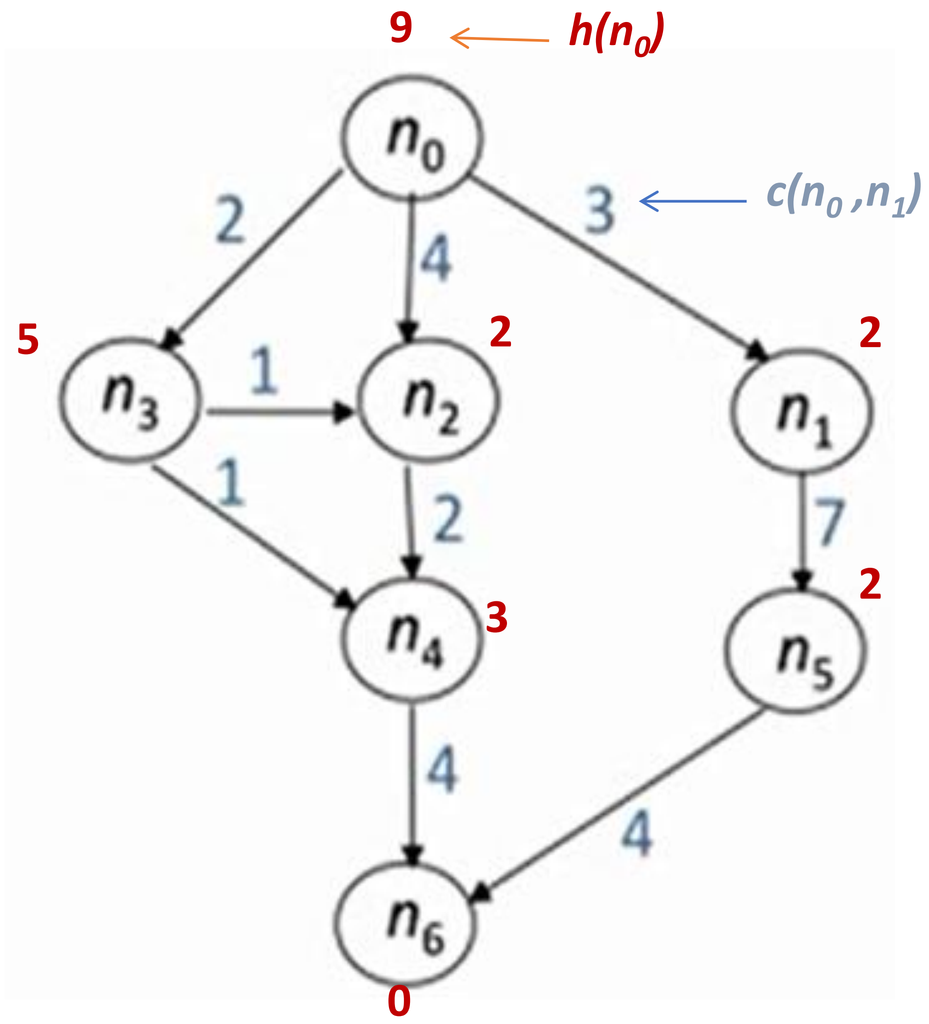
according to the **estimation function h**.

Greedy Best-First Search

Open List :

- (n_0,9,void)
- (n_2,2,n_0), (n_1,2,n_0), (n_3,5,n_0)
- (n_1,2,n_0), (n_4,3,n_2), (n_3,5,n_0)
- (n_5,2,n_1), (n_4,3,n_2), (n_3,5,n_0)
- (n_6,0,n_5), (n_4,3,n_2), (n_3,5,n_0)

Path : $n_0 \rightarrow n_1 \rightarrow n_5 \rightarrow n_6$



Heuristic-Based Search Algorithms

A* Search

- The utility measure is given by an evaluation function f

- For each node **n**: ***f*(*n*) = *g*(*n*) + *h*(*n*)**

  ○ ***g*(*n*)** *Is the cost till present to get **n***

  ○ ***h*(*n*)** Is the estimated cost to go from **n** to the **goal node**.

  ○ ***f*(*n*)** Is the total estimated cost to go from the **initial node** to the **goal node** going through **n**

h is said to be **admissible** if for all **n**: **h(n) ≤ c(n)**

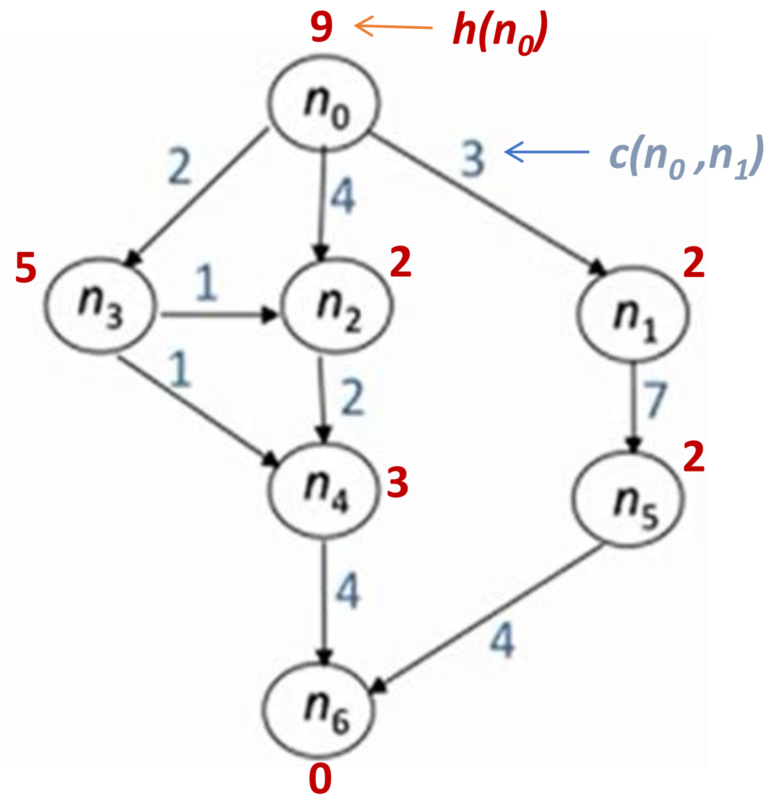
c(n) being the real cost leading from **n** to the **final state**

A* Search Algorithm

```
1. Declare two nodes n, ns
2. Declare two lists Open and Closed (initially empty)
3. Add initial node to Open
4. If Open is empty Then Exit the loop with a failure
5. Current node n = node at the head of Open
6. Remove n from Open and add it to Closed.
7. If n= goal Then Exit the loop and return the path
   Else : For each successor ns of n:
          - Initilize the value $g(ns) = g(n) + c(n,ns)$
          - Set parent of ns to n
          - If Open or Closed contains a node ns’=ns with $f(ns) \leq f(ns’)$
            Then remove ns’ from Open or Closed and insert ns into Open (with respect to the ascending order of f)
            Else : Insert ns into Open (with respect to the ascending order of f)
          - Go to 4.
```

A* Search Algorithm

Illustrative example: Path-Finding between two cities



n_0 : Departure city (initial node)

n_6 : Destination city(goal node)

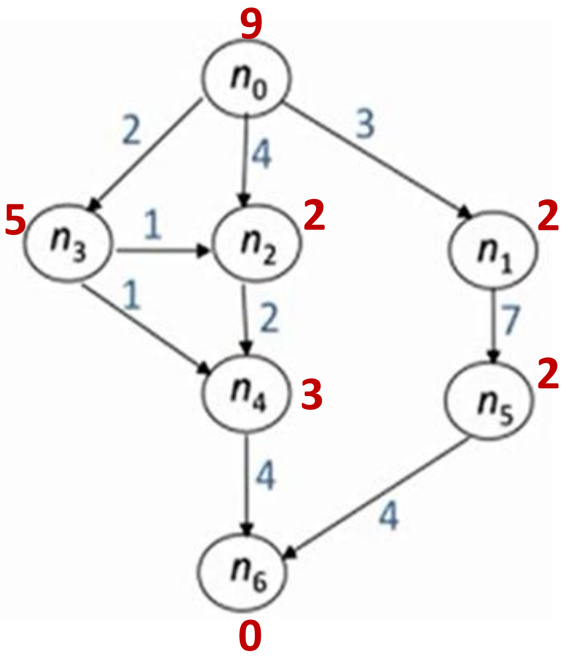
h : Direct distance between a city and the destination city (heuristic)

c : Real distance between two cities

A* Search Algorithm

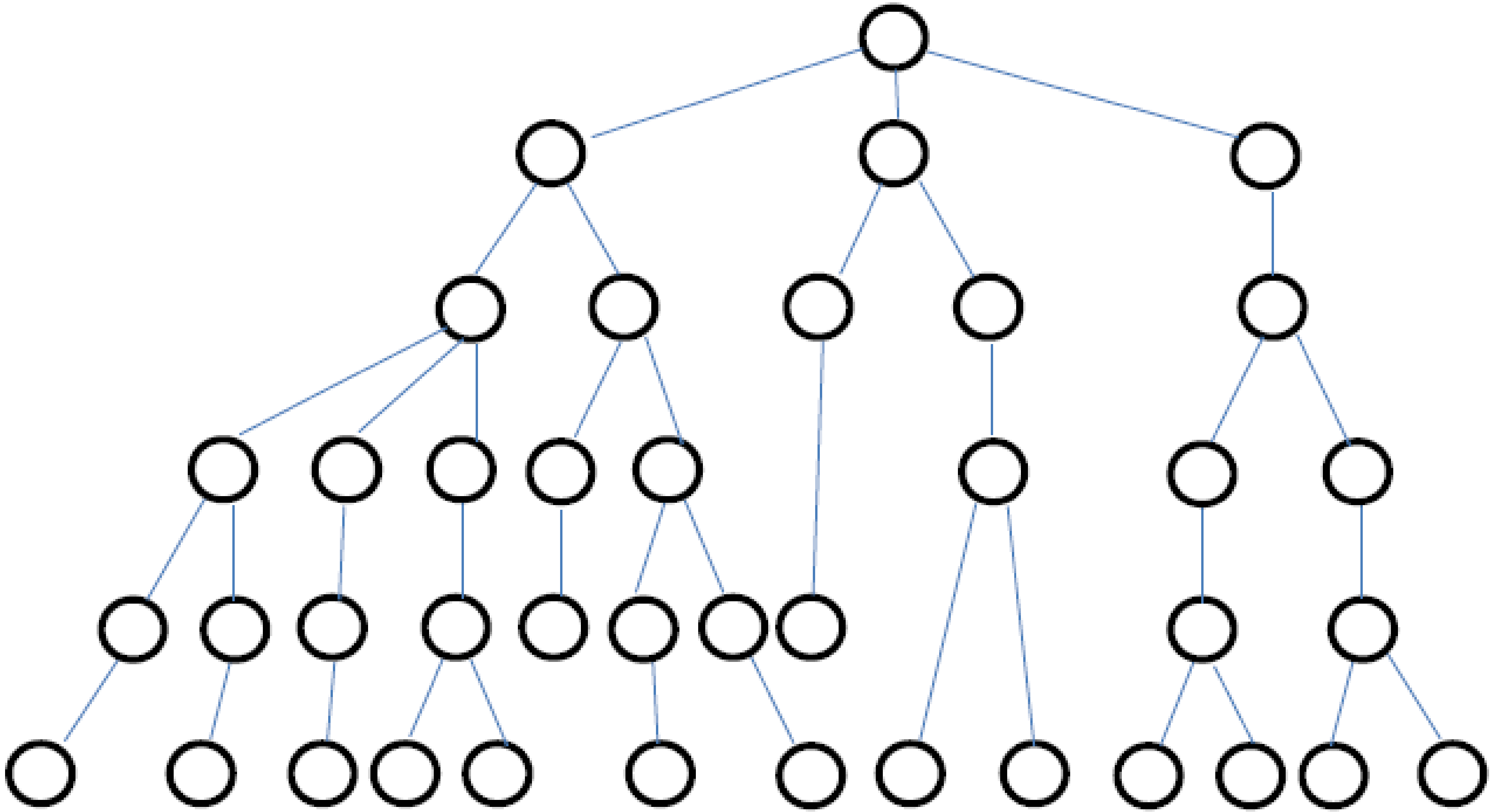
Illustrative example: Path-Finding between two cities

State of Open in each iteration	**State of Closed in each iteration**
(State, f, Parent)	**(State, f, Parent)**
1. (n_0, 9, void)	1. Vide
2. (n_1,5,n_0), (n_2,6,n_0), (n_3,7,n_0)	2. (n_0, 9, void)
3. (n_2,6,n_0), (n_3,7,n_0), (n_5,12,n_1)	3. (n_0, 9, void), (n_1,5,n_0)
4. (n_3,7,n_0), (n_4,9,n_2), (n_5,12,n_1)	4. (n_0, 9, void), (n_1,5,n_0), (n_2,6,n_0)
5. (n_2,5,n_3), (n_4,6,n_3), (n_5,12,n_1)	5. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0)
6. (n_4,6,n_3), (n_5,12,n_1)	6. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3)
7. (n_6,7,n_4), (n_5,12,n_1)	7. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3)
8. Solution : **n_0,n_3,n_4,n_6**	8. (n_0, 9, void), (n_1,5,n_0), (n_3,7,n_0), (n_2,5,n_3), (n_4,6,n_3), (n_6,7,n_4)

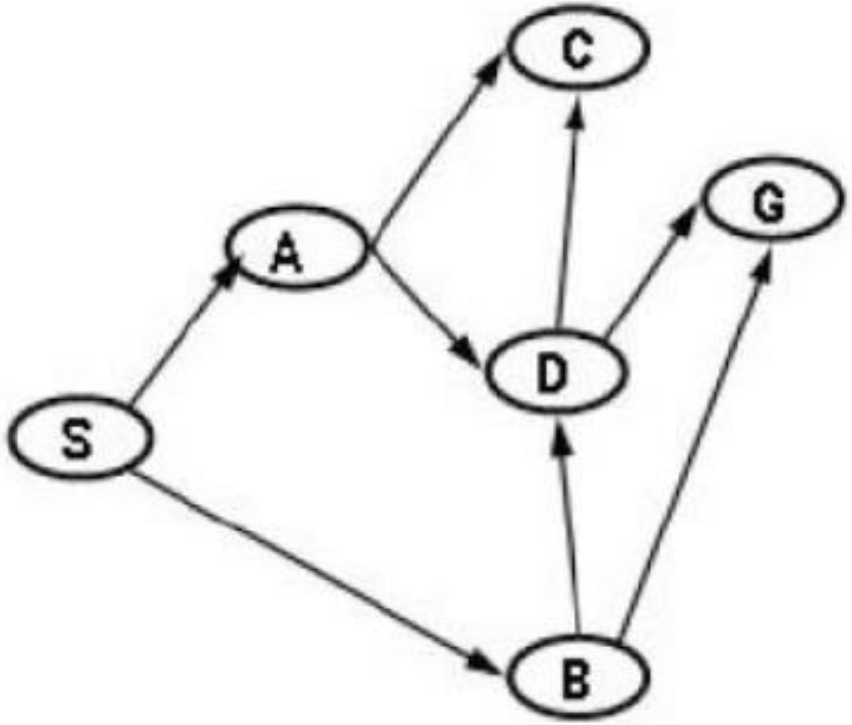


Exercise n° 1: Given the following search tree

- Number the different states of the tree according to their traversal orders using the two strategies Breadth-First followed by Depth-First.

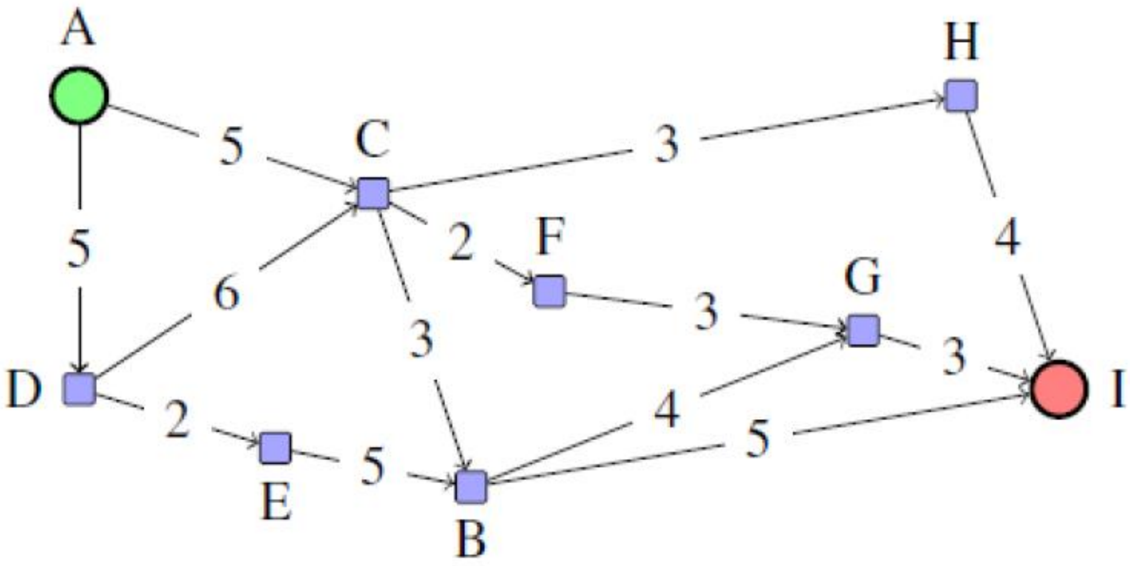


Exercise n°2: Transform the following graph into a search tree, then apply a breadth-first search followed by a depth-first search to find the state **G** from **S**. In case of conflicts between nodes, follow alphabetical order.



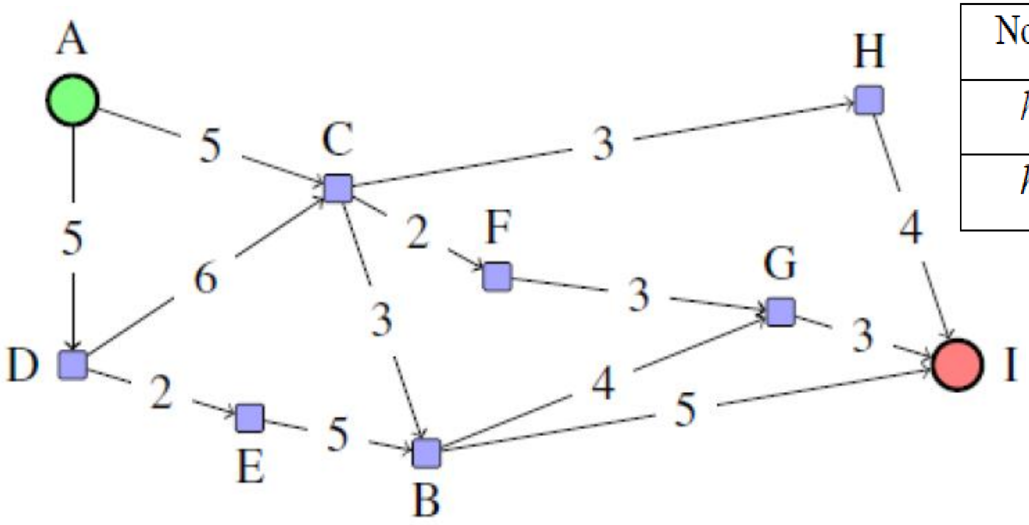
Exercise n° 3

We consider the following map. The objective is to find the optimal path between A and I. We also give two heuristics h1 and h2:



Nœud	A	B	C	D	E	F	G	H	I
h1	10	5	5	10	10	3	3	3	0
h2	10	2	8	11	6	2	1	5	0

1. Apply the strategies BFS, DFS and Uniform-Cost
2. Find the optimal (we minimize) path using the following algorithms:
   - a) Greedy Best-First Search using h2 as heuristic function
   - b) A* Search using h1



Nœud	A	B	C	D	E	F	G	H	I
h1	10	5	5	10	10	3	3	3	0
h2	10	2	8	11	6	2	1	5	0

1) Greedy Best-First Search using h2

[(A,10,void)]

[(C,8,A),(D,11,A)]

[(B,2,C),(F,2,C),(H,5,C),(D,11,A)]

[(I,0,B),(G,1,B),(F,2,C),(H,5,C)(D,11,A)]

The optimal path: A → C → B → I

2) A* Search using h1

[(A,10,void)]

[(C,10,A),(D,15,A)]

[(F,10,C), (H,11,C), (B,13,C),(D,15,A)]

[(H,11,C), (B,13,C), (G,13,F),(D,15,A)]

[(I,12,H)(B,13,C), (G,13,F),(D,15,A)]

Le chemin traversé est : A → C → H → I