Directed Graph for .NET

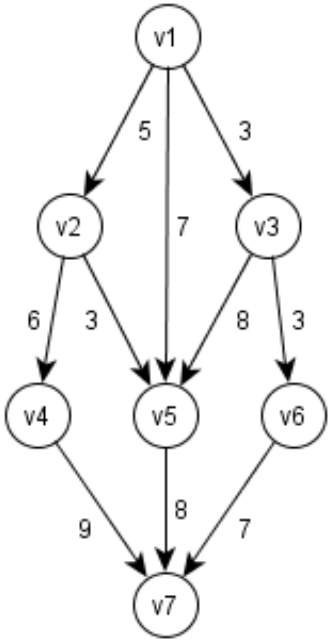
Introduction

Directed Graph is a common data structure specially for representing dependency-like relations. Directed graphs have various applications from project management to compiler design. Despite its vast usage, there is no implementation for directed graph in the .NET framework. In this article a simple implementation of Directed Graph as a library for the .NET platform using the C# language is presented. This library presents a simple and powerful implementation of directed graphs. The implementation tries to be as fast as possible and delivers these facilities:

- Adding and removing a vertex *O(1)*
- Adding and removing a edge *O(1)*
- Changing weight of a vertex or an edge *O(1)*
- Getting parents or children of any vertex *O(1)*
- Detecting cycles *O(e+v)*
- Computing the topological order *O(e+v)*
- Depth/Breath First Searches *O(e+v)*
- Computing the critical path *O(e+v)*
- Exporting to DOT language *O(e+v)*
- Needed Space *O(e)*

Background

A directed graph is a pair G=(V,E) where V is the set of vertices and E is the set of edges. An edge is represented by a pair (x,y) where x and y are vertices. Every edge or vertex can have a weight. Directed graphs have many application in field of computer science. Activity Graphs, Task Graphs, Application workflows and general dependency graphs are some major applications of directed graphs. A proper implementation of directed graph should provide some basic facilities e.g. inserting and removing vertices to/from the graph, inserting and removing edges to/from the graph, getting parents and children of a vertex and also some more advanced facilities e.g. cycle detection, finding critical path and depth/breath first searches. A sample directed graph is shown below:



Using the code

Using the class library is straight forward. For more ease of use, templates is not used and vertices are added by a string ID. When the graph is built, it can be manipulated by means of vertices' ID. For building more complicated graphs, for example a graph that its vertices are objects, programmer can use a translation table for converting

objects to IDs and vice versa.

Now we show how to work with the class library. First, we must create a new directed graph and insert some vertices and edges. Here we create the sample graph given in the background section:

```
var di_graph = new DirectedGraph();
di_graph.AddVertex("v1");
di_graph.AddVertex("v2");
di_graph.AddVertex("v3");
di_graph.AddVertex("v4");
di_graph.AddVertex("v5");
di_graph.AddVertex("v6");
di_graph.AddVertex("v7");
di_graph.AddEdge("v1", "v2", 5);
di_graph.AddEdge("v1", "v3", 3);
di_graph.AddEdge("v1", "v5", 7);
di_graph.AddEdge("v2", "v4", 6);
di_graph.AddEdge("v2", "v5", 3);
di_graph.AddEdge("v3", "v5", 8);
di_graph.AddEdge("v3", "v6", 3);
di_graph.AddEdge("v4", "v7", 9);
di_graph.AddEdge("v5", "v7", 8);
di_graph.AddEdge("v6", "v7", 7);
```

Now we can manipulate the graph. First, convert the graph to string:

```
Console.WriteLine("The Graph:");
Console.WriteLine(di_graph);
Console.WriteLine("=======================================");
```

Output:

```
The Graph:
[v1,v2,v3,v4,v5,v6,v7]
[(v1,v2),(v1,v3),(v1,v5),(v2,v4),(v2,v5),(v3,v5),(v3,v6),(v4,v7),(v5,v7),(v6,v7)]
```

Converting the graph to DOT language:

```
Console.WriteLine("The Graph in DOT lang:");
Console.WriteLine(di_graph.ToDotFormat());
Console.WriteLine("========================================");
```

Output:

```
The Graph in DOT lang:
digraph graphname {
v1;
v2;
v3;
v4;
v5;
v6;
v7;
v1 -> v2  [weight=5, label=5];
v1 -> v3  [weight=3, label=3];
v1 -> v5  [weight=7, label=7];
v2 -> v4  [weight=6, label=6];
v2 -> v5  [weight=3, label=3];
v3 -> v5  [weight=8, label=8];
v3 -> v6  [weight=3, label=3];
v4 -> v7  [weight=9, label=9];
v5 -> v7  [weight=8, label=8];
v6 -> v7  [weight=7, label=7];
}
```

Performing DFS and BFS searches:

```
//DFS
var dfs=di_graph.DepthFirstSearch("v1");
StringBuilder str_builder = new StringBuilder();
foreach (var node in dfs)
```

```
    str_builder.AppendFormat("{0},",node);
str_builder.Remove(str_builder.Length - 1, 1);
Console.WriteLine("DFS: {0}", str_builder.ToString());
Console.WriteLine("=======================================");

//BFS
var bfs = di_graph.BreathFirstSearch("v1");
str_builder = new StringBuilder();
foreach (var node in bfs)
    str_builder.AppendFormat("{0},", node);
str_builder.Remove(str_builder.Length - 1, 1);
Console.WriteLine("BFS: {0}", str_builder.ToString());
Console.WriteLine("=======================================");
```

Output:

```
DFS: v1,v5,v7,v3,v6,v2,v4
=======================================
BFS: v1,v2,v3,v5,v4,v6,v7
```

Topological Order:

```
//Topoligical Order
var topological = di_graph.TopologicalOrder();
str_builder = new StringBuilder();
foreach (var node in topological)
    str_builder.AppendFormat("{0},", node);
str_builder.Remove(str_builder.Length - 1, 1);
Console.WriteLine("Topological Order: {0}", str_builder.ToString());
Console.WriteLine("=======================================");
```

Output:

```
Topological Order: v1,v2,v3,v4,v5,v6,v7
```

Computing the critical path:

```
//Critical path
var critical_path = di_graph.GetCriticalPath();
str_builder = new StringBuilder();
foreach (var node in critical_path)
    str_builder.AppendFormat("{0},", node);
str_builder.Remove(str_builder.Length - 1, 1);
Console.WriteLine("Critical path: {0}", str_builder.ToString());
Console.WriteLine("Critical path length: {0}", di_graph.CriticalPathLength());
Console.WriteLine("=======================================");
```

Output:

```
Critical path: v1,v2,v4,v7
Critical path length: 20
```

The class library provides various methods that can be used to manipulate the graph roughly in every way. However, user can define new methods on demand. All of the public methods of DirectedGraph are listed below:

```
override string ToString()
string ToDotFormat()
DirectedGraph Clone()
static DirectedGraph MakeRandomDirectedAcyclicGraph(int vertex_count, double density)
void AddVertex(string vertex)
void AddVertex(string vertex, double weight)
void AddVertex(Vertex vertex)
void AddEdge(string from_vertex, string to_vertex)
void AddEdge(string from_vertex, string to_vertex, double weight)
void AddEdge(Edge edge)
void RemoveVertex(string vertex)
void UpdateVertex(string vertex,double weight)
void RemoveEdge(string from_vertex, string to_vertex)
void ZeroEdge(string from_vertex, string to_vertex)
void UpdateEdge(string from_vertex, string to_vertex, double new_weight)
bool HasVertex(string vertex)
bool HasEdge(string from_vertex, string to_vertex)
```

```
List<Edge> GetEdges()
List<Vertex> GetVertices()
int GetVertexCount()
int GetEdgeCount()
double GetVertexWeight(string vertex)
double GetEdgeWeight(string from_vertex, string to_vertex)
List<string> Children(string vertex)
List<string> Parents(string vertex)
List<string> GetRootVertices()
List<string> GetLeafVertices()
bool HasCycle()
List<string> TopologicalOrder()
List<string> DepthFirstSearch(string start_vertex)
List<string> BreathFirstSearch(string start_vertex)
List<string> GetCriticalPath()
double CriticalPathLength()
void MergeVertices(string vertex_id1, string vertex_id2)
```

Conclusion

In this article a simple implementation of Directed Graph was presented. The implementation is simple, easy to use and fast. For keeping the work simple, the DirectedGraph class does not use templates and just support manipulation of vertices using an ID string.

With Best Regards,

Saeed Shahrivari

December 27th, 2010

Last edited Dec 27, 2010 at 3:02 PM by SaeedSh, version 13

COMMENTS

ercalote Aug 2, 2013 at 1:01 PM

Unfortunately, the function MergeVertices is missing :(

Sign in to add a comment