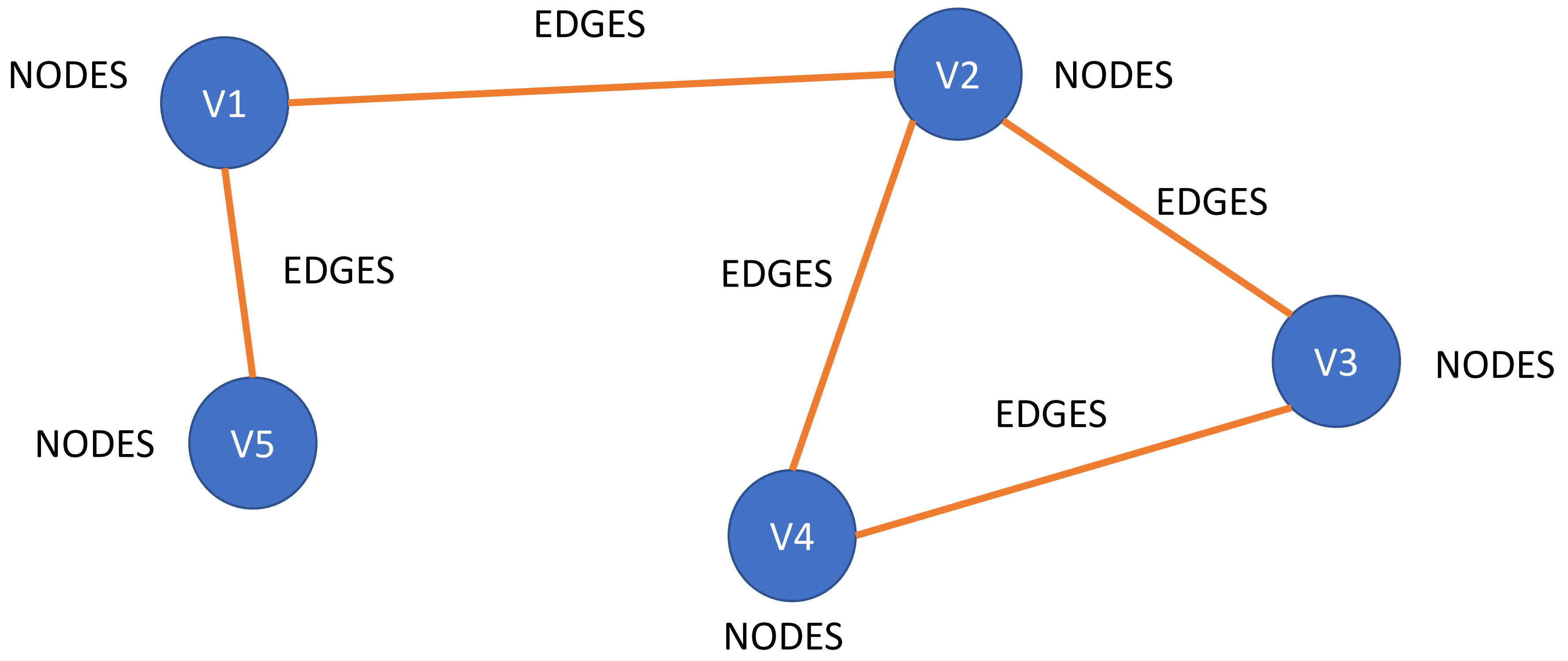
Review

Daniel Nogueira

dnogueira@ipca.pt

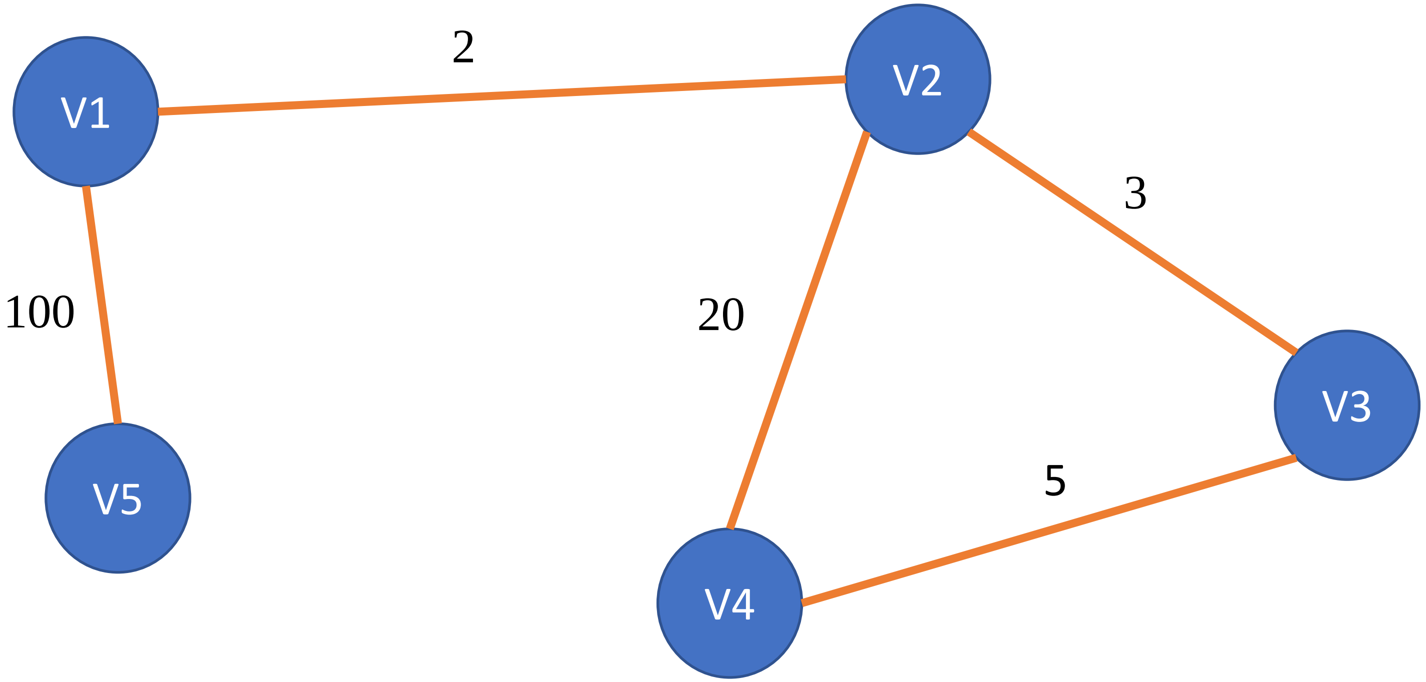
Graph - Representation

"Graphs are mathematical structures that allow you to encode **relationships** between pairs of **objects.** ".



Graph - Representation

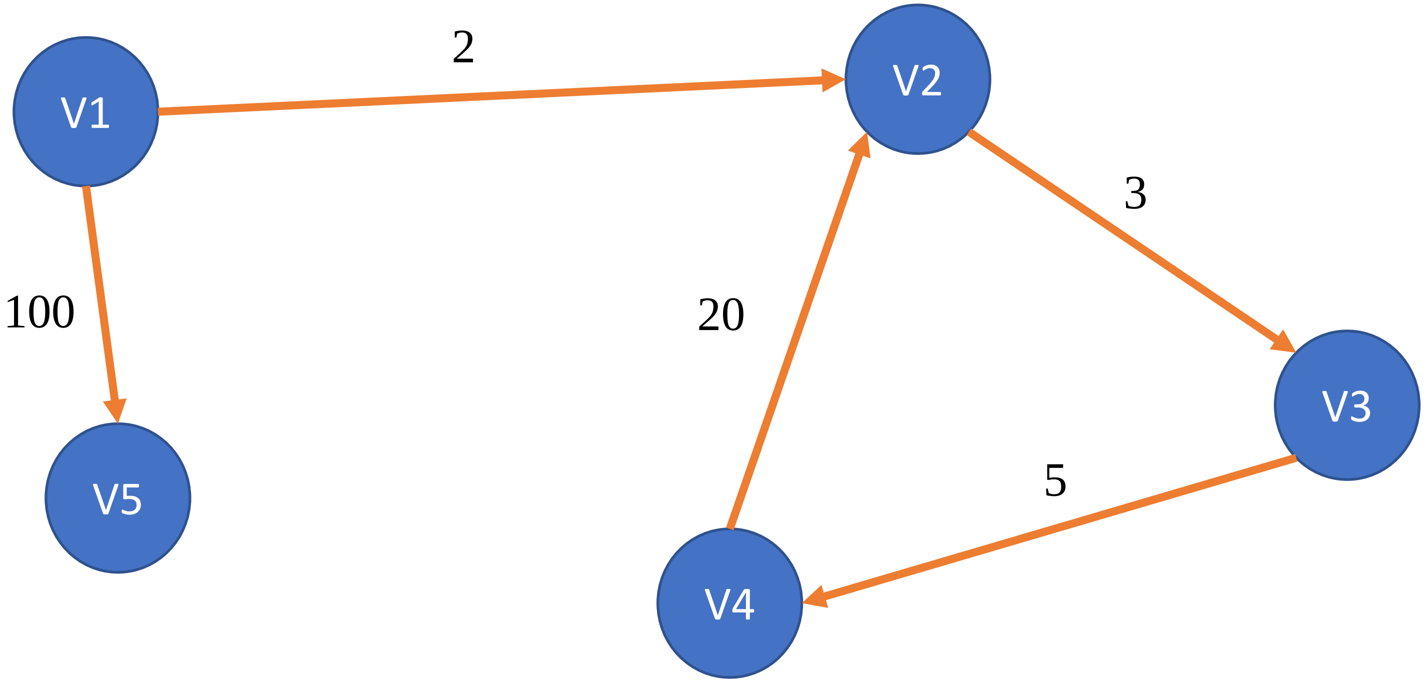
Adjacent List



Node	Connections
V1	V2, V5
V2	V1, V3, V4
V3	V2, V4
V4	V2, V3
V5	V1

Graph - Representation

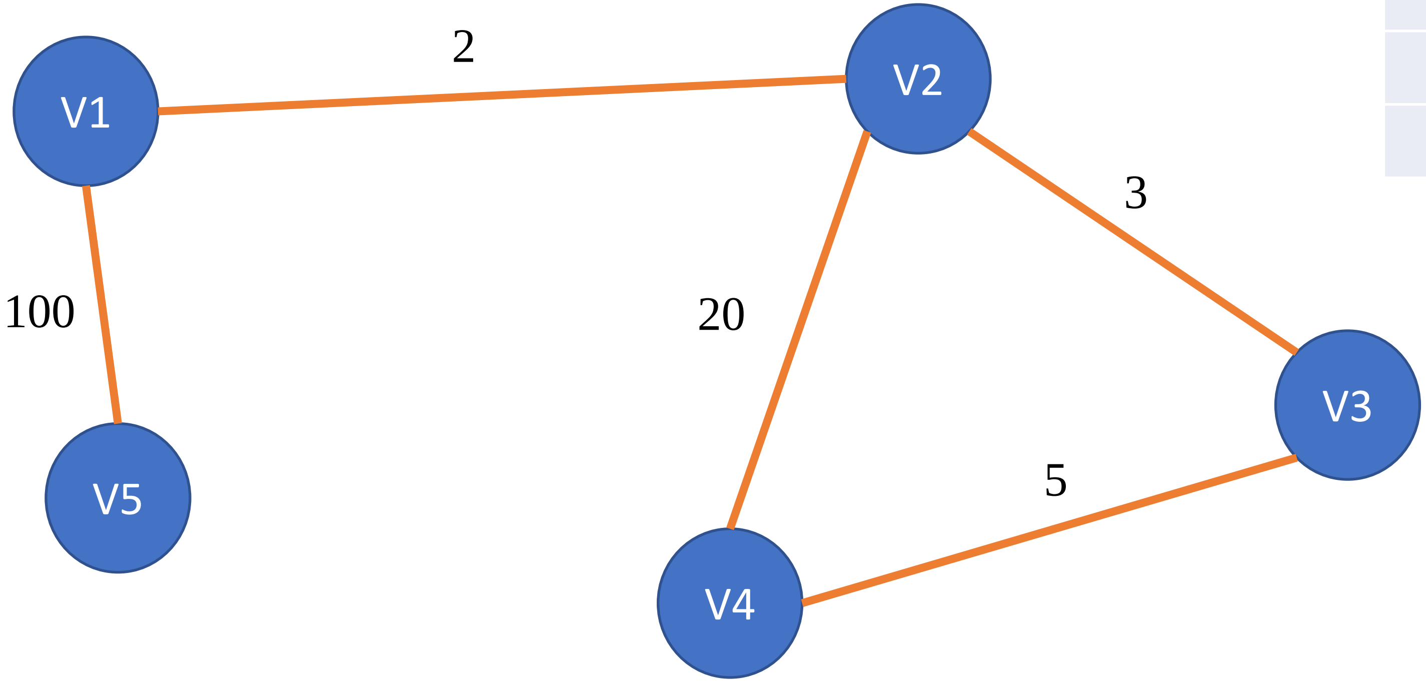
Adjacent List



Node	Connections
V1	V2, V5
V2	V3
V3	V4
V4	V2
V5	----

Graph - Representation

Adjacent Matrix

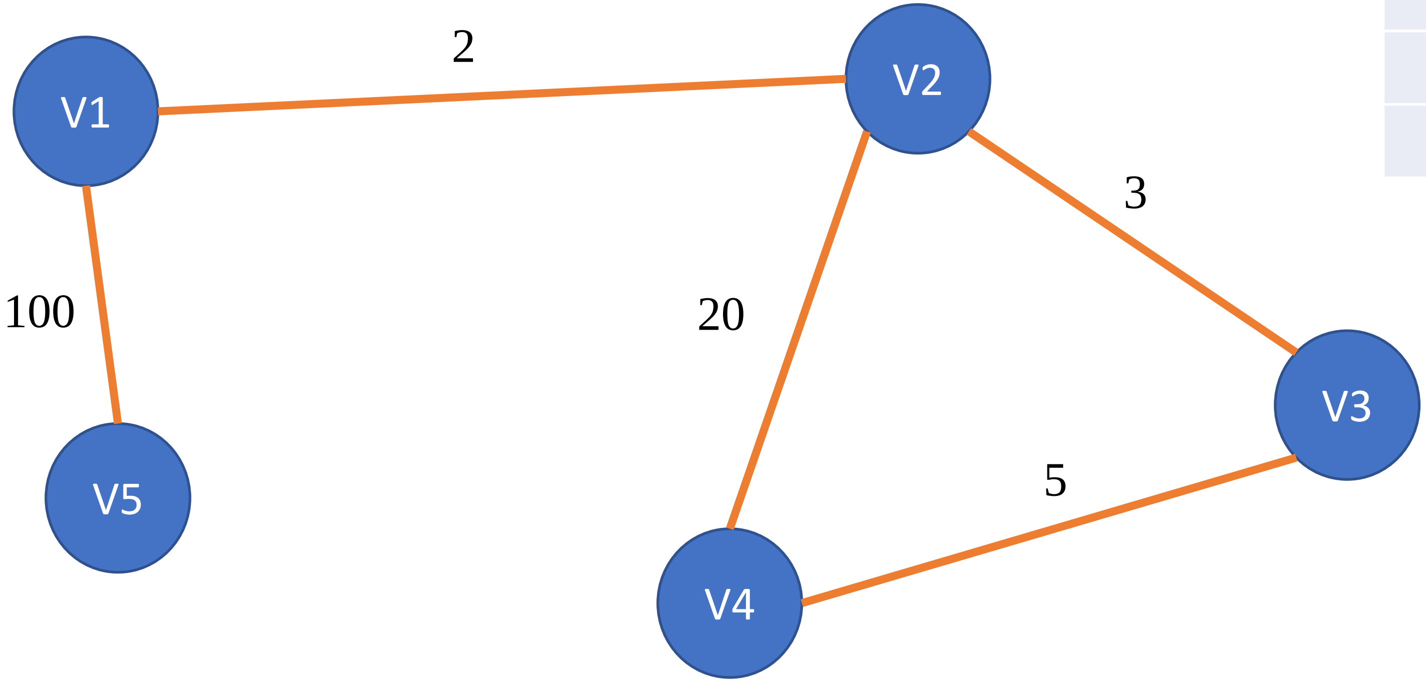


Node	V1	V2	V3	V4	V5
V1	0	1	0	0	1
V2	1	0	1	1	0
V3	0	1	0	1	0
V4	0	1	1	0	0
V5	1	0	0	0	0

No weighted

Graph - Representation

Adjacent Matrix

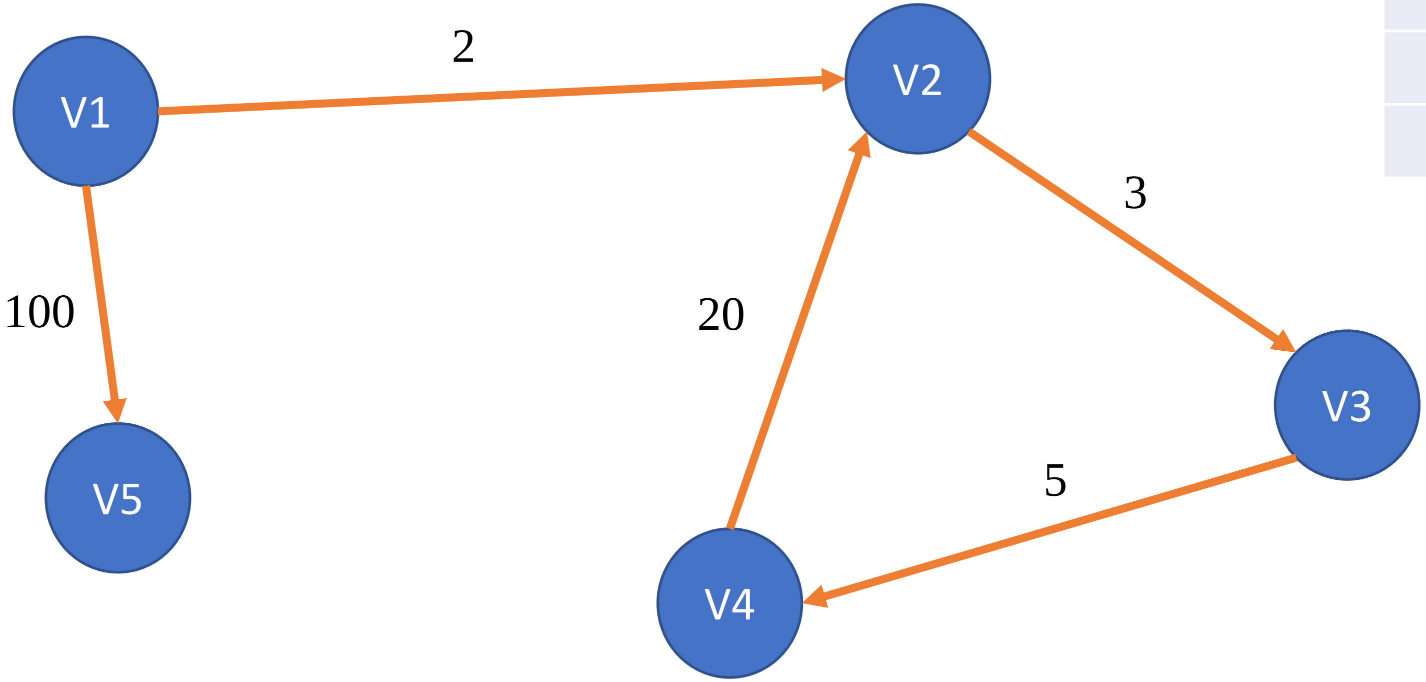


Node	V1	V2	V3	V4	V5
V1	0	2	0	0	100
V2	2	0	3	20	0
V3	0	3	0	5	0
V4	0	20	5	0	0
V5	100	0	0	0	0

Weighted

Graph - Representation

Adjacent Matrix

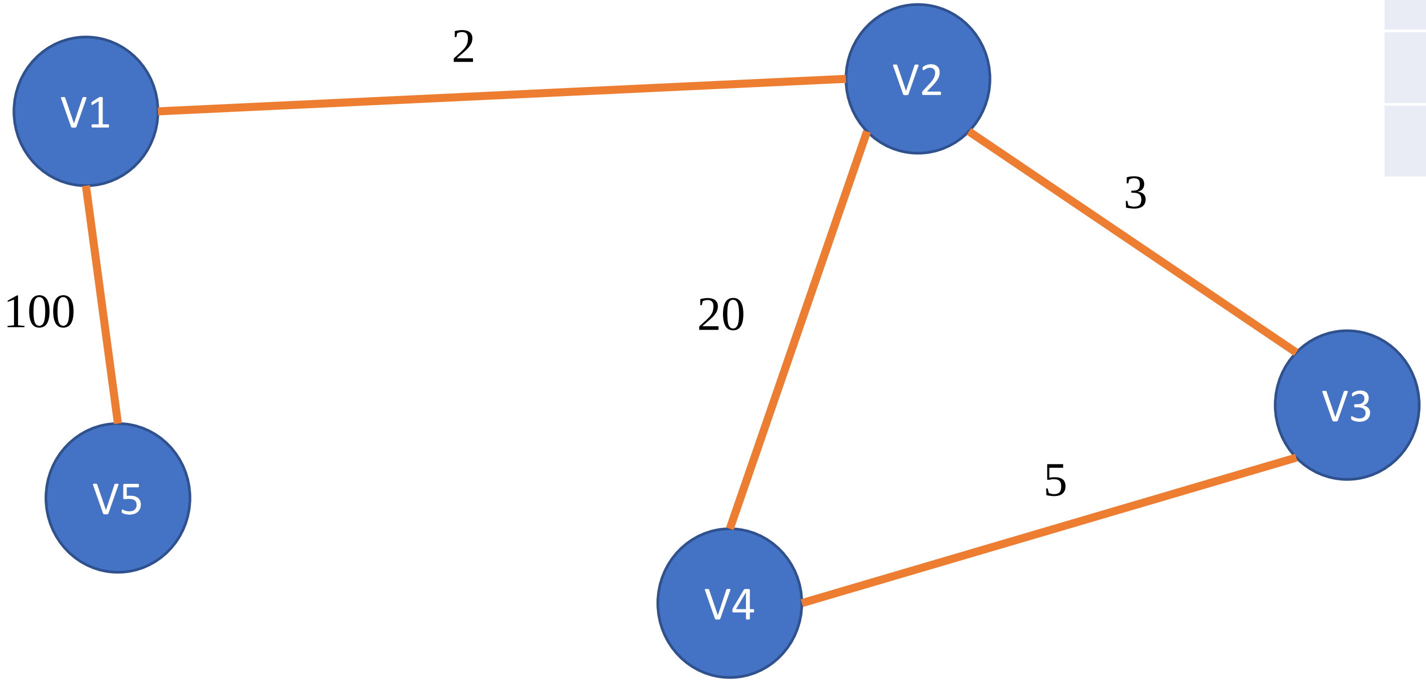


Node	V1	V2	V3	V4	V5
V1	0	2	0	0	100
V2	0	0	3	0	0
V3	0	0	0	5	0
V4	0	20	0	0	0
V5	0	0	0	0	0

Weighted

Graph - Representation

Adjacent Matrix

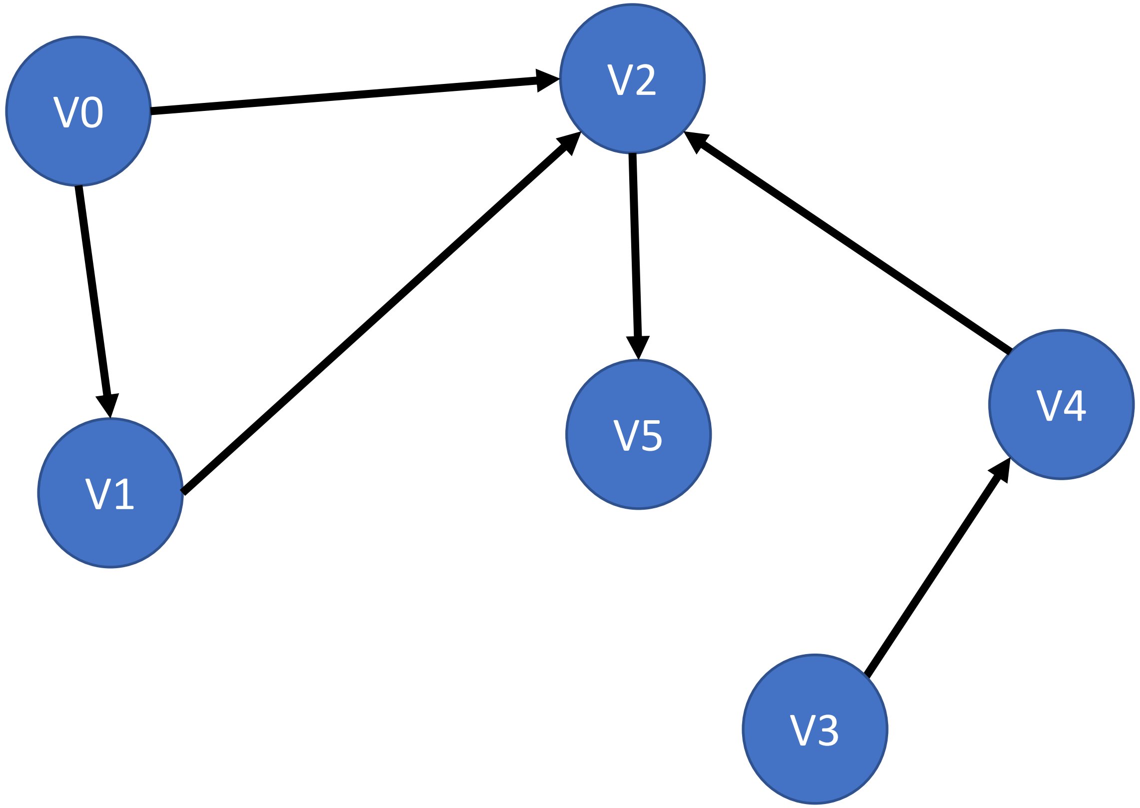


Node	V1	V2	V3	V4	V5
V1	∞	2	∞	∞	100
V2	2	∞	3	20	∞
V3	∞	3	∞	5	∞
V4	∞	20	5	∞	∞
V5	100	∞	∞	∞	∞

Weighted

Algorithms

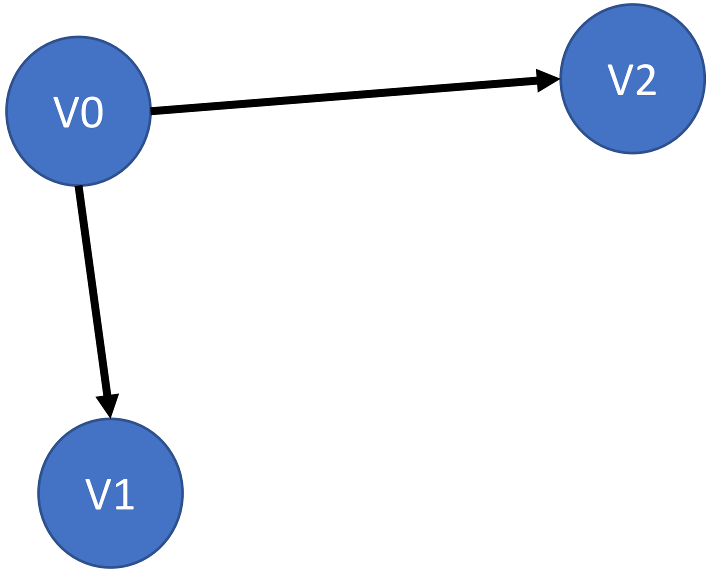
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
    - Choose an adjacent node not yet visited
    - Visit it
3. If it is a non-objective end node:
    - Return to this father
    - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

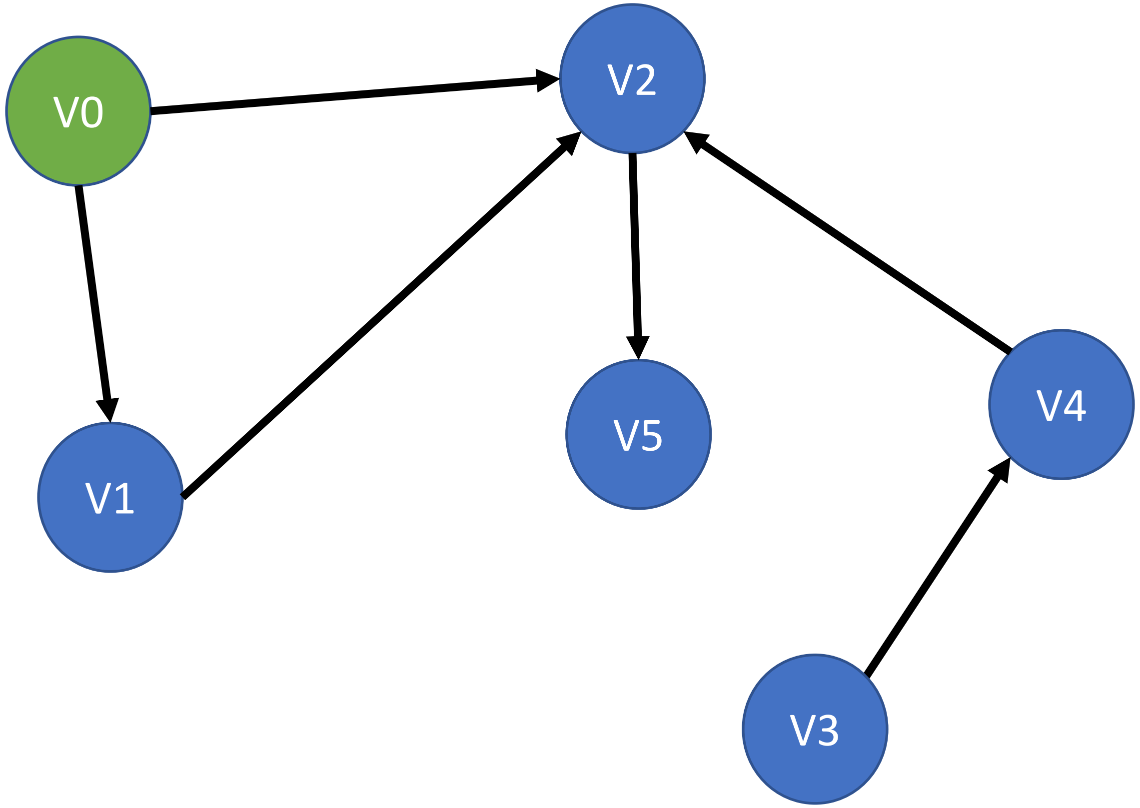
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

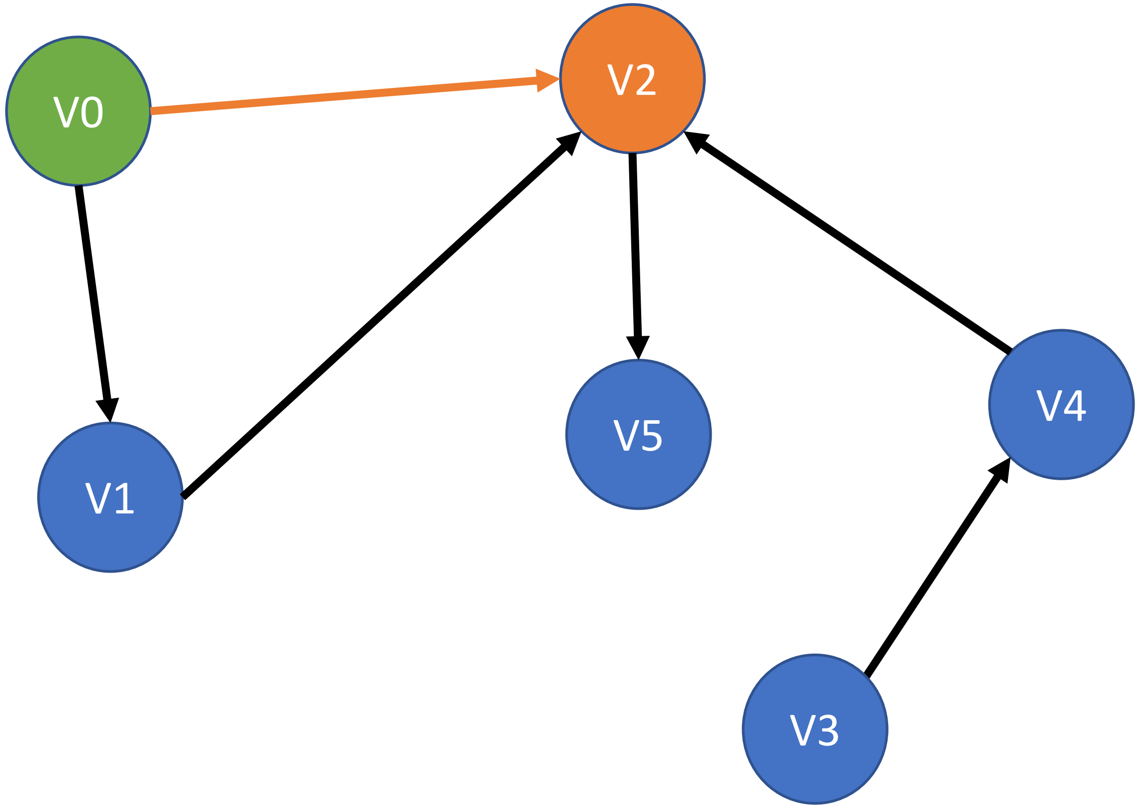
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
 - Choose an adjacent node not yet visited
 - Visit it
3. If it is a non-objective end node:
 - Return to this father
 - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

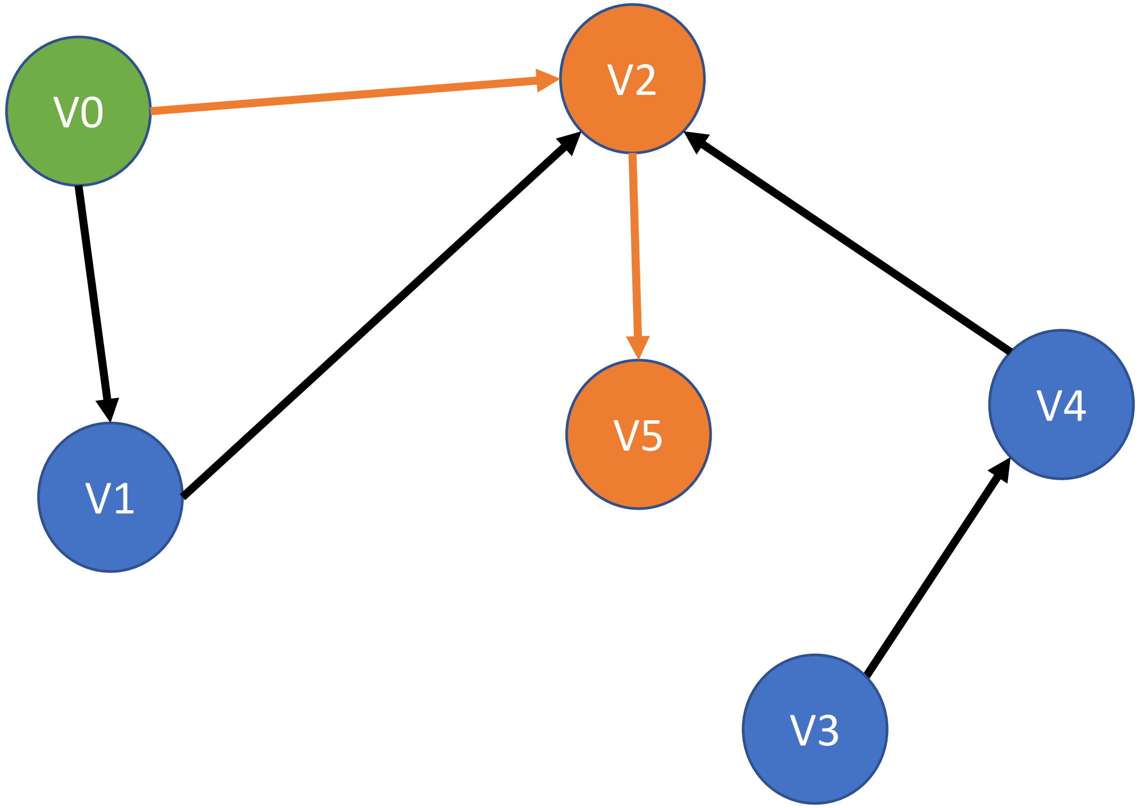
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

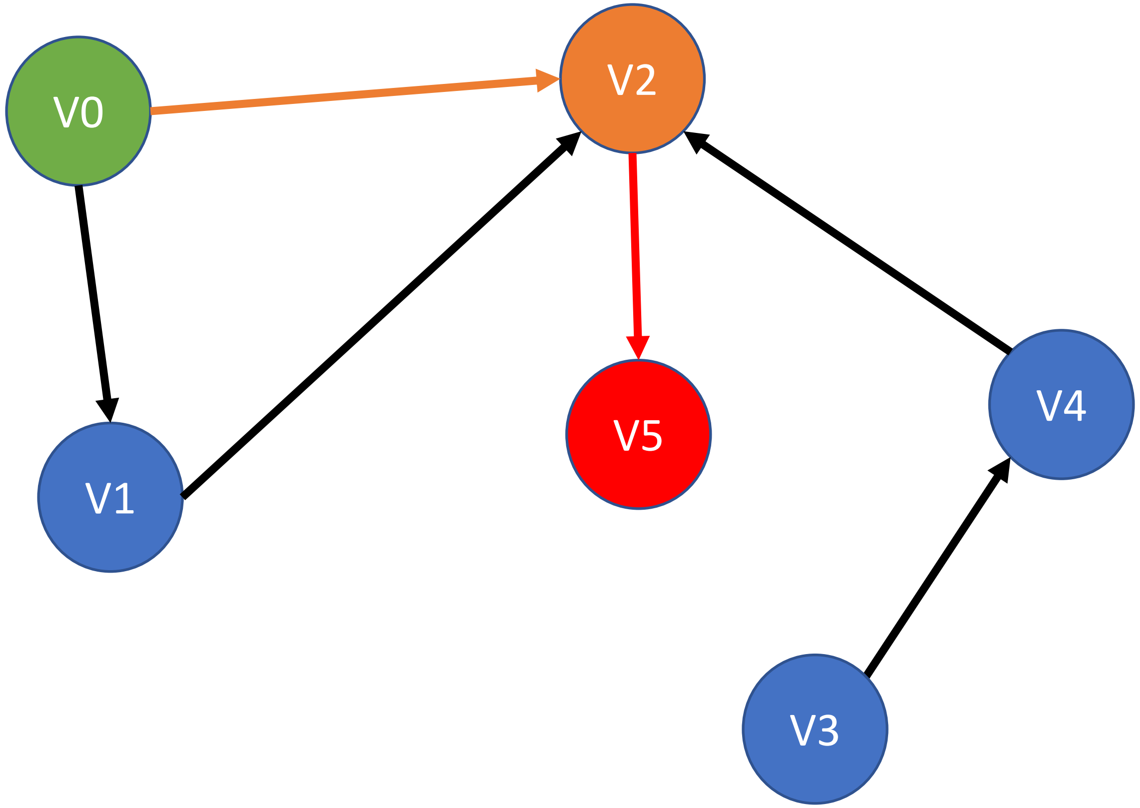
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

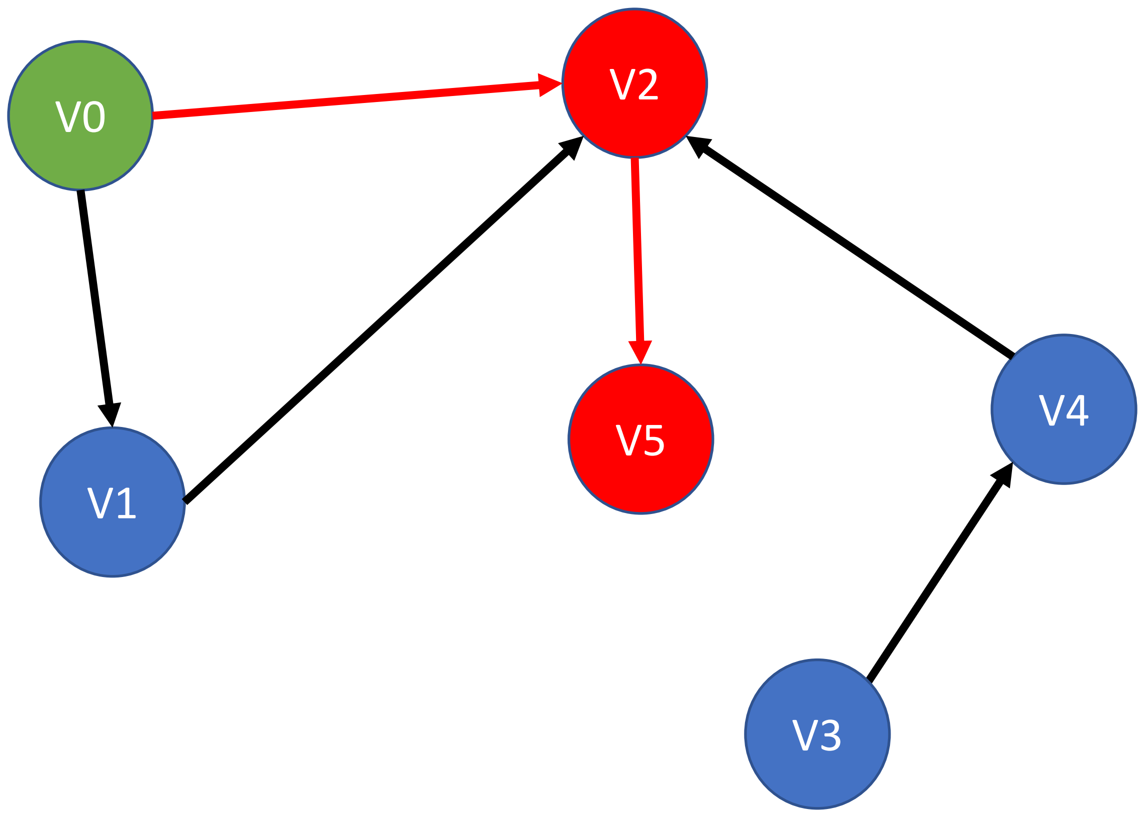
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

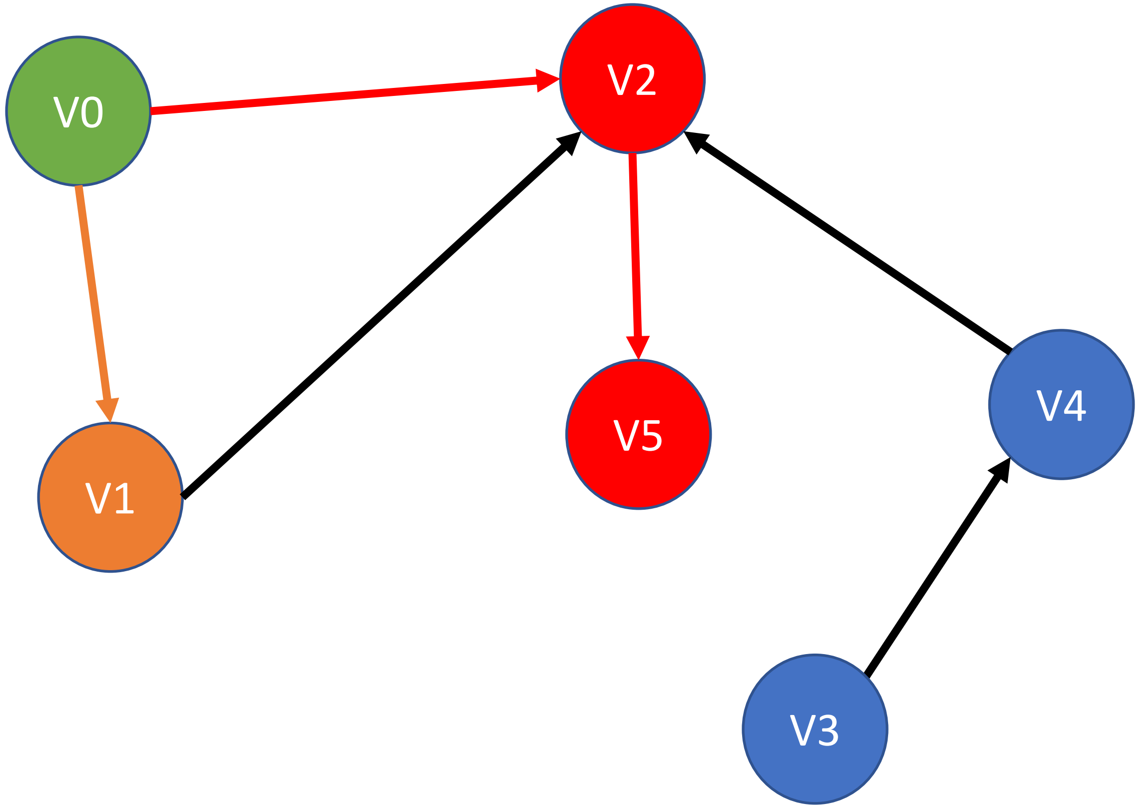
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

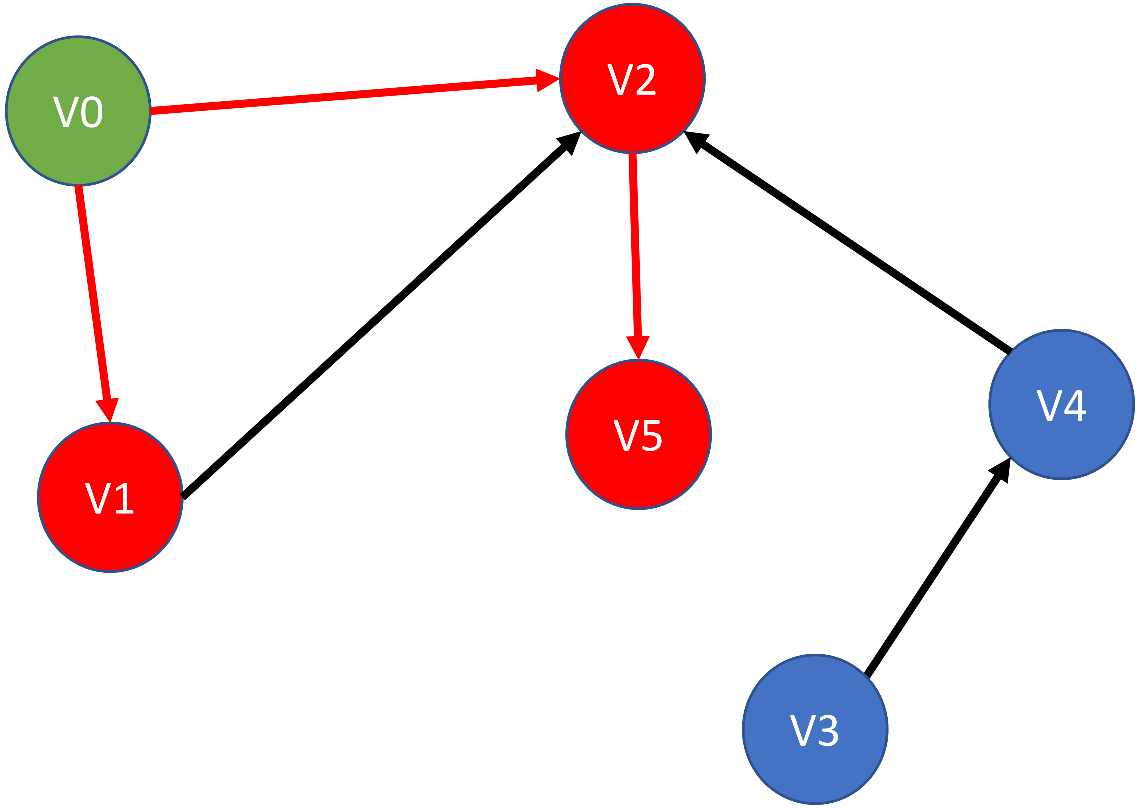
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

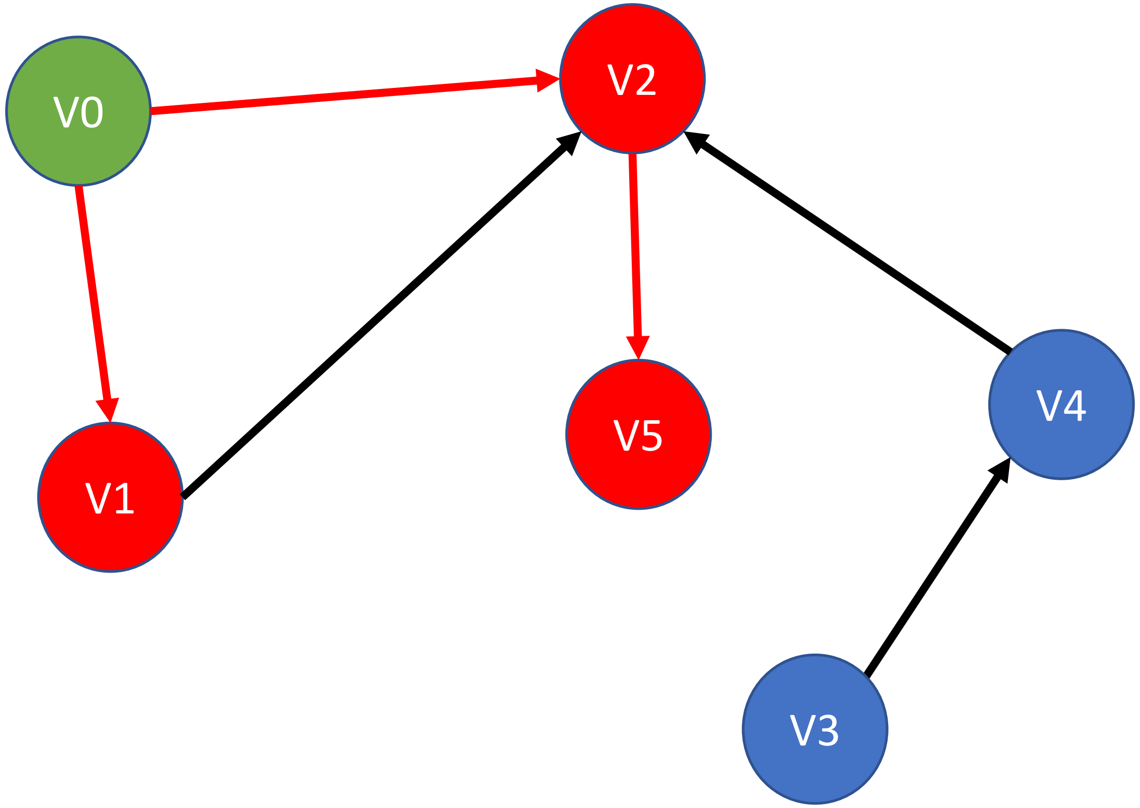
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

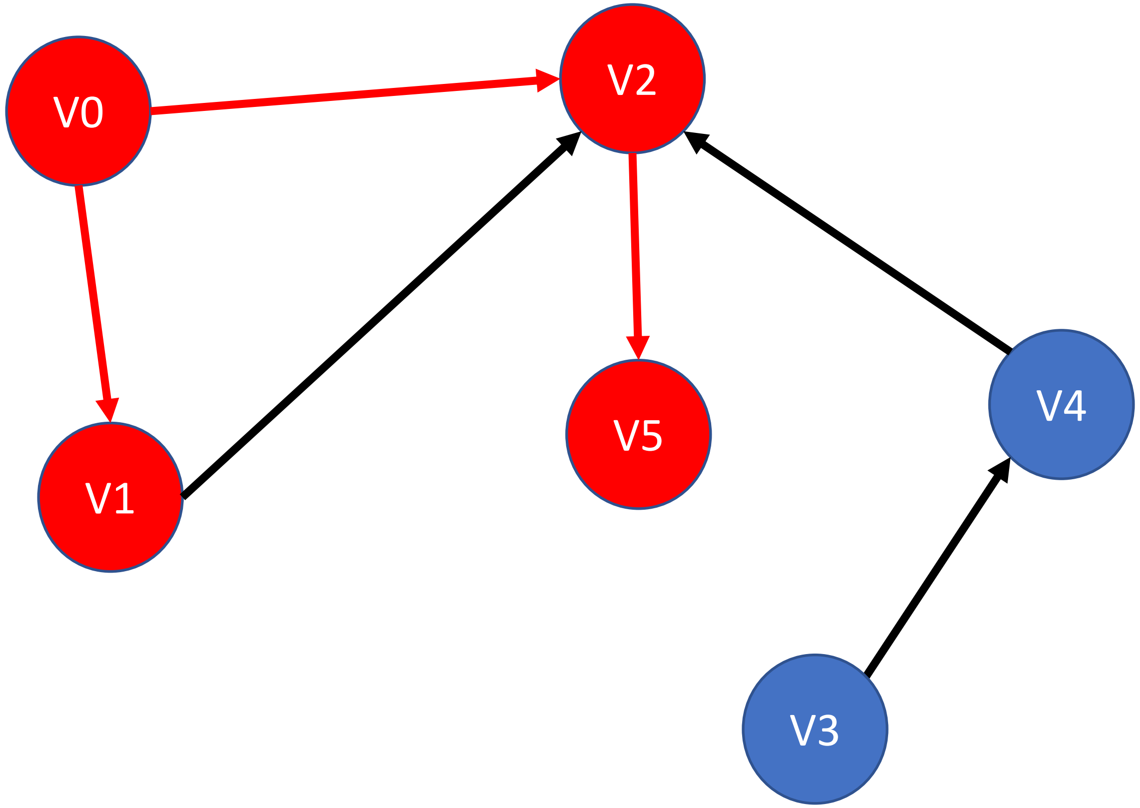
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

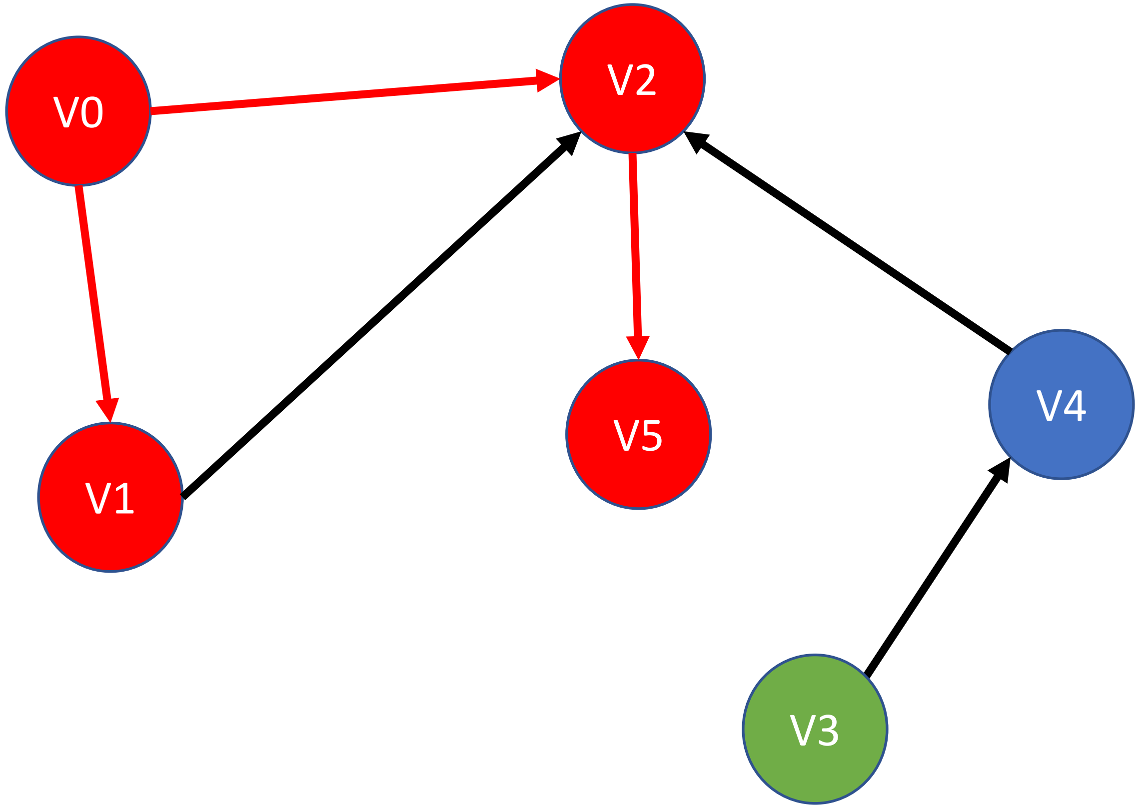
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

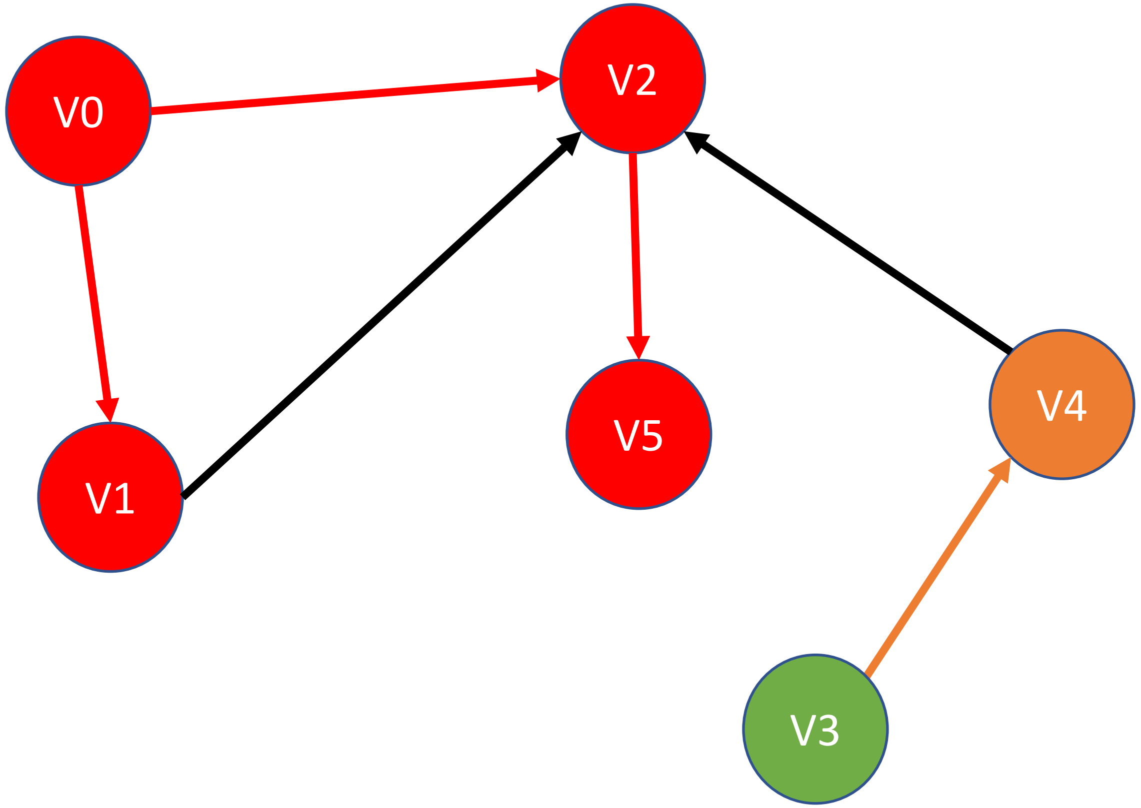
Depth-First Search (DFS)



1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
   - Choose an adjacent node not yet visited
   - Visit it
3. If it is a non-objective end node:
   - Return to this father
   - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

Depth-First Search (DFS)

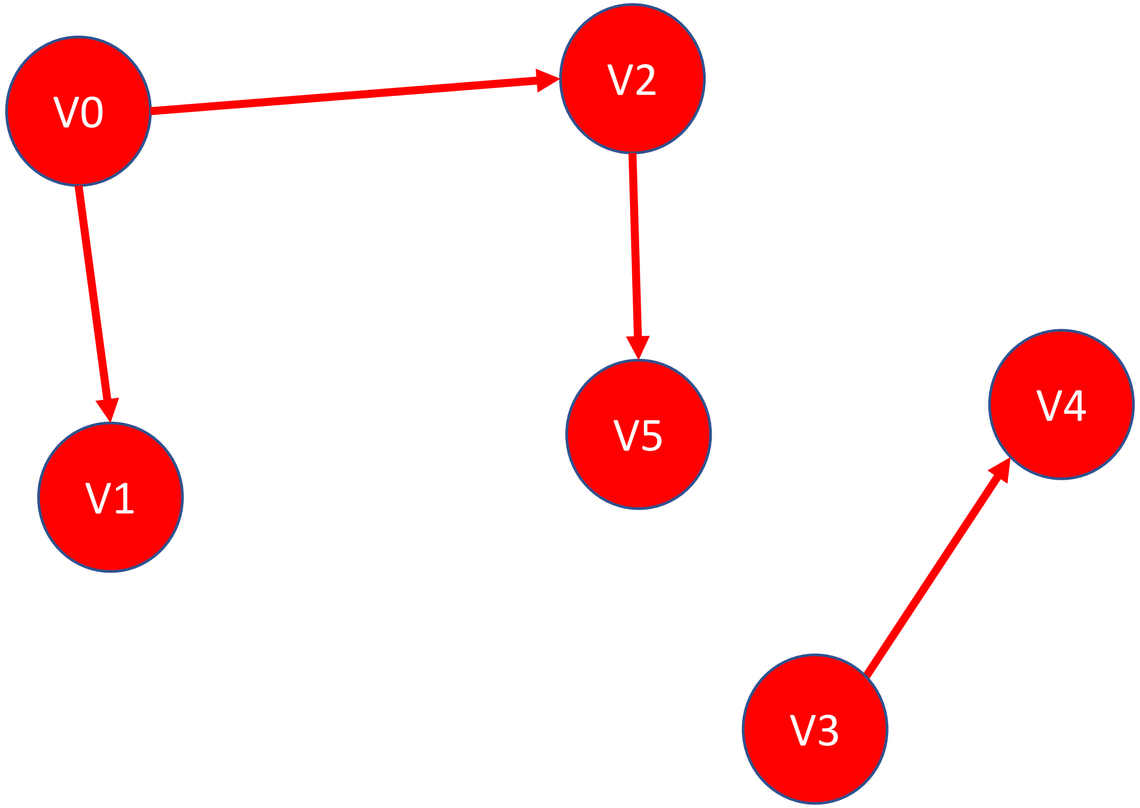


1. Set a start node
2. While this is not an objective or final node (node whose adjacency has already been visited):
    - Choose an adjacent node not yet visited
    - Visit it
3. If it is a non-objective end node:
    - Return to this father
    - If there is a father, repeat. If there is no parent, choose another start node

Algorithms

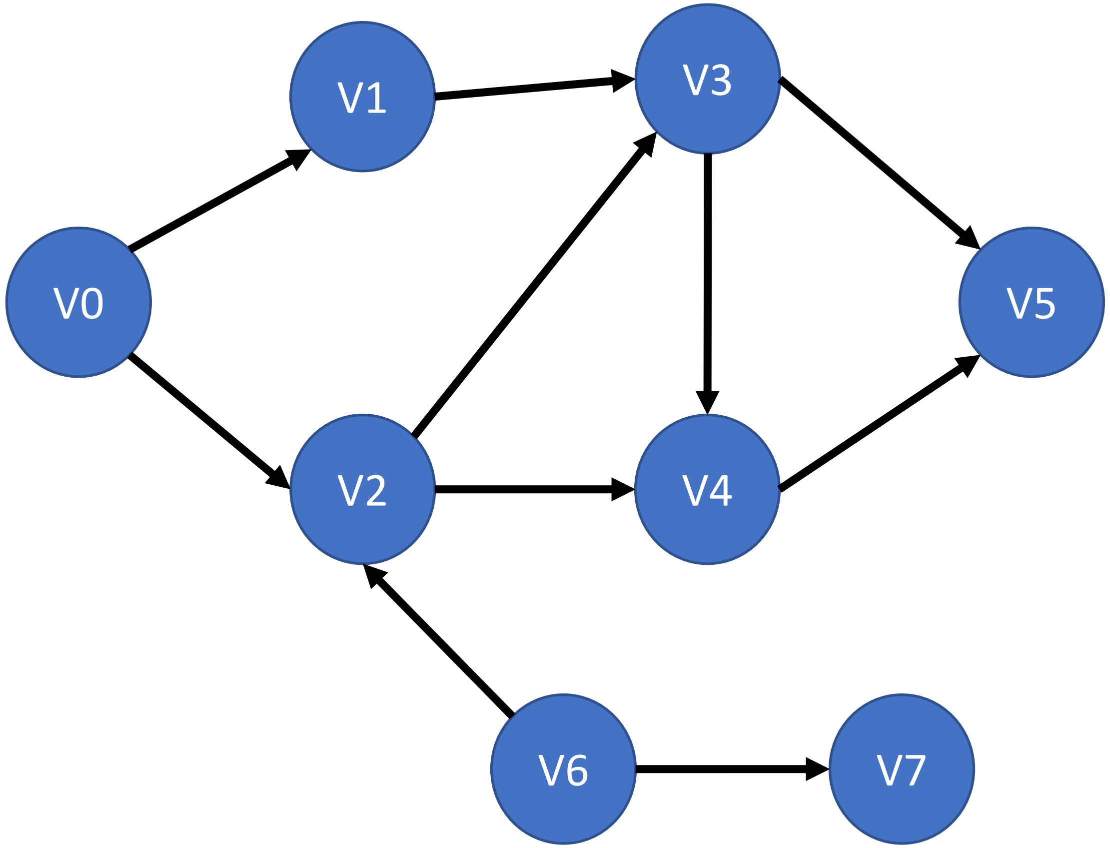
Depth-First Search (DFS)

Two Trees or Forest



Algorithms

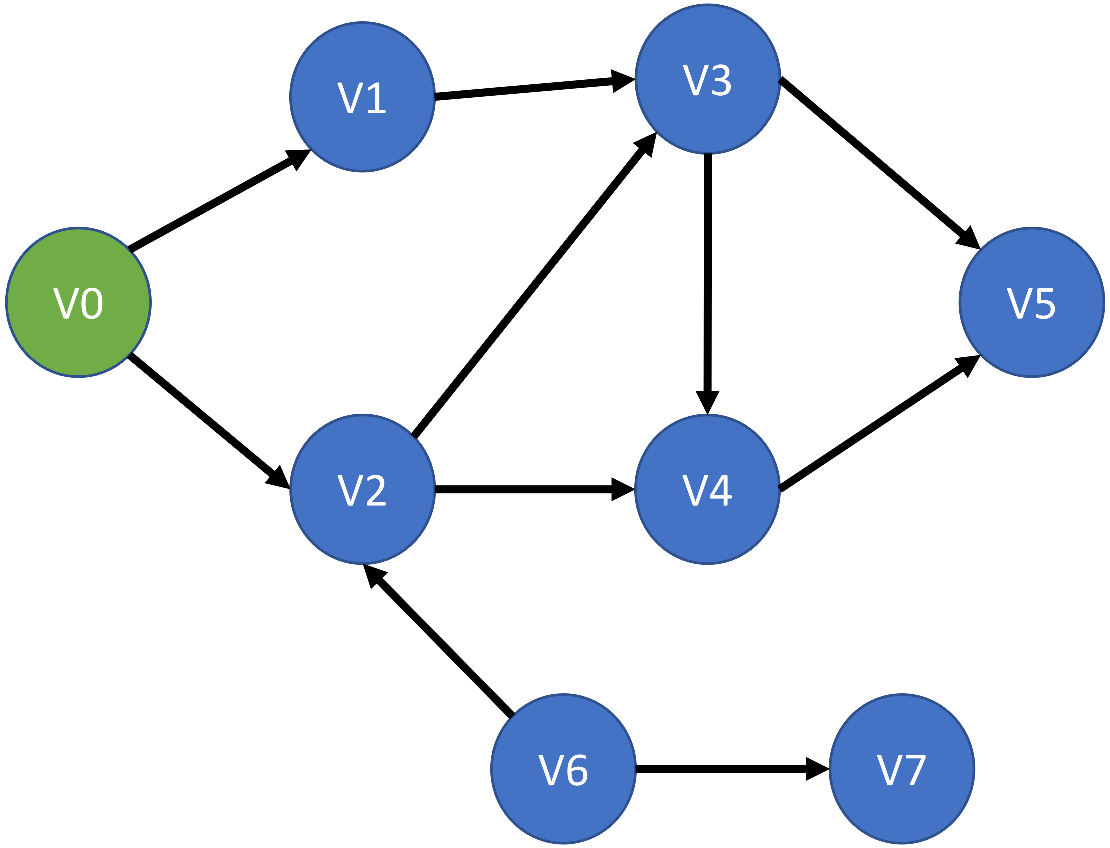
Breadth-First Search (BFS)



1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
    - Remove the 1st node from the queue, u
    - For each neighbour v of u:
        * If v is not explored:
            ** Mark v as explored
            ** Put v at the end of the queue
4. Repeat from another starting node, if there is one

Algorithms

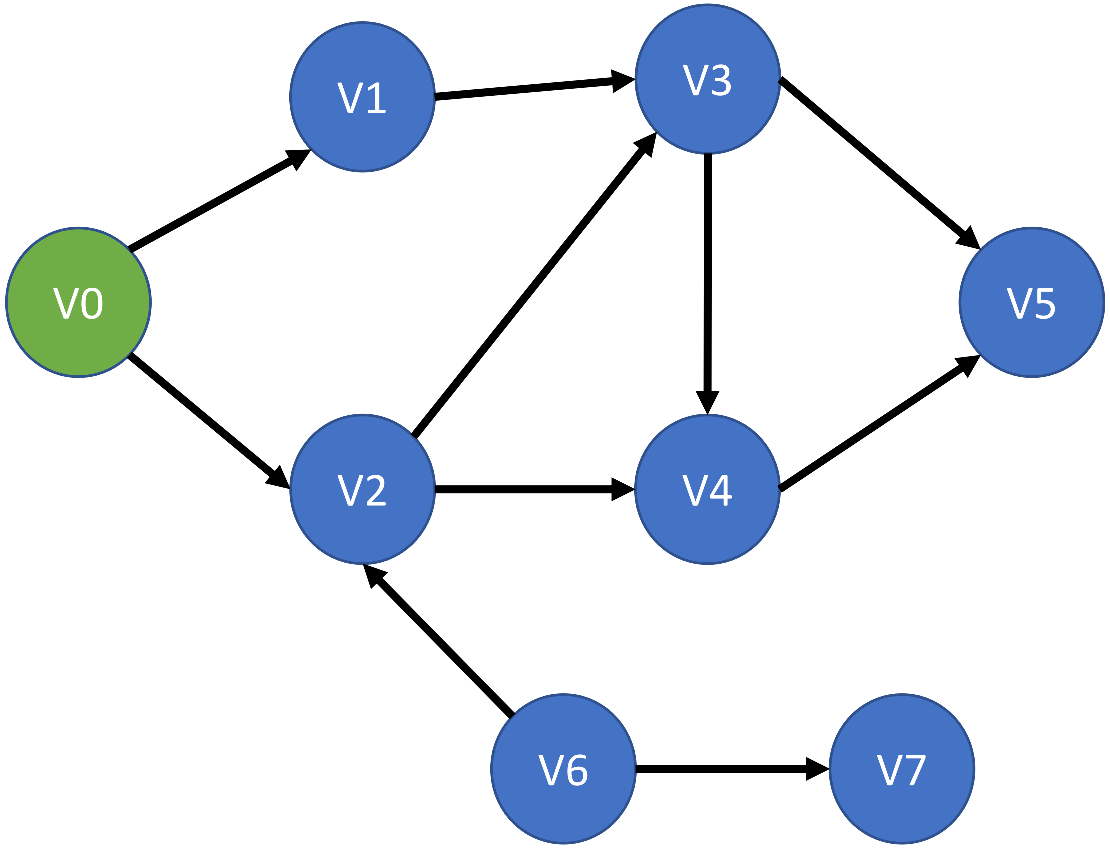
Breadth-First Search (BFS)



1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
    - Remove the 1st node from the queue, u
    - For each neighbour v of u:
        * If v is not explored:
            ** Mark v as explored
            ** Put v at the end of the queue
4. Repeat from another starting node, if there is one

Algorithms

Breadth-First Search (BFS)

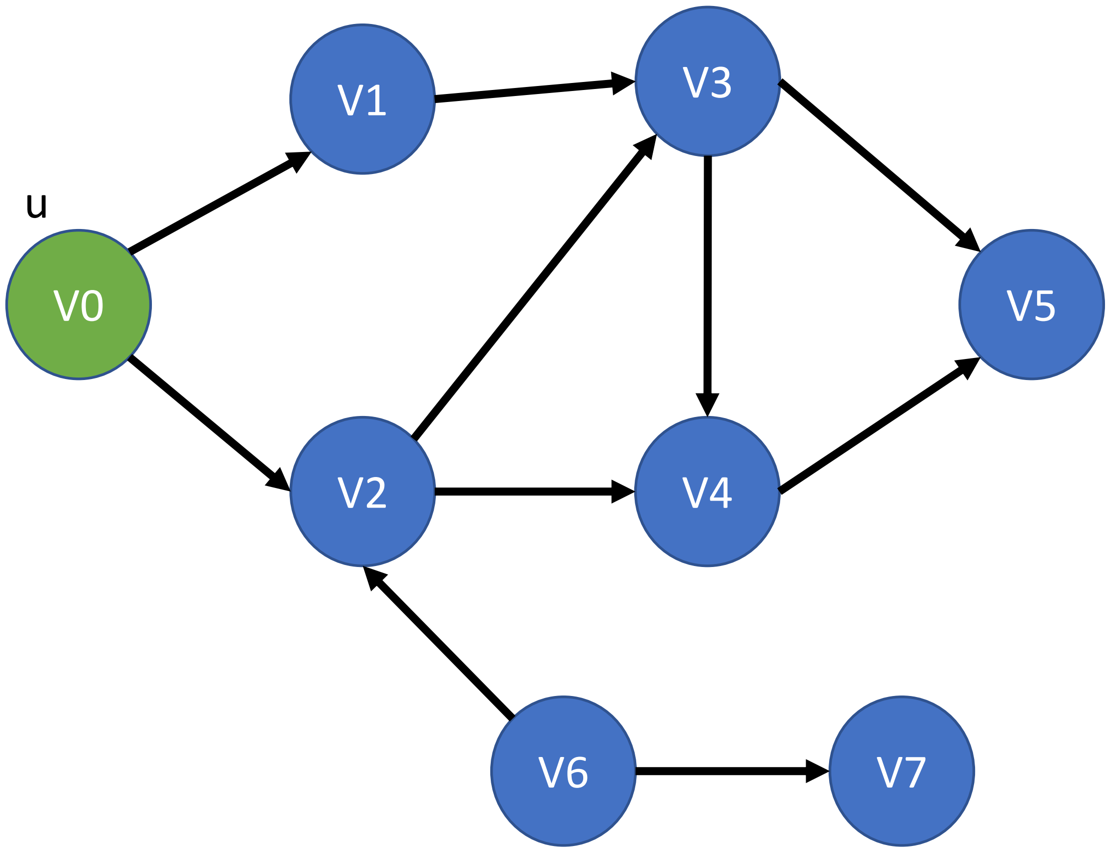


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
    - Remove the 1st node from the queue, u
    - For each neighbour v of u:
        * If v is not explored:
            ** Mark v as explored
            ** Put v at the end of the queue
4. Repeat from another starting node, if there is one

V0

Algorithms

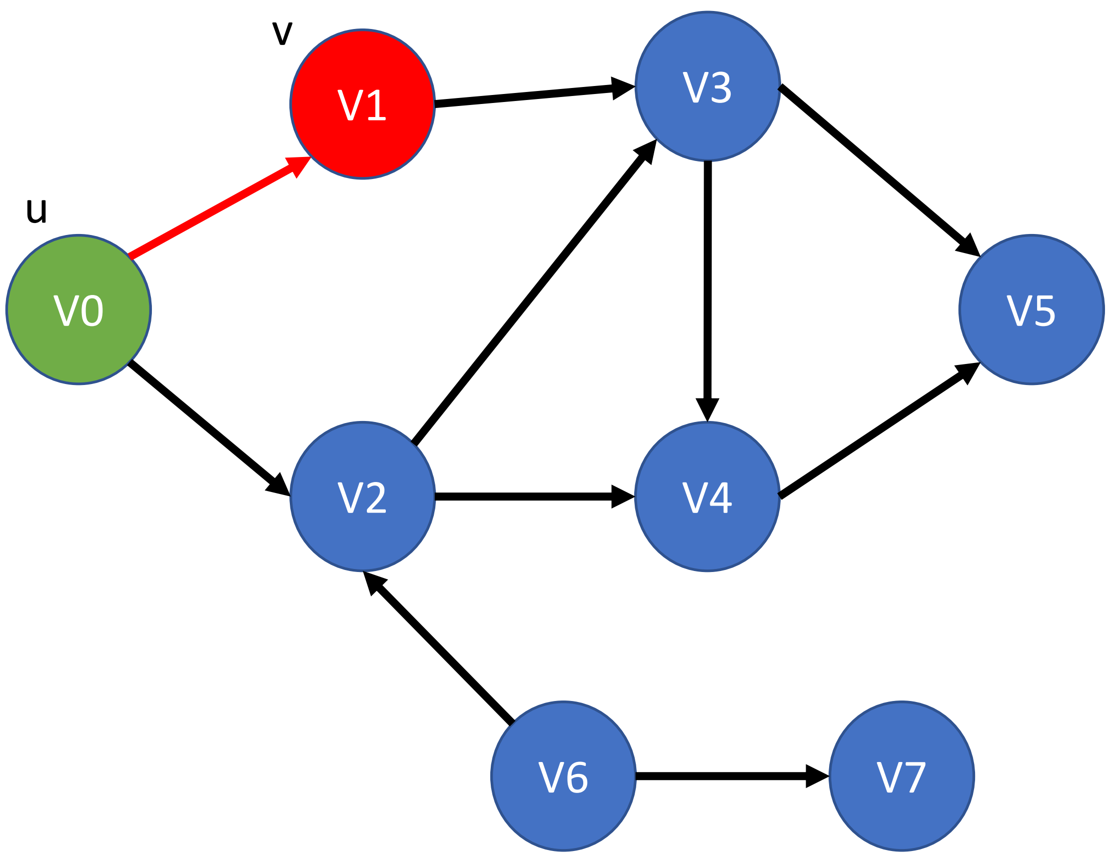
Breadth-First Search (BFS)



1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

Algorithms

Breadth-First Search (BFS)

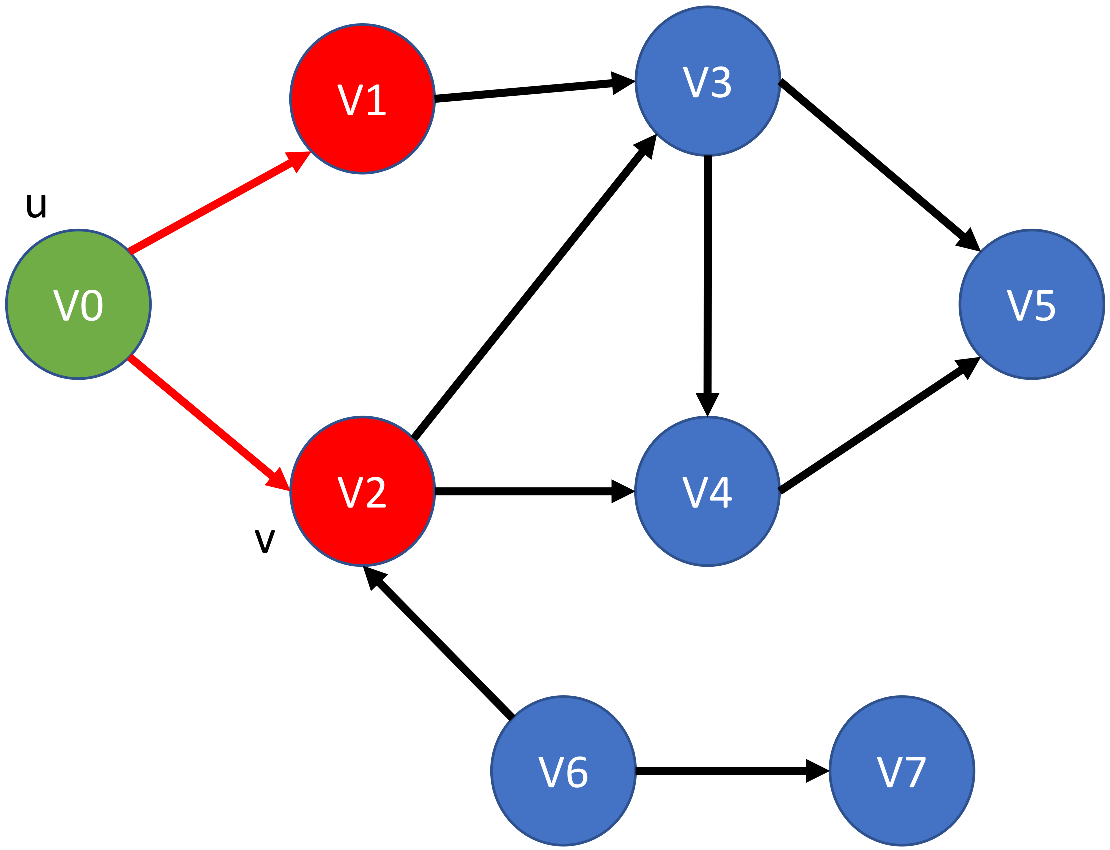


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V1

Algorithms

Breadth-First Search (BFS)

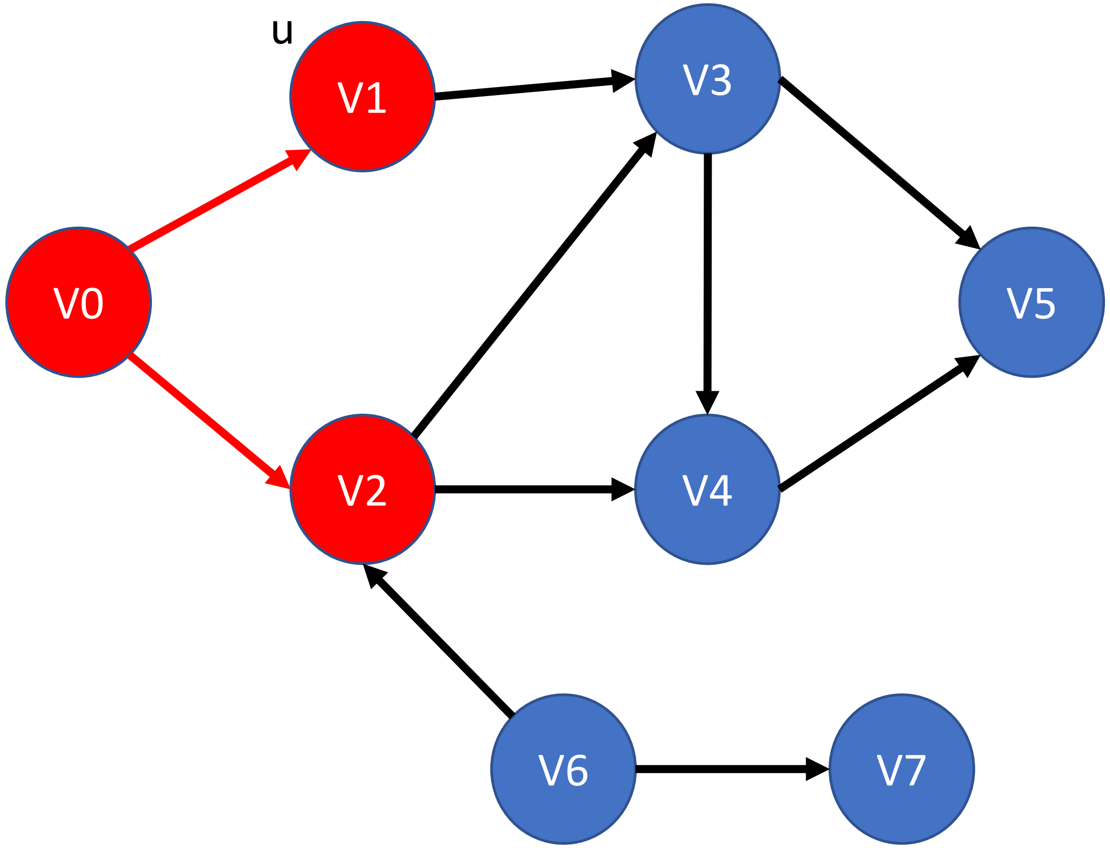


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V1	V2

Algorithms

Breadth-First Search (BFS)

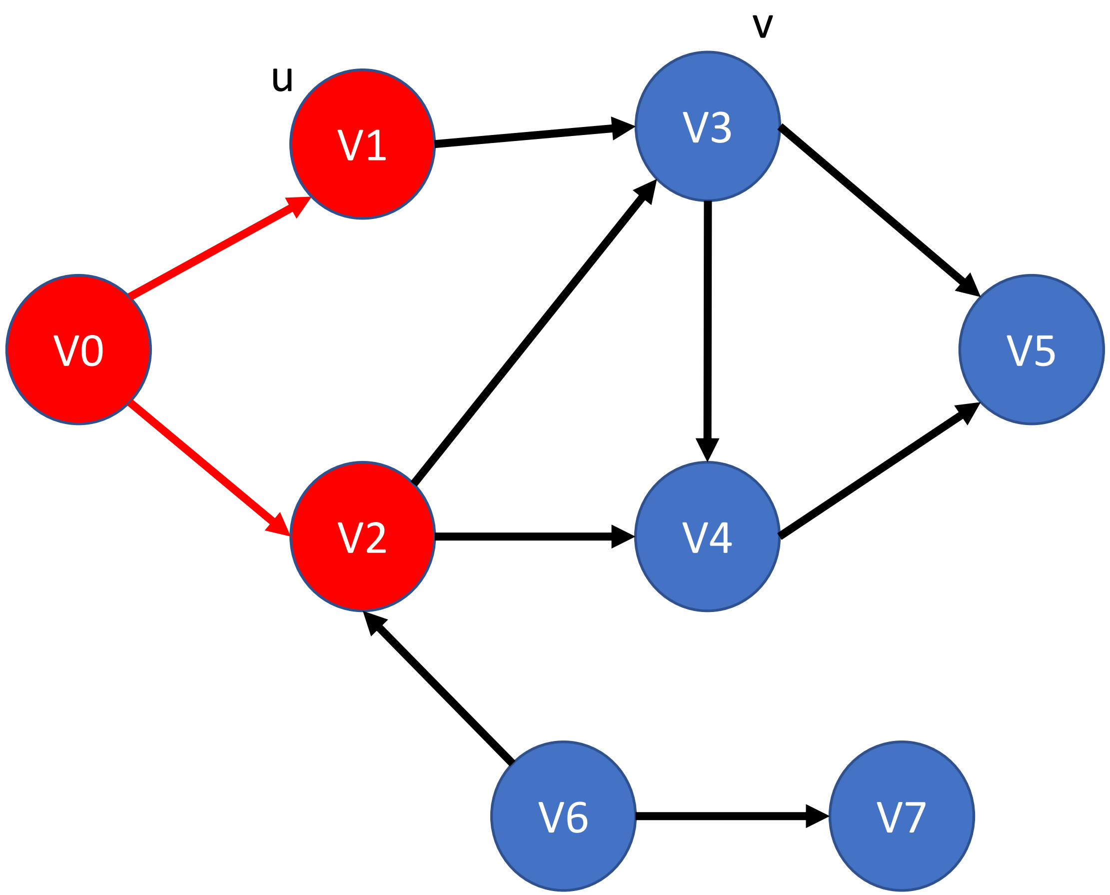


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V2

Algorithms

Breadth-First Search (BFS)

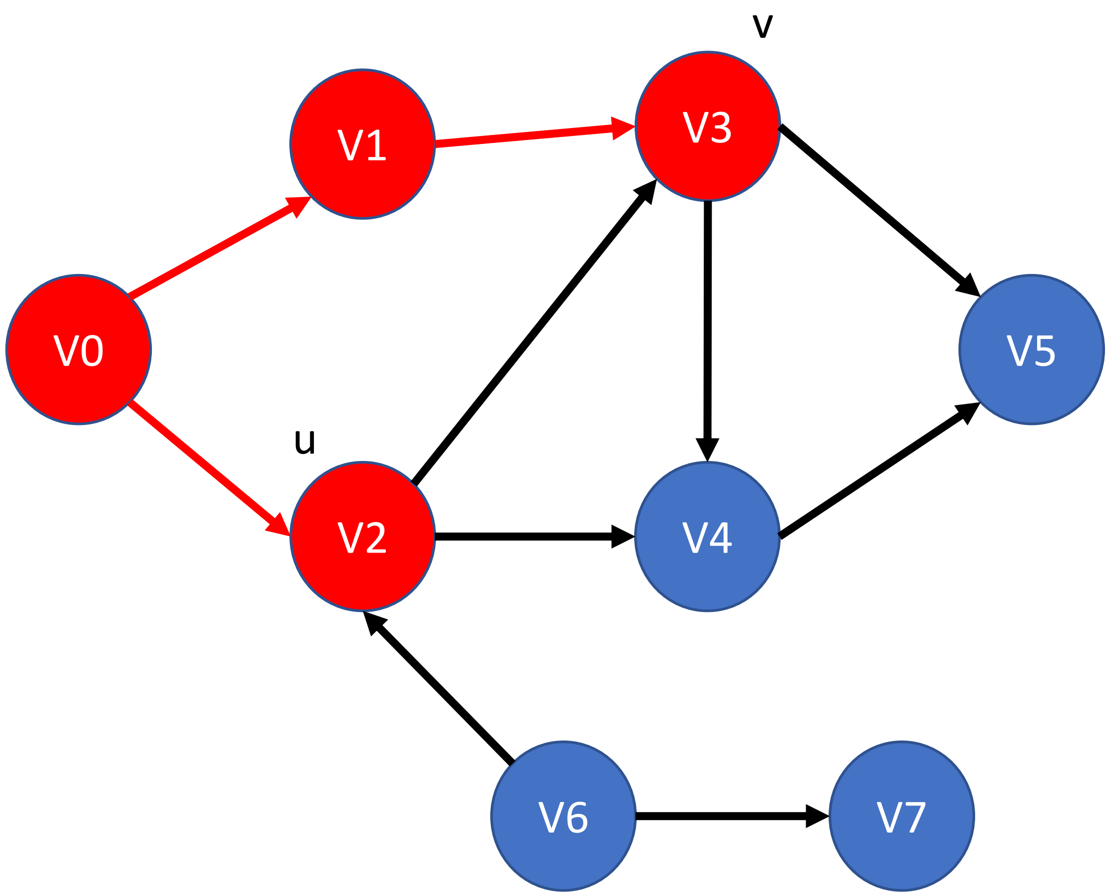


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V2	V3

Algorithms

Breadth-First Search (BFS)

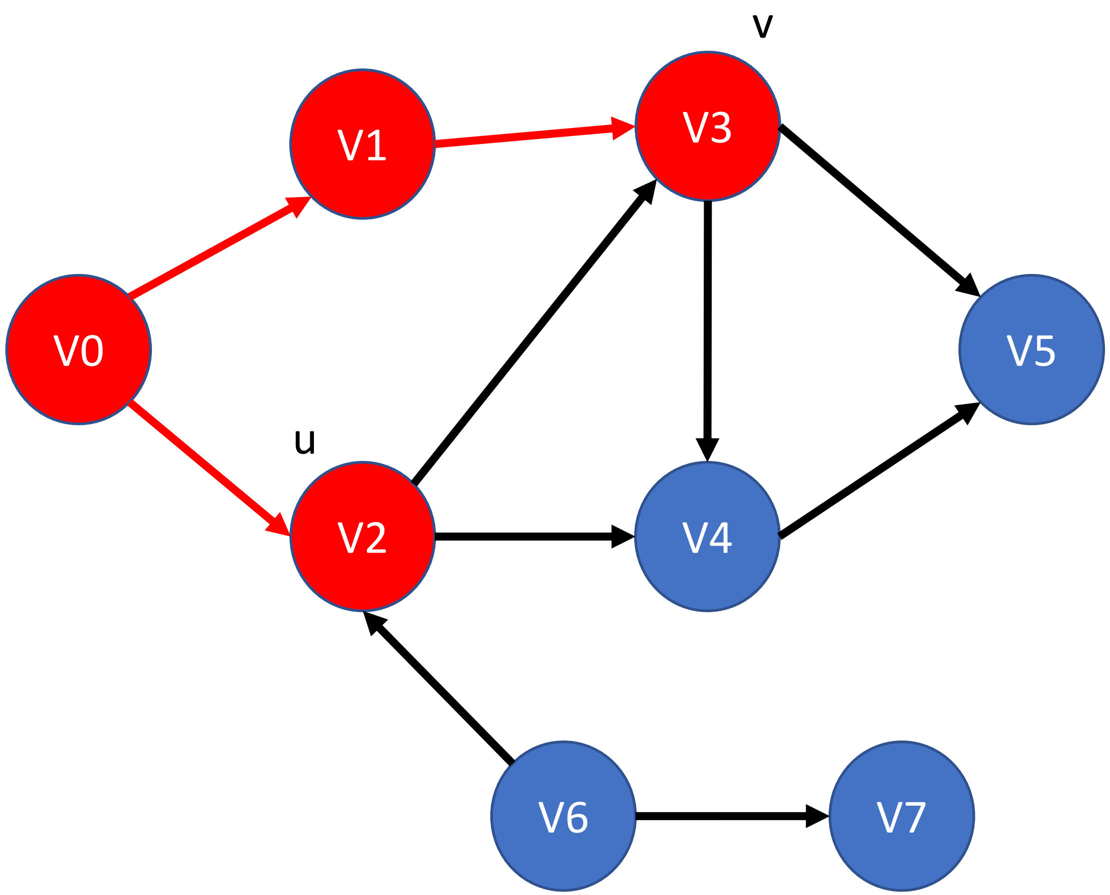


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V2	V3

Algorithms

Breadth-First Search (BFS)

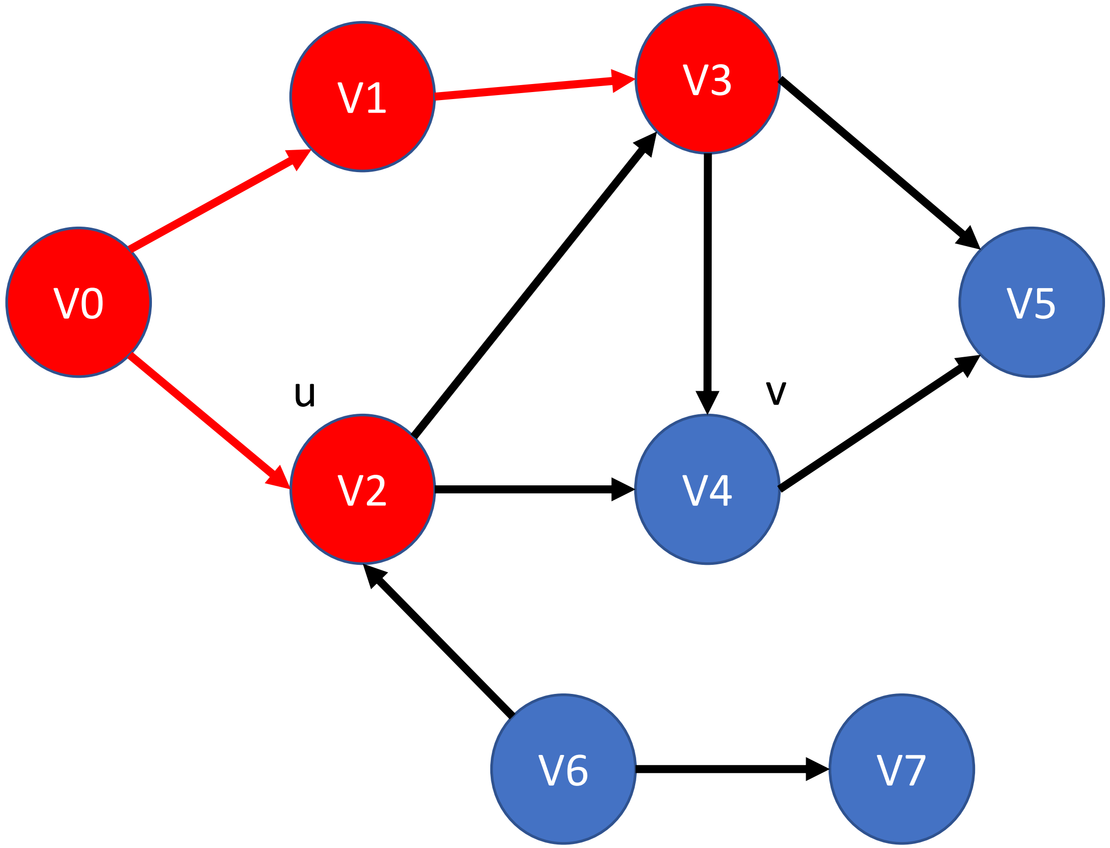


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V3

Algorithms

Breadth-First Search (BFS)

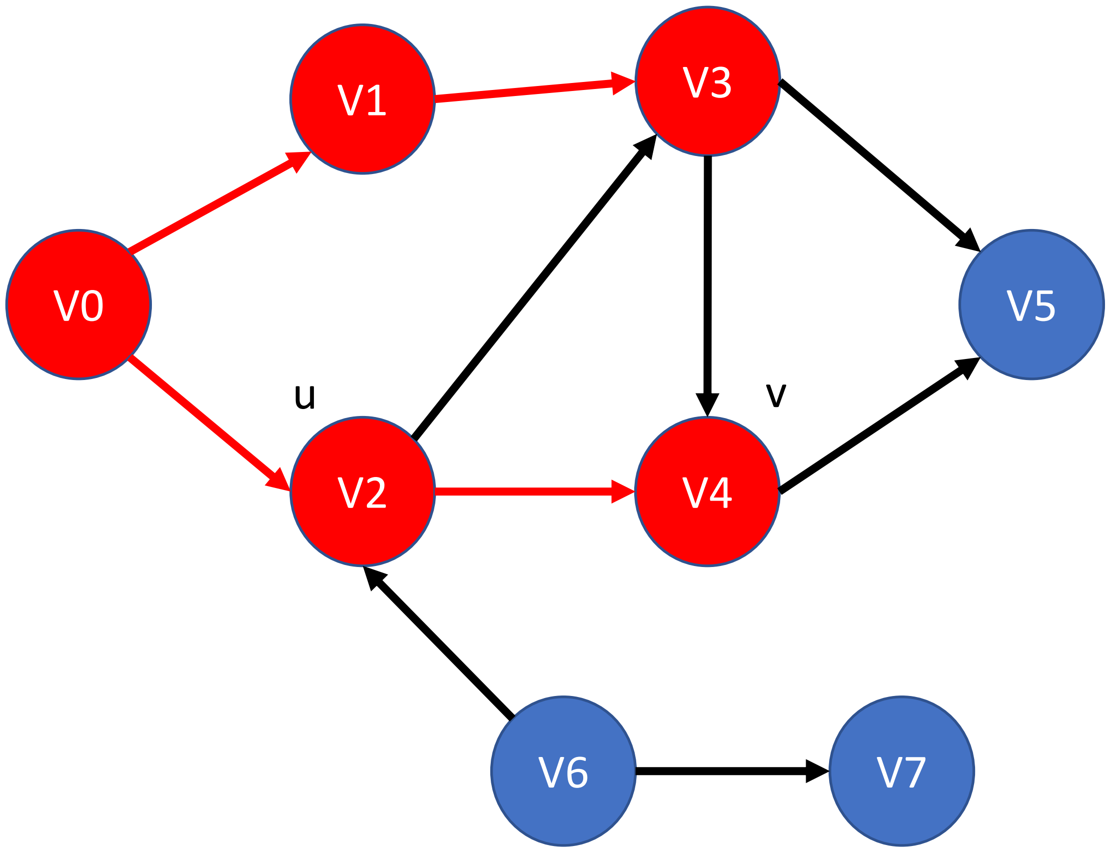


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V3	V4

Algorithms

Breadth-First Search (BFS)

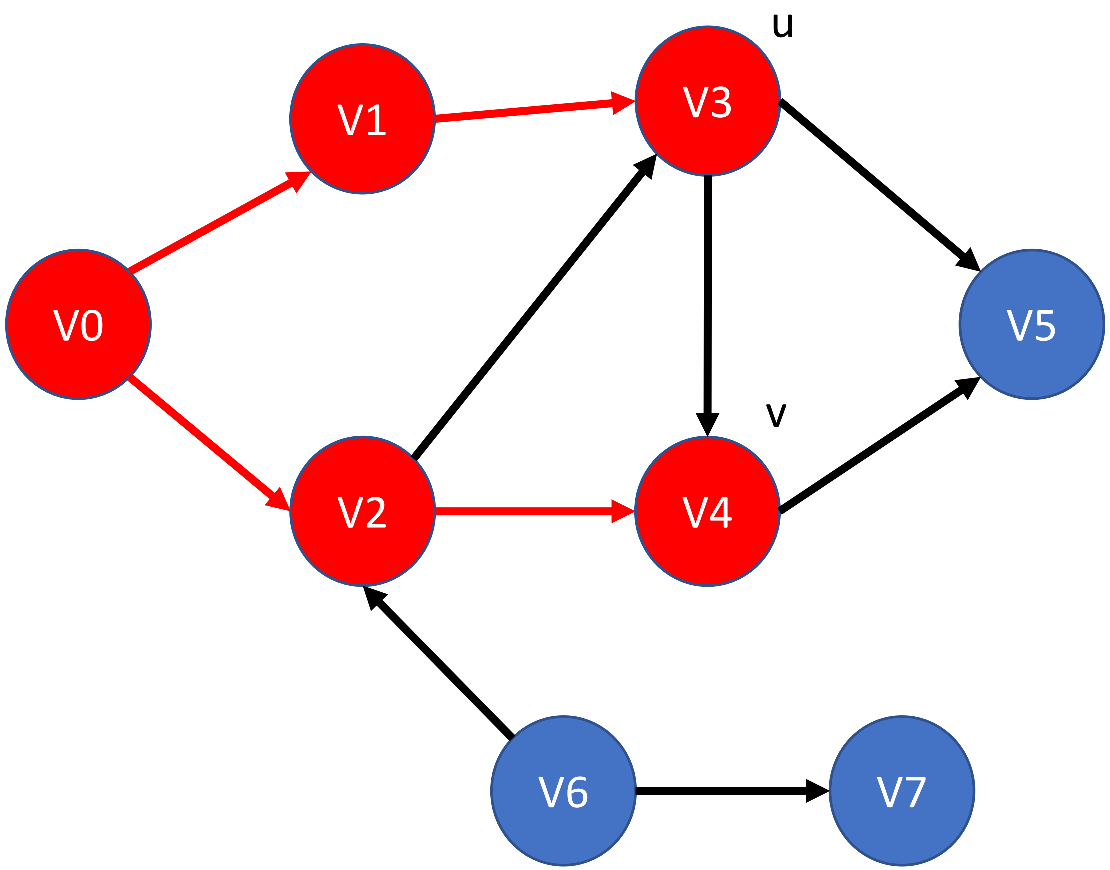


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V3	V4

Algorithms

Breadth-First Search (BFS)

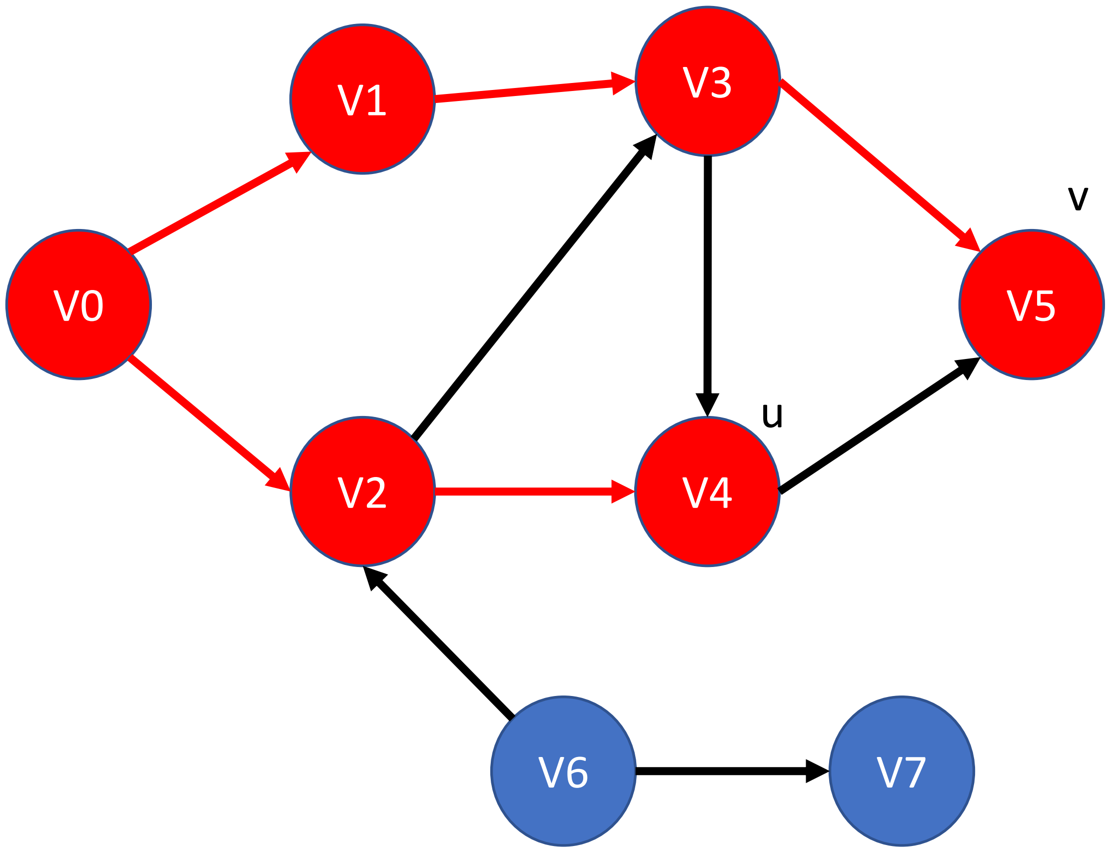


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V4

Algorithms

Breadth-First Search (BFS)

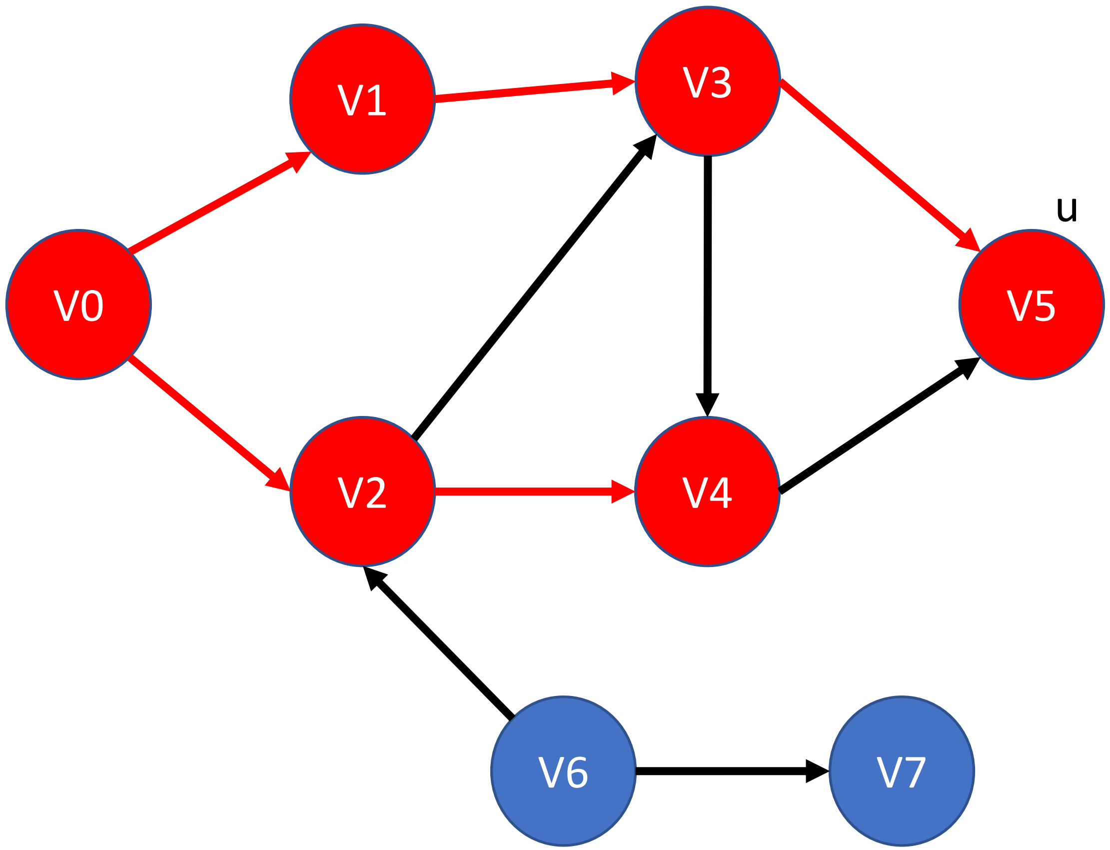


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V5

Algorithms

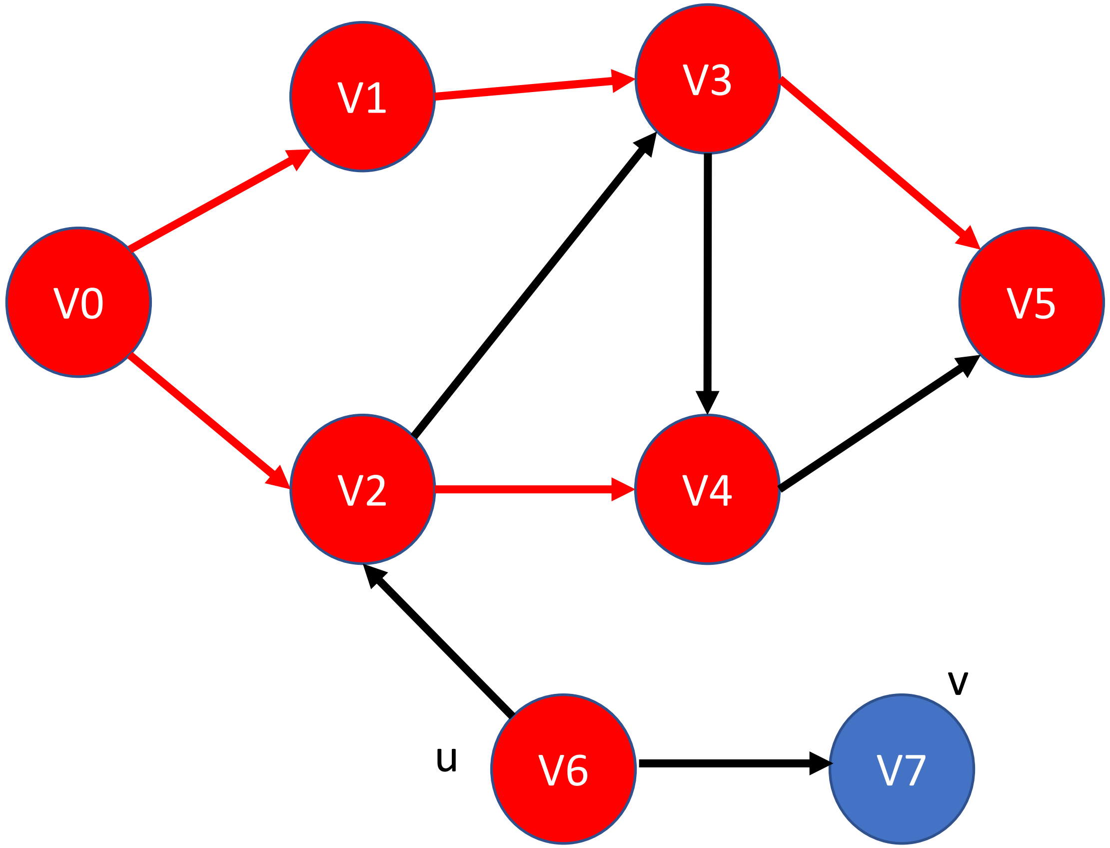
Breadth-First Search (BFS)



1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
    - Remove the 1st node from the list, u
    - For each neighbour v of u:
        * If v is not explored:
            ** Mark v as explored
            ** Put v at the end of the list
4. Repeat from another starting node, if there is one

Algorithms

Breadth-First Search (BFS)

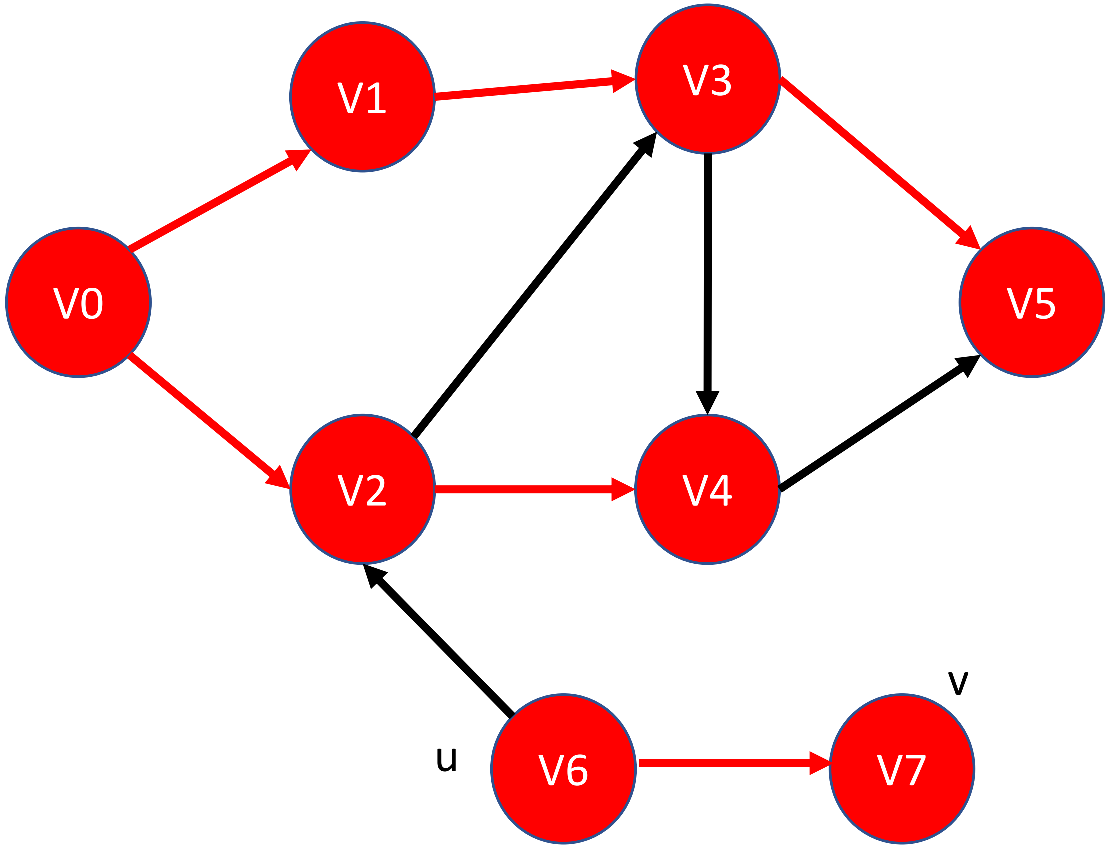


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V6

Algorithms

Breadth-First Search (BFS)

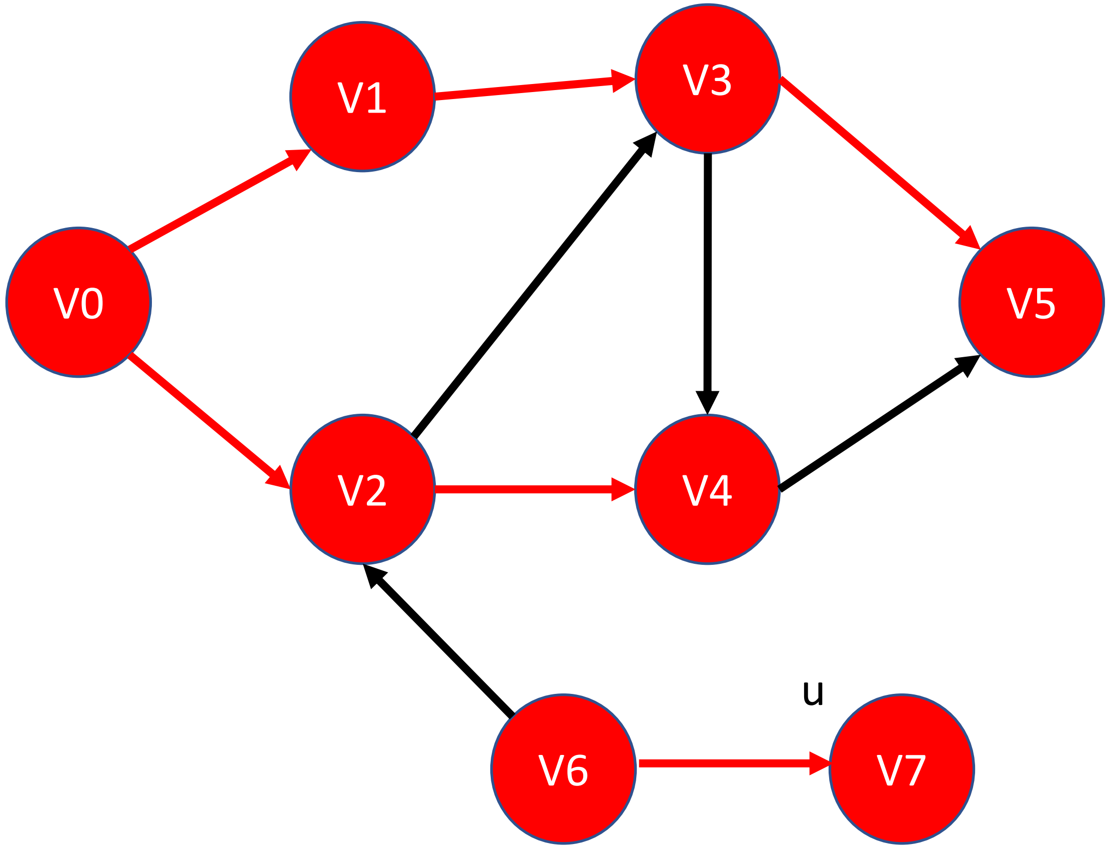


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V6	V7

Algorithms

Breadth-First Search (BFS)

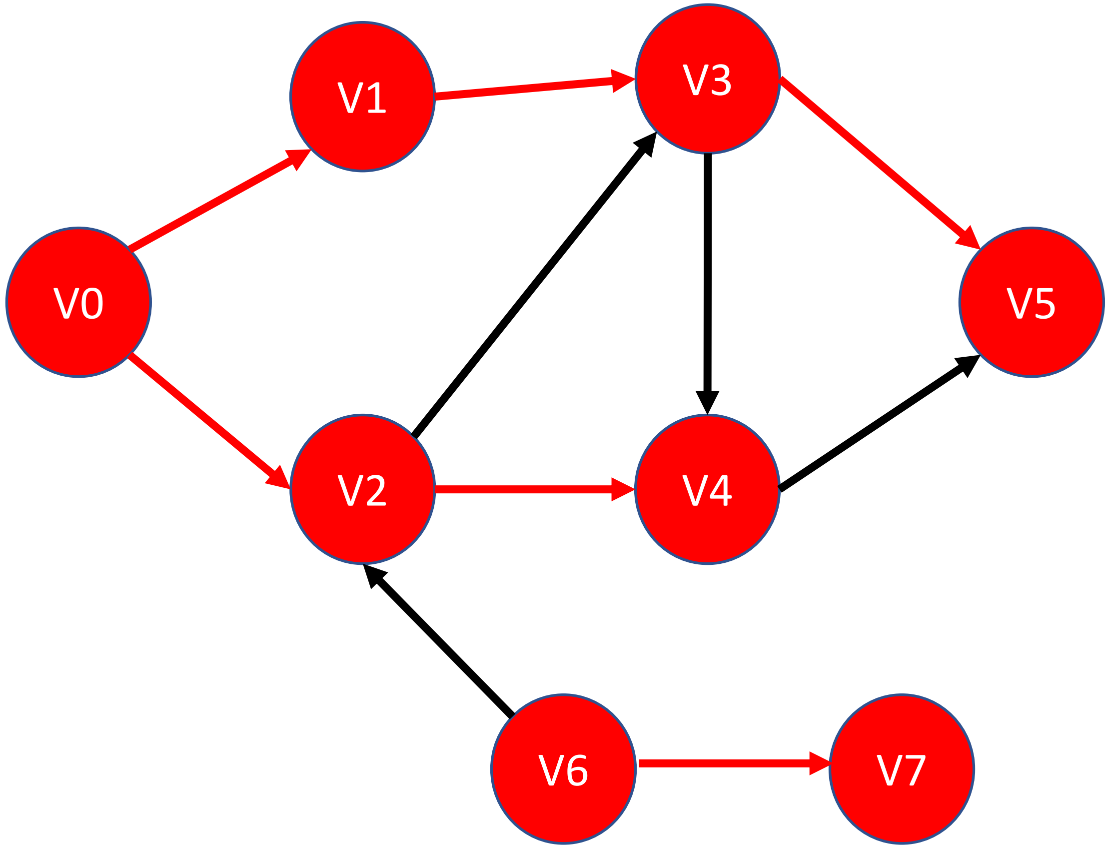


1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

V7

Algorithms

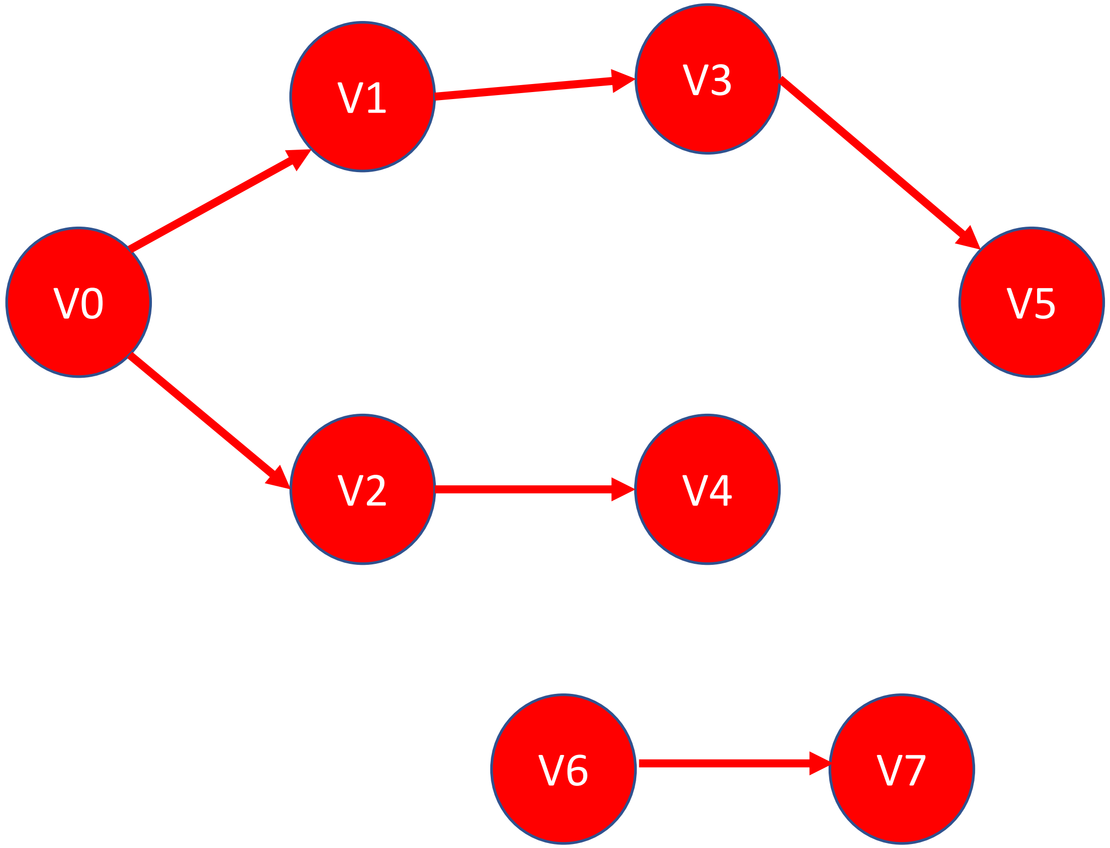
Breadth-First Search (BFS)



1. Define an initial node, marking it as explored
2. Put it on the list
3. As long as the queue is not empty:
   - Remove the 1st node from the list, u
   - For each neighbour v of u:
     * If v is not explored:
       ** Mark v as explored
       ** Put v at the end of the list
4. Repeat from another starting node, if there is one

Algorithms

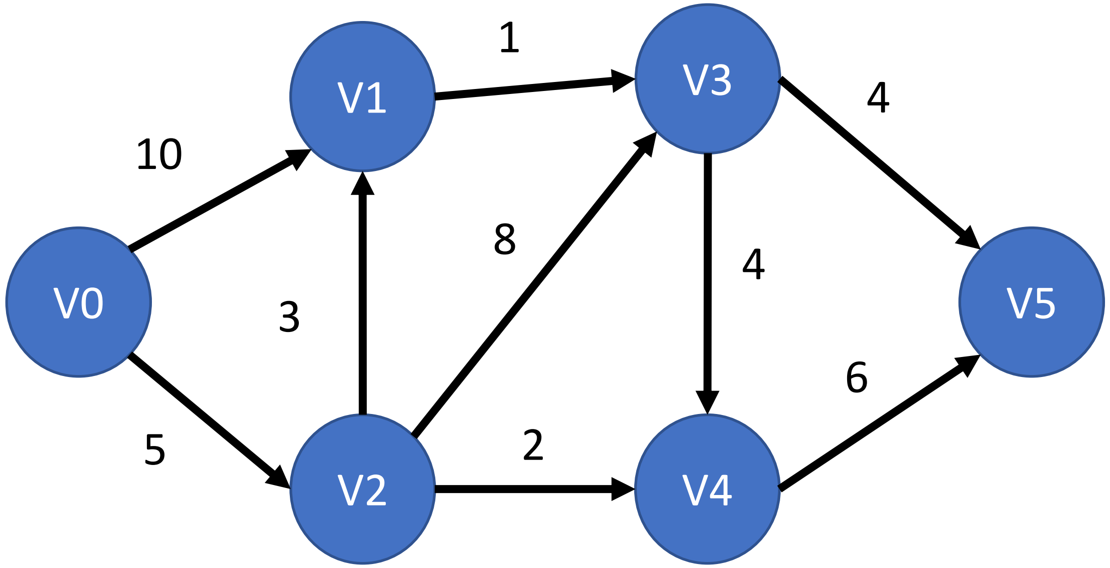
Breadth-First Search (BFS)



Two Trees or Forest

Algorithms

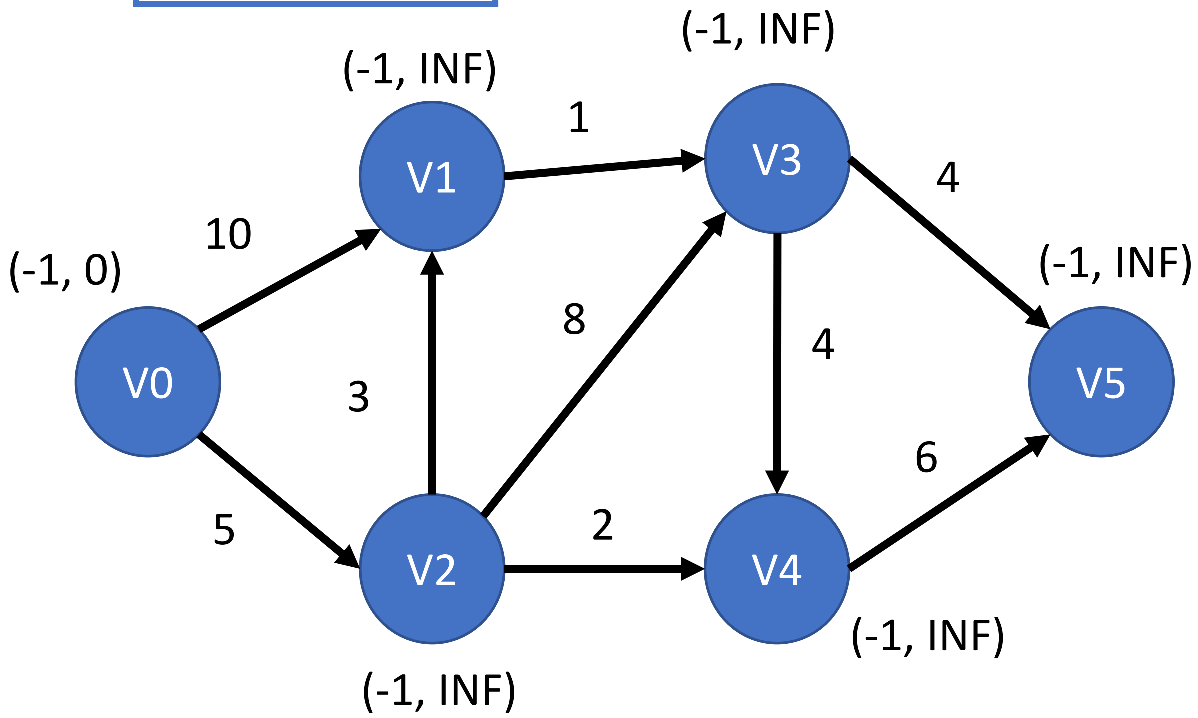
Dijkstra



1. Initialize the graph with d(s) = 0, d(v) = INF, for all v ≠ s, and p(v) = -1 for all *v*
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose *u* whose estimate is the smallest among the open
   * Close *u*
   * For every open node *v* adjacent to *u*: relax edge (*u,v*)

Algorithms

Dijkstra

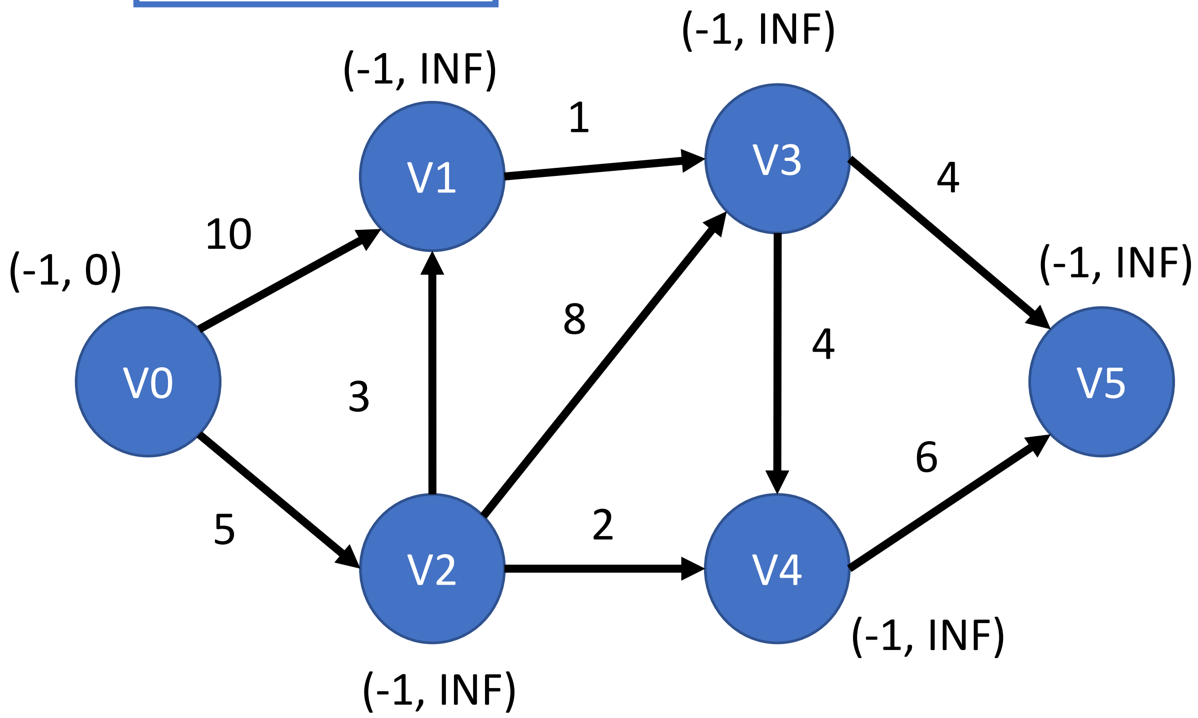


NOTATION ➡ (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose <u>*u*</u> whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

Algorithms

Dijkstra

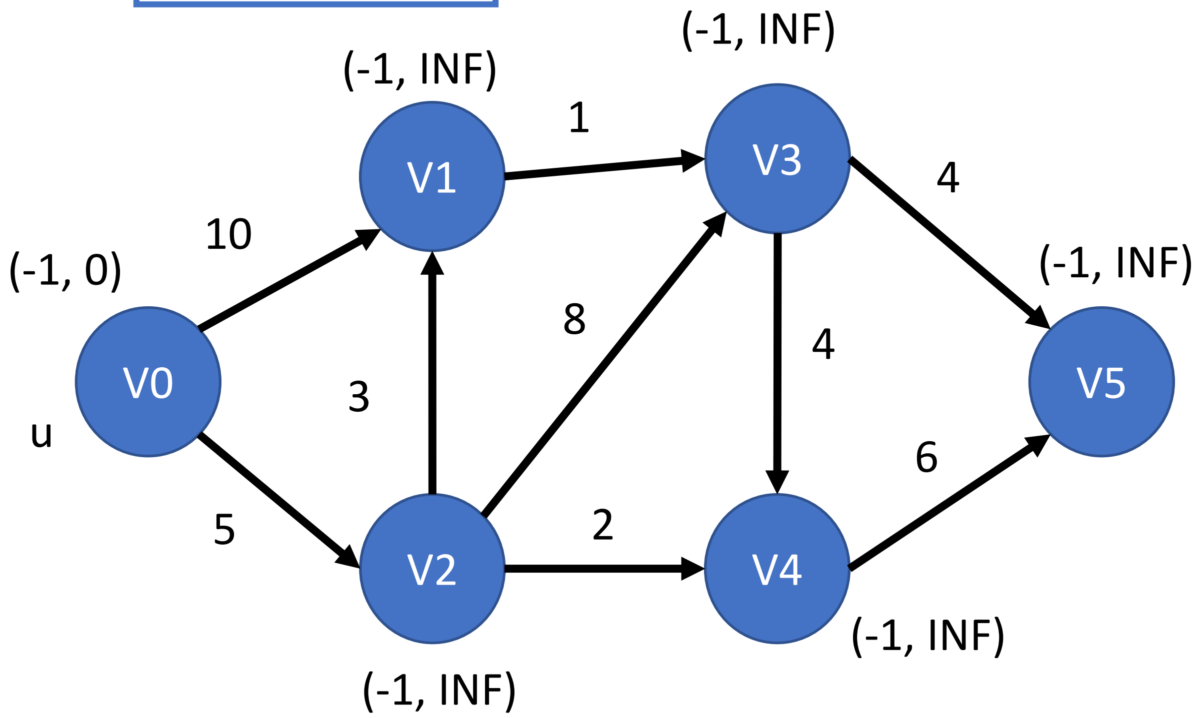


NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>v</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose *u* whose estimate is the smallest among the open
   * Close *u*
   * For every open node *v* adjacent to *u*: relax edge (*u,v*)

Algorithms

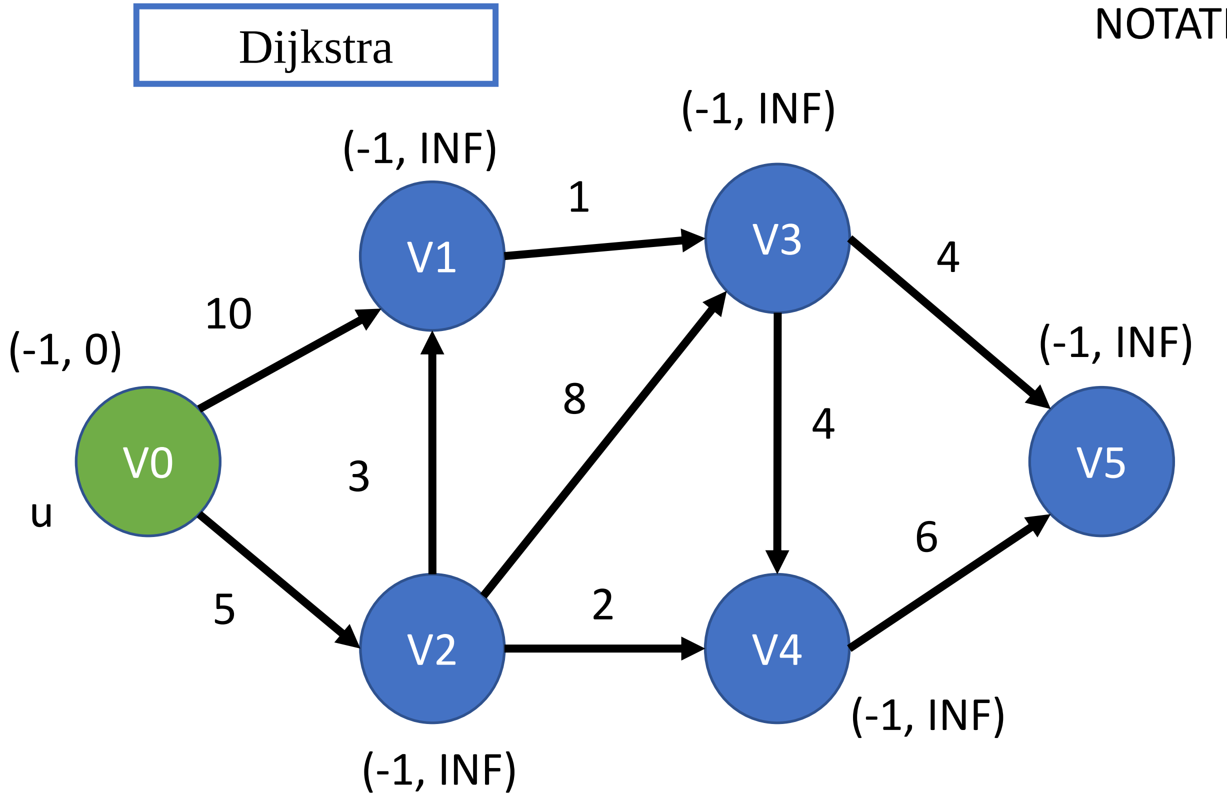
Dijkstra



NOTATION ➡ (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

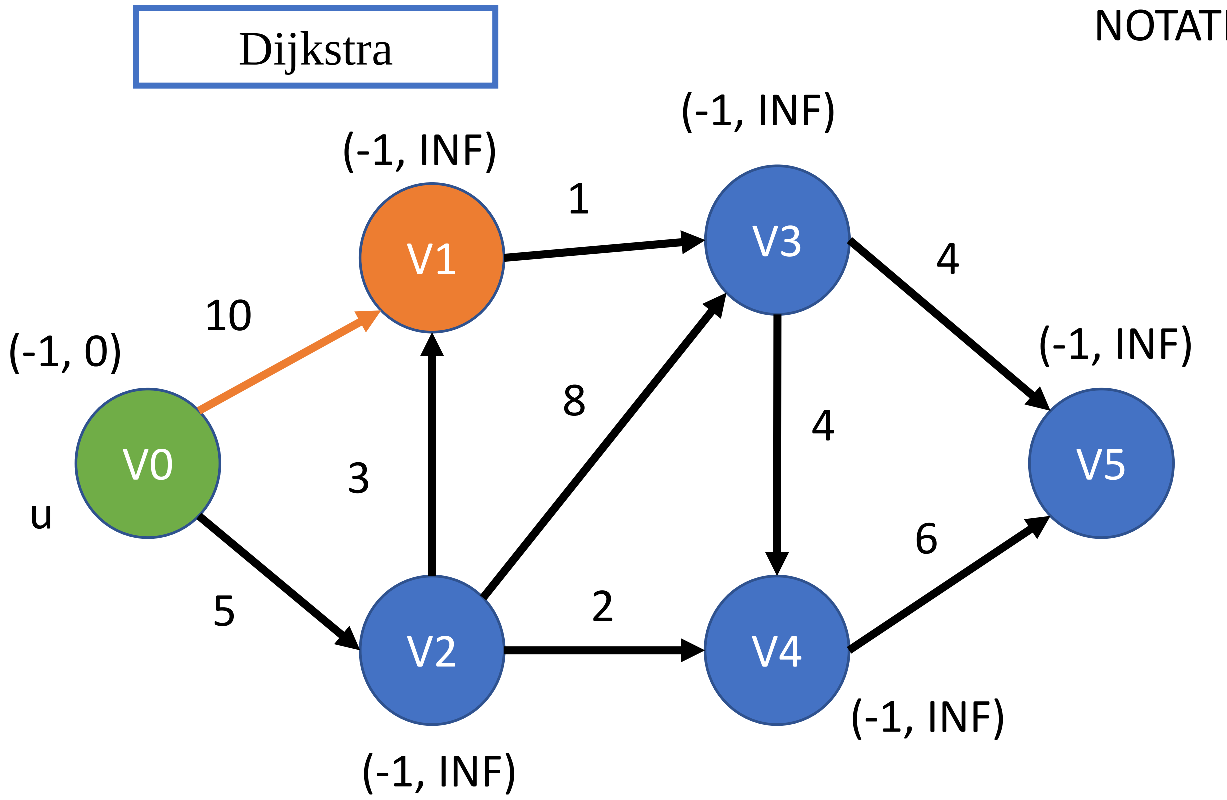
Algorithms



NOTATION ➡ (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

Algorithms

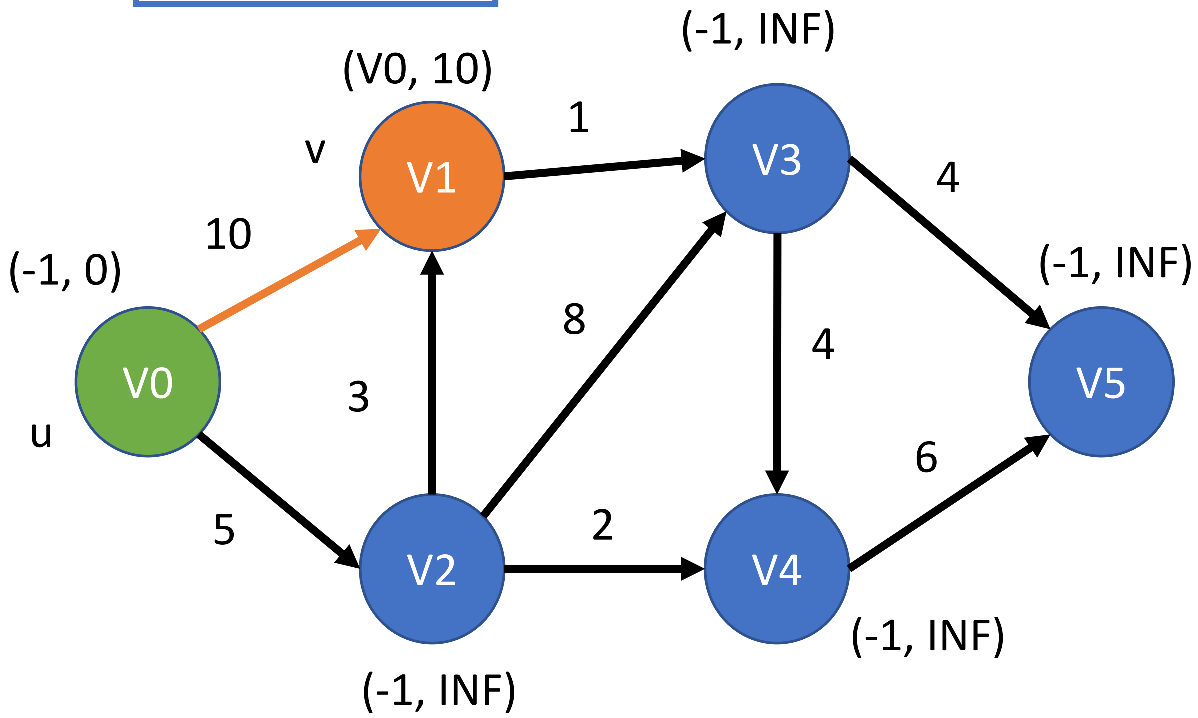


NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all v ≠ s, and p(v) = -1 for all v
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close u
   * For every open node v adjacent to u: relax edge (u,v)

Algorithms

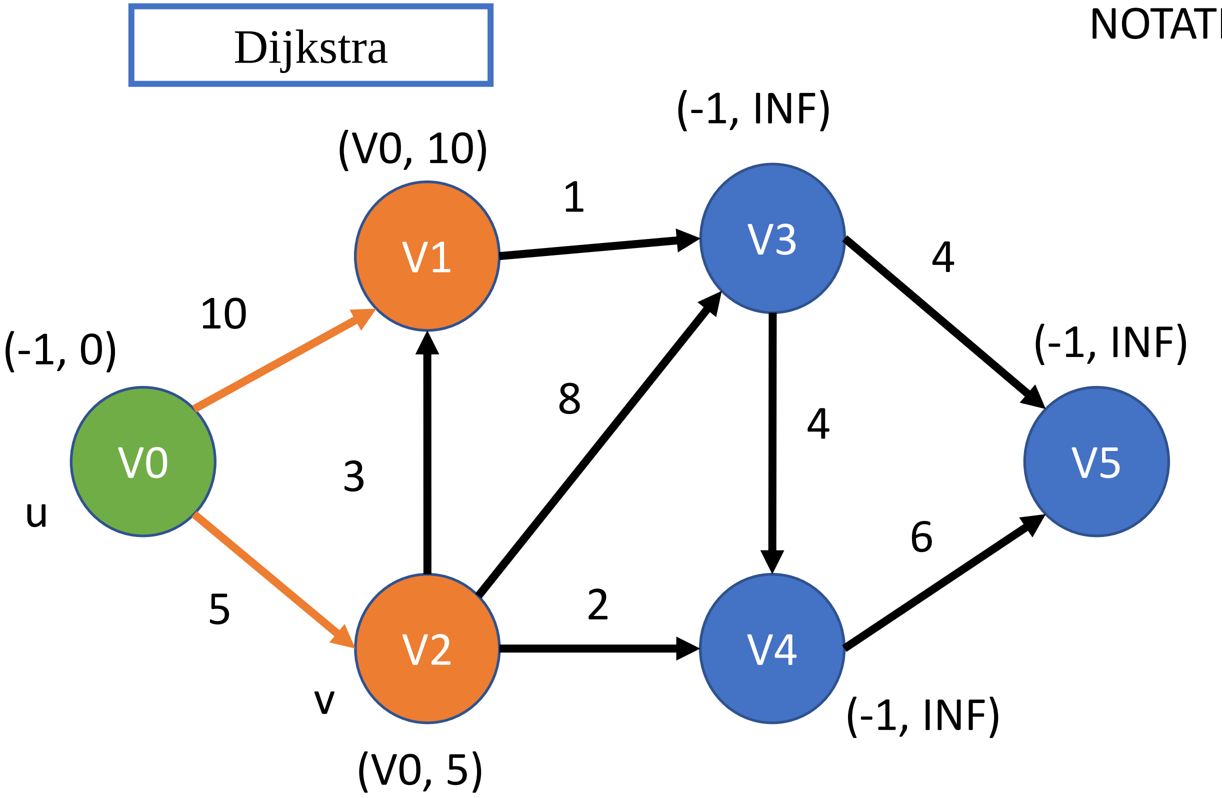
Dijkstra



NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>v</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>u</u>
   * For every open node <u>v</u> adjacent to <u>u</u>: relax edge (<u>u,v</u>)

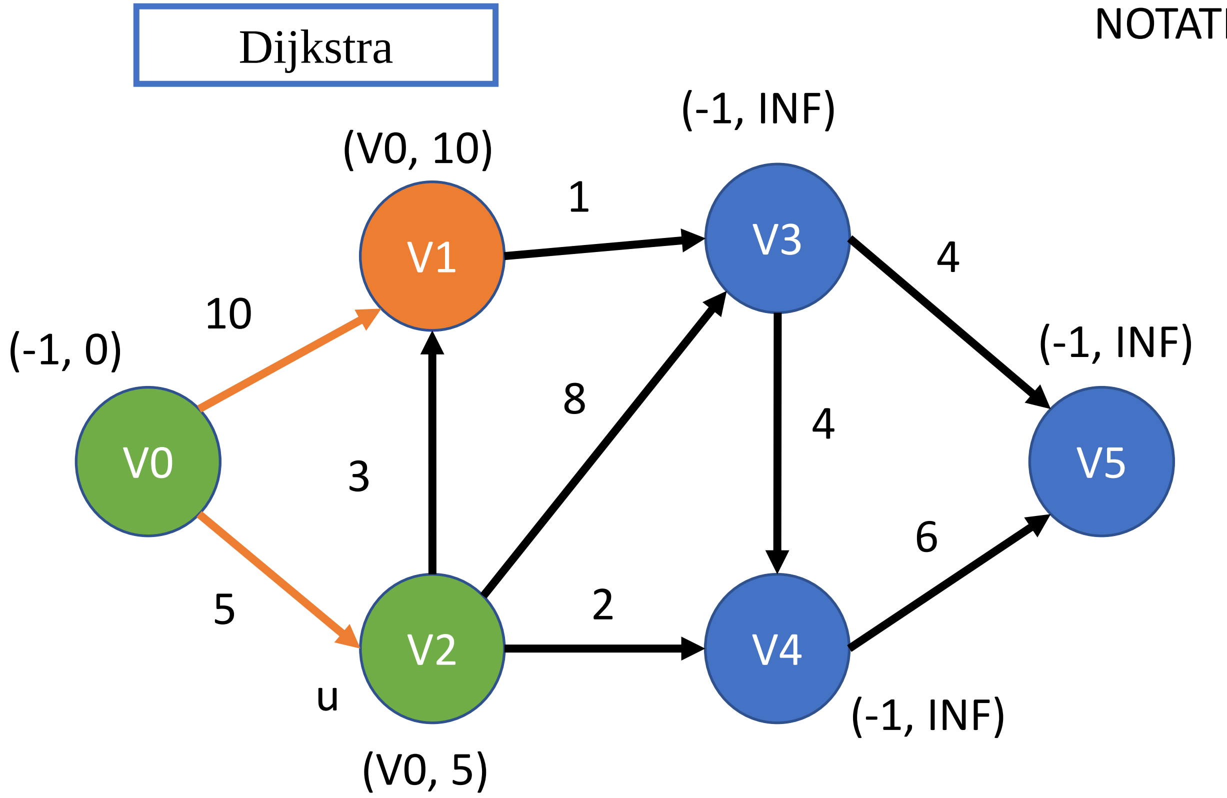
Algorithms



NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all v ≠ s, and p(v) = -1 for all *v*
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close *u*
   * For every open node *v* adjacent to *u*: relax edge (*u,v*)

Algorithms

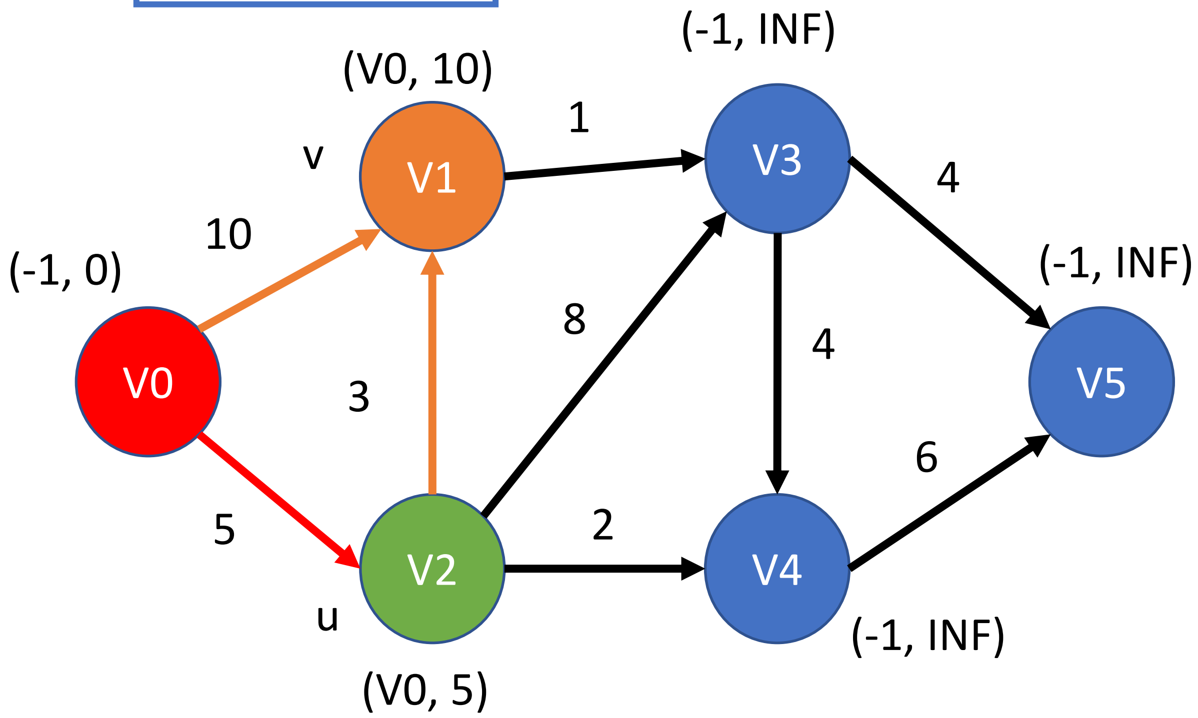


NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all v ≠ s, and p(v) = -1 for all *v*
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close *u*
   * For every open node *v* adjacent to *u*: relax edge (*u,v*)

Algorithms

Dijkstra

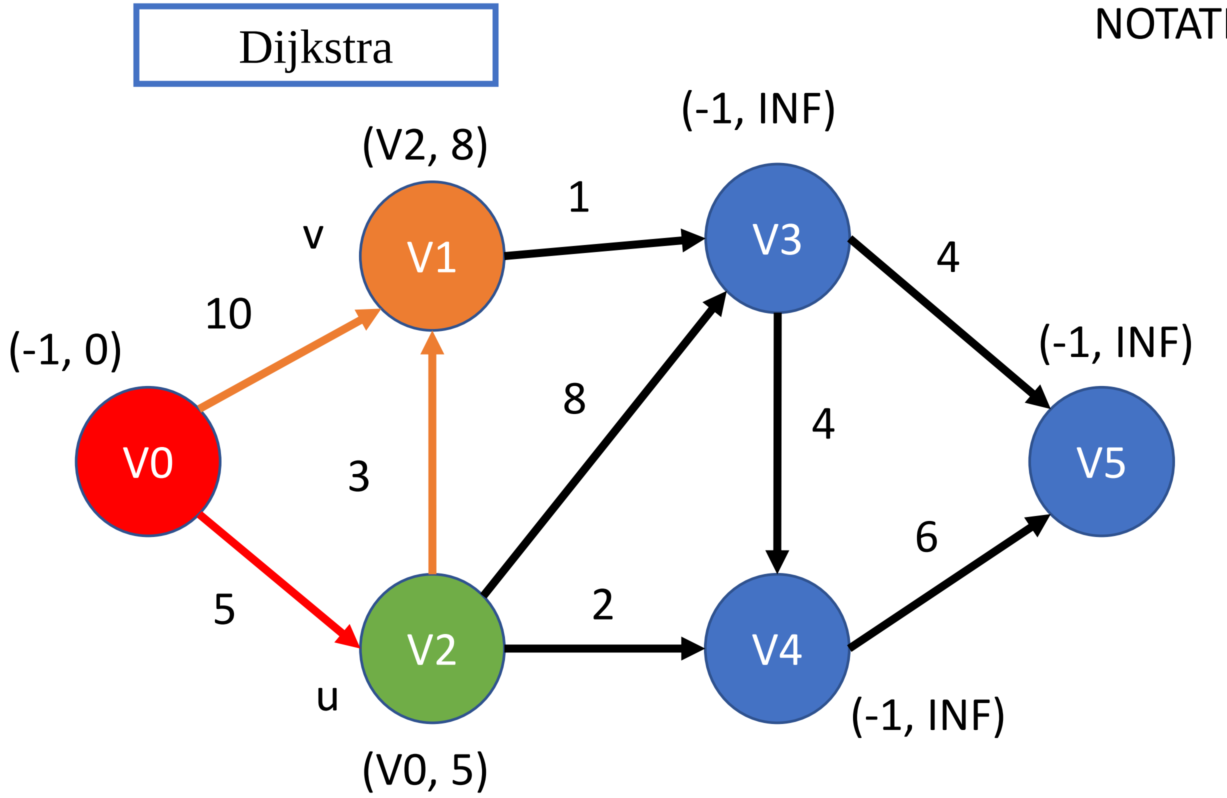


NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>v</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

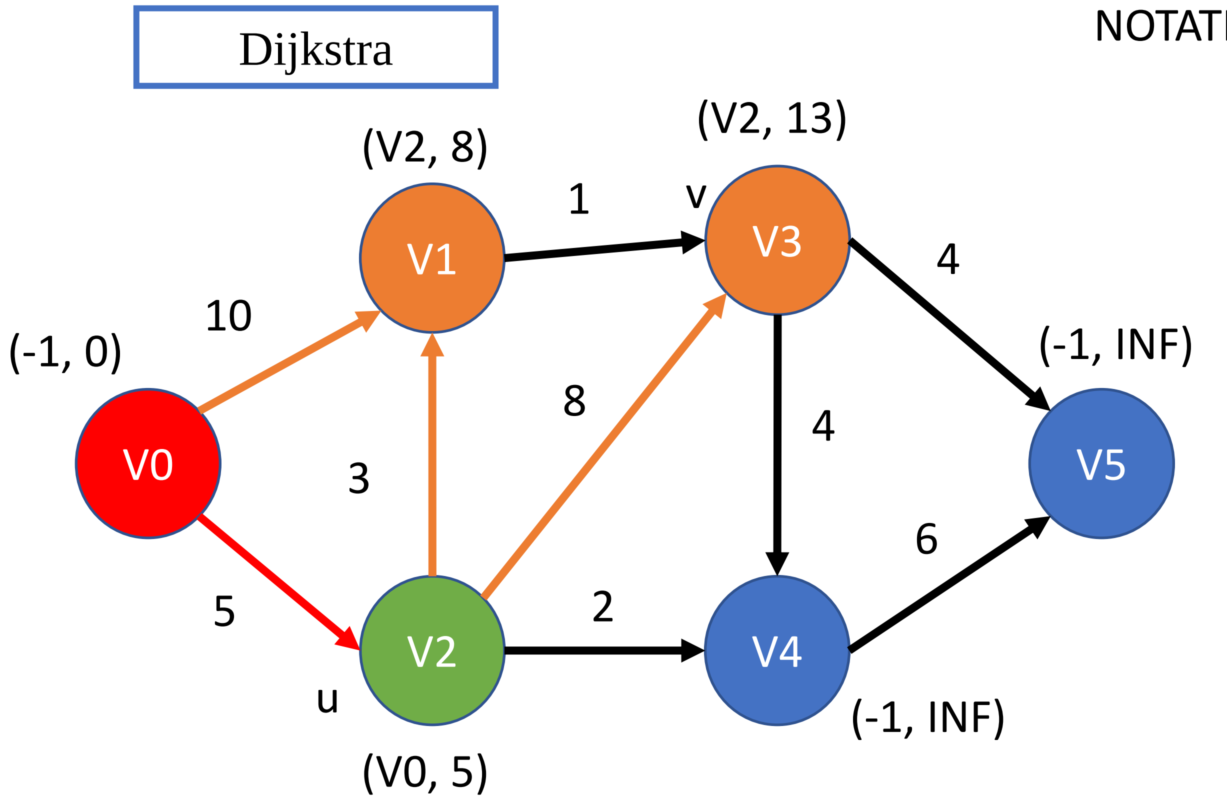
Algorithms

NOTATION → (father node, weigth)



1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

Algorithms

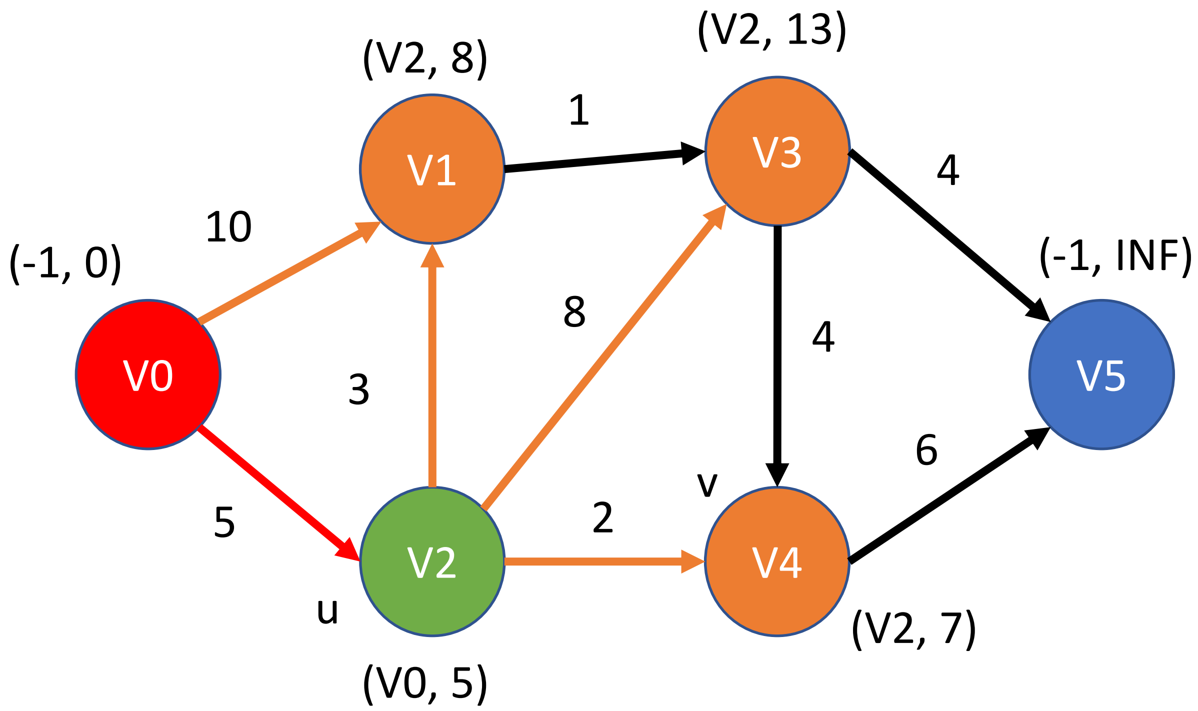


NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>v</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

Algorithms

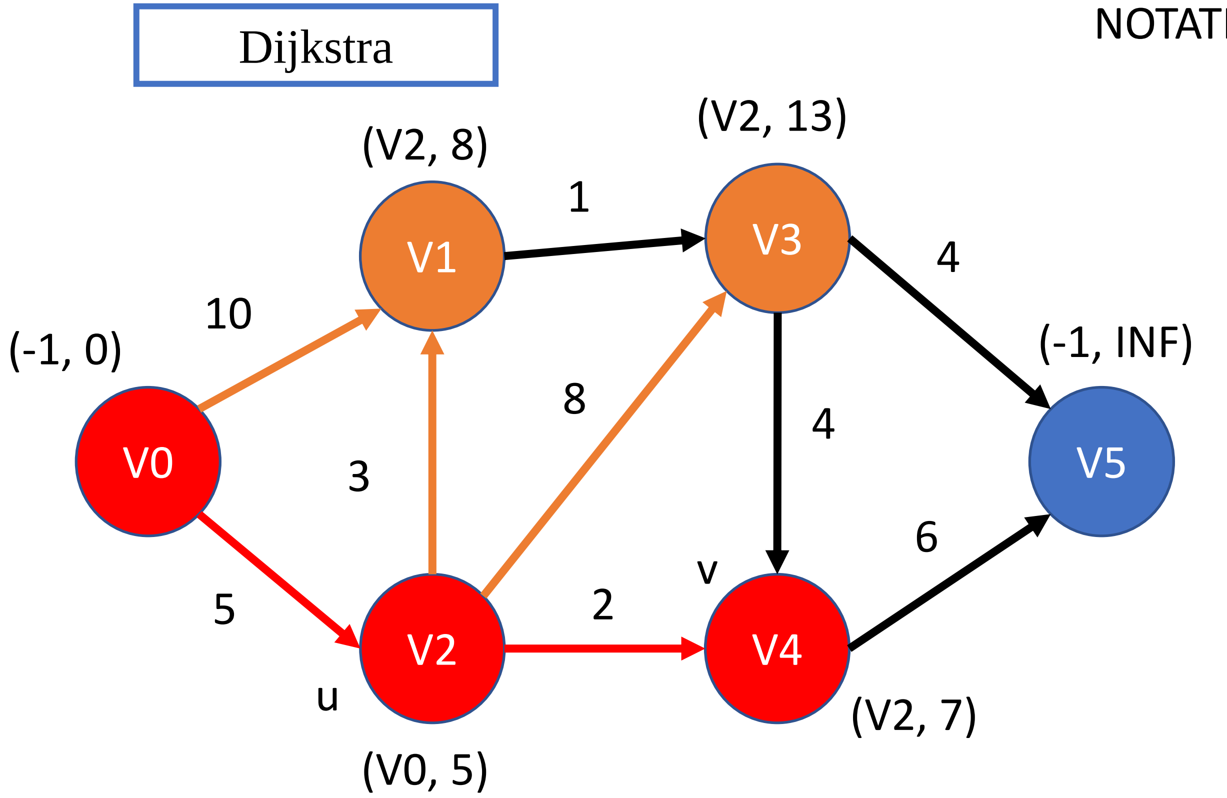
Dijkstra



NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

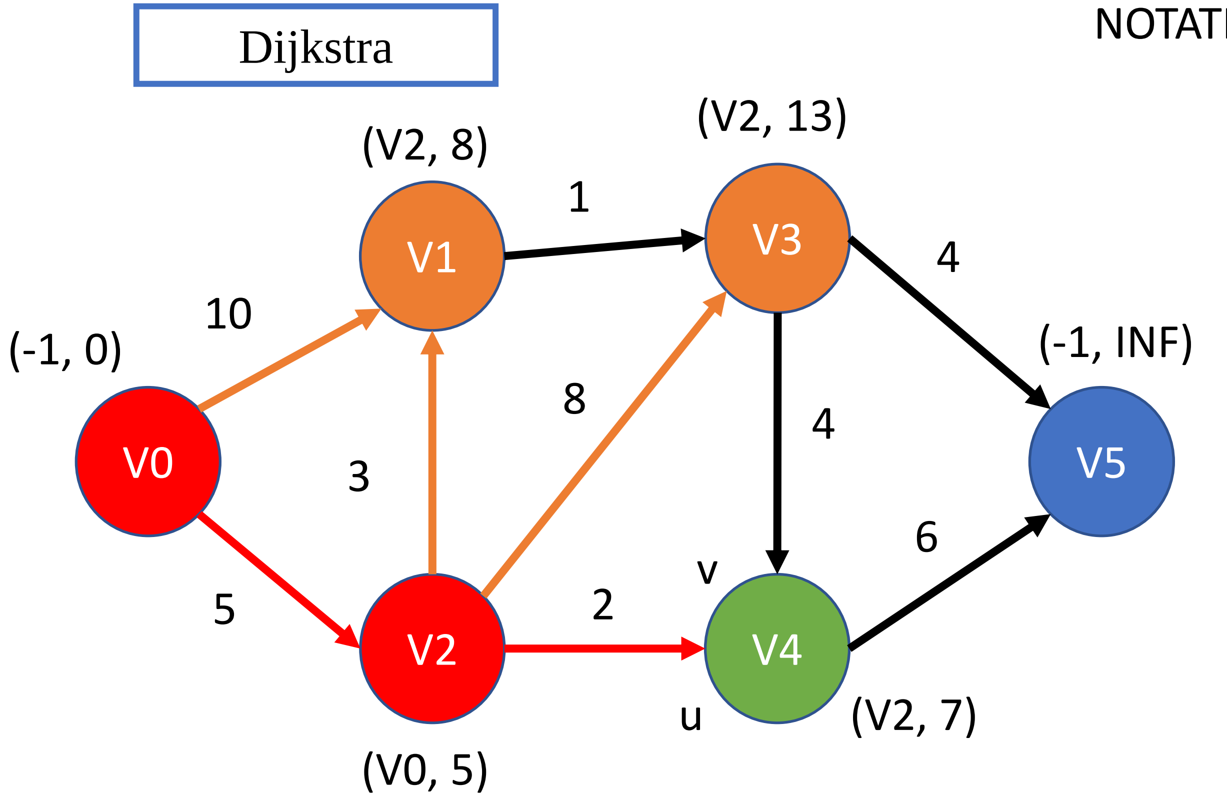
Algorithms



NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

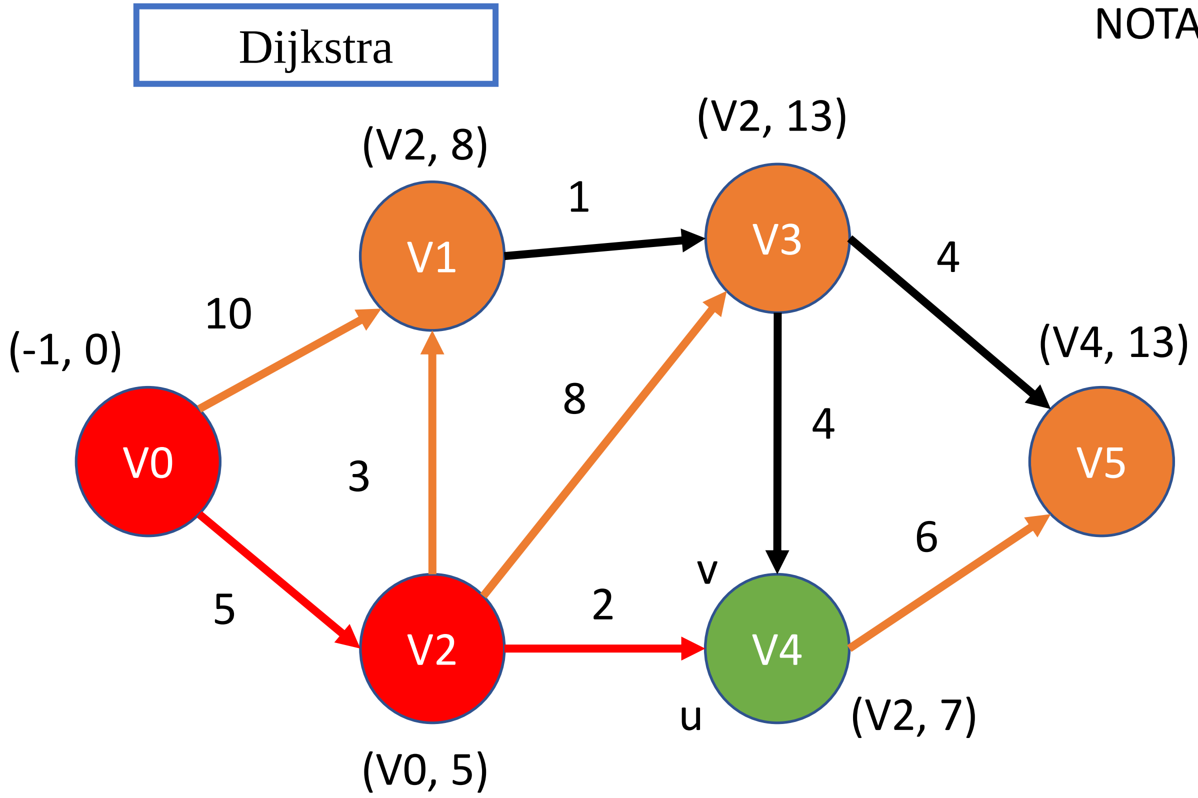
Algorithms



NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

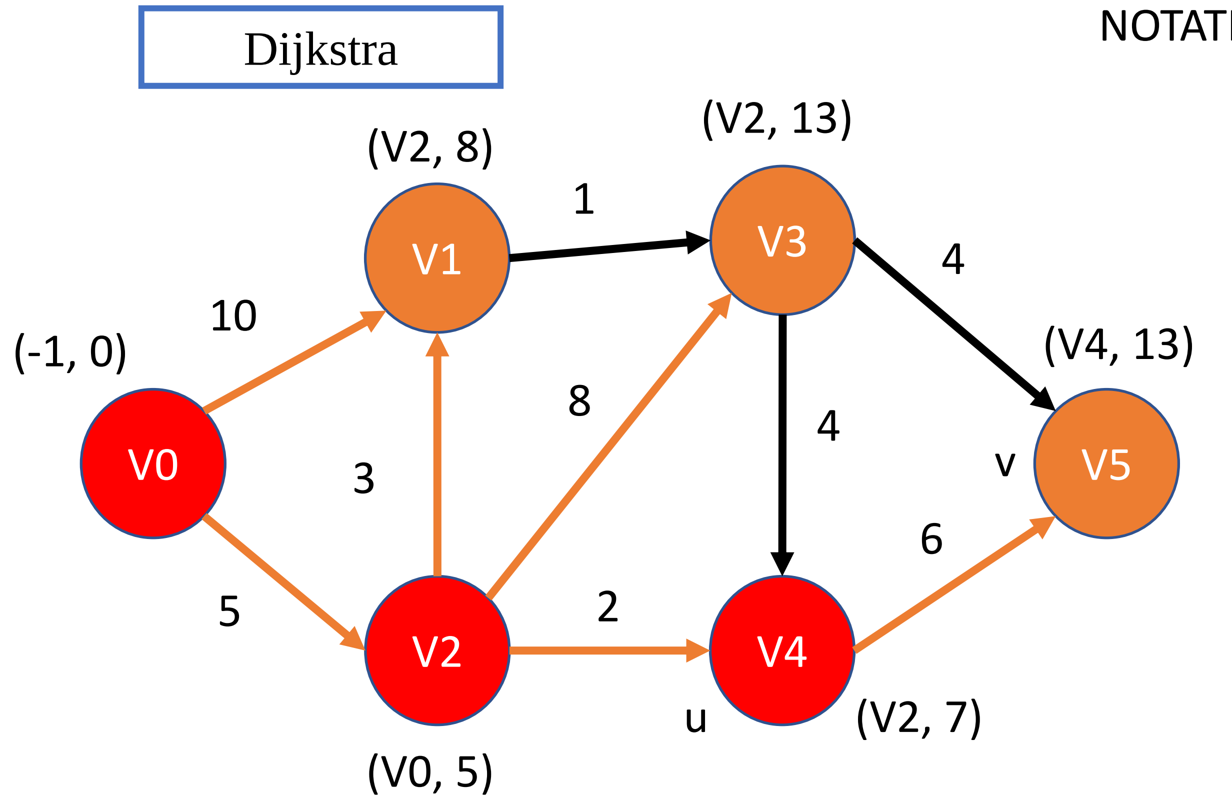
Algorithms



NOTATION ➡ (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all v ≠ s, and p(v) = -1 for all *v*
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close *u*
   * For every open node *v* adjacent to *u*: relax edge (*u*,*v*)

Algorithms

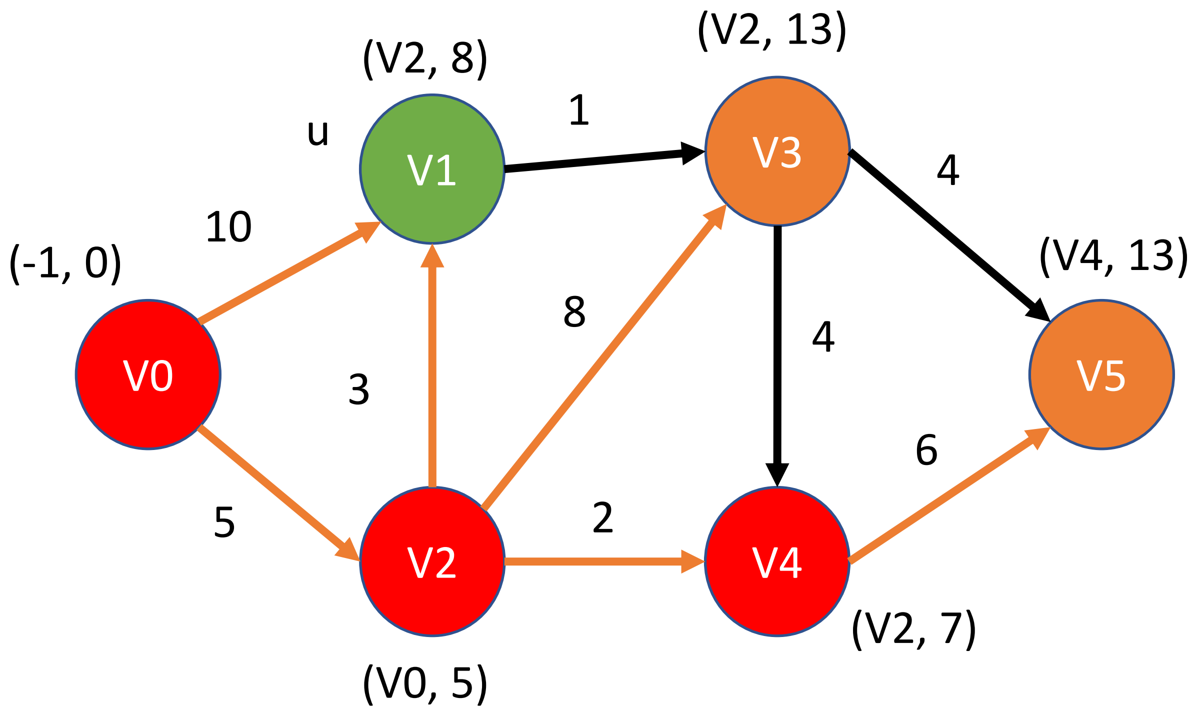


NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

Algorithms

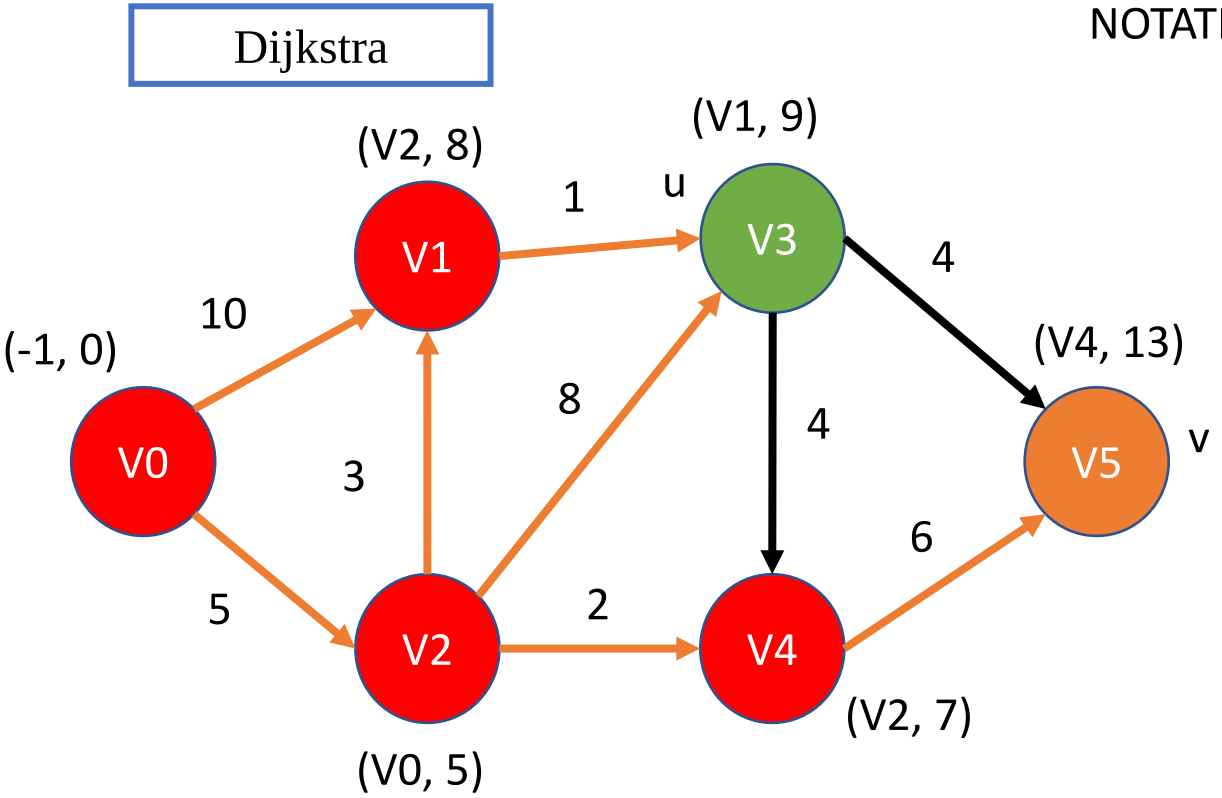
Dijkstra



NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

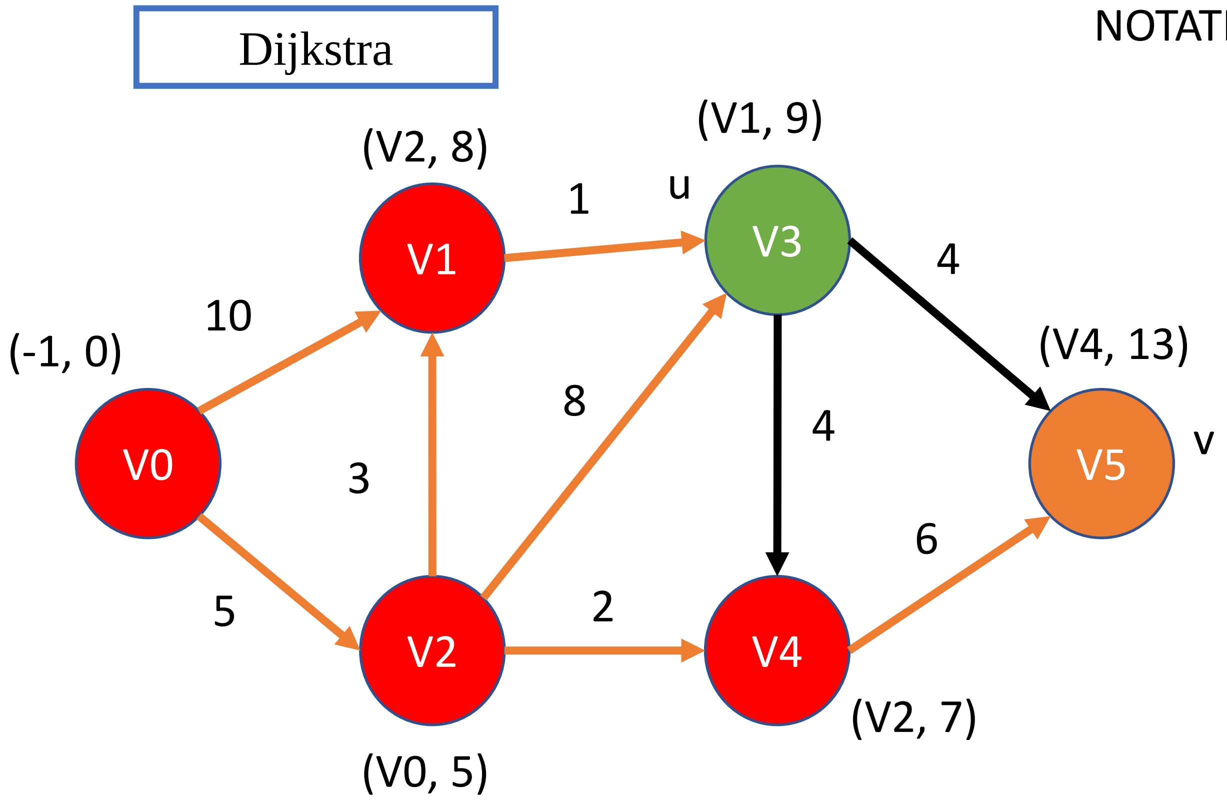
Algorithms



NOTATION ➡ (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

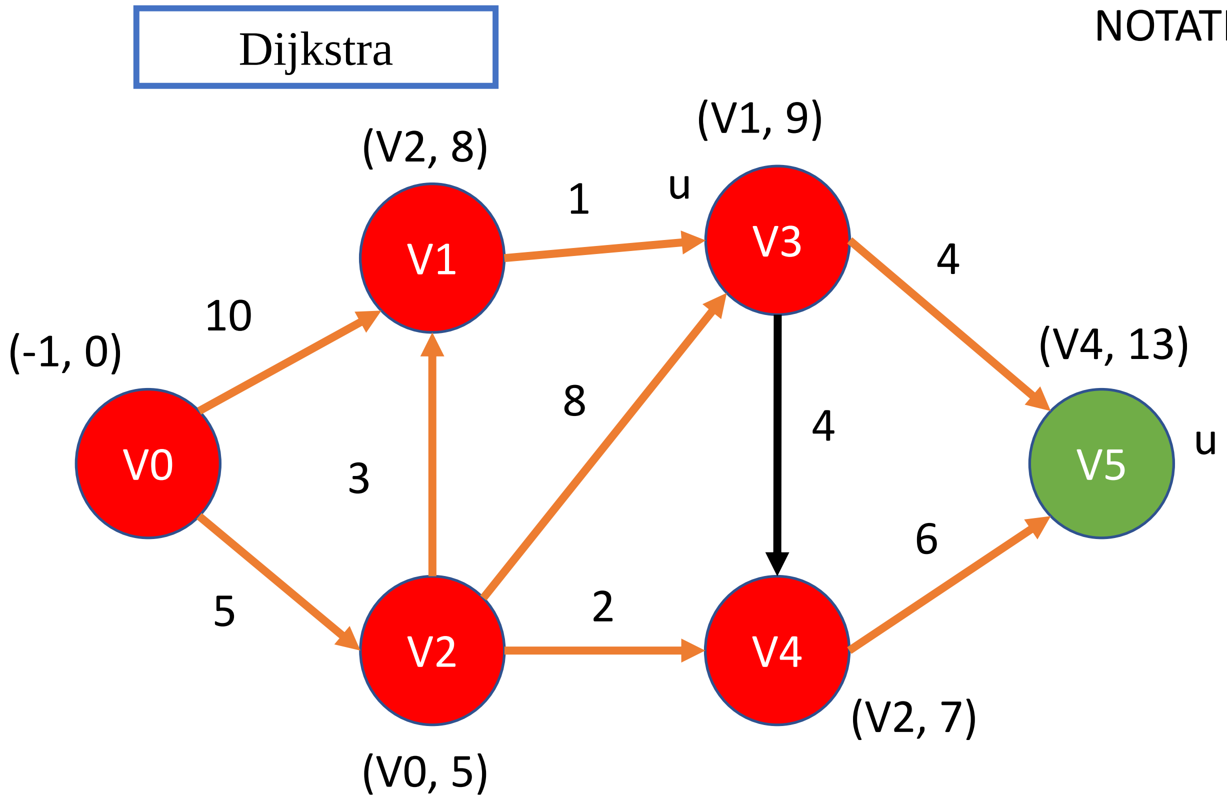
Algorithms



NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

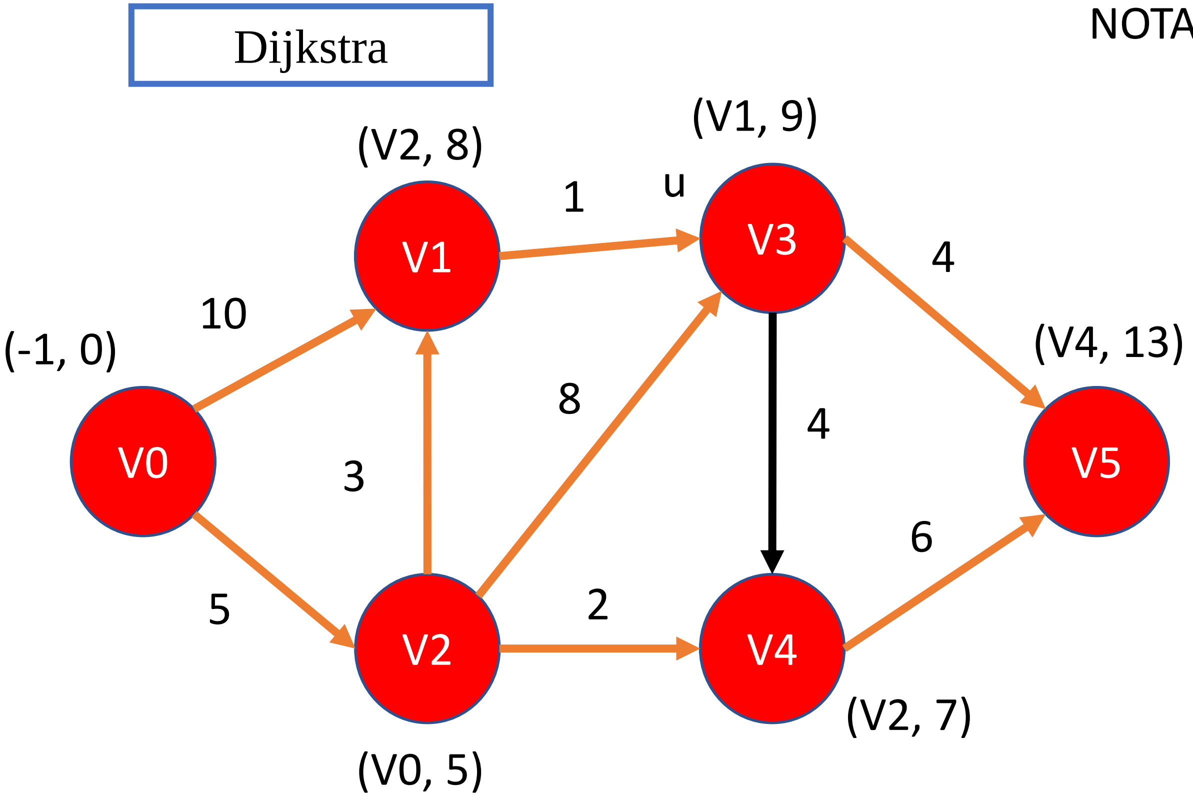
Algorithms



NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>v</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>u</u>
   * For every open node <u>v</u> adjacent to <u>u</u>: relax edge (<u>u,v</u>)

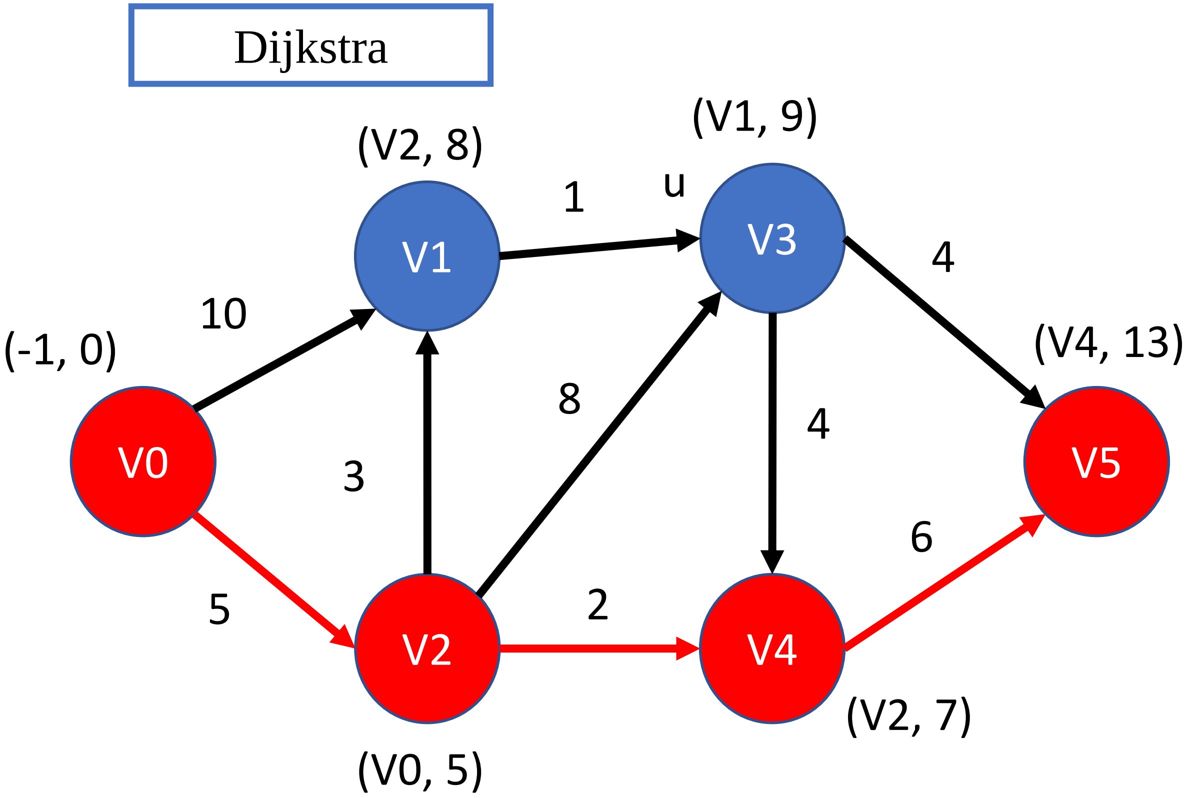
Algorithms



NOTATION → (father node, weigth)

1. Initialize the graph with d(s) = 0, d(v) = INF, for all <u>v ≠ s</u>, and p(v) = -1 for all <u>*v*</u>
2. Make open(v) = True for every v in the graph
3. As long as there is an open vertex:
   * Choose u whose estimate is the smallest among the open
   * Close <u>*u*</u>
   * For every open node <u>*v*</u> adjacent to <u>*u*</u>: relax edge (<u>*u,v*</u>)

Algorithms

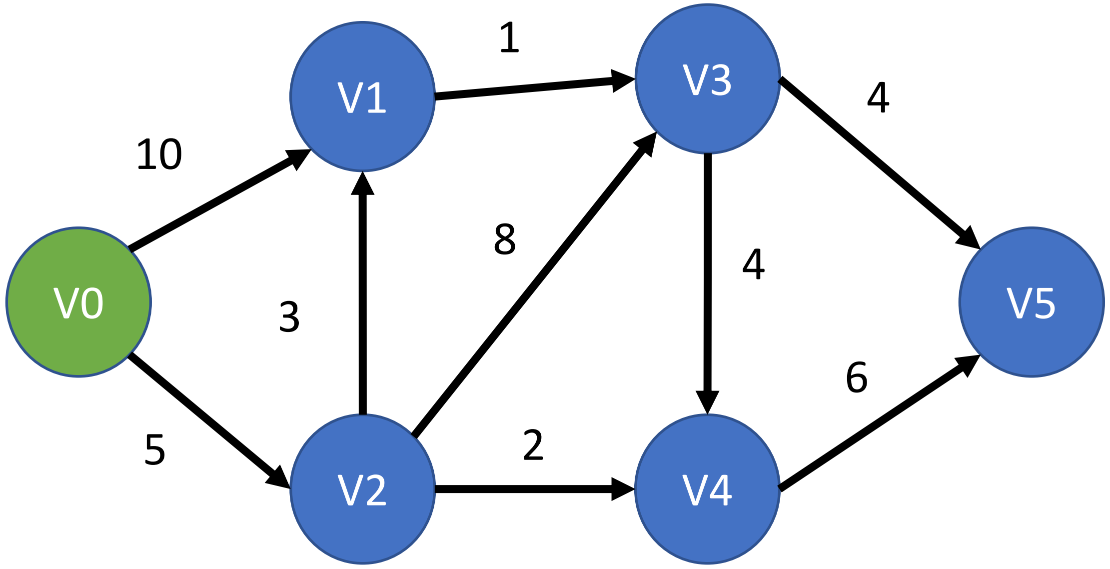


the shortest distance V0 => V5

V5, V4, V2, V0

Algorithms

A*

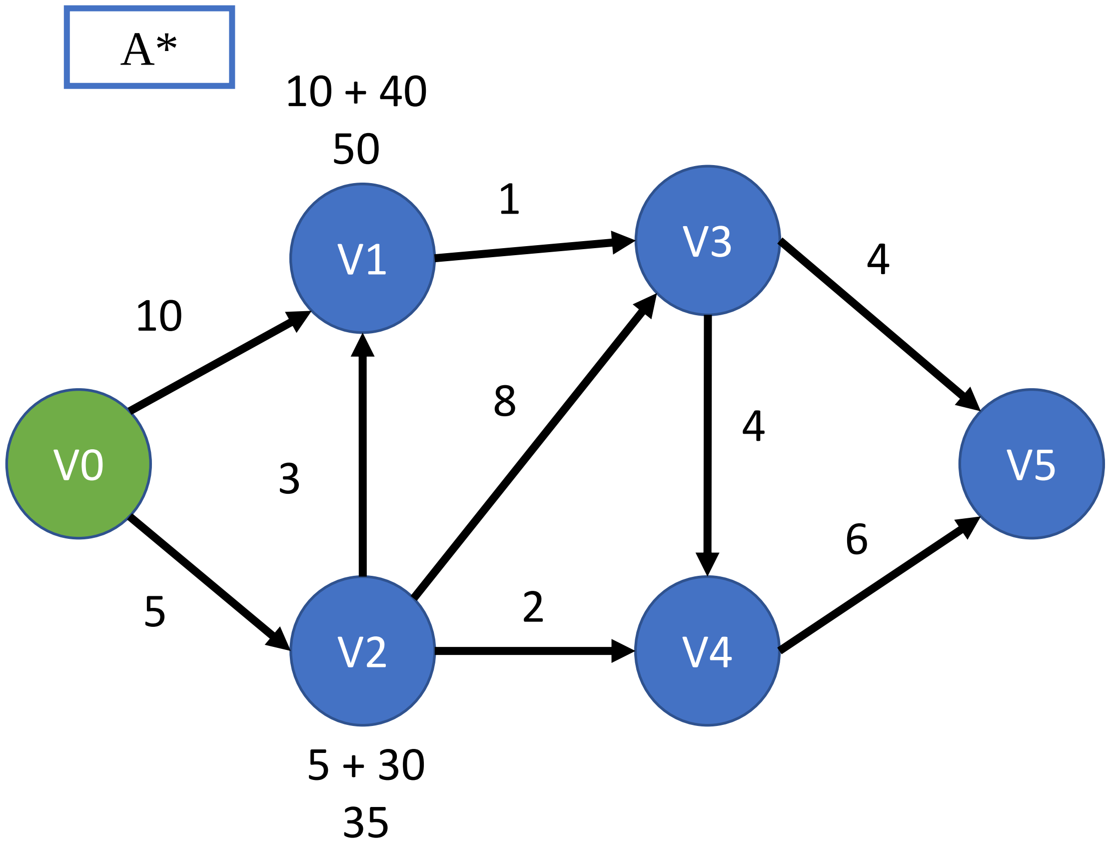


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

h(n) →

	V5
V0	50
V1	40
V2	30
V3	20
V4	10

Algorithms

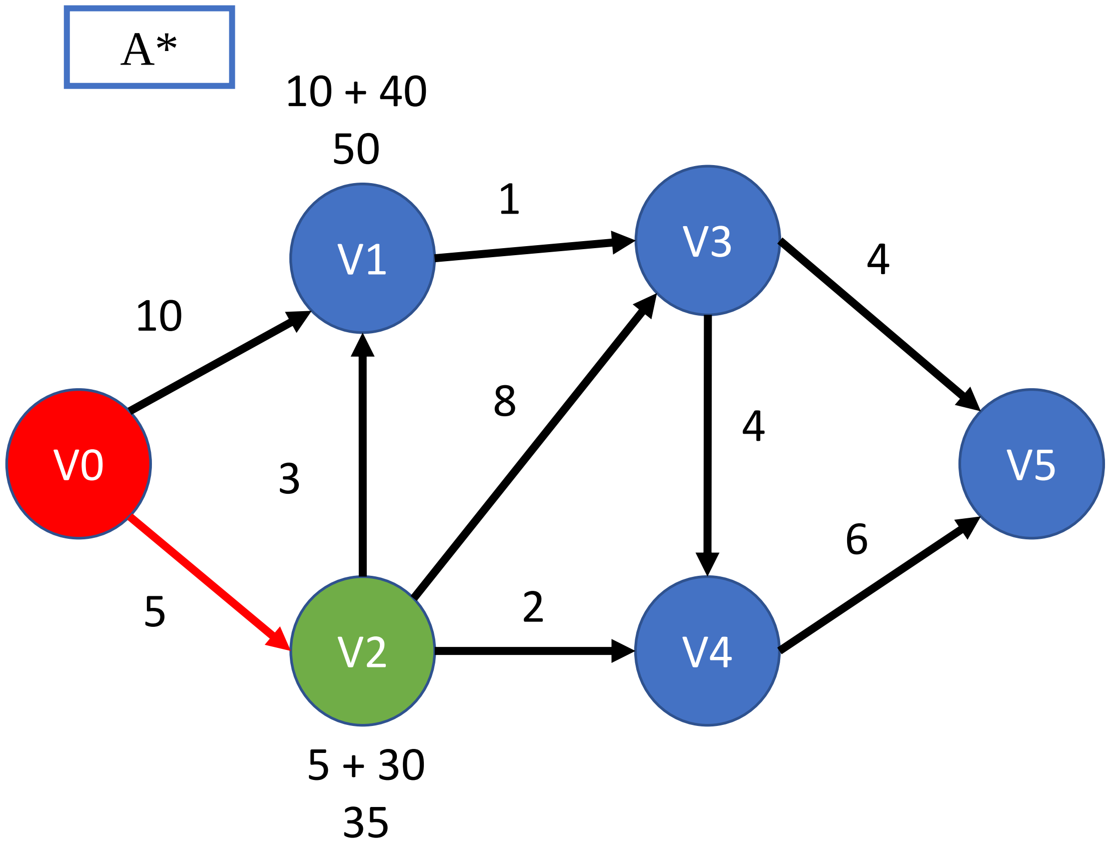


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

h(n)

	V5
V0	50
V1	40
V2	30
V3	20
V4	10

Algorithms

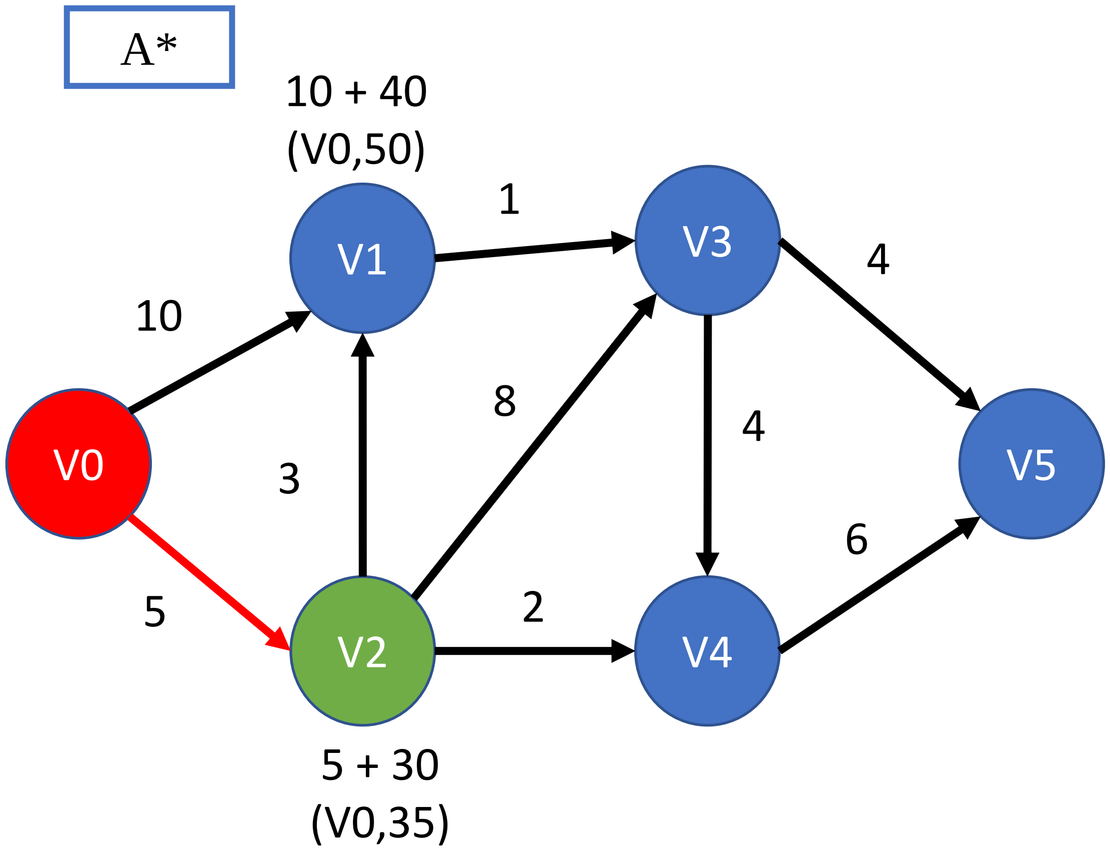


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

h(n)

	V5
V0	50
V1	40
V2	30
V3	20
V4	10

Algorithms

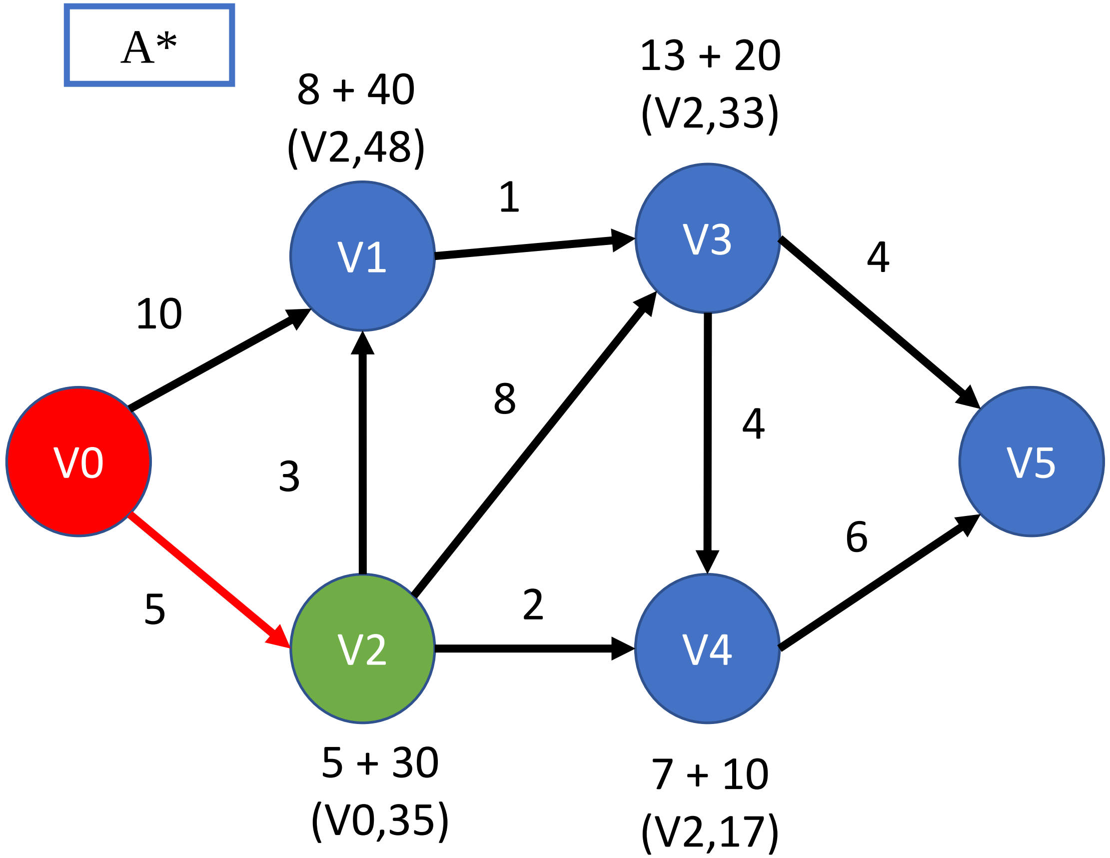


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

h(n) →

	V5
V0	50
V1	40
V2	30
V3	20
V4	10

Algorithms

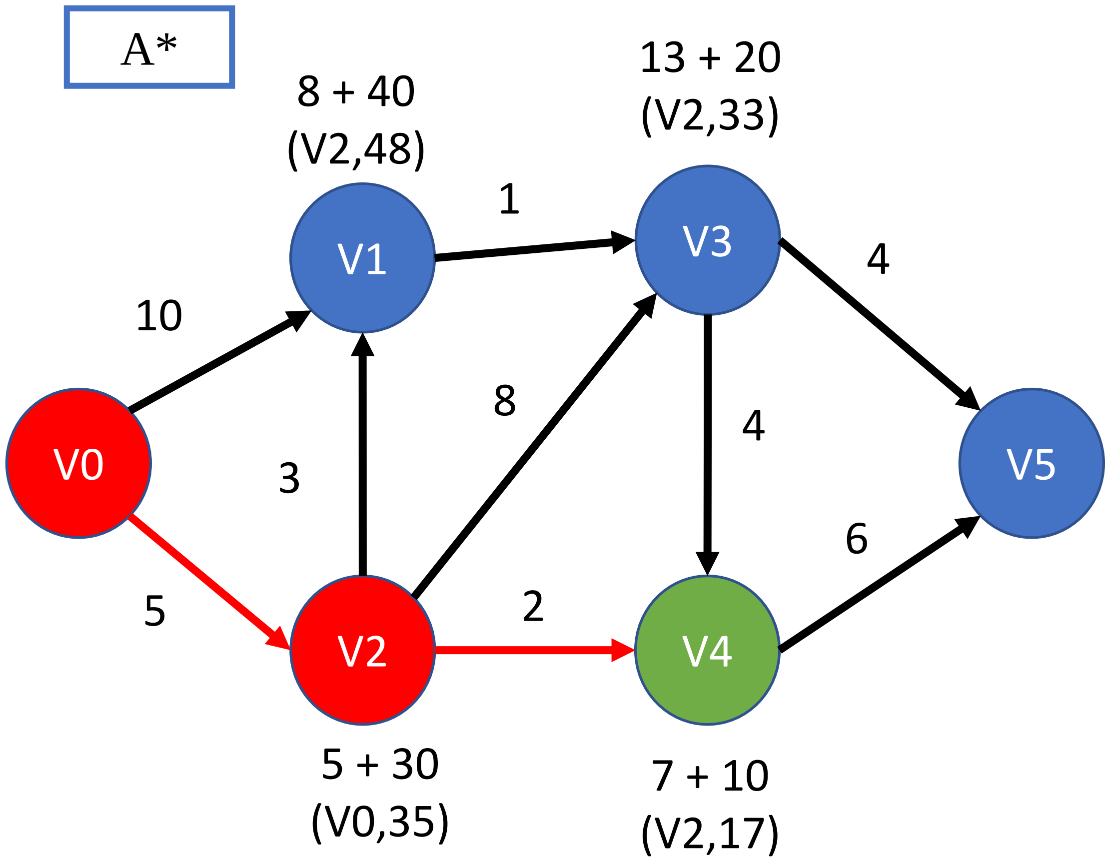


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

h(n)

	V5
V0	50
V1	40
V2	30
V3	20
V4	10

Algorithms

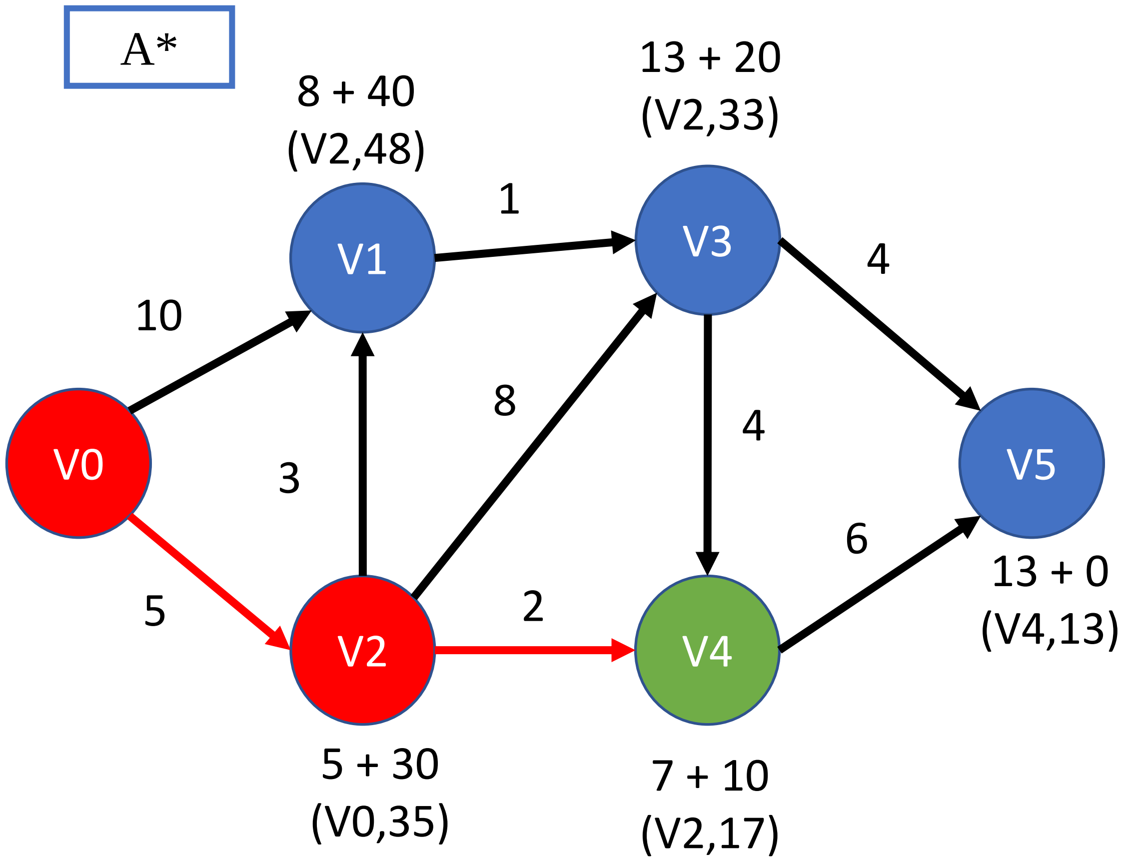


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

h(n) →

	V5
V0	50
V1	40
V2	30
V3	20
V4	10

Algorithms

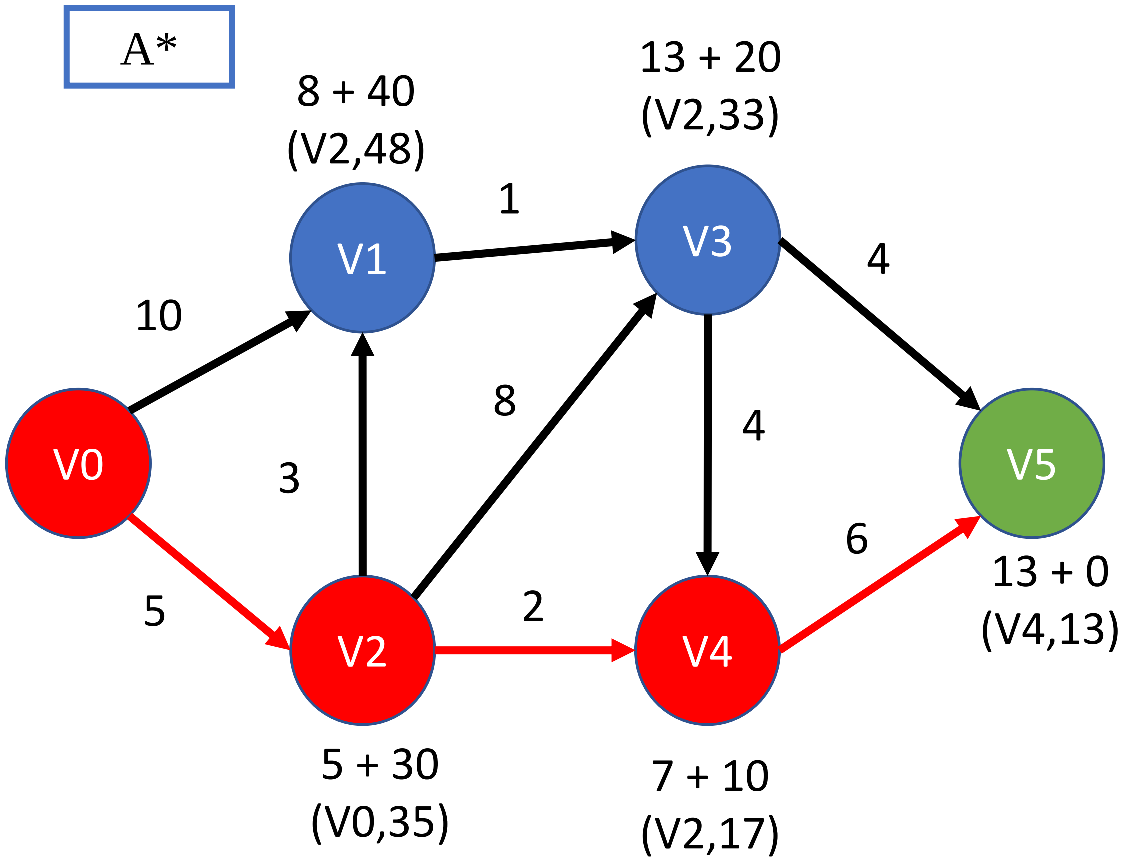


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

h(n) →

	V5
V0	50
V1	40
V2	30
V3	20
V4	10

Algorithms

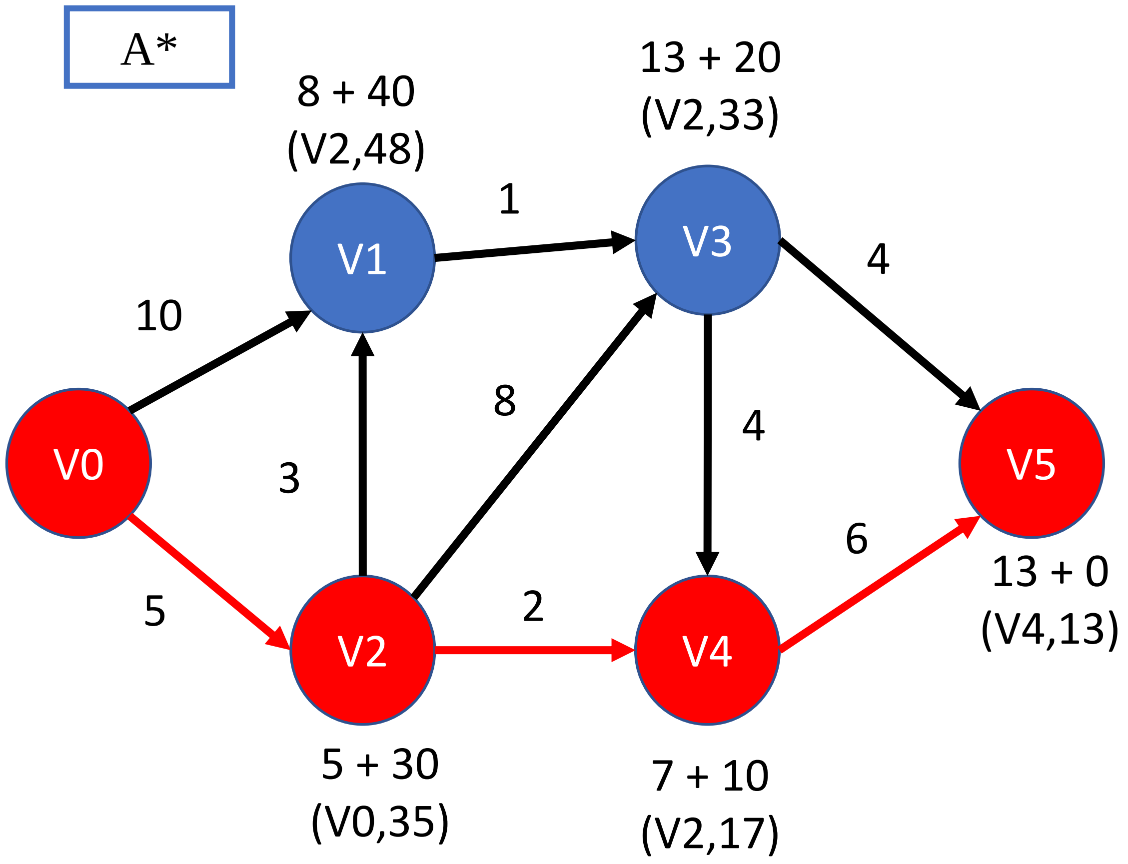


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

h(n)

	V5
V0	50
V1	40
V2	30
V3	20
V4	10

Algorithms

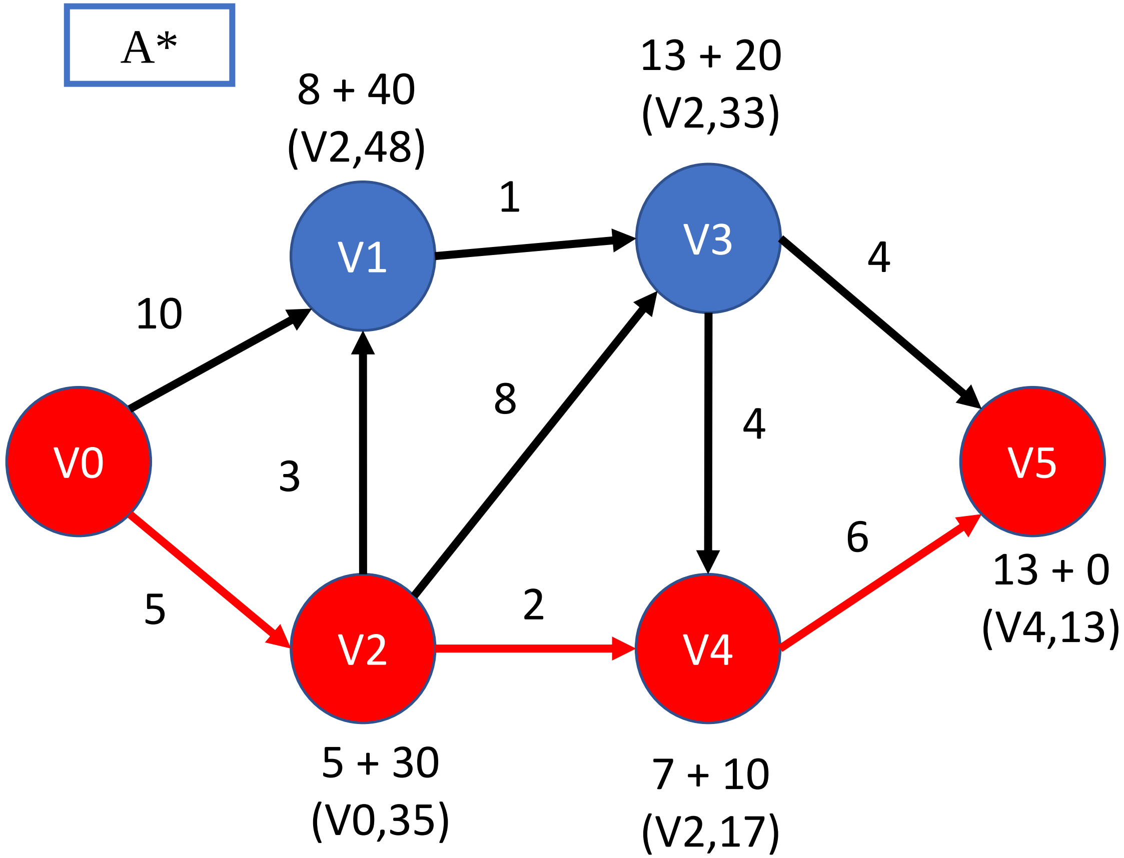


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

h(n) →

	V5
V0	50
V1	40
V2	30
V3	20
V4	10

Algorithms



the shortest distance V0 => V5

V5, V4, V2, V0

Algorithms

manhattan((x1, y1), (x2, y2)) = |x1 — x2| + |y1 — y2|

euclidean((x1, y1), (x2, y2)) = sqrt(x^2 + y^2)

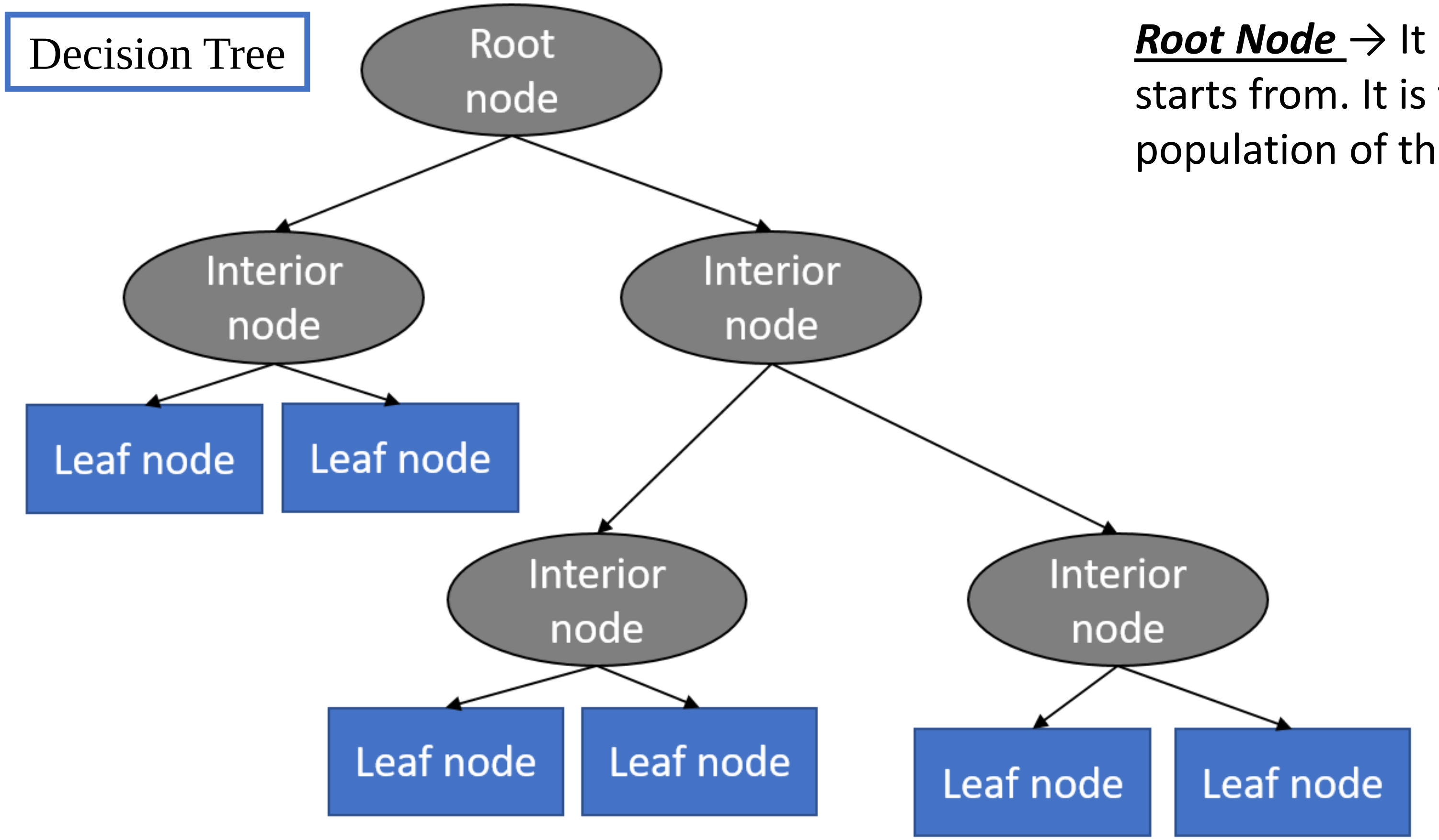
A*

0	1	1	1	1
1	1	1	1	1
1				1
1	1	1	1	1
1	1	1	1	1

h(n)

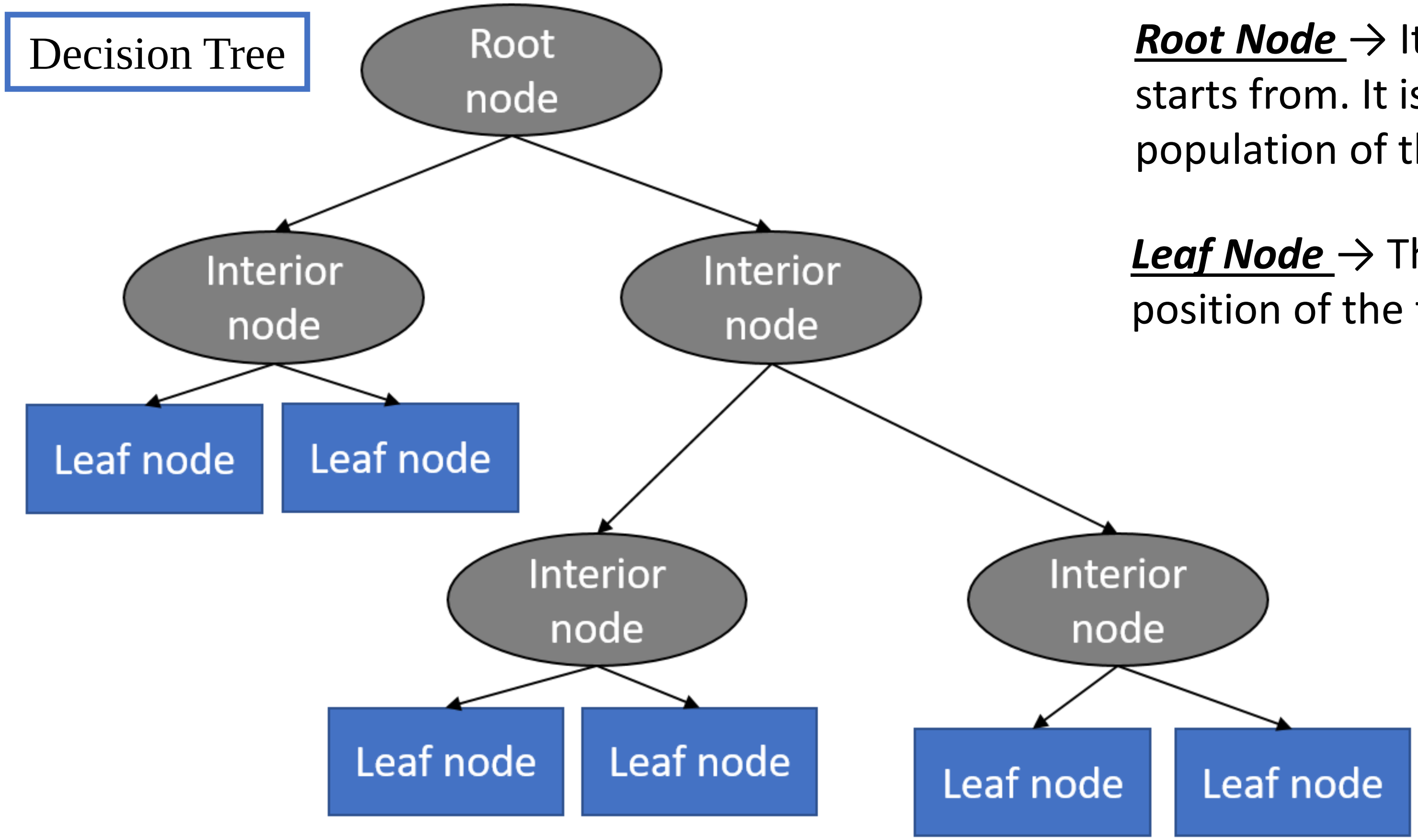
8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	4	3	2	1
4	3	2	1	0

Definitions



<u>Root Node</u> → It is the base node from where the entire tree starts from. It is the first node that represent the entire population of the tree.

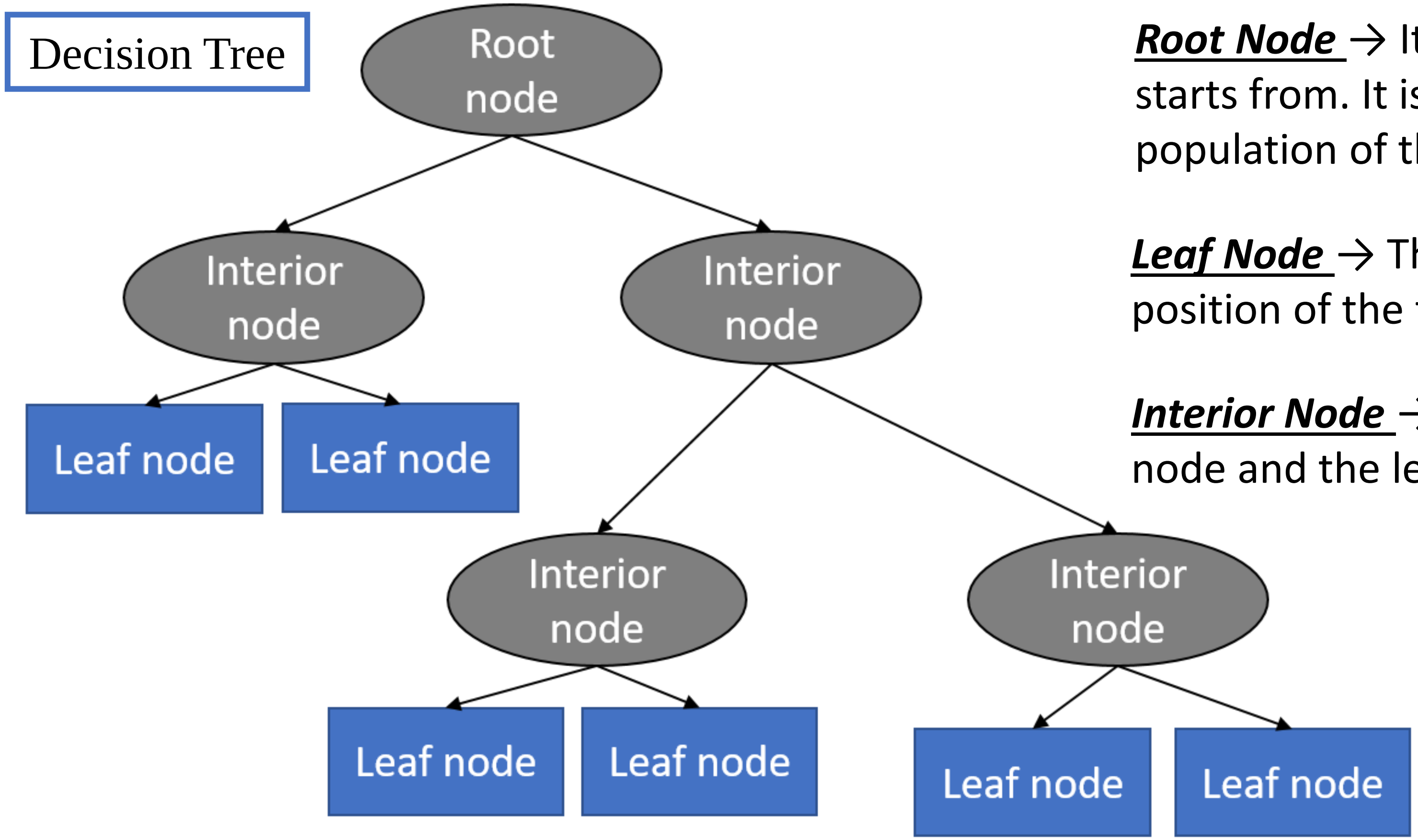
Definitions



Root Node → It is the base node from where the entire tree starts from. It is the first node that represent the entire population of the tree.

Leaf Node → The nodes which are present at the end or last position of the tree.

Definitions

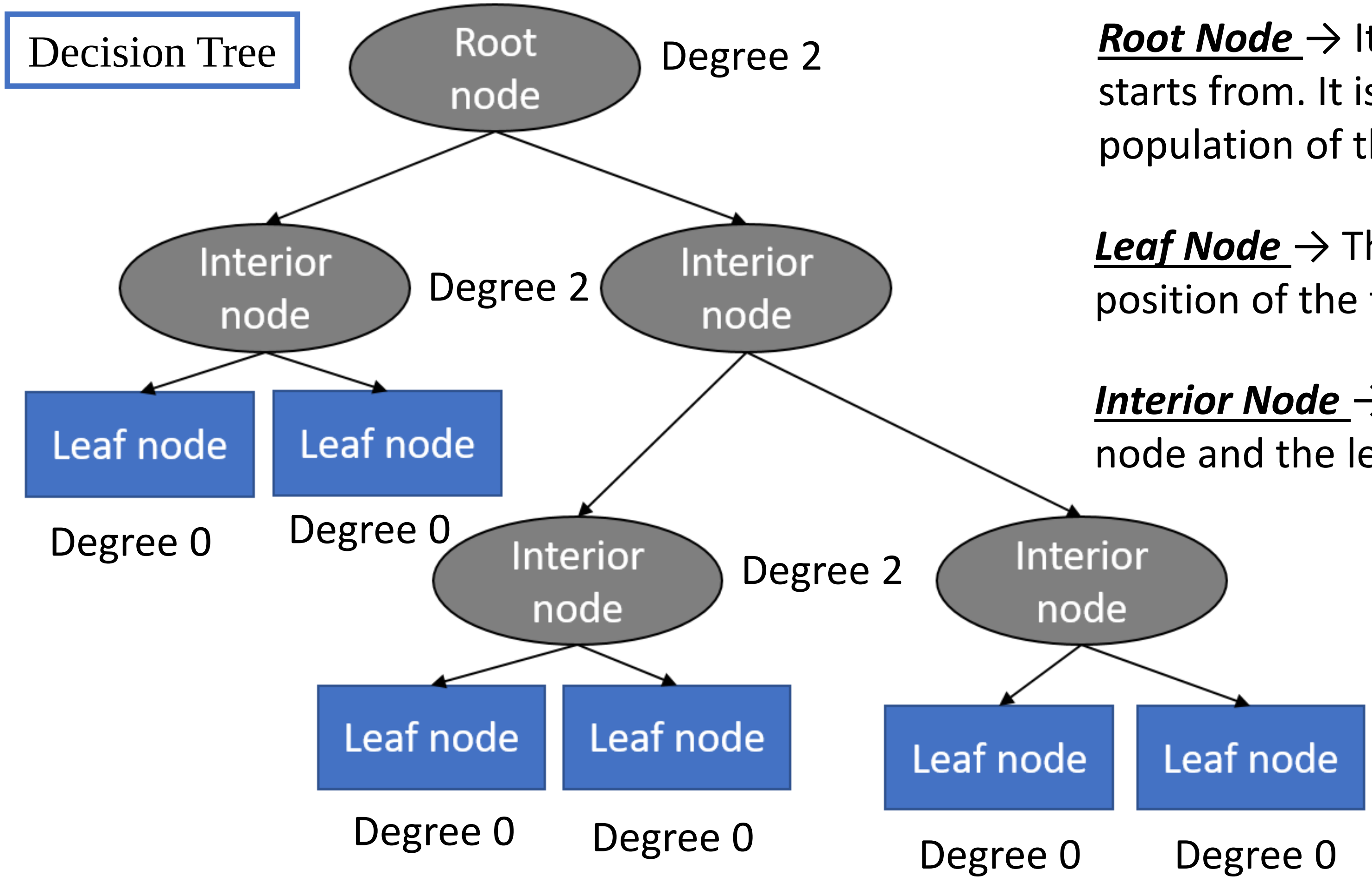


Root Node → It is the base node from where the entire tree starts from. It is the first node that represent the entire population of the tree.

Leaf Node → The nodes which are present at the end or last position of the tree.

Interior Node → The intermediate nodes between the root node and the leaf node.

Definitions

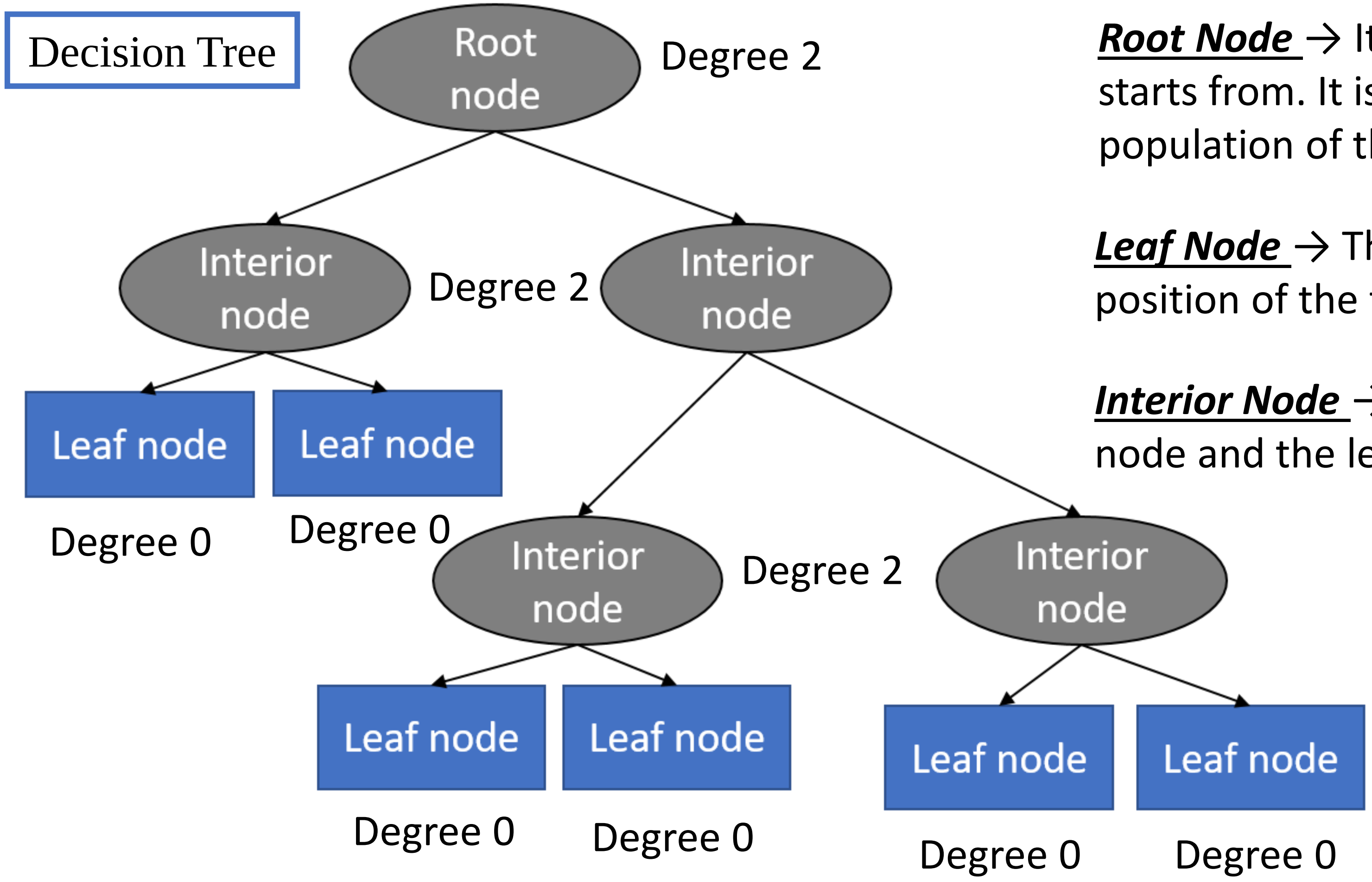


Root Node → It is the base node from where the entire tree starts from. It is the first node that represent the entire population of the tree.

Leaf Node → The nodes which are present at the end or last position of the tree.

Interior Node → The intermediate nodes between the root node and the leaf node.

Definitions



Root Node → It is the base node from where the entire tree starts from. It is the first node that represent the entire population of the tree.

Leaf Node → The nodes which are present at the end or last position of the tree.

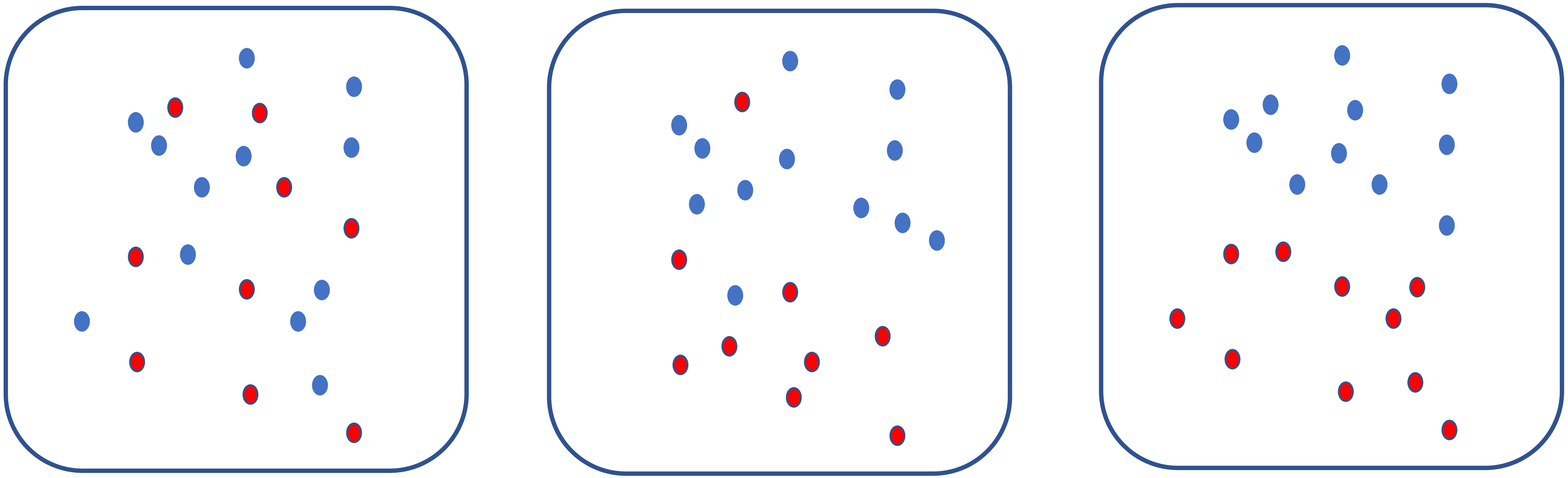
Interior Node → The intermediate nodes between the root node and the leaf node.

Binary Decision Tree → Node Degree < 3

Definitions

Decision Tree

Entropy →
- It is an information theory metric that measures the impurity or uncertainty in a group of observations.
- It helps to decide the best attribute for start for start making decisions.
- It helps in telling the attribute with highest information gain.
- It is the presence of impurity (degree of randomness).



Very Impure Less Impure Minimum Impure

Definitions

Decision Tree

Information Gain →

- It define information gain as a measure of how much information a feature provides about a class.
- It is the decrease or reduction in entropy after a dataset is split on the basis of an attribute so that it helps to decide which attribute should be selected as the decision node.
- It helps to determine the order of attributes in the nodes of a decision tree.
- Constructing a Decision tree is all about finding the attribute that returns highest information gain.

$$Gain = E_{parent} - E_{children}$$

- $Gain$ represents information gain
- E_{parent} is the entropy of the parent node
- $E_{children}$ is the entropy of the child nodes

Definitions

Gini Index →

- Gini impurity is a function that determines how well a decision tree was split.
- It helps to determine which splitter is best so that we can build a pure decision tree.
- Gini impurity ranges values from 0 to 0.5

Consider a dataset with N classes →

$$\text{Gini Index} = 1 - \sum_{i=1}^{n} (P_i)^2$$

Pi denotes the probability of an element being classified for a class ***i***.

Definitions

Máquinas de Estado Finitos

Finite State Machine (FSM)

- É um modelo matemático utilizado para representar os diversos comportamentos e as respetivas transições entre estes em um programa. Ou seja, é **composta por estados e transições**.
- Cada estado representa o sistema em um determinado momento no tempo e não é possível uma máquina de estados estar em dois estados ao mesmo tempo.
- É um conjunto de estados finitos que funcionam como intermediários entre uma relação de entrada e saídas. Desta forma, a saída dependerá do estado das entradas naquele momento
- Quando uma máquina está em um estado, ela aguarda que as condições para uma transição sejam atingidas
- Cada máquina possui um estado inicial, onde a máquina começa, e pode ter um ou mais estados finais, indicando que a máquina terminou a tarefa computacional.

Definitions

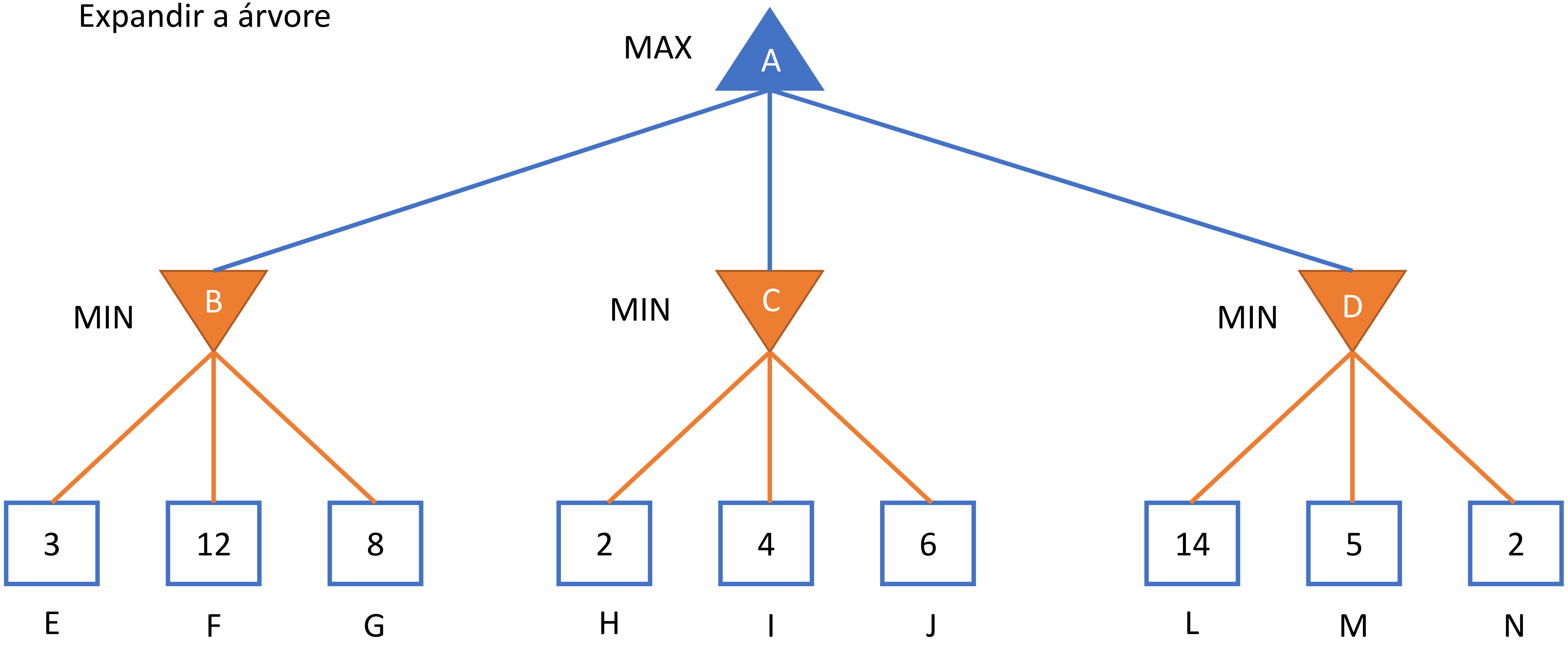
Árvore de Comportamentos

Behaviour Tree (BTs)

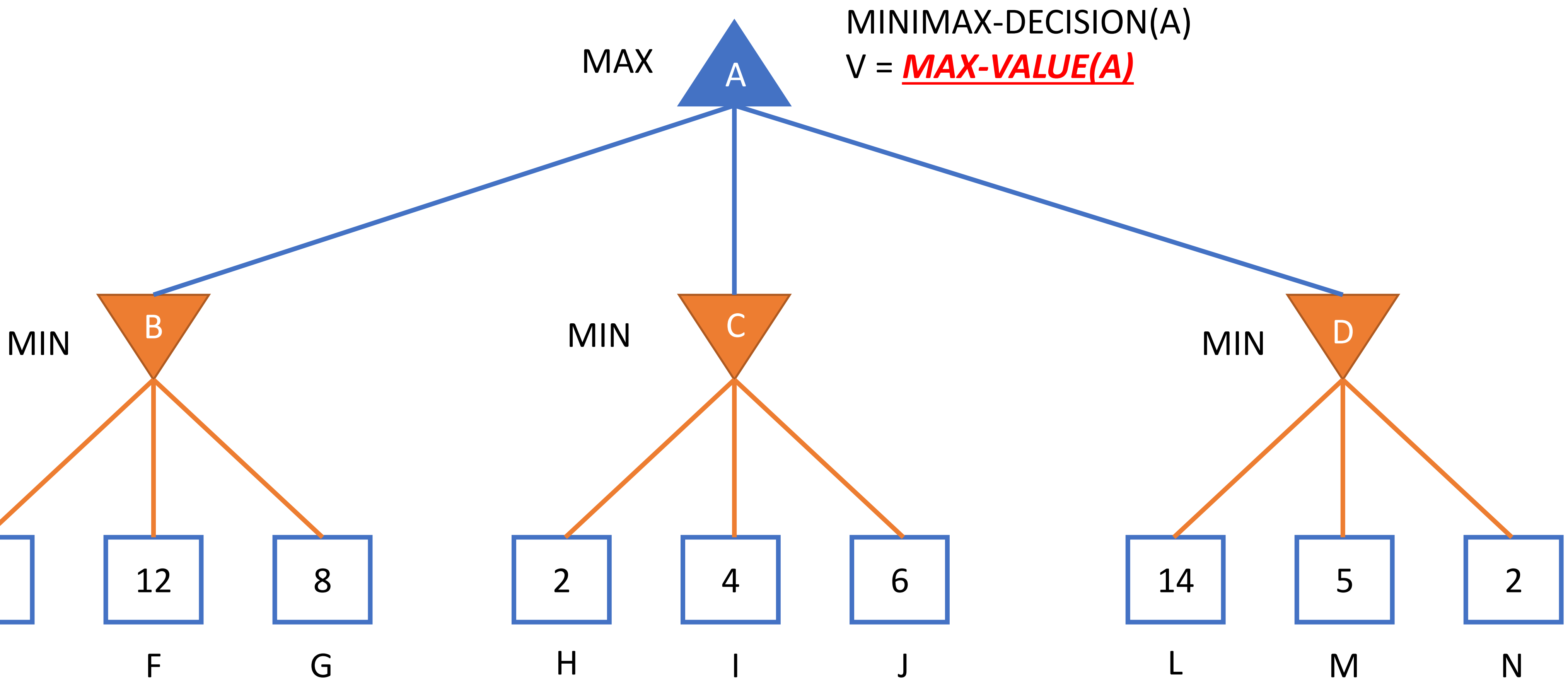
- A árvore de comportamento é uma árvore de nós hierárquicos que controlam o fluxo de tomada de decisão de uma entidade de Inteligência Artificial (IA)
- É uma arquitetura de IA que fornece aos **Non Player Characters (NPC)** do jogo a capacidade de selecionar comportamentos e executá-los, por meio de uma arquitetura semelhante a uma árvore que define operações lógicas simples.
- Tem sistemas semelhantes as de uma Máquina de Estado Finita
- Os estados das BTs são chamadas de **Tarefas**.
- Nas folhas estão os comandos reais que controlam a entidade da IA, e formando os ramos estão vários tipos de nós utilitários que controlam a caminhada da IA pelas árvores para alcançar as sequências de comandos mais adequadas à situação.

MiniMax Algorithm

Expandir a árvore



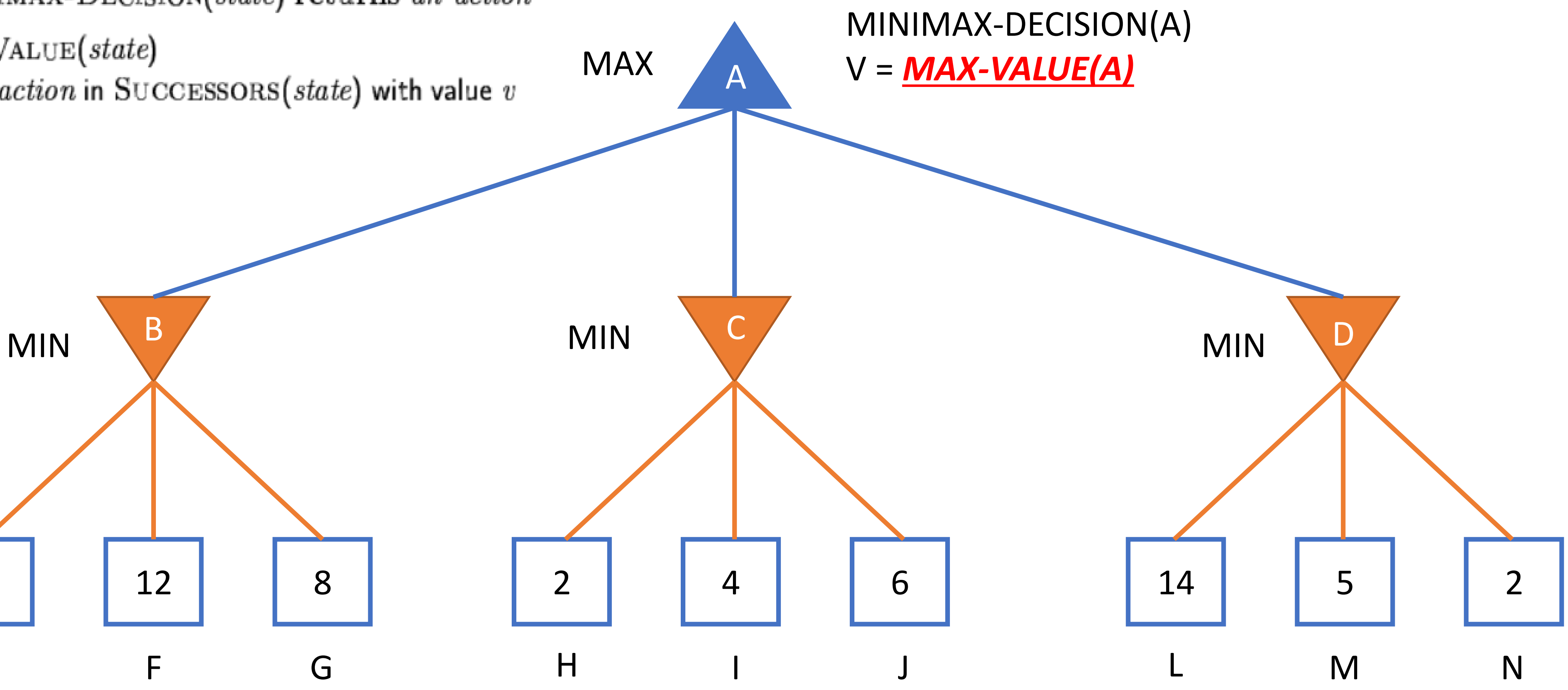
MiniMax Algorithm



MiniMax Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*

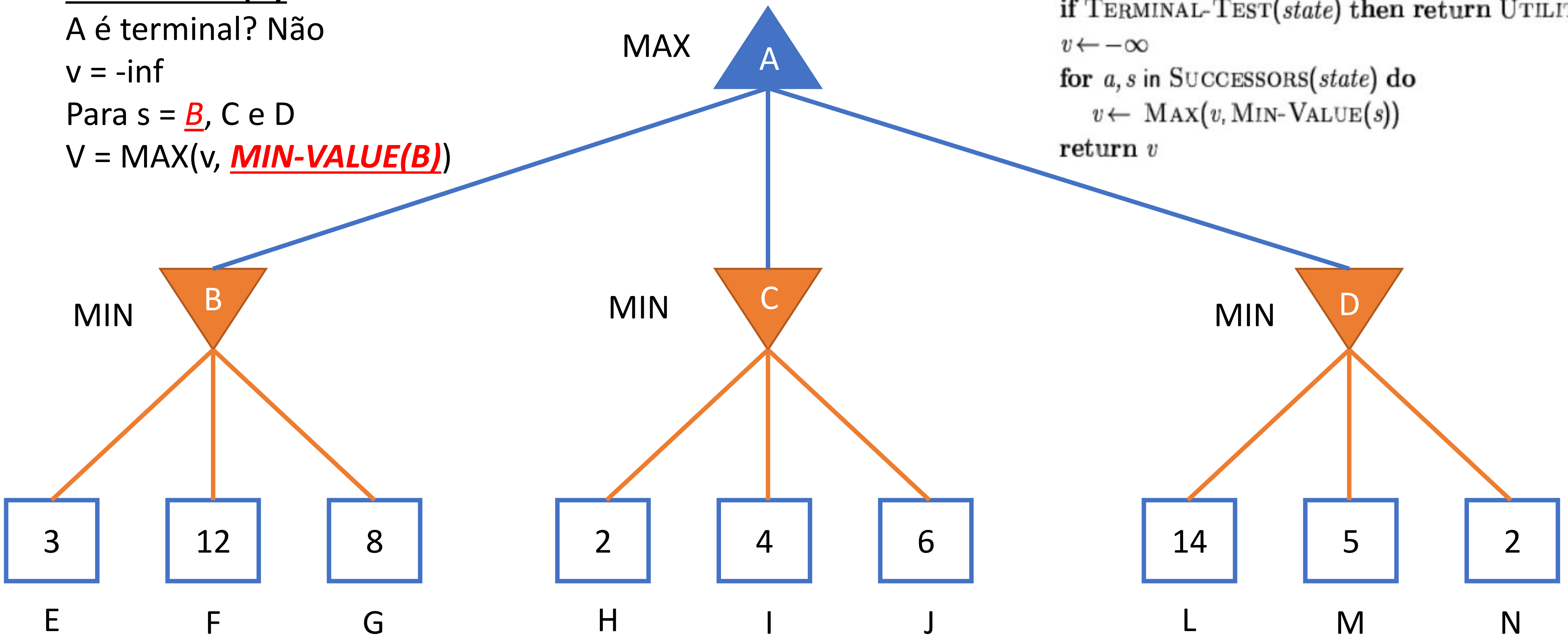
$v \leftarrow$ MAX-VALUE(*state*)
return the *action* in SUCCESSORS(*state*) with value v



MiniMax Algorithm

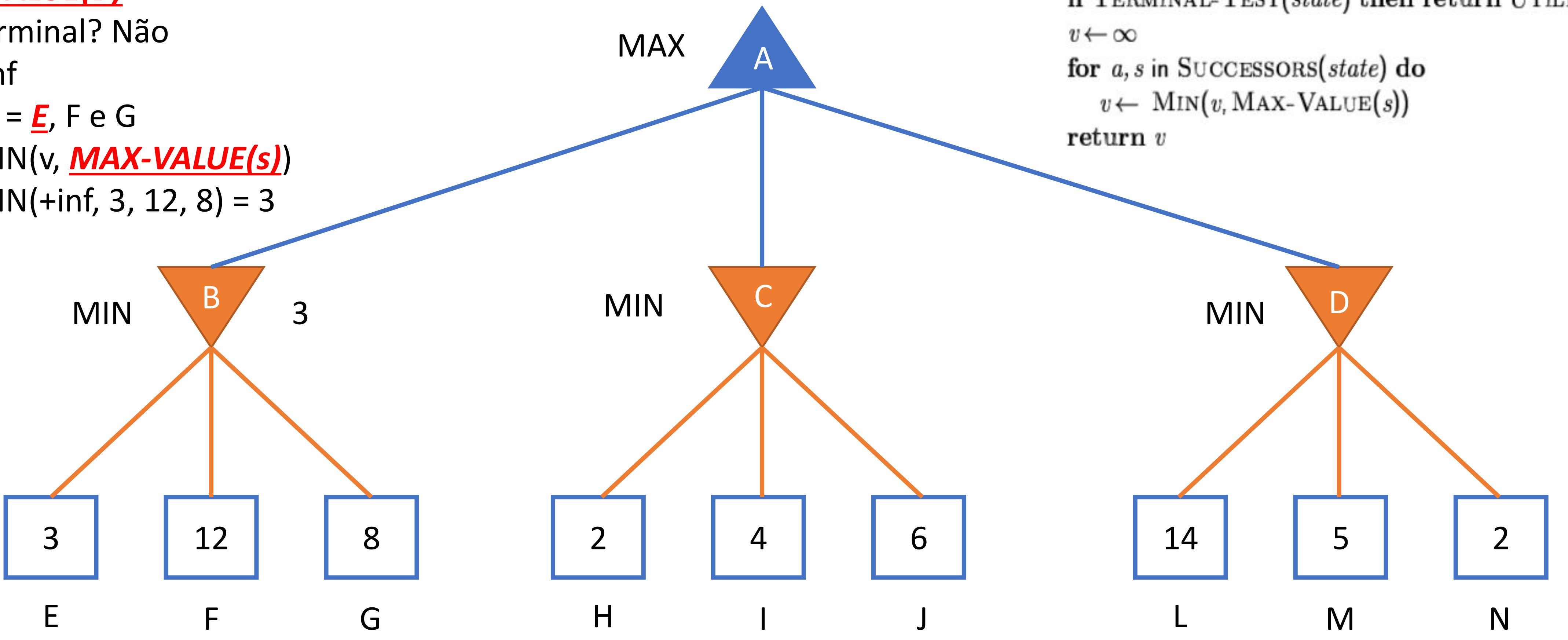
<u>MAX-VALUE(A)</u>
A é terminal? Não
v = -inf
Para s = <u>*B*</u>, C e D
V = MAX(v, ***<u>MIN-VALUE(B)</u>***)

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s))
  return v



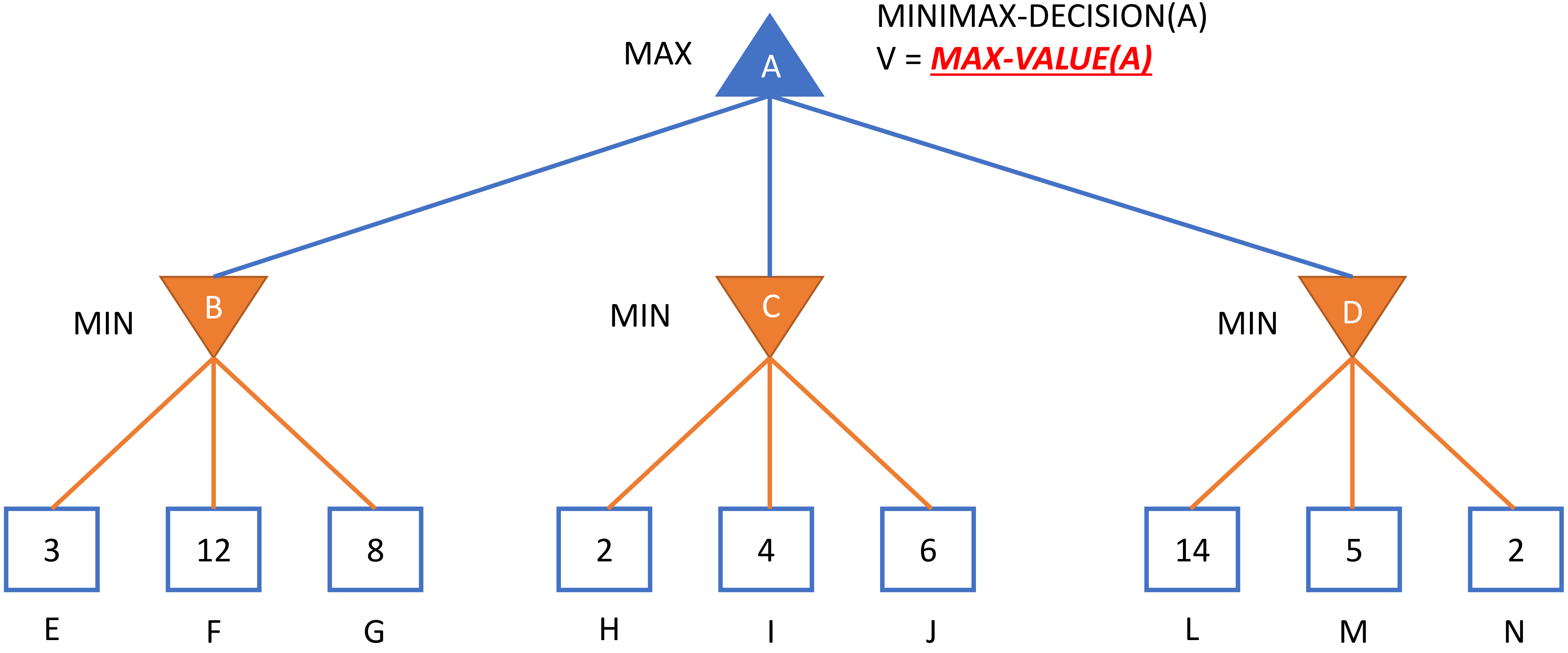
MiniMax Algorithm

MIN-VALUE(B)
B é terminal? Não
v = +inf
Para s = ***E***, F e G
V = MIN(v, ***MAX-VALUE(s)***)
V = MIN(+inf, 3, 12, 8) = 3



function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(s))
  return v

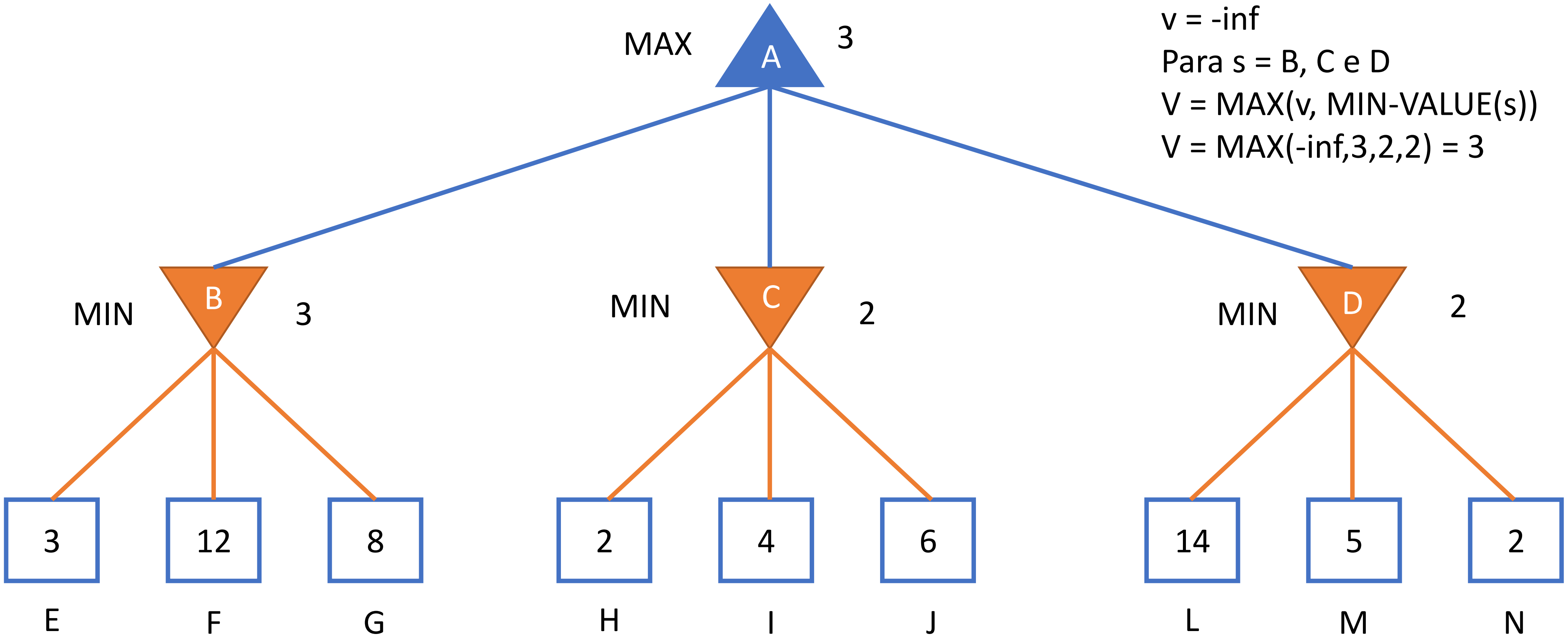
MiniMax Algorithm



MINIMAX-DECISION(A)
V = ***MAX-VALUE(A)***

MAX-VALUE(E)
E é terminal? Sim
v = UTILITY(E) = 3

MiniMax Algorithm



MAX-VALUE(A)

A é terminal? Não

v = -inf

Para s = B, C e D

V = MAX(v, MIN-VALUE(s))

V = MAX(-inf,3,2,2) = 3

MiniMax Algorithm

Poda (Pruning) α-β

[α, β]

MAX

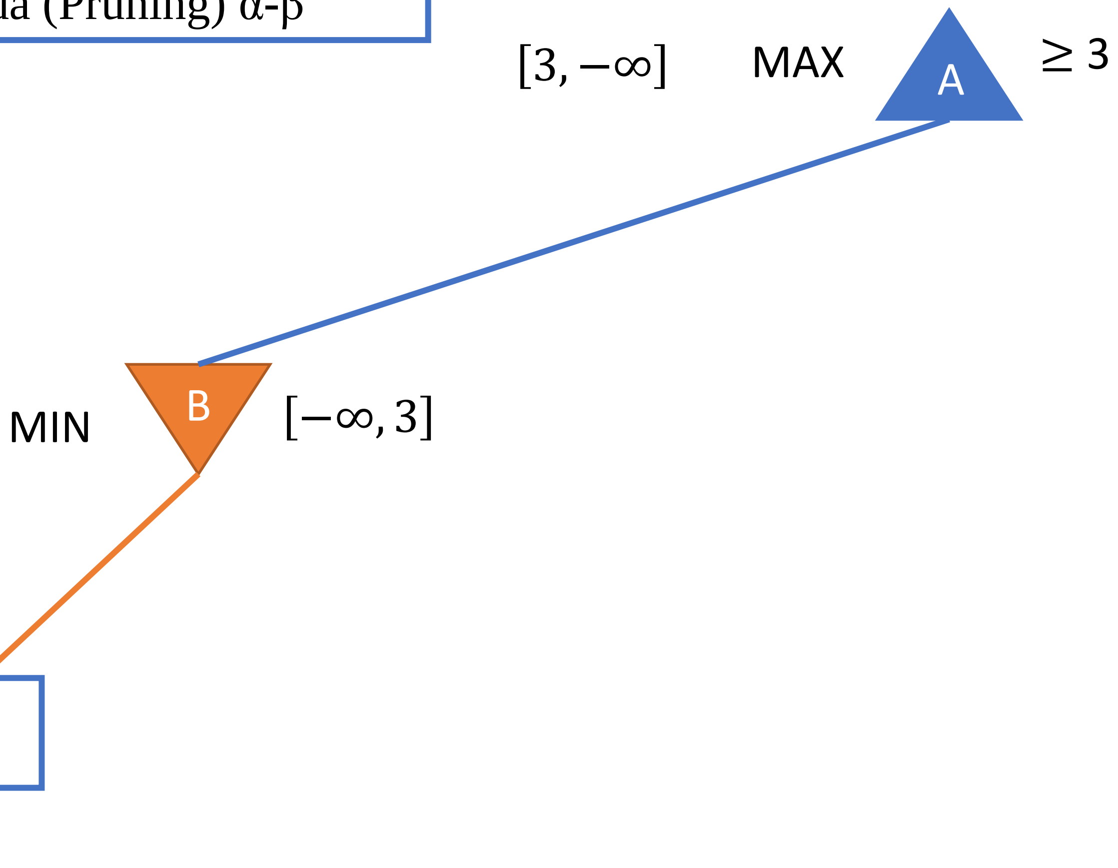
MiniMax Algorithm

Poda (Pruning) α-β

$[+\infty, -\infty]$ MAX

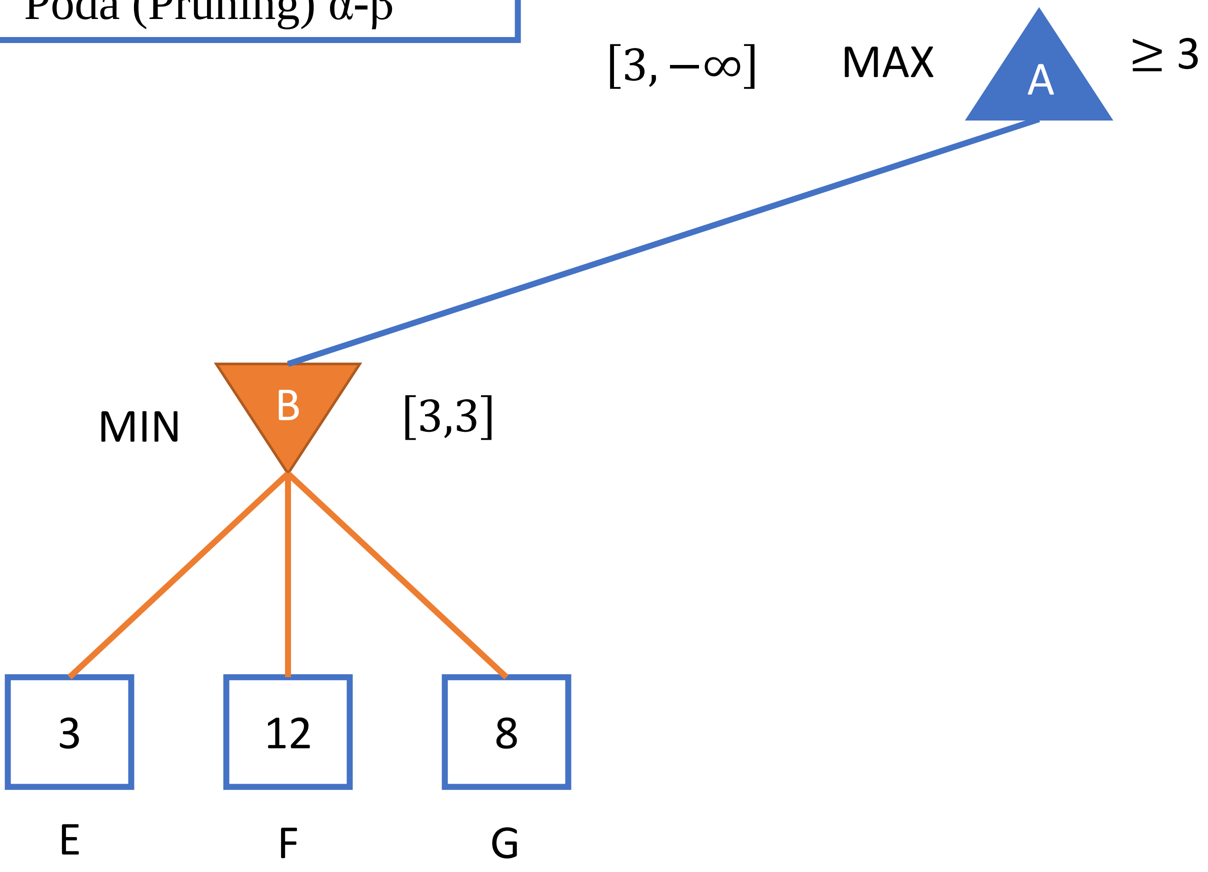
MiniMax Algorithm

Poda (Pruning) α-β



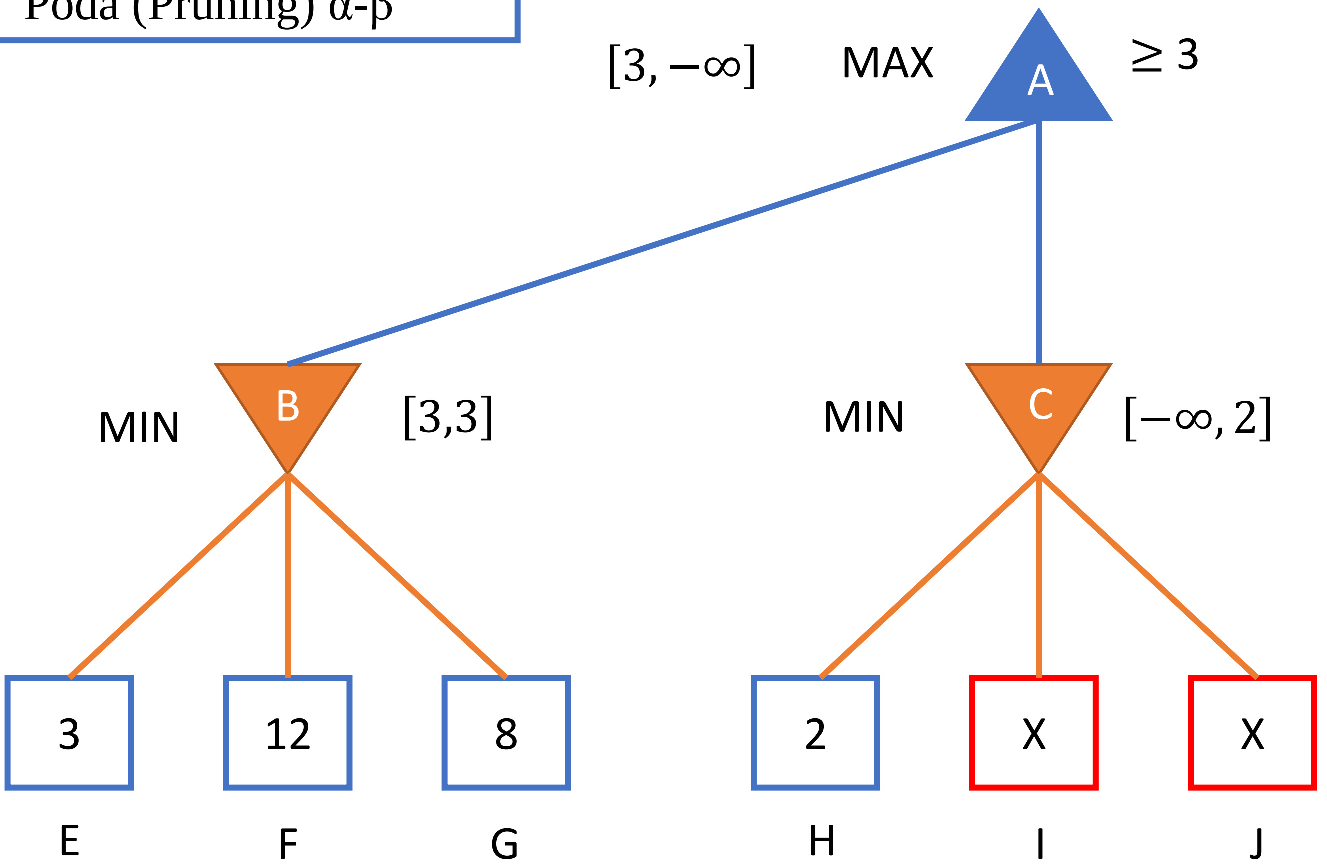
MiniMax Algorithm

Poda (Pruning) α-β



MiniMax Algorithm

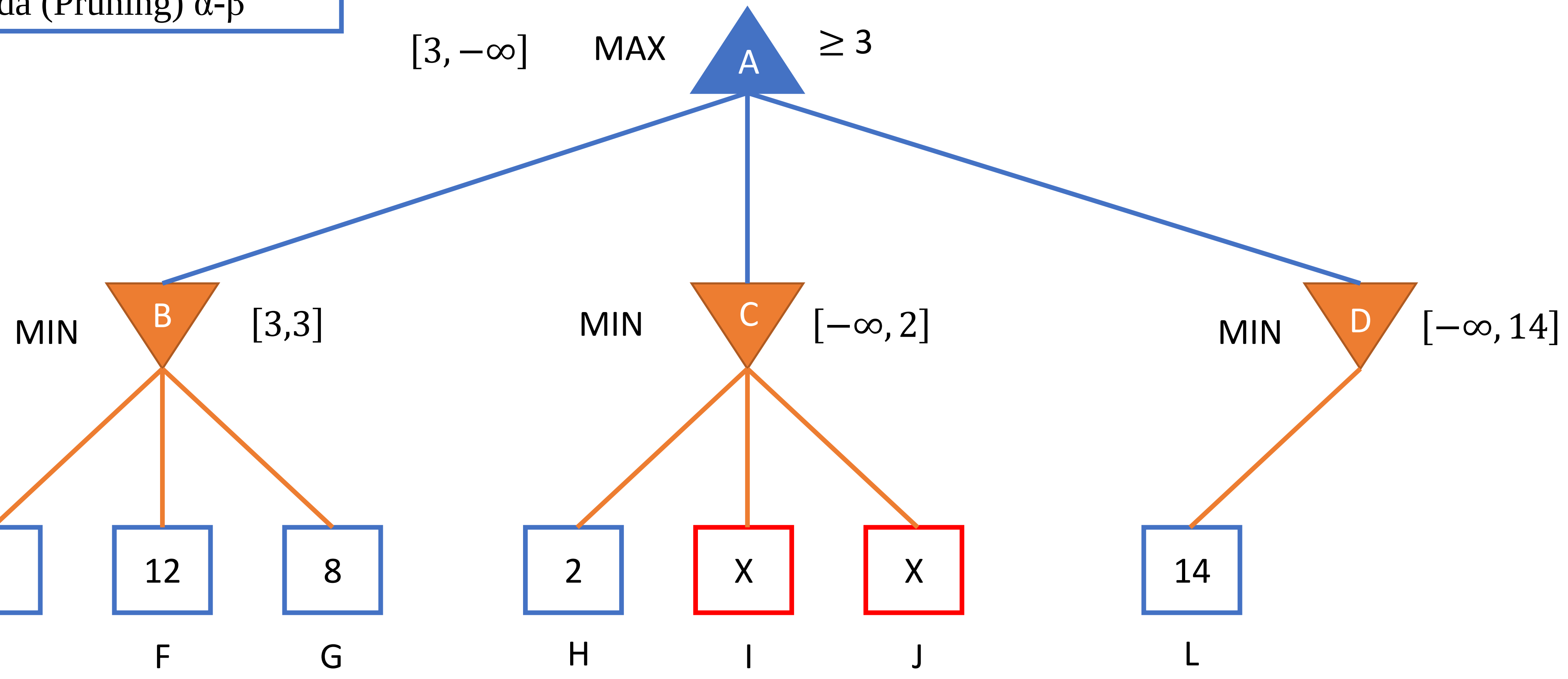
Poda (Pruning) α-β



Não olha as outras folhas => 2 < 3

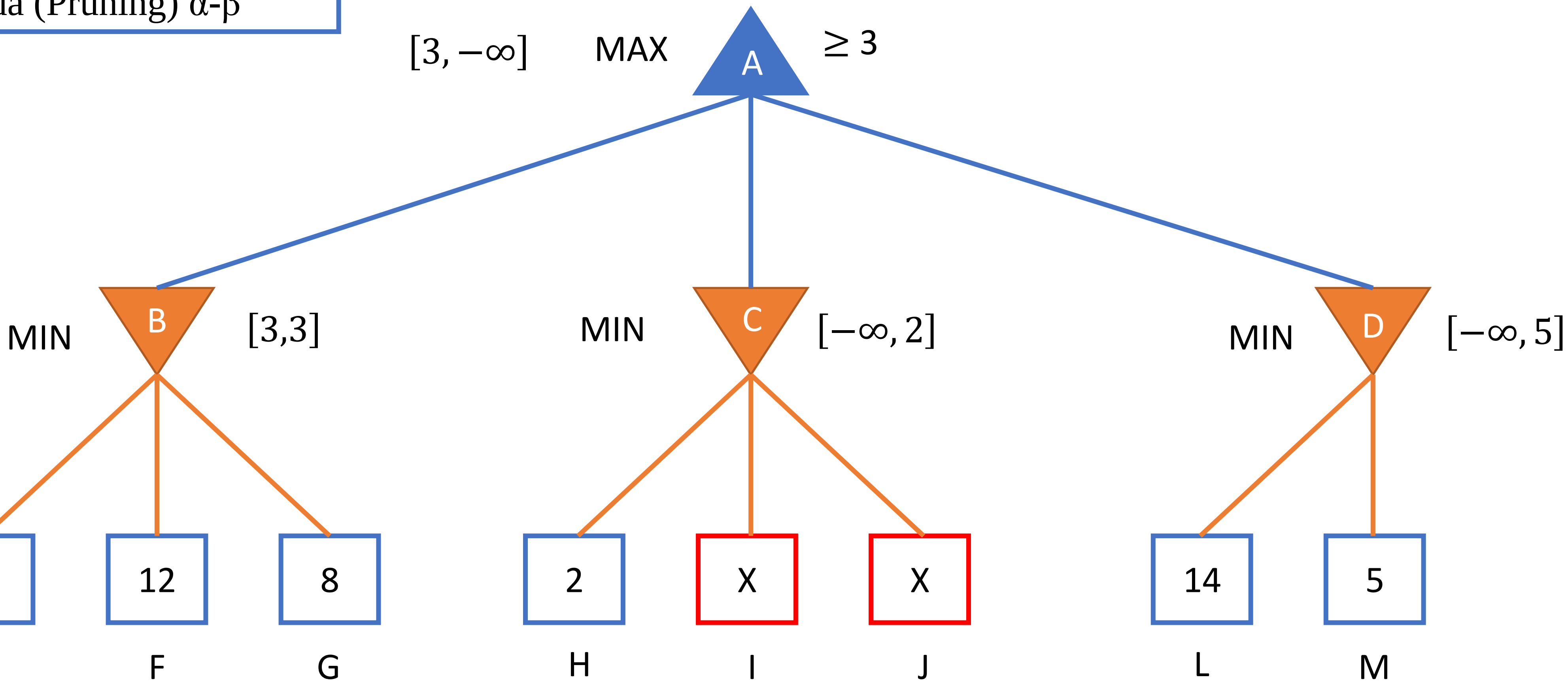
MiniMax Algorithm

Poda (Pruning) α-β



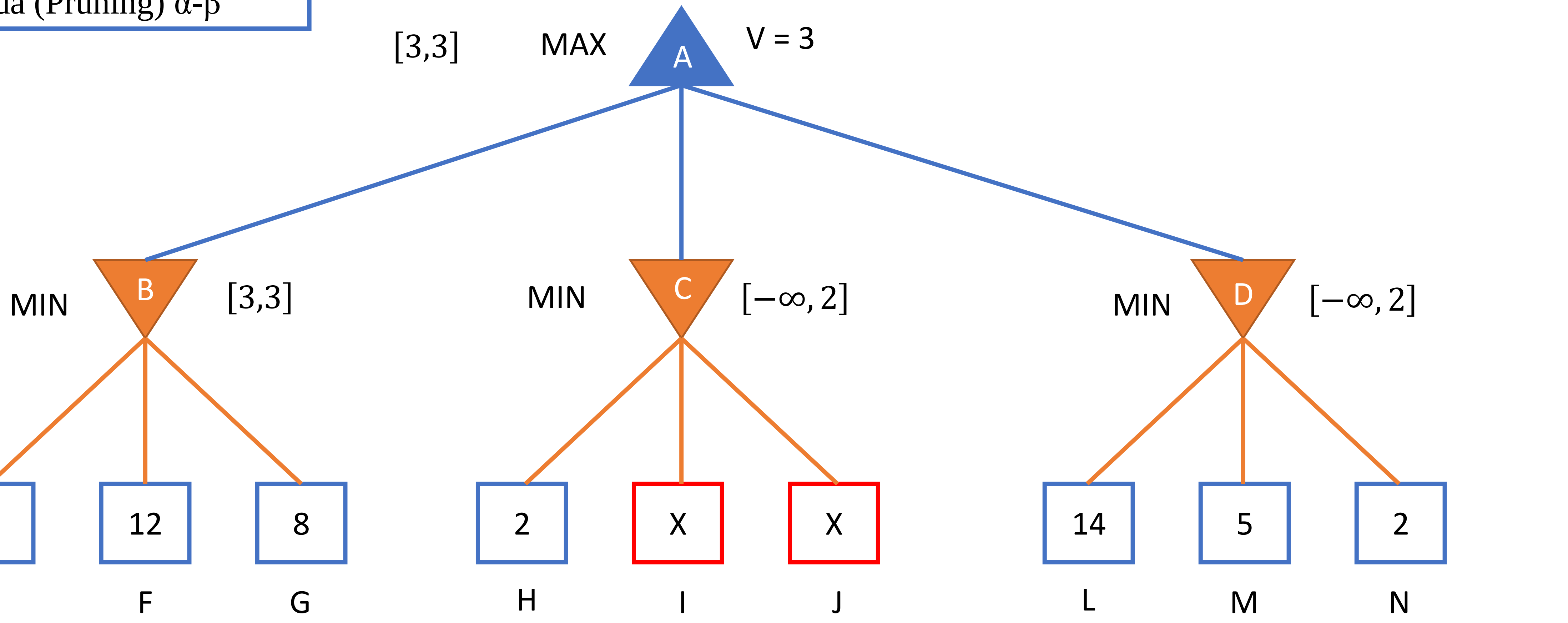
MiniMax Algorithm

Poda (Pruning) α-β



MiniMax Algorithm

Poda (Pruning) α-β



Naïve Bayes

Advantages

- Naïve Bayes based on the independence assumption

Training is very easy and fast; just requiring considering each attribute in each class separately

Test is straightforward; just looking up tables or calculating conditional probabilities with normal distributions

- A popular model

Performance competitive to most of state-of-the-art classifiers even in presence of violating independence assumption

Many successful applications, e.g., spam mail filtering.

Issues

- Violation of Independence Assumption
- Zero conditional probability Problem

Naïve Bayes

Play Tennis

PlayTennis: training examples

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Naïve Bayes

Play Tennis

PlayTennis: training examples

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Outlook	Play=*Yes*	Play=*No*
Sunny	2/9	3/5
Overcast	4/9	0/5
Rain	3/9	2/5

Play=*Yes – 9*

Play=*No – 5*

Naïve Bayes

Play Tennis

Outlook	Play=*Yes*	Play=*No*
Sunny	2/9	3/5
Overcast	4/9	0/5
Rain	3/9	2/5

Temperature	Play=*Yes*	Play=*No*
Hot	2/9	2/5
Mild	4/9	2/5
Cool	3/9	1/5

Humidity	Play=*Yes*	Play=No
High	3/9	4/5
Normal	6/9	1/5

Wind	Play=*Yes*	Play=*No*
Strong	3/9	3/5
Weak	6/9	2/5

$P(\text{Play}=Yes) = 9/14$ $P(\text{Play}=No) = 5/14$

Naïve Bayes

Play Tennis

Given a new instance,

$\mathbf{x}'$=(Outlook=*Sunny,* Temperature=*Cool,* Humidity=*High,* Wind=*Strong*)

Naïve Bayes

Play Tennis

Given a new instance,

$\mathbf{x}'$=(Outlook=*Sunny,* Temperature=*Cool,* Humidity=*High,* Wind=*Strong*)

– Look up tables

P(Outlook=*Sunny*|Play=*Yes*) = 2/9

P(Temperature=*Cool*|Play=*Yes*) = 3/9

P(Huminity=*High*|Play=*Yes*) = 3/9

P(Wind=*Strong*|Play=*Yes*) = 3/9

P(Play=*Yes*) = 9/14

P(Outlook=*Sunny*|Play=*No*) = 3/5

P(Temperature=*Cool*|Play==*No*) = 1/5

P(Huminity=*High*|Play=*No*) = 4/5

P(Wind=*Strong*|Play=*No*) = 3/5

P(Play=*No*) = 5/14

Naïve Bayes

Play Tennis

Given a new instance,

$\mathbf{x}'$=(Outlook=*Sunny*, Temperature=*Cool*, Humidity=*High*, Wind=*Strong*)

- Look up tables

P(Outlook=*Sunny*|Play=*Yes*) = 2/9

P(Temperature=*Cool*|Play=*Yes*) = 3/9

P(Huminity=*High*|Play=*Yes*) = 3/9

P(Wind=*Strong*|Play=*Yes*) = 3/9

P(Play=*Yes*) = 9/14

P(Outlook=*Sunny*|Play=*No*) = 3/5

P(Temperature=*Cool*|Play==*No*) = 1/5

P(Huminity=*High*|Play=*No*) = 4/5

P(Wind=*Strong*|Play=*No*) = 3/5

P(Play=*No*) = 5/14

- MAP rule

$P(Yes|\mathbf{x}')$: $[P(Sunny|Yes)P(Cool|Yes)P(High|Yes)P(Strong|Yes)]P(Play=Yes) = 0.0053$

$P(No|\mathbf{x}')$: $[P(Sunny|No)\,P(Cool|No)P(High|No)P(Strong|No)]P(Play=No) = 0.0206$

Naïve Bayes

Play Tennis

Given a new instance,

$\mathbf{x}'$=(Outlook=*Sunny*, Temperature=*Cool*, Humidity=*High*, Wind=*Strong*)

- Look up tables

P(Outlook=*Sunny*|Play=*Yes*) = 2/9	P(Outlook=*Sunny*|Play=*No*) = 3/5
P(Temperature=*Cool*|Play=*Yes*) = 3/9	P(Temperature=*Cool*|Play==*No*) = 1/5
P(Huminity=*High*|Play=*Yes*) = 3/9	P(Huminity=*High*|Play=*No*) = 4/5
P(Wind=*Strong*|Play=*Yes*) = 3/9	P(Wind=*Strong*|Play=*No*) = 3/5
P(Play=*Yes*) = 9/14	P(Play=*No*) = 5/14

- MAP rule

P(*Yes*|$\mathbf{x}'$): [P(*Sunny*|Yes)P(*Cool*|Yes)P(*High*|Yes)P(*Strong*|Yes)]P(Play=*Yes*) = 0.0053

P(*No*|$\mathbf{x}'$): [P(*Sunny*|No) P(*Cool*|No)P(*High*|No)P(*Strong*|No)]P(Play=*No*) = 0.0206

Given the fact P(*Yes*|$\mathbf{x}'$) < P(*No*|$\mathbf{x}'$), we label $\mathbf{x}'$ to be "*No*".

Genetic Algorithm

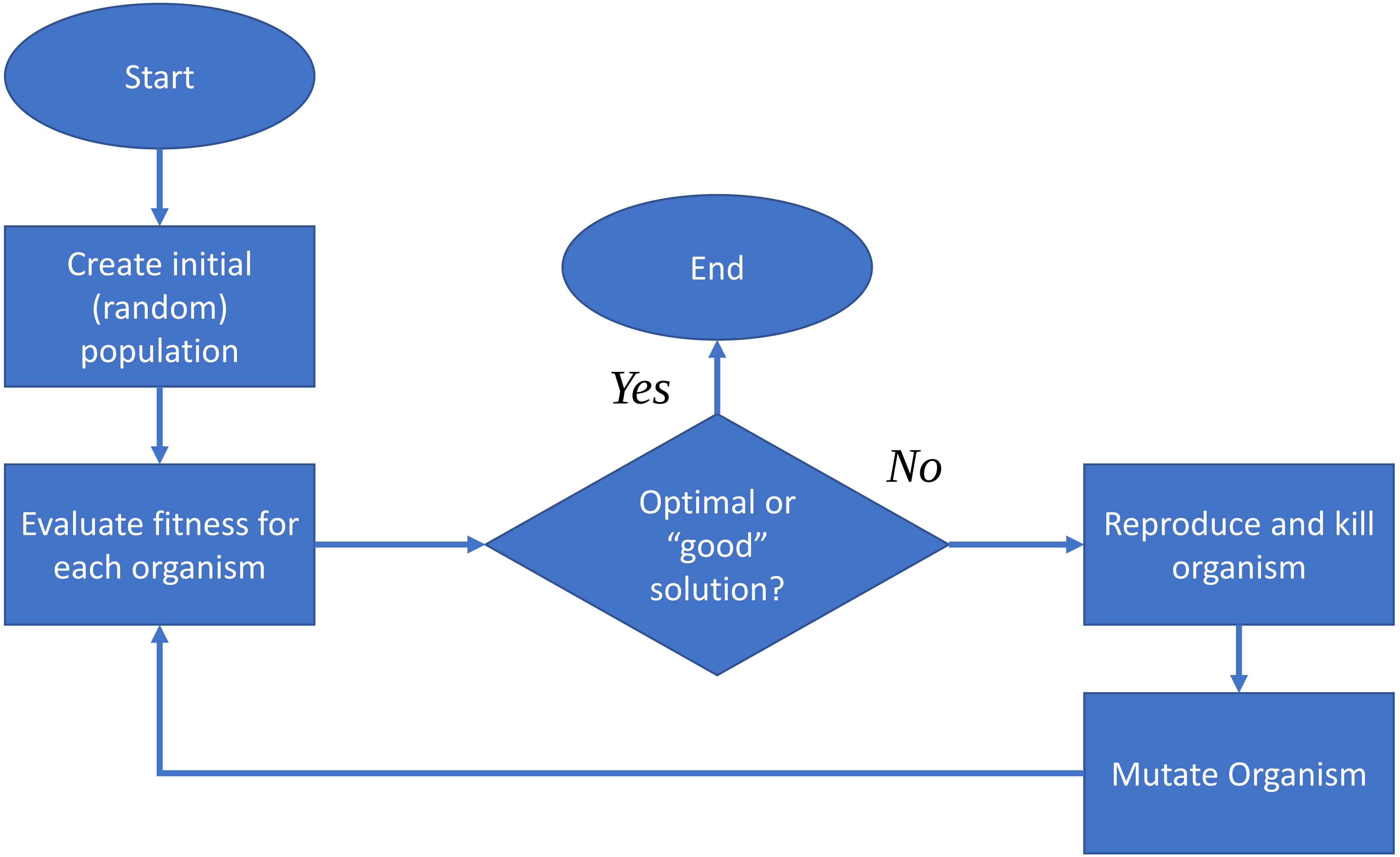
What is it?

- Genetic Algorithms are a class of procedures, with well-defined distinct steps.
- This class is based on analogies to biological concepts already tested to exhaustion.
- Each distinct step can have several different versions.

What are they good for?

- Search and Optimization
- Widely used, with success, in problems that are difficult to handle using traditional techniques
- Efficiency X Flexibility

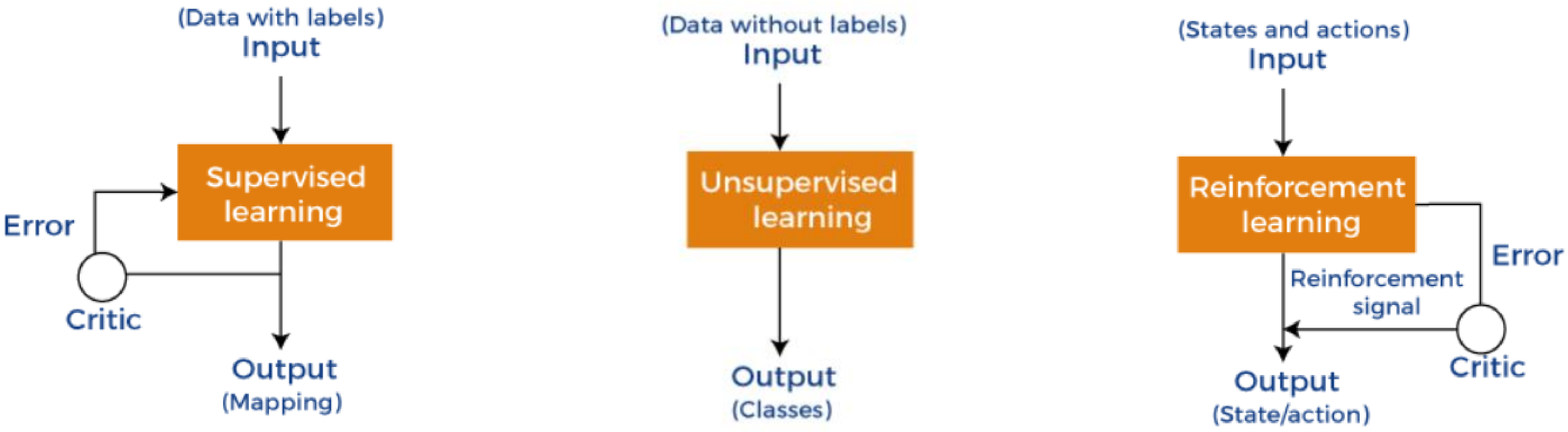
Genetic Algorithm



Metrics

Machine Learning Models

Reference: Input (Dataset)



Metrics

Machine Learning Models

Classification Models

Regression Models

Reference: Output

Classificação: É quando o objetivo é atribuir um rótulo a uma entrada.

EXEMPLO:

Imagine que queremos classificar personagens do jogo em "aliados" ou "inimigos" com base em características como o comportamento e a aparência. Usamos a classificação para prever a categoria à qual cada personagem pertence, ajudando o jogador a diferenciar oponentes de aliados em tempo real.

Regressão: É usada para prever um valor contínuo.

EXEMPLO:

Suponha que queremos prever a pontuação de um jogador com base em ações específicas realizadas no jogo. Utilizando um modelo de regressão, podemos estimar a pontuação futura, ajudando a criar sistemas de progressão e feedback para o jogador.

Metrics

Accuracy

Accuracy mede o quão próximo o resultado está do valor real que você estava tentando alcançar. Em outras palavras, é o quão perto você atinge o que almeja.

Accuracy pode ser usada em uma instancia.

Precision

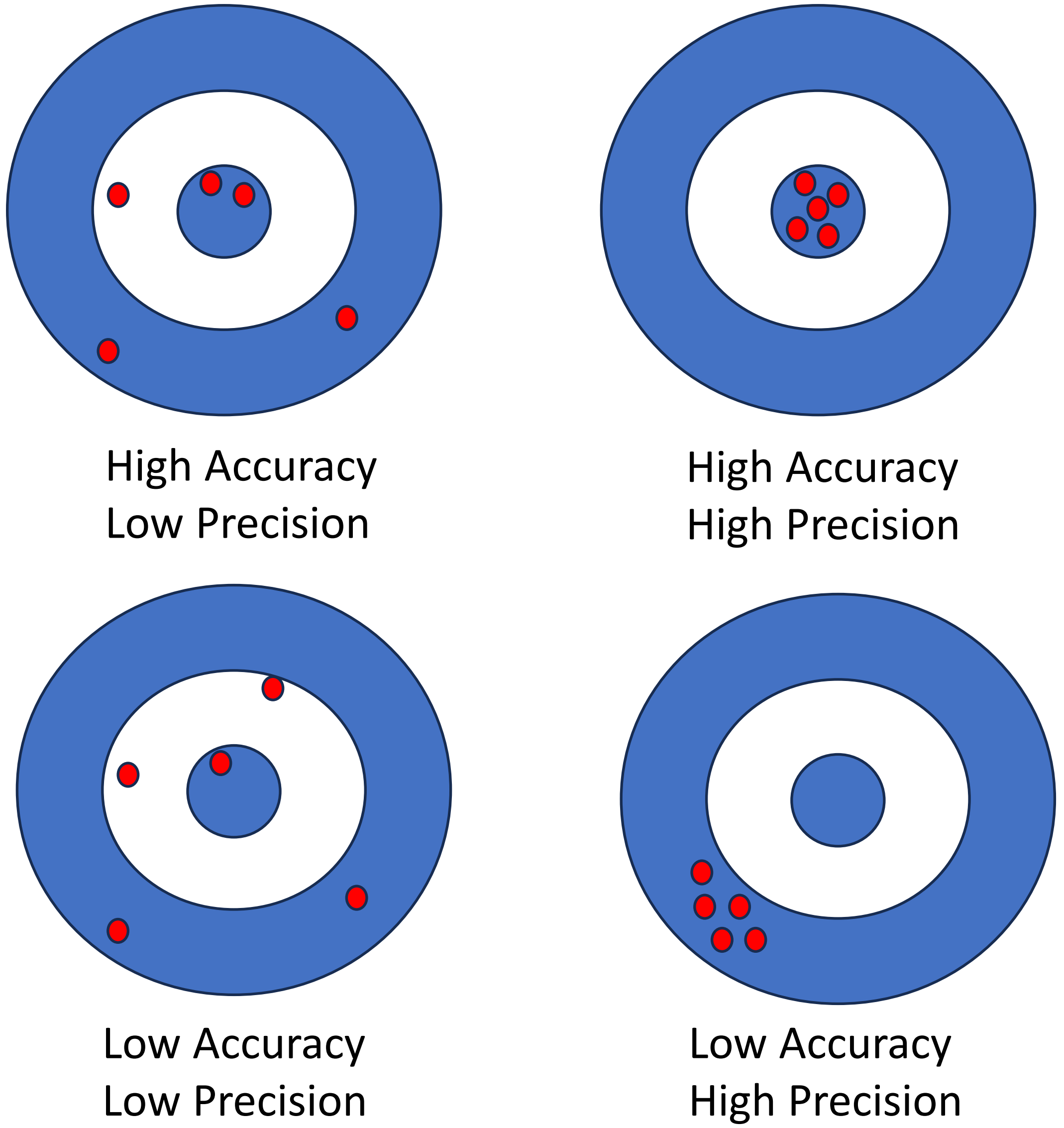
Precision mede a proximidade entre seus resultados.

Precision é usada ao longo do tempo

Metrics

$$Accurancy = \frac{TP}{TP + FP + FN + TN}$$

$$Precision = \frac{TP}{TP + FP}$$

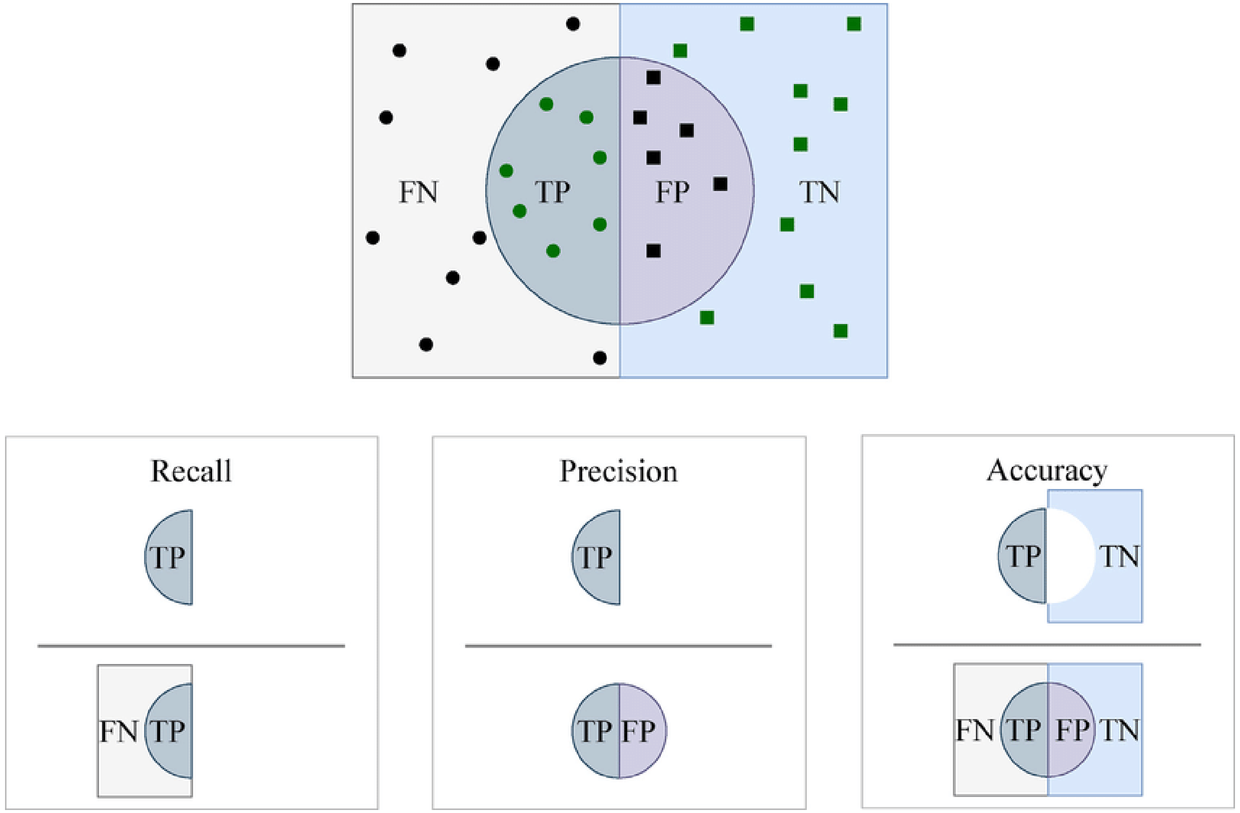


Metrics

$$Accurancy = \frac{TP}{TP + FP + FN + TN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$



Metrics

$$Accurancy = \frac{TP}{TP + FP + FN + TN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

- **F1 Score:**
  - **Definição:** é uma métrica de avaliação que combina as métricas de precision e recall em um único número, fornecendo uma medida geral do desempenho de um modelo.
  - **Foco:** é particularmente útil para encontrar um equilíbrio entre a precision e a capacidade de recuperar todos os casos positivos (recall). O F1 Score é calculado pela média harmônica da precision e recall.

- O F1 Score varia de 0 a 1, onde 1 indica um modelo perfeito que atinge tanto alta precisão quanto alta revocação.
- É especialmente útil quando as consequências de falsos positivos e falsos negativos são críticas e você deseja encontrar um equilíbrio entre esses dois tipos de erros.
- É amplamente utilizado em problemas de classificação binária, como detecção de spam, diagnóstico médico, ou classificação de sentimentos.

$$F1\ score = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{TP}{TP + \frac{FP}{2} + \frac{FN}{2}}$$

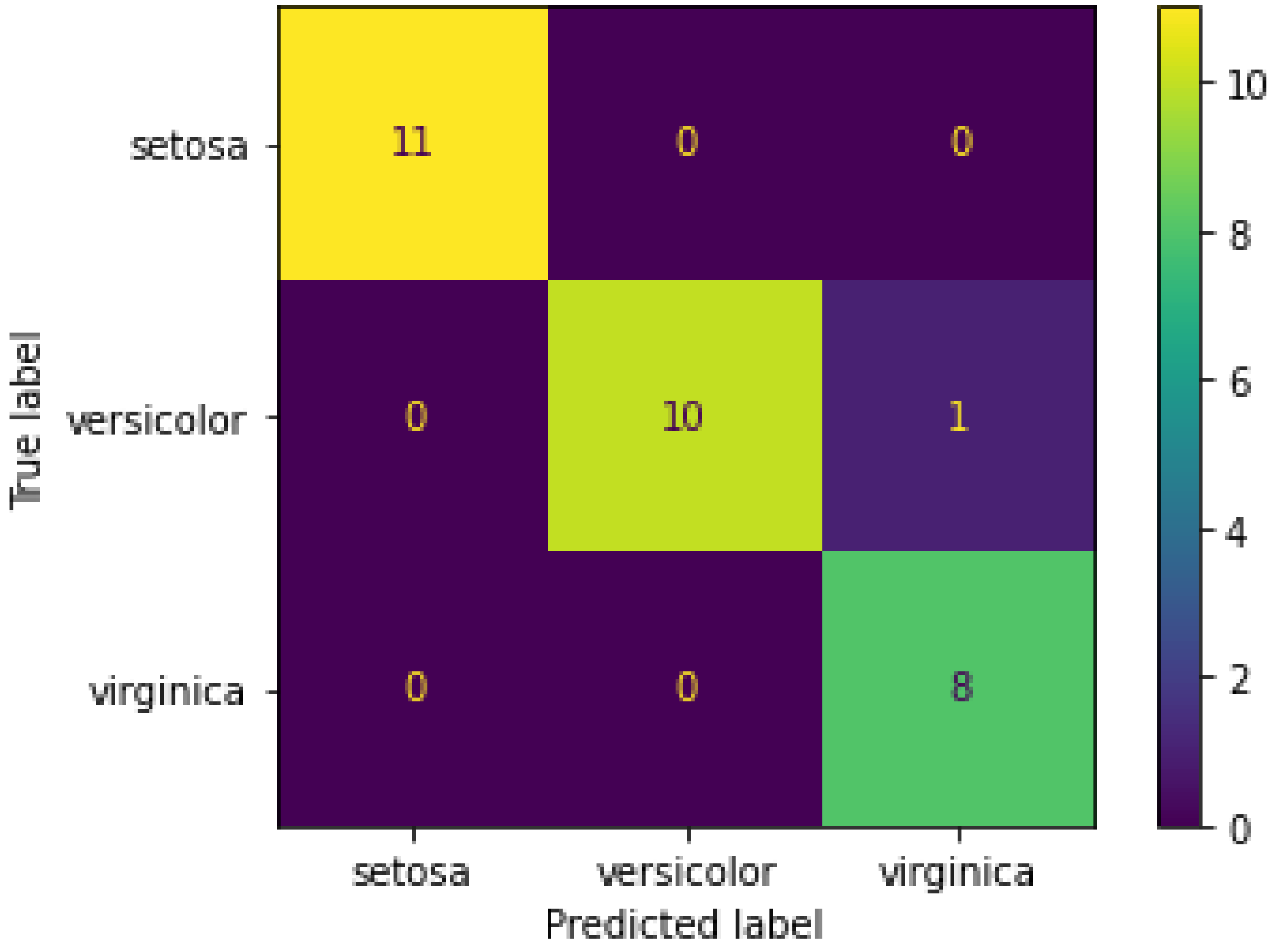
Metrics

Confusion Matrix

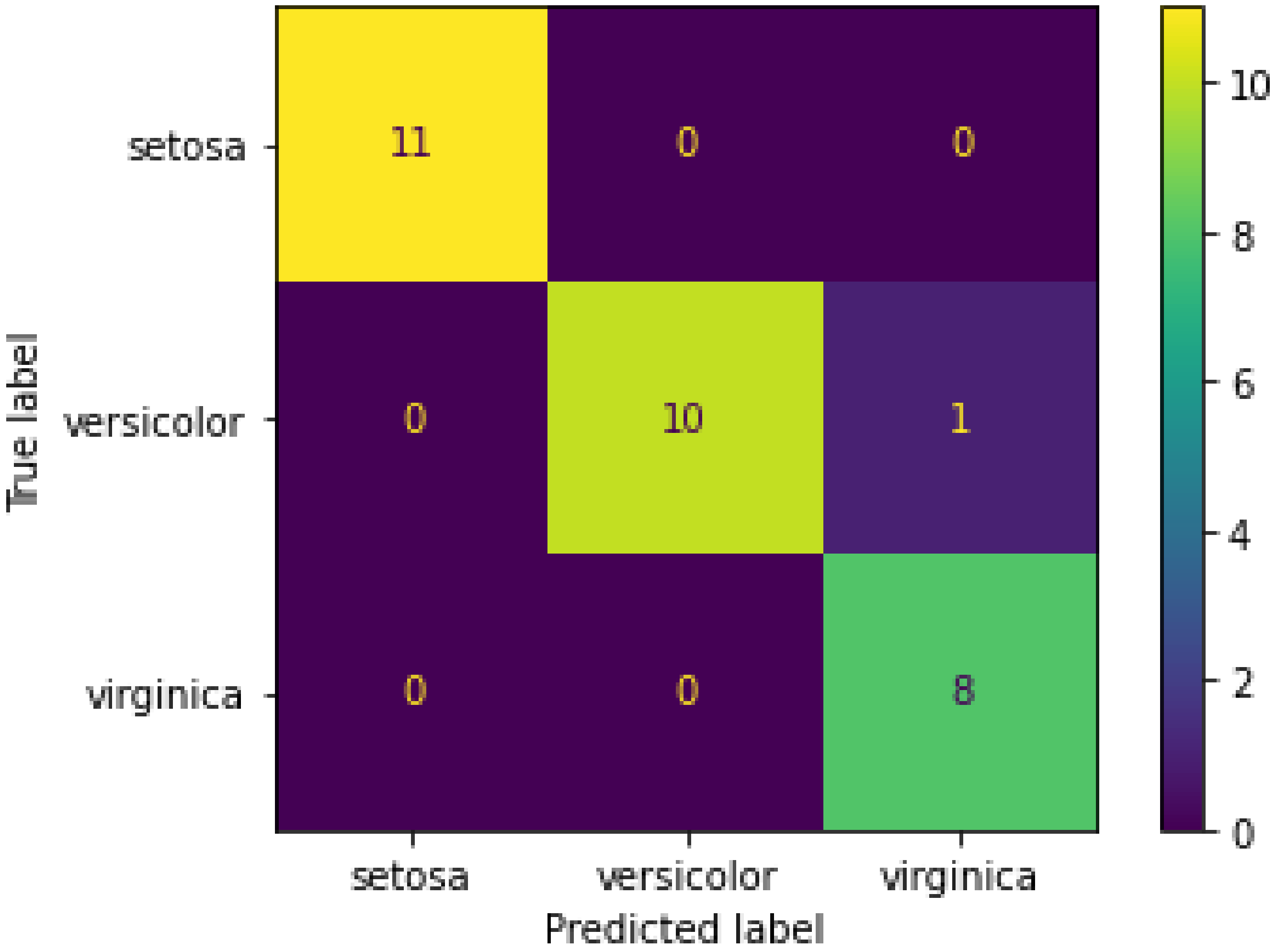
- 3 classes (setosa, versicolor, virginica)
- 30 test samples (test dataset):

  - ✓ Setosa – 11 samples (11 predicted)
  - ✓ Versicolor – 11 samples (10 predicted)
  - ✓ Virginica – 8 samples (9 predicted)

➢ 1 sample predicted Virginica => Versicolor (real)

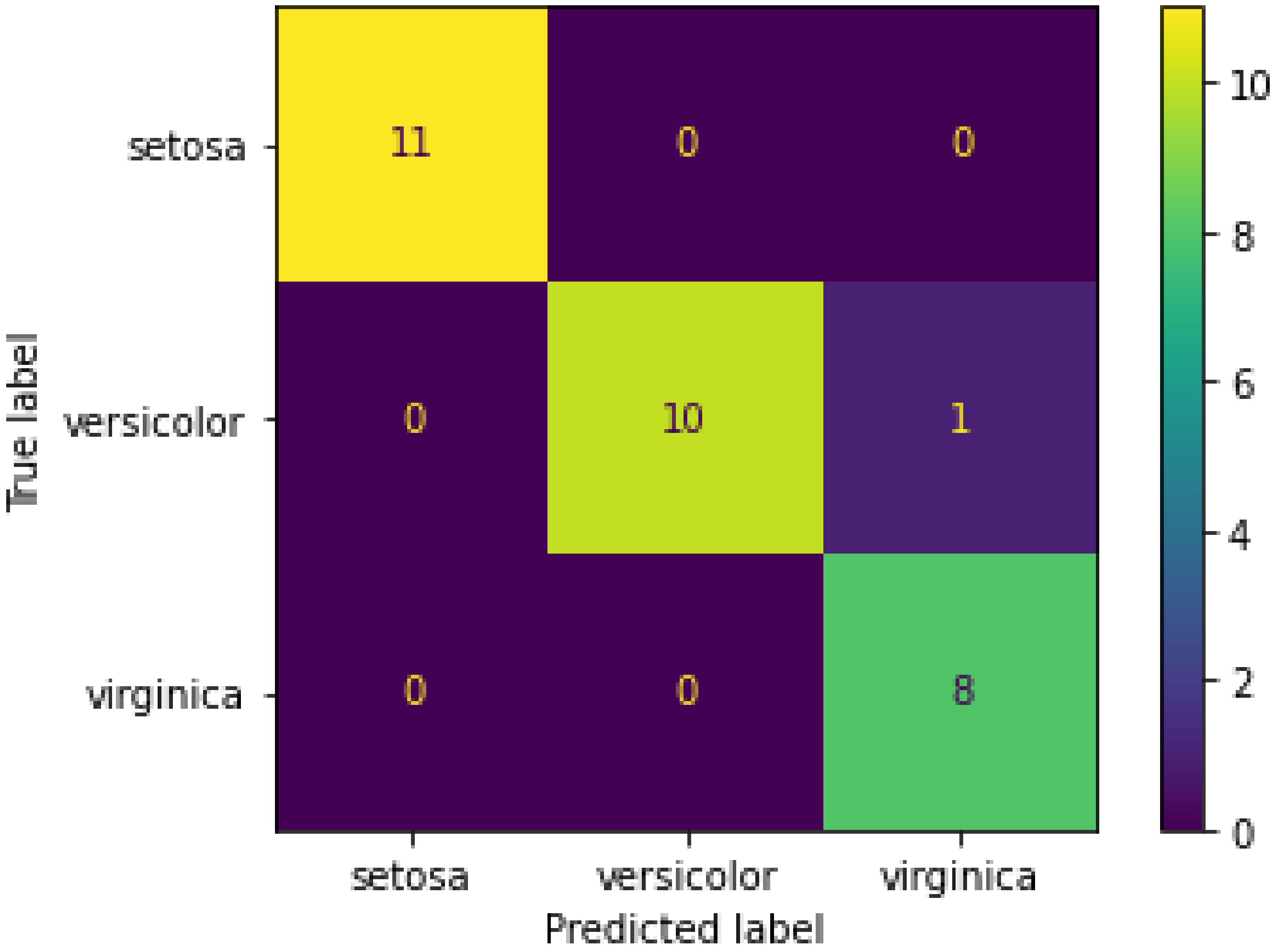


Metrics



$$Accurancy = \frac{11 + 10 + 8}{11 + 11 + 8} = \frac{29}{30} = 0.9667$$

Metrics



Setosa: TP = 11 | FP = 0 | FN = 0

Versicolor: TP = 10 | FP = 0 | FN = 1

Virginica: TP = 8 | FP = 1 | FN = 0

$$Precision_{setosa} = \frac{11}{11+0} = \frac{11}{11} = 1$$

$$Recall_{setosa} = \frac{11}{11+0} = \frac{11}{11} = 1$$

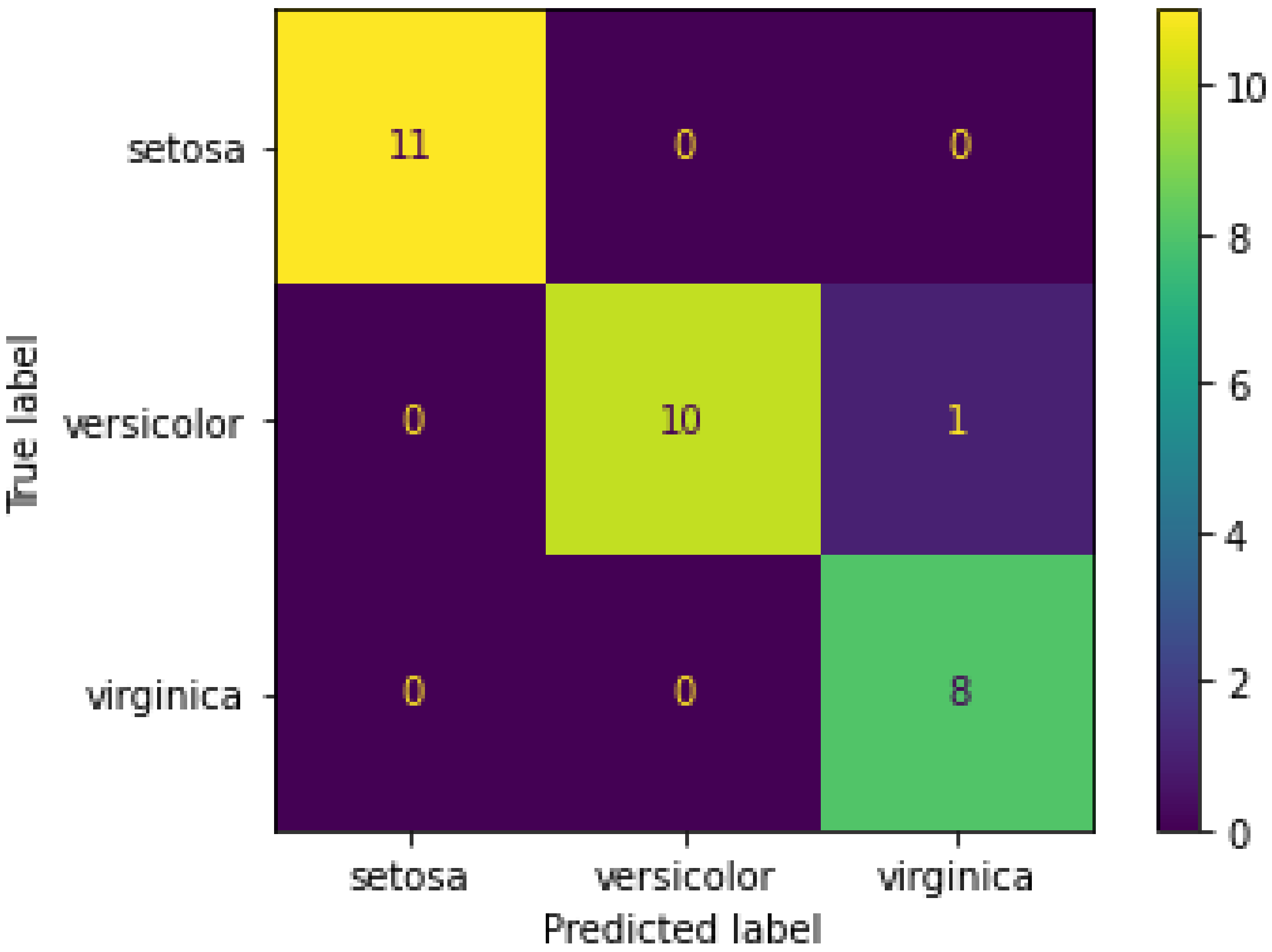
$$Precision_{versicolor} = \frac{10}{10+0} = \frac{10}{10} = 1$$

$$Recall_{versicolor} = \frac{10}{10+1} = \frac{10}{11} = 0.9091$$

$$Precision_{virginica} = \frac{8}{8+1} = \frac{8}{9} = 0.8889$$

$$Recall_{virginica} = \frac{8}{8+0} = \frac{8}{8} = 1$$

Metrics



Setosa: TP = 11 | FP = 0 | FN = 0

Versicolor: TP = 10 | FP = 0 | FN = 1

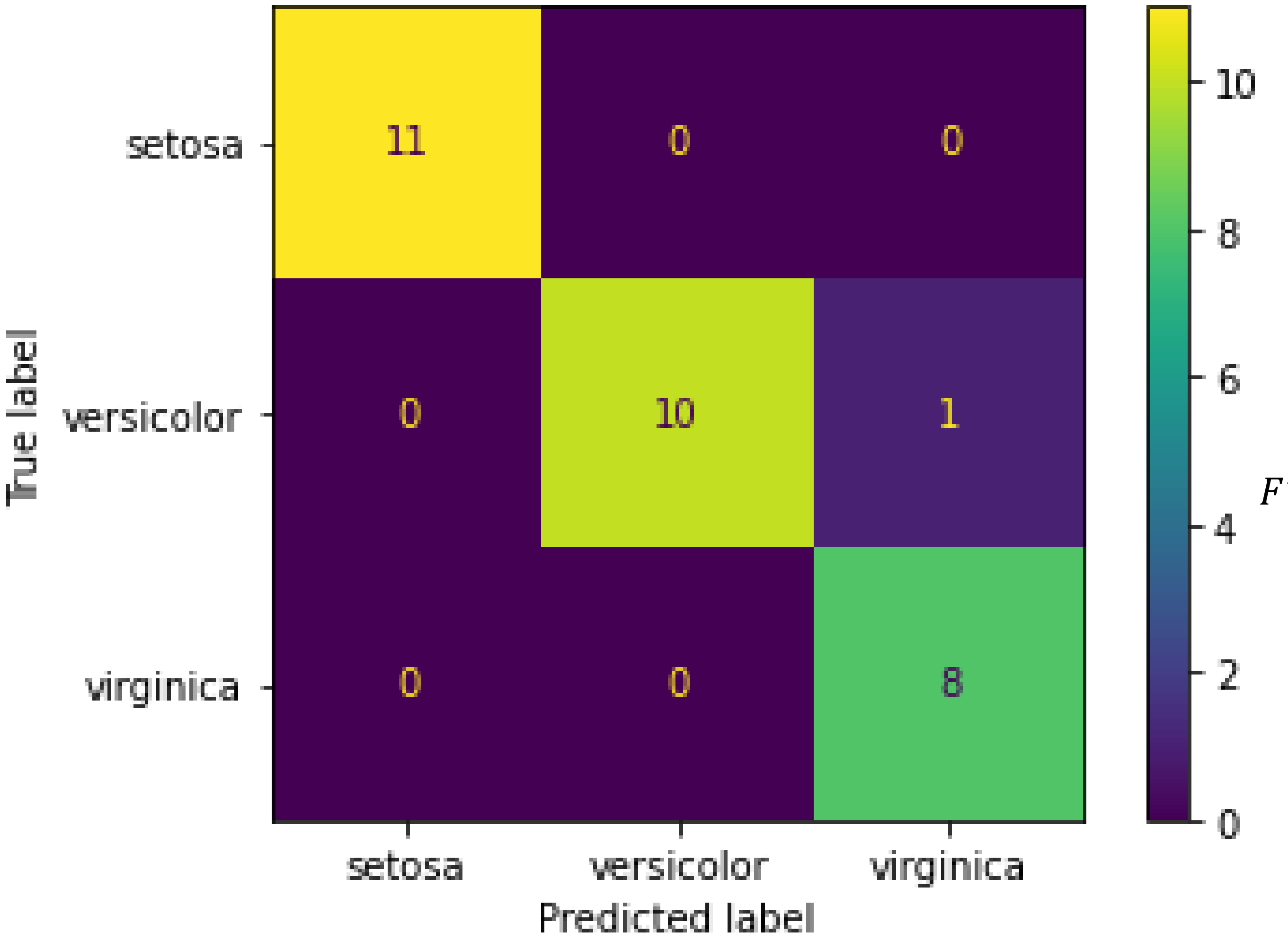
Virginica: TP = 8 | FP = 1 | FN = 0

$$\begin{cases} F1_{setosa} = \frac{2 * 1 * 1}{1 + 1} = \frac{2}{2} = 1 \\ W_{setosa} = \frac{11}{30} = 0.3666 \end{cases}$$

$$\begin{cases} F1_{versicolor} = \frac{2 * 1 * 0.9091}{1 + 0.9091} = \frac{1.8182}{1.9091} = 0.9524 \\ W_{versicolor} = \frac{11}{30} = 0.3666 \end{cases}$$

$$\begin{cases} F1_{virginica} = \frac{2 * 0,8889 * 1}{0,8889 + 1} = \frac{1.7778}{1.8889} = 0.9412 \\ W_{virginica} = \frac{8}{30} = 0.2667 \end{cases}$$

Metrics



Setosa: TP = 11 | FP = 0 | FN = 0

Versicolor: TP = 10 | FP = 0 | FN = 1

Virginica: TP = 8 | FP = 1 | FN = 0

$$F1\ score = \sum_{i=1}^{n} W_i * F1_i$$

$$F1\ score = (1 * 0.3666) + (0.9524*0.3666)+(0.9412*0.2667)$$

$$F1\ score = 0.9668$$

Daniel Nogueira

dnogueira@ipca.pt