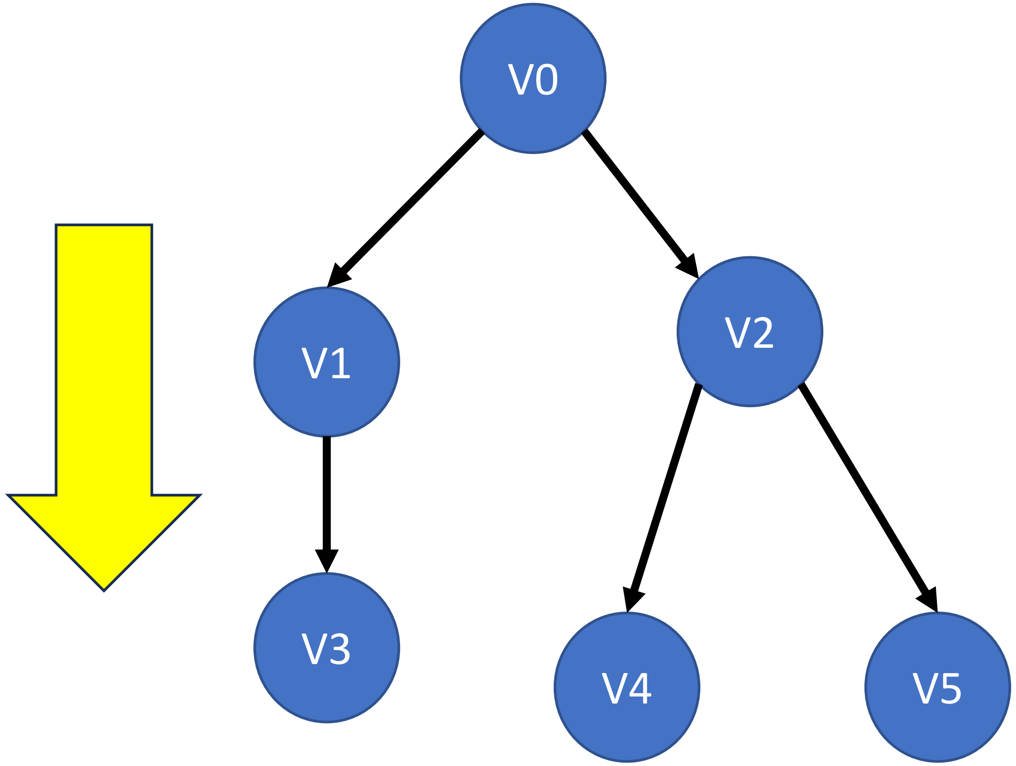
Path-finding

Daniel Nogueira

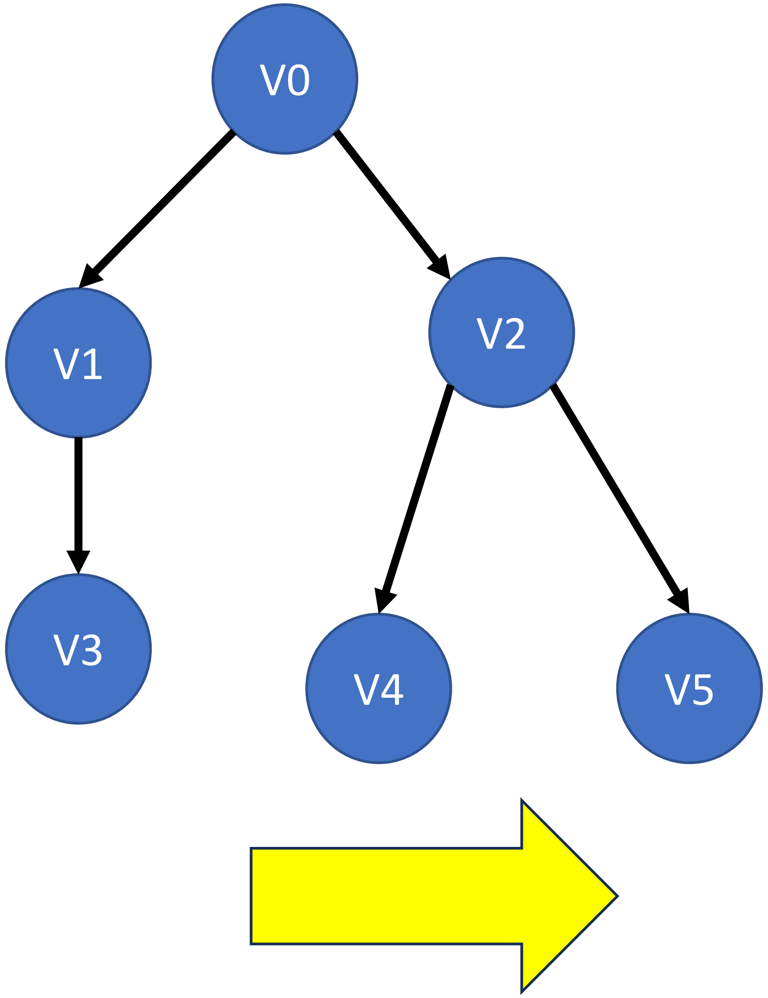
dnogueira@ipca.pt

Algorithms

Depth-First Search (DFS)

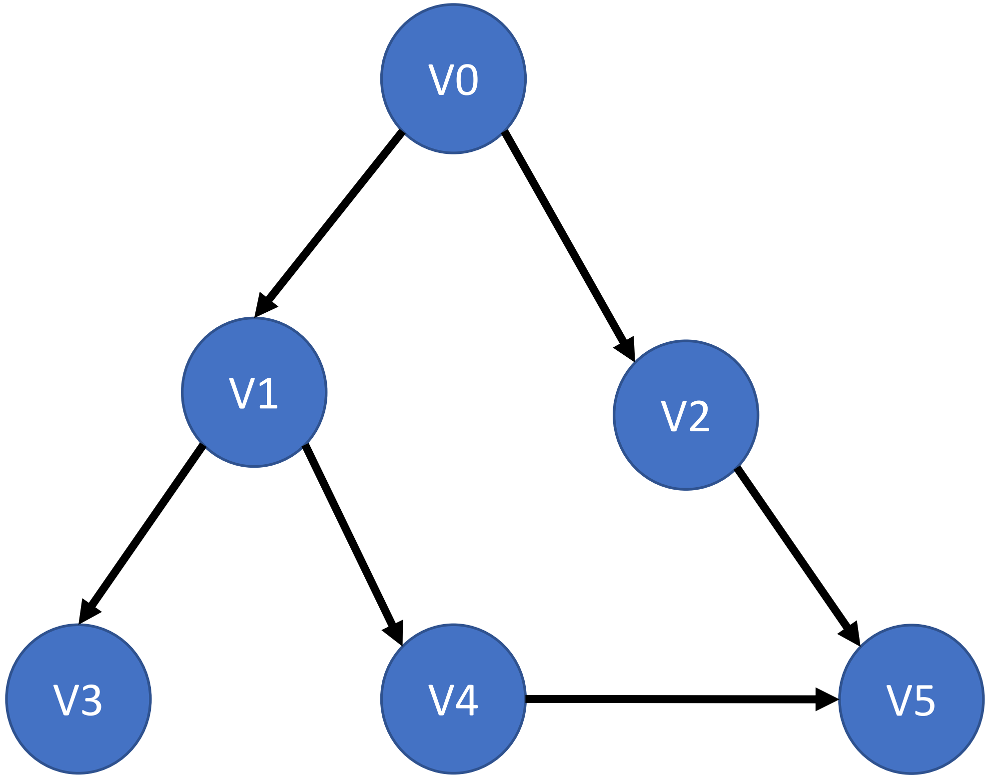


Breadth-First Search (BFS)



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

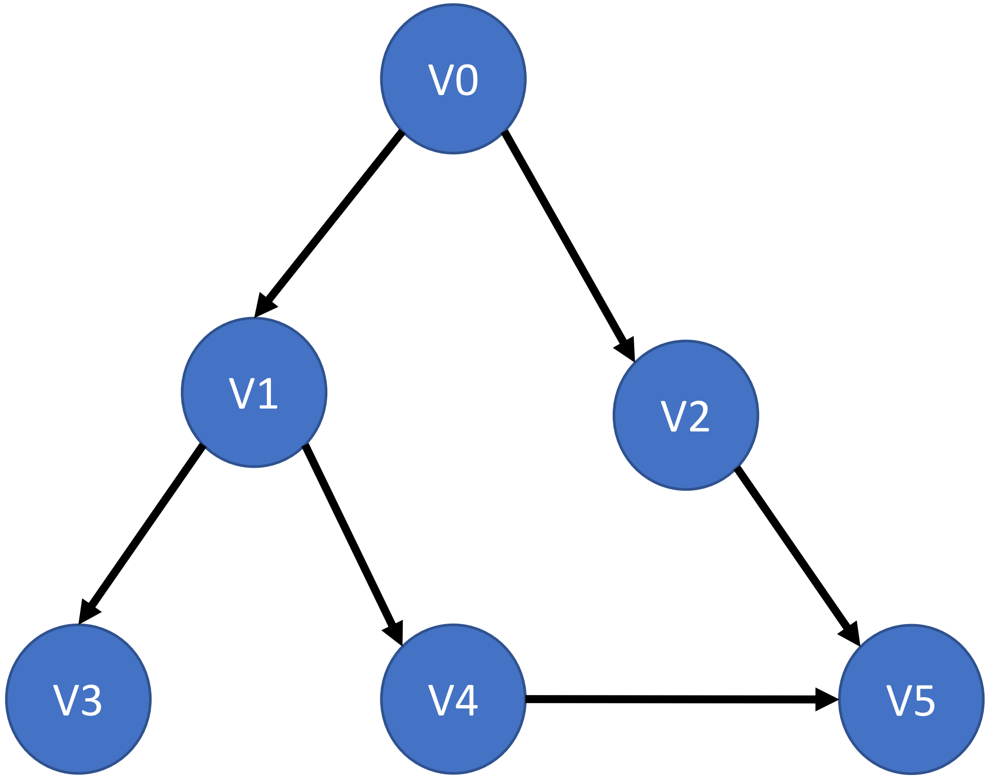
Depth-First Search (DFS)

```
using System;
using System.Collections.Generic;

class Program
{
    static Dictionary<string, List<string>> graph = new Dictionary<string, List<string>>
    {
        { "V0", new List<string> { "V1", "V2" } },
        { "V1", new List<string> { "V3", "V4" } },
        { "V2", new List<string> { "V5" } },
        { "V3", new List<string>() },
        { "V4", new List<string> { "V5" } },
        { "V5", new List<string>() }
    };

    static HashSet<string> visited = new HashSet<string>();

    static void Main(string[] args)
    {
        string startNode = "V0";

        Console.WriteLine("Following is the Depth-First Search:");
        DFS(startNode);

        Console.ReadLine();
    }

    static void DFS(string node)
    {
        if (!visited.Contains(node))
        {
            Console.WriteLine(node);
            visited.Add(node);

            if (graph.ContainsKey(node))
            {
                foreach (string neighbor in graph[node])
                {
                    DFS(neighbor);          }         }      }   }   }
```

```
private void DFSRecursive(string vertex, HashSet<string> visited)
    {
        visited.Add(vertex);
        Console.WriteLine("Visiting vertex: " + vertex);

        foreach (string neighbor in adjacencyList[vertex])
        {
            if (!visited.Contains(neighbor))
            {
                DFSRecursive(neighbor, visited);
            }       }    }  }
class Program
{
    static void Main(string[] args)
    {
        Graph graph = new Graph();

        // Adicionar vértices
        graph.AddVertex("V0");
        graph.AddVertex("V1");
        graph.AddVertex("V2");
        graph.AddVertex("V3");
        graph.AddVertex("V4");
        graph.AddVertex("V5");

        // Adicionar arestas
        graph.AddEdge("V0", "V1");
        graph.AddEdge("V0", "V2");
        graph.AddEdge("V1", "V3");
        graph.AddEdge("V1", "V4");
        graph.AddEdge("V2", "V5");
        graph.AddEdge("V4", "V5");

        Console.WriteLine("DFS starting from vertex V0:");
        graph.DFS("V0");    } }
```

Algorithms

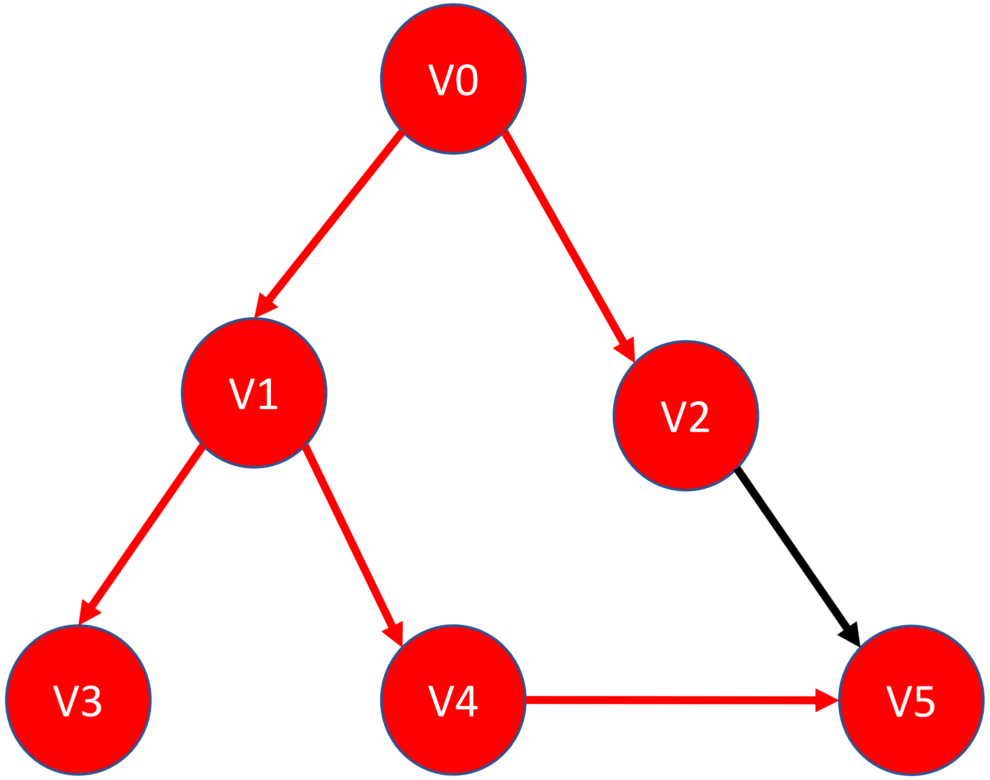
Depth-First Search (DFS)




```
def dfs(graph, vertex, visited):
    # Marcar o vértice como visitado
    visited[vertex] = True
    print("Visitando vértice:", vertex)

    # Recursivamente visitar os vértices adjacentes não visitados
    for neighbor in graph[vertex]:
        if not visited[neighbor]:
            dfs(graph, neighbor, visited)

# Grafo representado como um dicionário de adjacências
graph = {
    'V0': ['V1', 'V2'],
    'V1': ['V3', 'V4'],
    'V2': ['V5'],
    'V3': [],
    'V4': ['V5'],
    'V5': []
}

# Inicializar um vetor de visitados
visited = {vertex: False for vertex in graph}

# Chamar o DFS a partir de todos os vértices não visitados
for vertex in graph:
    if not visited[vertex]:
        dfs(graph, vertex, visited)
```

Algorithms

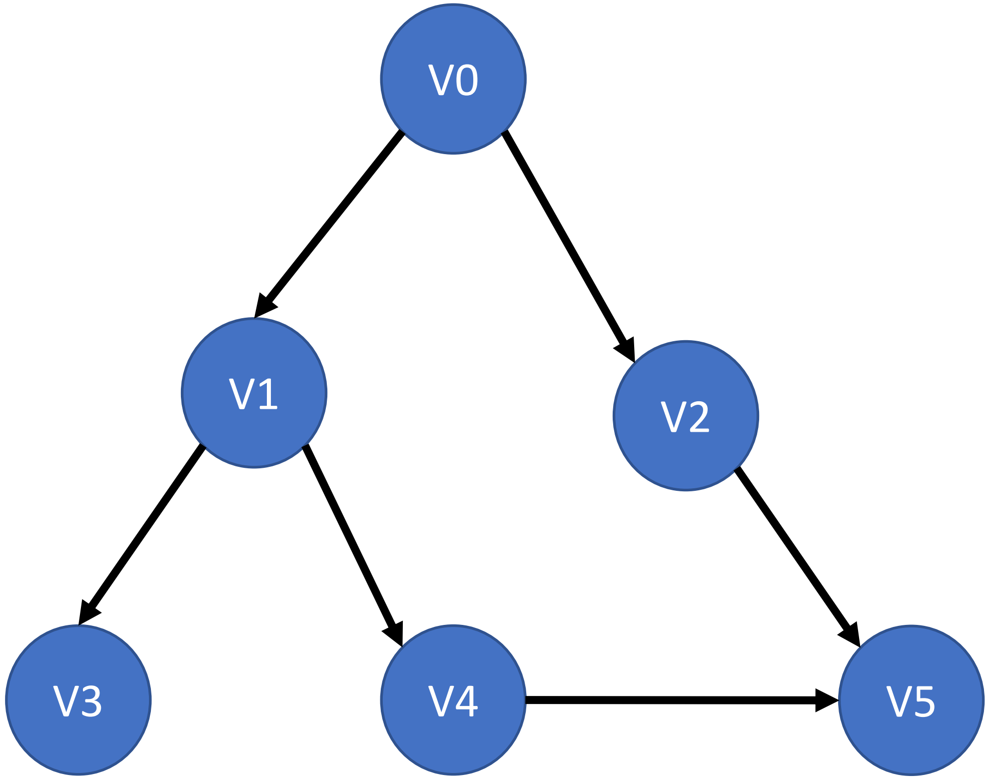
Depth-First Search (DFS)




```
Following is the Depth-First Search
V0
V1
V3
V4
V5
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

Algorithms

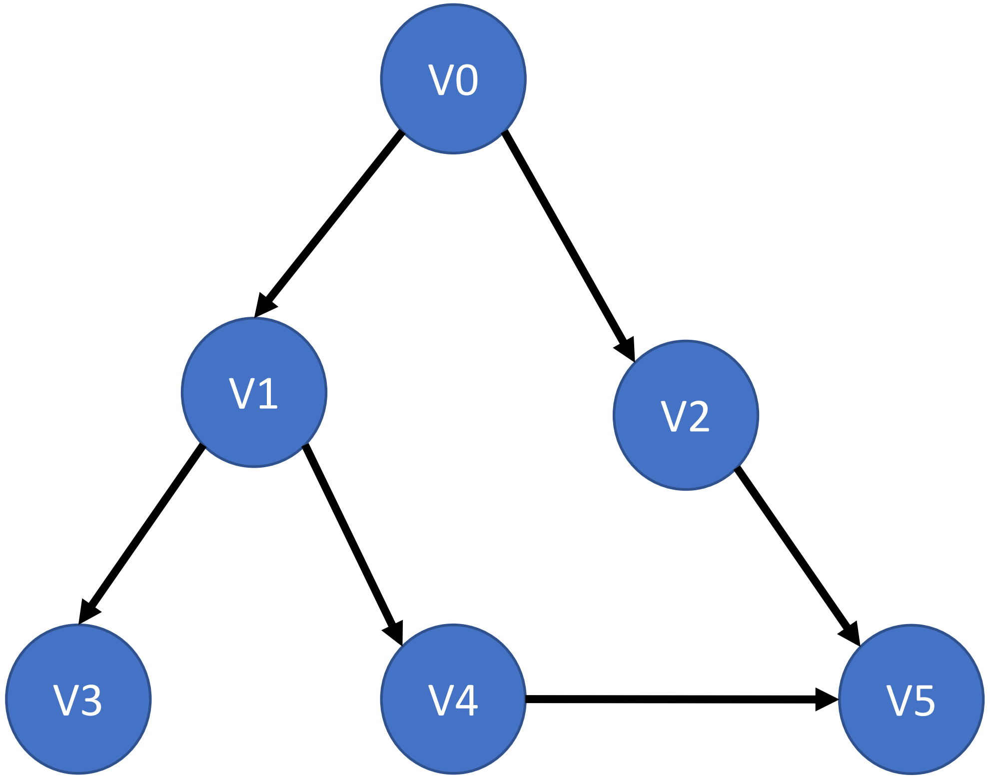
Breadth-First Search (BFS)

```
using System;
using System.Collections.Generic;

class Program
{
   static Dictionary<string, List<string>> graph = new Dictionary<string, List<string>>
   {
      { "V0", new List<string> { "V1", "V2" } },
      { "V1", new List<string> { "V3", "V4" } },
      { "V2", new List<string> { "V5" } },
      { "V3", new List<string>() },
      { "V4", new List<string> { "V5" } },
      { "V5", new List<string>() }
   };

   static List<string> visited = new List<string>();
   static Queue<string> queue = new Queue<string>();

   static void Main(string[] args)
   {
      string startNode = "V0";

      Console.WriteLine("Following is the Breadth-First Search:");
      BFS(startNode);

      Console.ReadLine();
   }
```

```
static void BFS(string node)
   {
      visited.Add(node);
      queue.Enqueue(node);

      while (queue.Count > 0)
      {
         string s = queue.Dequeue();
         Console.Write(s + " ");

         if (graph.ContainsKey(s))
         {
            foreach (string neighbor in graph[s])
            {
               if (!visited.Contains(neighbor))
               {
                  visited.Add(neighbor);
                  queue.Enqueue(neighbor);
               }
            }
         }
      }
   }
}
```

Algorithms

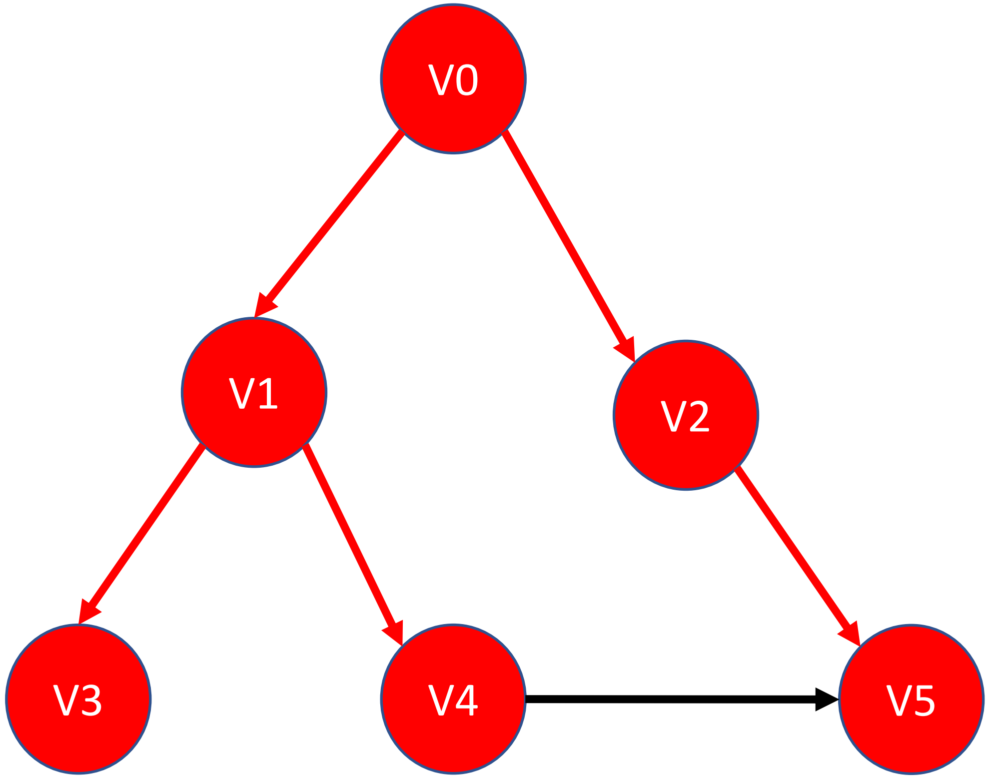
Breadth-First Search (BFS)

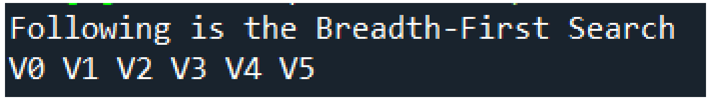



```python
graph = {
    'V0' : ['V1','V2'],
    'V1' : ['V3', 'V4'],
    'V2' : ['V5'],
    'V3' : [],
    'V4' : ['V5'],
    'V5' : []
}

visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    s = queue.pop(0)
    print (s, end = " ")

    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, 'V0')
```

Algorithms

Breadth-First Search (BFS)





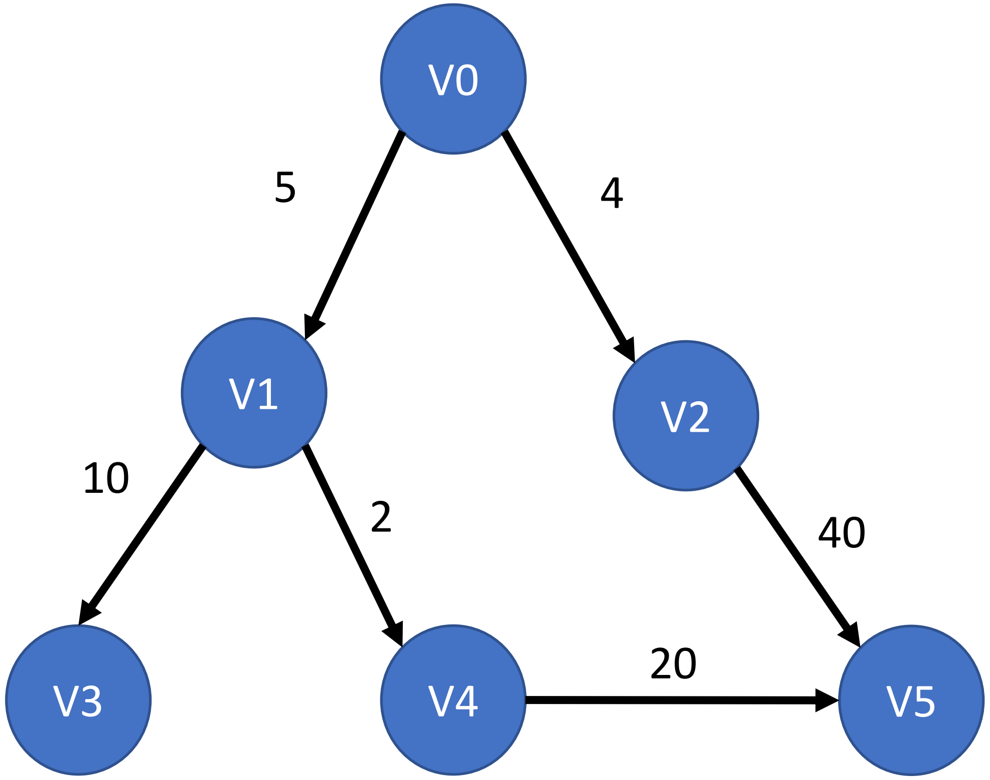
```
Following is the Breadth-First Search
V0 V1 V2 V3 V4 V5
```

DFS V0 V1 V3 V4 V5 V2

BFS V0 V1 V2 V3 V4 V5

Algorithms

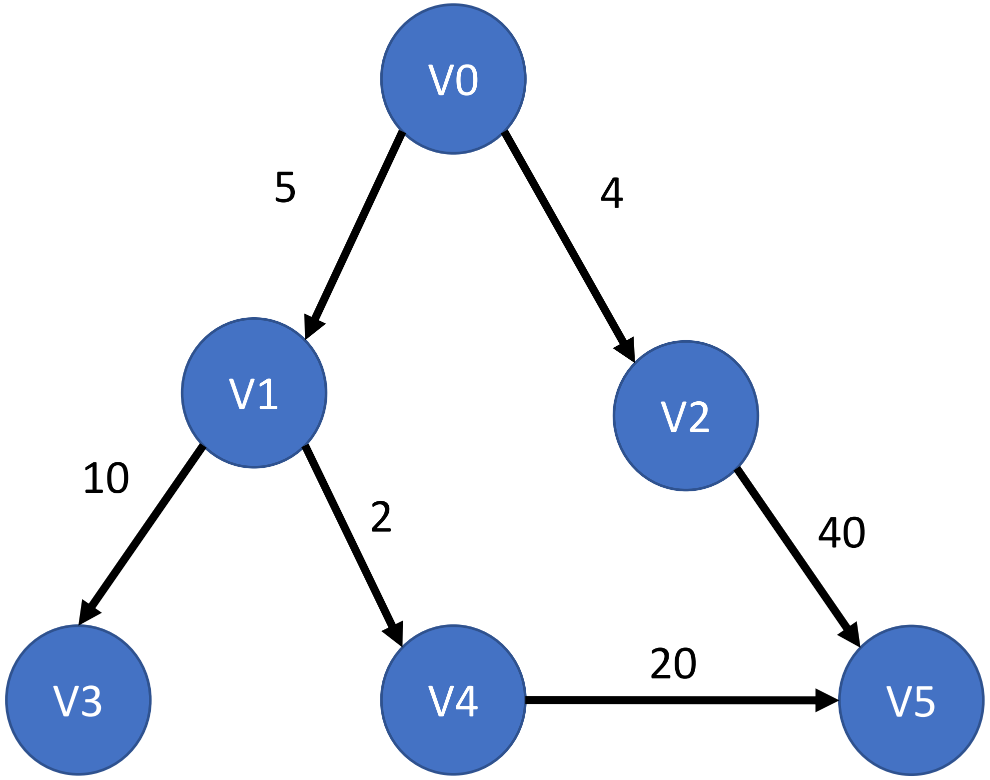
Dijkstra



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

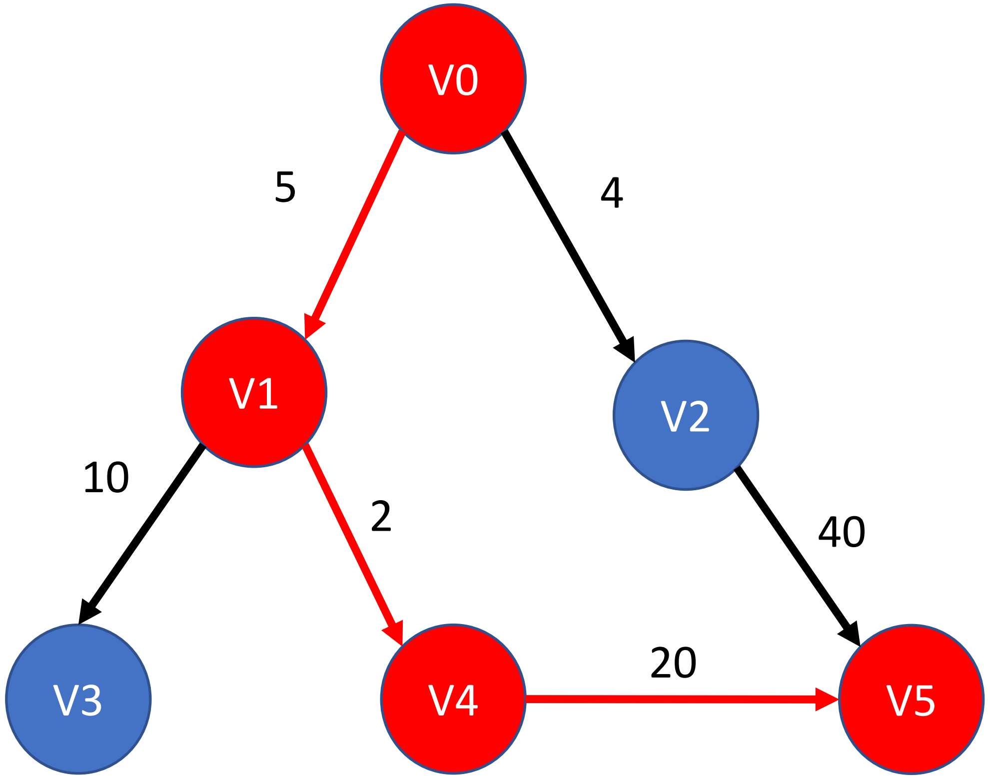
Dijkstra




```python
def dijkstra_algorithm(graph, start_node):
    unvisited_nodes = list(graph.get_nodes())
    shortest_path = {}
    previous_nodes = {}
    # We'll use max_value to initialize the "infinity" value of the unvisited nodes
    max_value = sys.maxsize
    for node in unvisited_nodes:
        shortest_path[node] = max_value
    # However, we initialize the starting node's value with 0
    shortest_path[start_node] = 0
    while unvisited_nodes:
        current_min_node = None
        for node in unvisited_nodes: # Iterate over the nodes
            if current_min_node == None:
                current_min_node = node
            elif shortest_path[node] < shortest_path[current_min_node]:
                current_min_node = node
        # The code block below retrieves the current node's neighbors and updates their distances
        neighbors = graph.get_outgoing_edges(current_min_node)
        for neighbor in neighbors:
            tentative_value = shortest_path[current_min_node] + graph.value(current_min_node, neighbor)
            if tentative_value < shortest_path[neighbor]:
                shortest_path[neighbor] = tentative_value
                # We also update the best path to the current node
                previous_nodes[neighbor] = current_min_node
        unvisited_nodes.remove(current_min_node)

    return previous_nodes, shortest_path
```

Algorithms

Dijkstra




```
We found the following best path with a value of 27.
V0 -> V1 -> V4 -> V5
```

Algorithms

Dijkstra

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        Dictionary<string, Dictionary<string, int>> initGraph = new Dictionary<string, Dictionary<string, int>>()
        {
            { "V0", new Dictionary<string, int> { { "V1", 5 }, { "V2", 4 } } },
            { "V1", new Dictionary<string, int> { { "V3", 10 }, { "V4", 2 } } },
            { "V2", new Dictionary<string, int> { { "V5", 40 } } },
            { "V3", new Dictionary<string, int>() },
            { "V4", new Dictionary<string, int> { { "V5", 20 } } },
            { "V5", new Dictionary<string, int>() }
        };

        Graph graph = new Graph(initGraph);
        string startNode = "V0";

        Dictionary<string, int> shortestPath = DijkstraAlgorithm(graph, startNode);
        PrintResult(shortestPath, startNode, "V5");

        Console.ReadLine();
    }
```

```
class Graph
{
    private Dictionary<string, Dictionary<string, int>> graph;

    public Graph(Dictionary<string, Dictionary<string, int>> initGraph)
    {
        graph = new Dictionary<string, Dictionary<string, int>>(initGraph);
    }

    public bool ContainsNode(string node)
    {
        return graph.ContainsKey(node);
    }

    public Dictionary<string, int> GetEdges(string node)
    {
        return graph[node];
    }
}
```

Algorithms

Dijkstra

```
static Dictionary<string, int> DijkstraAlgorithm(Graph graph, string startNode)
 {
    Dictionary<string, int> shortestPath = new Dictionary<string, int>();
    HashSet<string> visited = new HashSet<string>();

    foreach (var node in graph.GetEdges(startNode).Keys)
    {
       shortestPath[node] = int.MaxValue;
    }
    shortestPath[startNode] = 0;
    while (true)
    {
       string currentNode = null;
       int minDistance = int.MaxValue;

       foreach (var node in graph.GetEdges(startNode).Keys)
       {
          if (!visited.Contains(node) && shortestPath[node] < minDistance)
          {
             currentNode = node;
             minDistance = shortestPath[node];
          }
       }
       if (currentNode == null)
       {
          break;
       }
       visited.Add(currentNode);
       foreach (var kvp in graph.GetEdges(currentNode))
       {
          string neighbor = kvp.Key;
          int weight = kvp.Value;
          if (shortestPath[currentNode] + weight < shortestPath[neighbor])
          {
             shortestPath[neighbor] = shortestPath[currentNode] + weight;
          }
       }
    }
    return shortestPath;
 }
 static void PrintResult(Dictionary<string, int> shortestPath, string startNode, string endNode)
 {
    List<string> path = new List<string>();
    string currentNode = endNode;

    while (currentNode != startNode)
    {
       path.Add(currentNode);
       foreach (var kvp in shortestPath)
       {
          if (kvp.Key == currentNode)
          {
             currentNode = kvp.Value.ToString();
             break;
          }
       }
    }

    path.Add(startNode);
    path.Reverse();

    Console.WriteLine($"Shortest Path from {startNode} to {endNode} with a value of {shortestPath[endNode]}:");
    Console.WriteLine(string.Join(" -> ", path));
 }
}
```

Algorithms

A*

A* is an informed search algorithm. Informed Search signifies that the algorithm has extra information, to begin with. It is a complete as well as an optimal solution for solving path and grid problems.

Optimal – find the least cost from the starting point to the ending point. Complete – It means that it will find all the available paths from start to end.

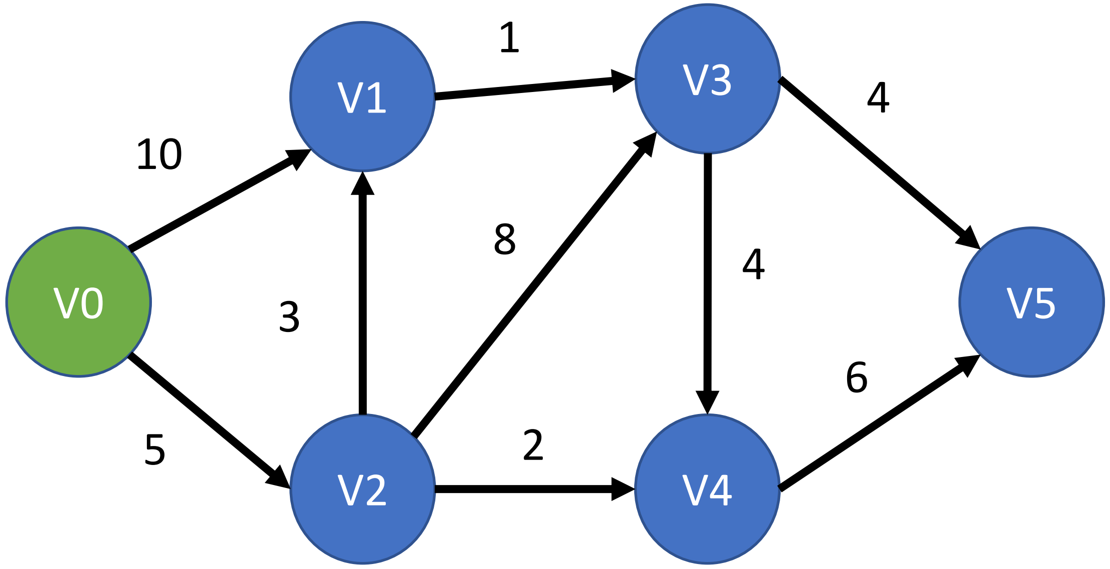
The Algorithm

$$f(n) = g(n) + h(n)$$

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.

Algorithms

A*

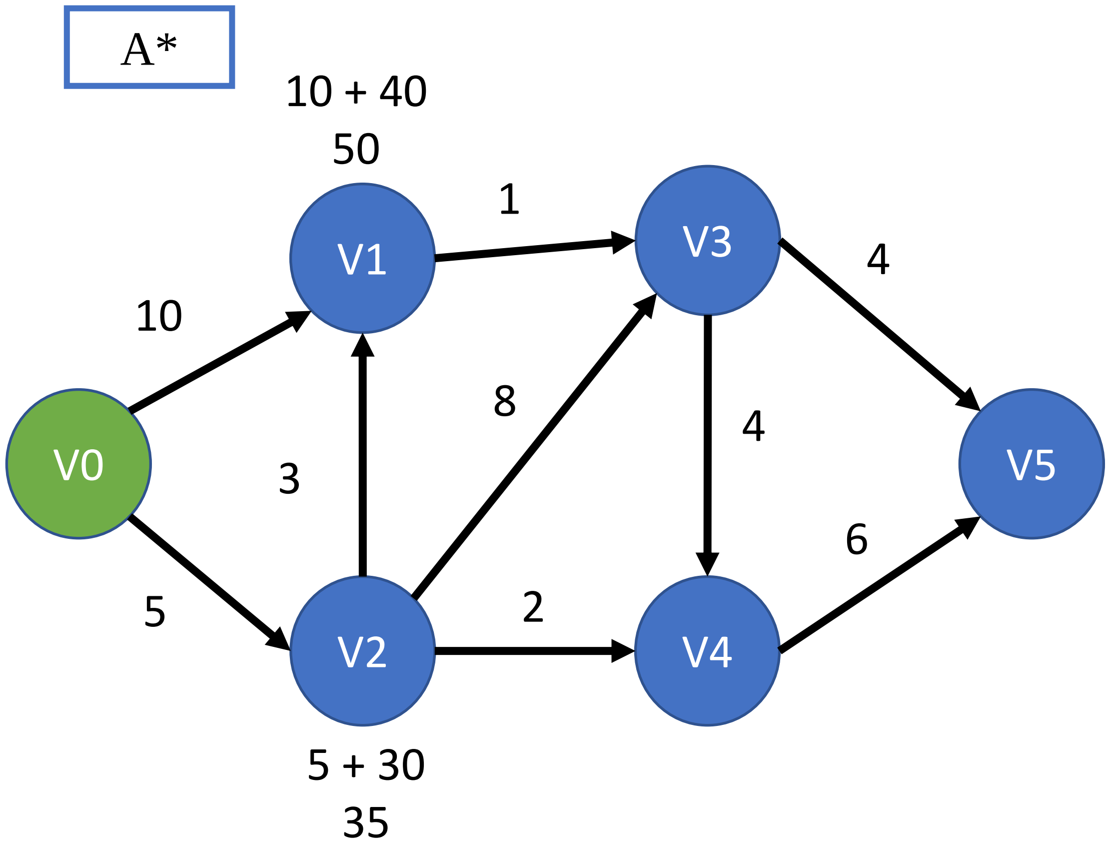


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors,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



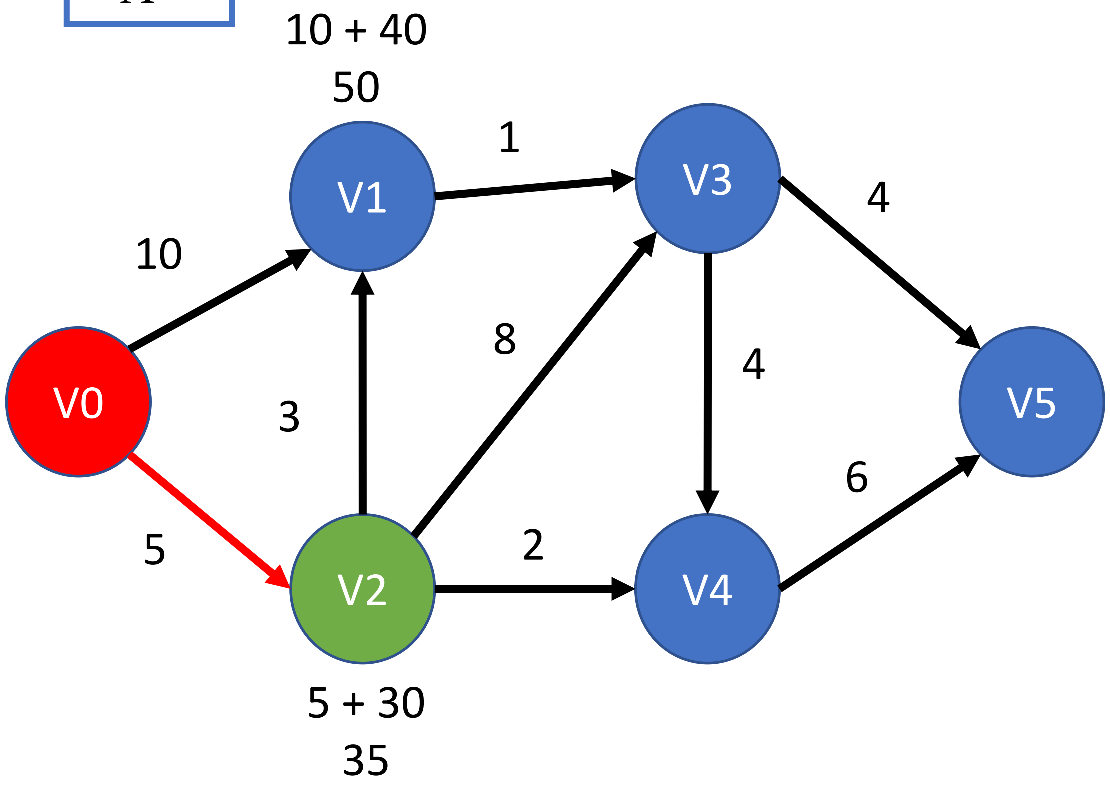
1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors,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

A*



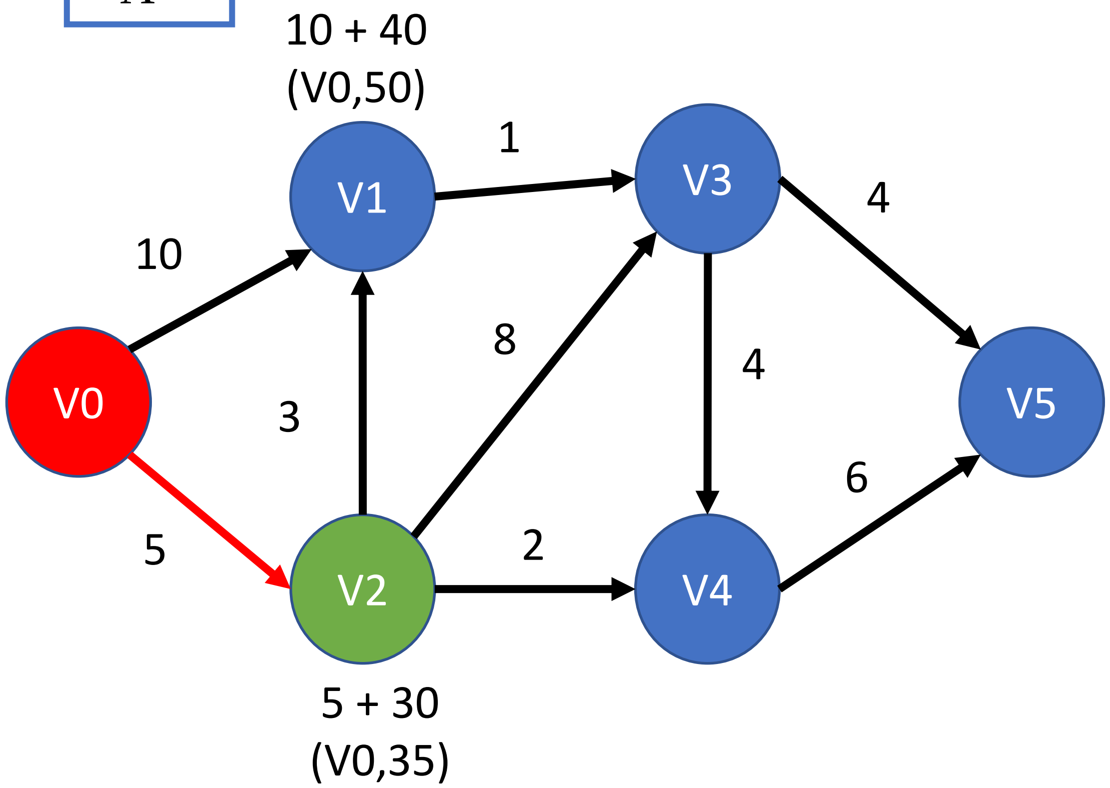
1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors,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

A*

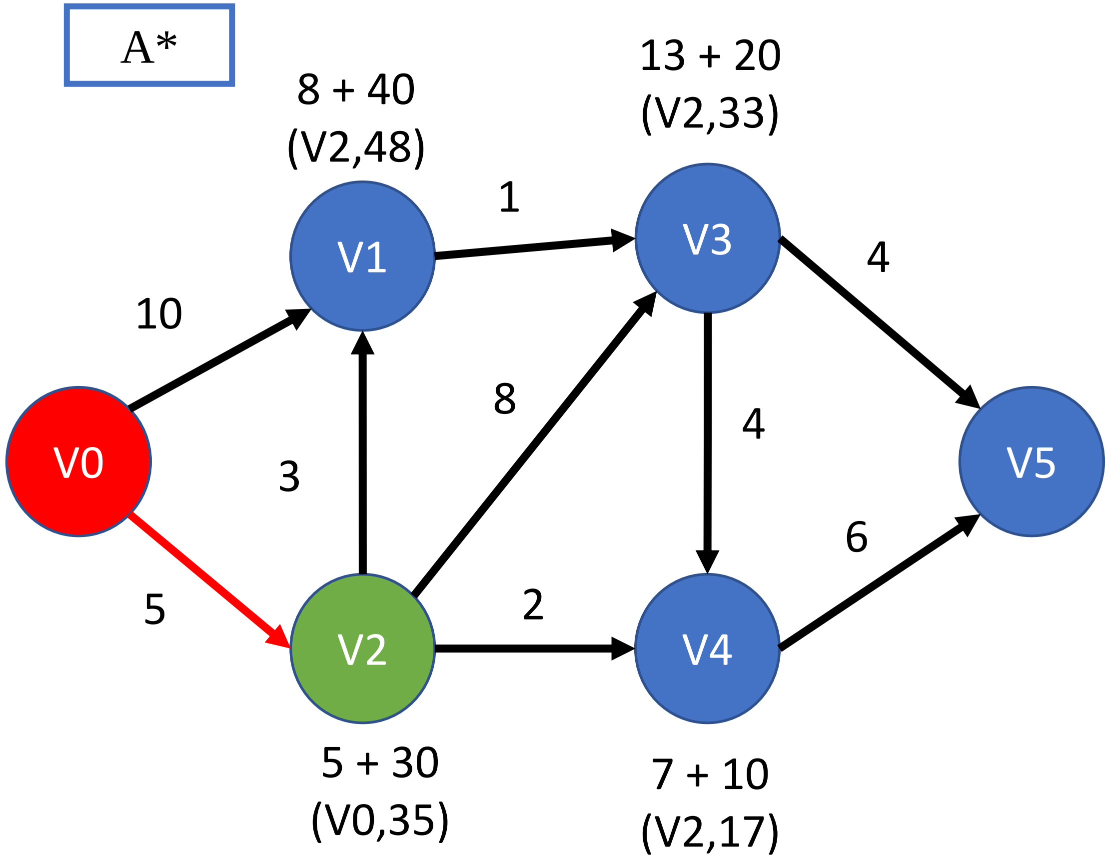


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors,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

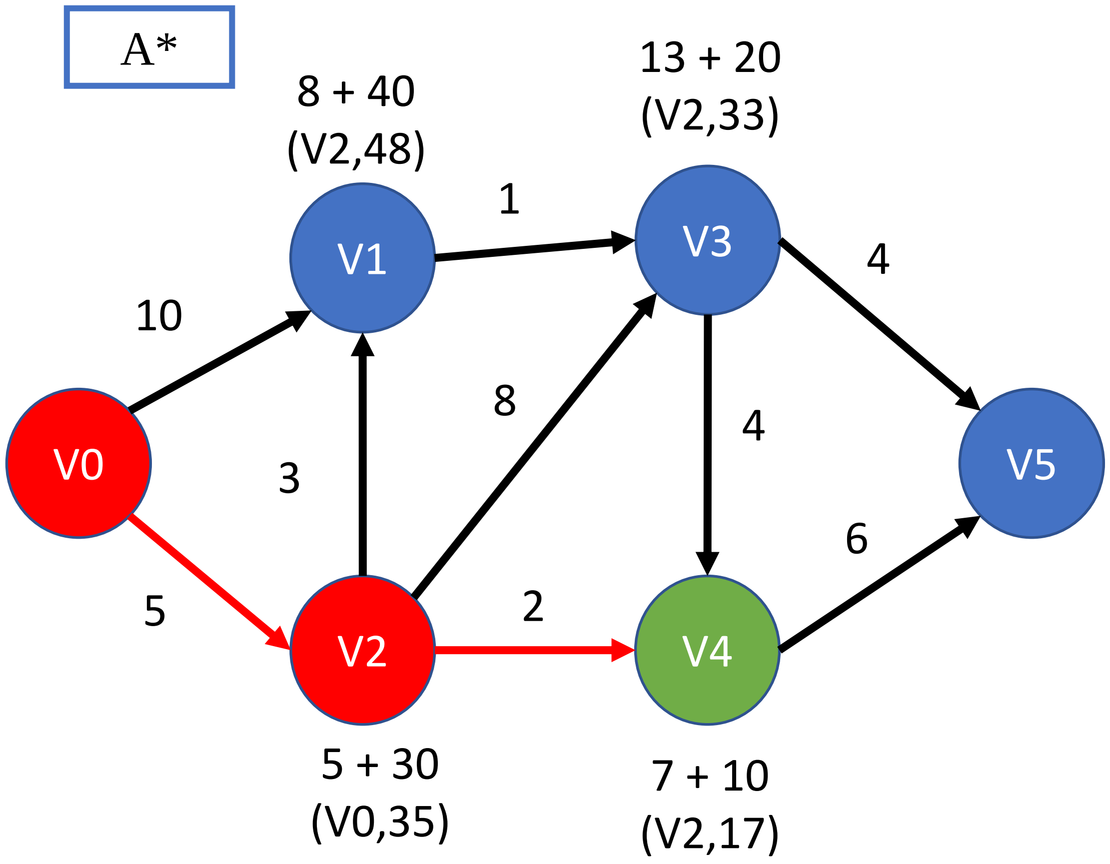


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
    * If it is a goal node, then stop and return to success.
    * Else remove the node from OPEN, and find all its successors.
    * Find the f (n) value of all the successors,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

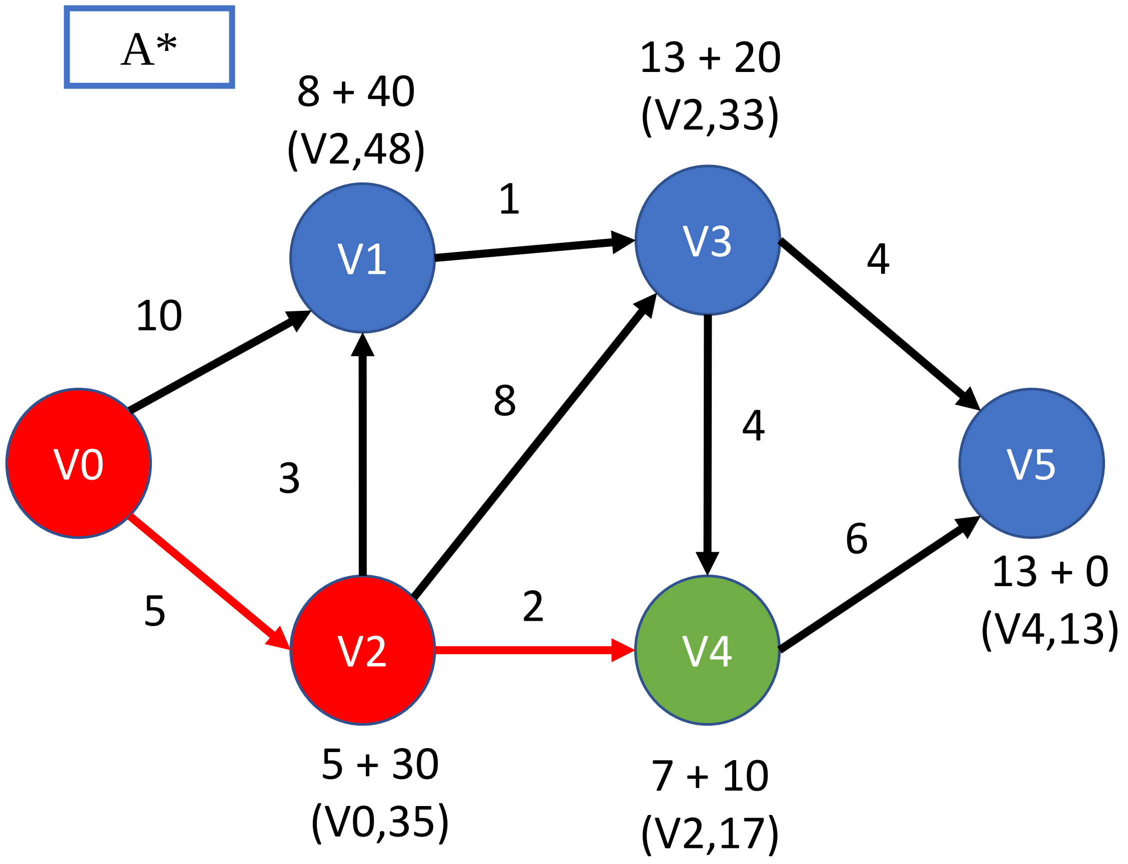


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
    * If it is a goal node, then stop and return to success.
    * Else remove the node from OPEN, and find all its successors.
    * Find the f (n) value of all the successors,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

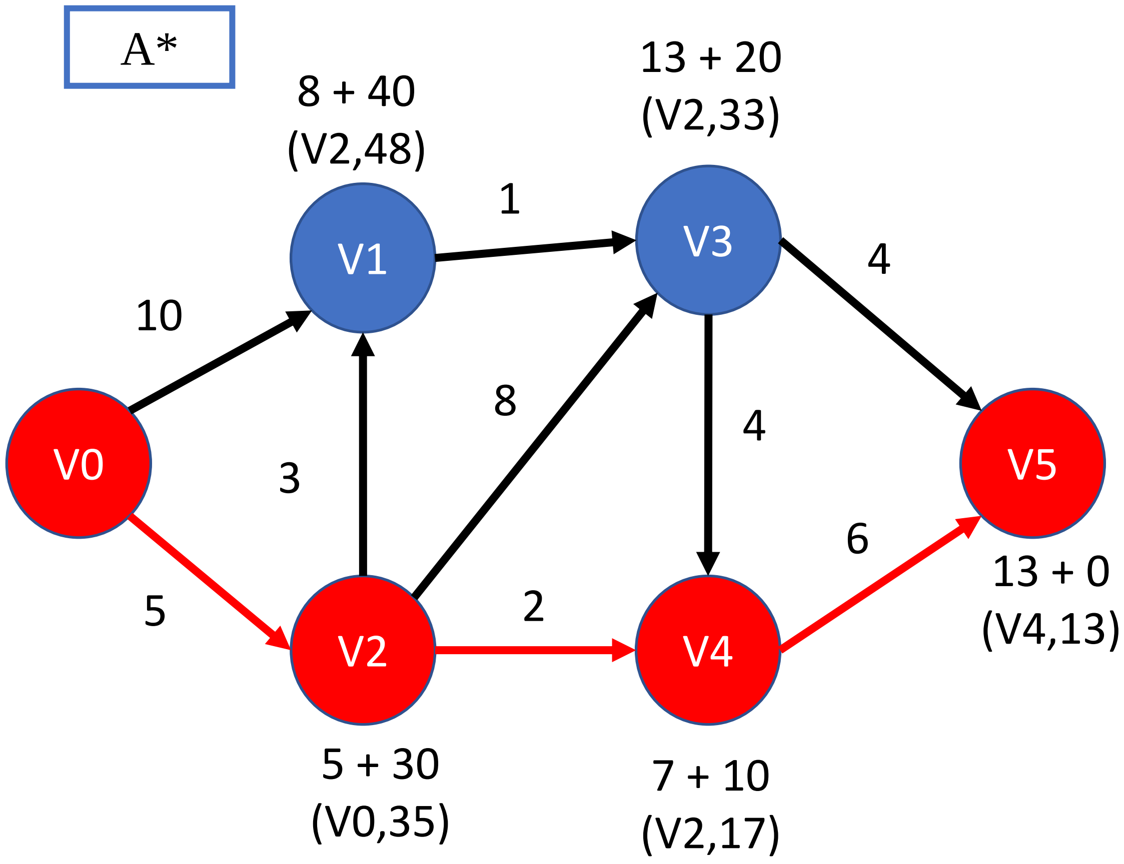


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors,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

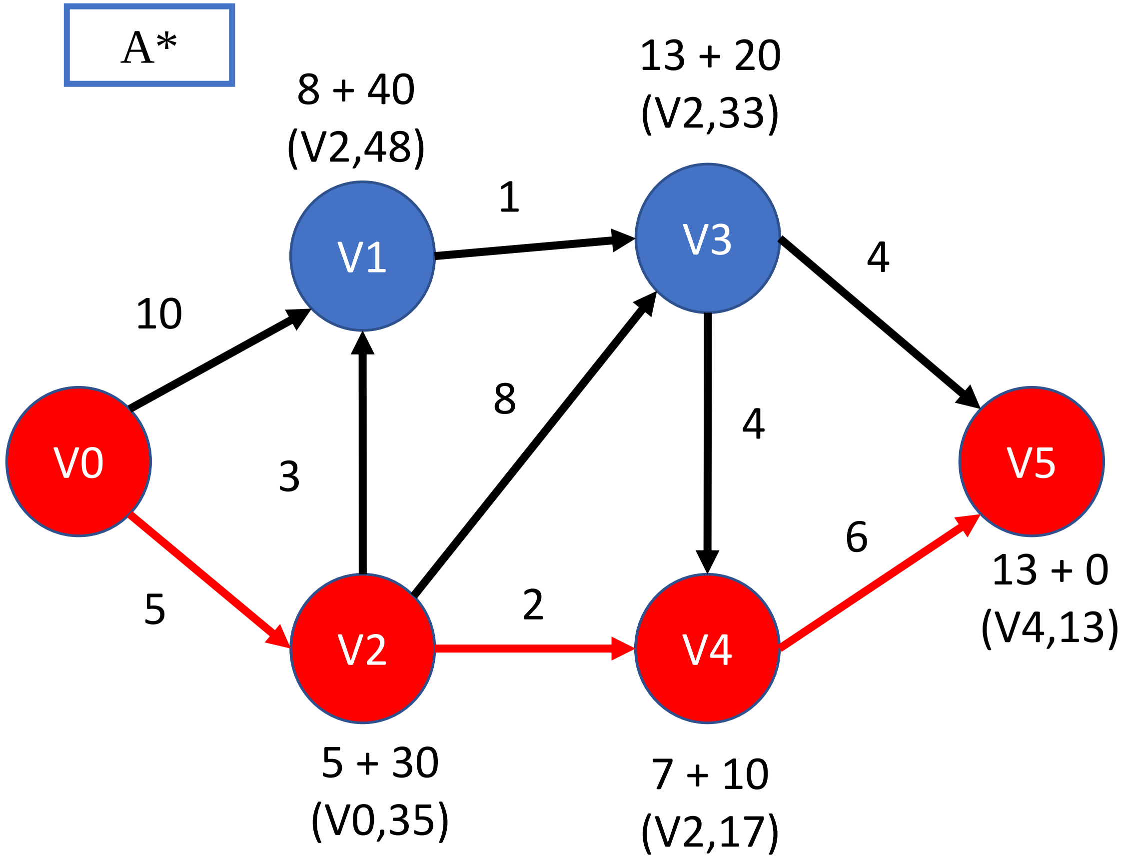


1. Place the starting node into OPEN and find its f (n) value.
2. Remove the node from OPEN, having the smallest f (n) value.
   * If it is a goal node, then stop and return to success.
   * Else remove the node from OPEN, and find all its successors.
   * Find the f (n) value of all the successors,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



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

Algorithms

A*

0	1+7 8	1	1	1
1+7 8	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

Algorithms

A*

0	1+7 8	1	1	1
1+7 8	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

Algorithms

A*

0	1+7 8	2+6 8	1	1
1+7 8	2+6 8	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

Algorithms

A*

0	1+7 8	2+6 8	1	1
1+7 8	2+6 8	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

Algorithms

A*

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

Algorithms

A*

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

Algorithms

A*

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

Algorithms

A*

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

Algorithms

A*

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

Algorithms

A*

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

Algorithms

A*

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

Algorithms

A*

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

Algorithms

A*

1	1+7 8	2+6 8	3+5 8	4+4 8
1+7 8	2+6 8	3+5 8	4+4 8	5+3 8
1				6+2 8
1	1	1	8+2 10	7+1 8
1	1	1	1	8+0 8

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

Algorithms

A*

0	1+7 8	2+6 8	3+5 8	4+4 8
1+7 8	2+6 8	3+5 8	4+4 8	5+3 8
1				6+2 8
1	1	1	8+2 10	7+1 8
1	1	1	1	8+0 8

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

Daniel Nogueira

dnogueira@ipca.pt