A* IMPLEMENTATION

In this example, I have created a PriorityQueue class. In priorityqueue, it has a selector which select according to F value in my example. Also there are Enqueue, Dequeue and Any methods.

```
class PriorityQueue<TElement,TKey>
{
    private SortedDictionary<TKey, Queue<TElement>> dictionary;
    private Func<TElement, TKey> selector;

    public PriorityQueue(Func<TElement,TKey> selector)
    {
        dictionary = new SortedDictionary<TKey, Queue<TElement>>();
        this.selector = selector;
    }

    public bool Any()
    {
        return (dictionary.Count >0);
    }

    public void Enqueue(TElement item)
    {
        TKey key = selector(item);
        Queue<TElement> queue;
        if(!dictionary.TryGetValue(key,out queue))
        {
            queue = new Queue<TElement>();
            dictionary.Add(key, queue);
        }
        queue.Enqueue(item);
    }

    public TElement Dequeue()
    {
        if (dictionary.Count == 0)
            throw new Exception("No Items to Dequeue: ");
        var key = dictionary.Keys.First();

        var queue = dictionary[key];
        var output = queue.Dequeue();
        if (queue.Count == 0)
            dictionary.Remove(key);

        return output;
```

I have created the graph with the given shape by my Node and Graph classes.

```
private void Go()
{
    Node<string> S = new Node<string>("S", 4);
    Node<string> A = new Node<string>("A", 2);
    Node<string> B = new Node<string>("B", 6);
    Node<string> C = new Node<string>("C", 2);
    Node<string> D = new Node<string>("D", 3);
    Node<string> G = new Node<string>("G", 0);

    Graph<string> graph = new Graph<string>();

    graph.Add(S, G, 12);
    graph.Add(S, A, 1);
    graph.Add(A, B, 3);
    graph.Add(A, C, 1);
    graph.Add(B, D, 3);
    graph.Add(C, D, 1);
    graph.Add(C, G, 2);
    graph.Add(D, G, 3);

    Func<Node<string>, int> func = delegate (Node<string> item) { return item.F; };
    PriorityQueue<Node<string>, int> priorityQueue = new PriorityQueue<Node<string>, int>(func);
    priorityQueue.Enqueue(S);


    Searches<string> searches = new Searches<string>();
    List<Node<string>> list=searches.AStar(priorityQueue,G);
    TextBoxWriter(list);
    graph.UnvisitAll();
}
```

And written the A* algorithm also some private methods that they are for finding the path.

```
public List<Node<T>> AStar(PriorityQueue<Node<T>,int> openList, Node<T> goal)
{
    Stack<Node<T>> stack = new Stack<Node<T>>();
    while (openList.Any())
    {
        Node<T> current = openList.Dequeue();
        PathArrange(stack,current);
        current.IsVisited = true;
        if (current == goal)
        {
            return StackToList(stack);
        }
        foreach (var temp in current.Neighbors)
        {
            if (temp.neighbor.IsVisited == true)
            {
                break;
            }
            else
            {
                temp.neighbor.HandG[0, 1] = current.HandG[0, 1] + temp.DistanceToNeighbor;
                //heuristic value is already defined at the beginning of declaration.
                temp.neighbor.Depth = current.Depth + 1;
                openList.Enqueue(temp.neighbor);
            }
        }
    }
    return StackToList(stack);


}
```

Additional methods arranges the path from start to end directly. It uses stack. When we dequeue if the depth is bigger than the stack element it arranges the path.

StackToList method returns the reversed list. Two of the methods are private because user shouldn’t use them they are only for class itself.

```
private void PathArrange(Stack<Node<T>> stack, Node<T> current)
{
    if (!stack.Any())
    {
        stack.Push(current);
        return;
    }
    while (stack.Peek().Depth >= current.Depth)
    {
        stack.Pop();
    }
    stack.Push(current);

}

private List<Node<T>> StackToList(Stack<Node<T>> stack)
{
    List<Node<T>> temp = stack.ToList();
    temp.Reverse();
    return temp;
}
```

Results with A* and Breadth First Search from S to G.

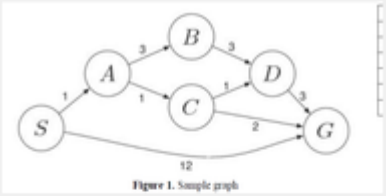

Figure 1. Sample graph

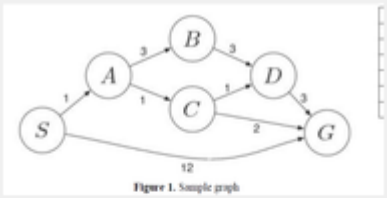

Figure 1. Sample graph

OUTPUT :

```
Depth    Searched(F) : Neighbors
0  S(4) : G A
1  A(3) : B C
2  C(4) : D G
3  G(4) :
```

OUTPUT :

```
Depth    Searched(F) : Neighbors
0  S(4) : G A
1  G(0) :
```