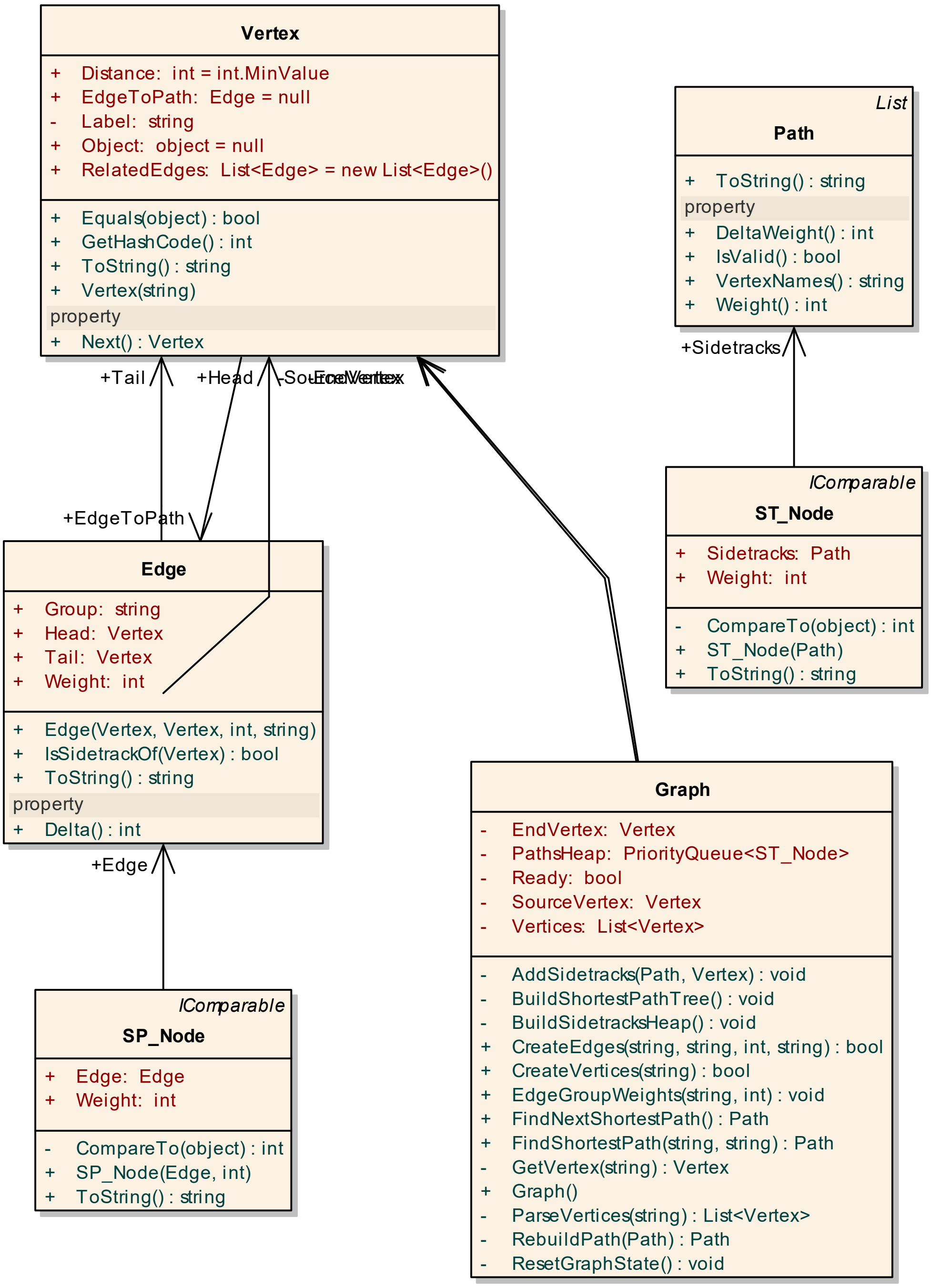
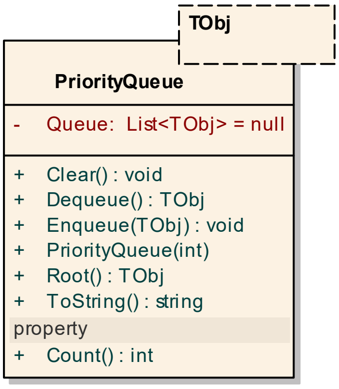

Vertex
+ Distance: int = int.MinValue
+ EdgeToPath: Edge = null
- Label: string
+ Object: object = null
+ RelatedEdges: List<Edge> = new List<Edge>()
+ Equals(object) : bool
+ GetHashCode() : int
+ ToString() : string
+ Vertex(string)
property
+ Next() : Vertex
+Tail
+Head
-SourceVertex
-EndVertex
+EdgeToPath
List
Path
+ ToString() : string
property
+ DeltaWeight() : int
+ IsValid() : bool
+ VertexNames() : string
+ Weight() : int
+Sidetracks
IComparable
ST_Node
+ Sidetracks: Path
+ Weight: int
- CompareTo(object) : int
+ ST_Node(Path)
+ ToString() : string
Edge
+ Group: string
+ Head: Vertex
+ Tail: Vertex
+ Weight: int
+ Edge(Vertex, Vertex, int, string)
+ IsSidetrackOf(Vertex) : bool
+ ToString() : string
property
+ Delta() : int
+Edge
IComparable
SP_Node
+ Edge: Edge
+ Weight: int
- CompareTo(object) : int
+ SP_Node(Edge, int)
+ ToString() : string
Graph
- EndVertex: Vertex
- PathsHeap: PriorityQueue<ST_Node>
- Ready: bool
- SourceVertex: Vertex
- Vertices: List<Vertex>
- AddSidetracks(Path, Vertex) : void
- BuildShortestPathTree() : void
- BuildSidetracksHeap() : void
+ CreateEdges(string, string, int, string) : bool
+ CreateVertices(string) : bool
+ EdgeGroupWeights(string, int) : void
+ FindNextShortestPath() : Path
+ FindShortestPath(string, string) : Path
- GetVertex(string) : Vertex
+ Graph()
- ParseVertices(string) : List<Vertex>
- RebuildPath(Path) : Path
- ResetGraphState() : void


TObj
PriorityQueue
- Queue: List<TObj> = null
+ Clear() : void
+ Dequeue() : TObj
+ Enqueue(TObj) : void
+ PriorityQueue(int)
+ Root() : TObj
+ ToString() : string
property
+ Count() : int

BuildShortestPathTree method:

```
Reset Graph State
Start from endpoint vertex
Set distance for endpoint vertex = 0

Create a queue for pending edges to evaluate (called “the fringe” in some texts)

Loop
    If there is pending vertex to evaluate
        Traverse all incoming and valid edges for this vertex
            Store them into the fringe

    Extract next shortest weighted edge from the fringe
    If there is no pending edge
        Finished!

    If tail (vertex) of extracted edge has not been evaluated
        Distance from vertex to endpoint = edge’s weight + distance from edge’s head to endpoint
        Mark this edge as a shortest path edge for current vertex
    else
        Do not traverse this vertex in next loop
End Loop
```

BuildSidetracksHeap method:

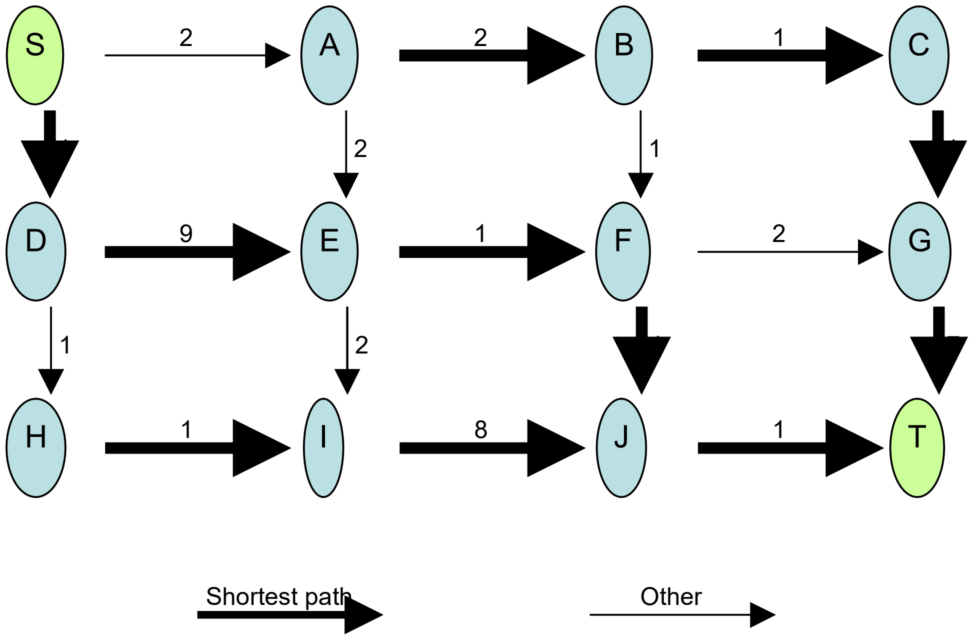
```
Create a heap (queue) to store sidetracks for each path (weight is the sum of deltas of sidetracks) *
Store in queue an empty sidetrack list, representing the shortest path

Start to evaluate from start vertex in shortest path
Loop
    For each sidetrack outcoming from current vertex
        Create a copy of previous vertex’s sidetrack list
        Add sidetrack to list
        Store sidetrack list into the heap
    End for
    If there is a next vertex in shortest path
        Recursively evaluate sidetracks for next vertex
    else
        Finished!
End Loop
```

Sidetracks are those outcoming edges that are not part of the shortest path
Delta of a sidetrack is the extra distance to take comparing with the shortest path
Shortest path edges always have a delta = 0

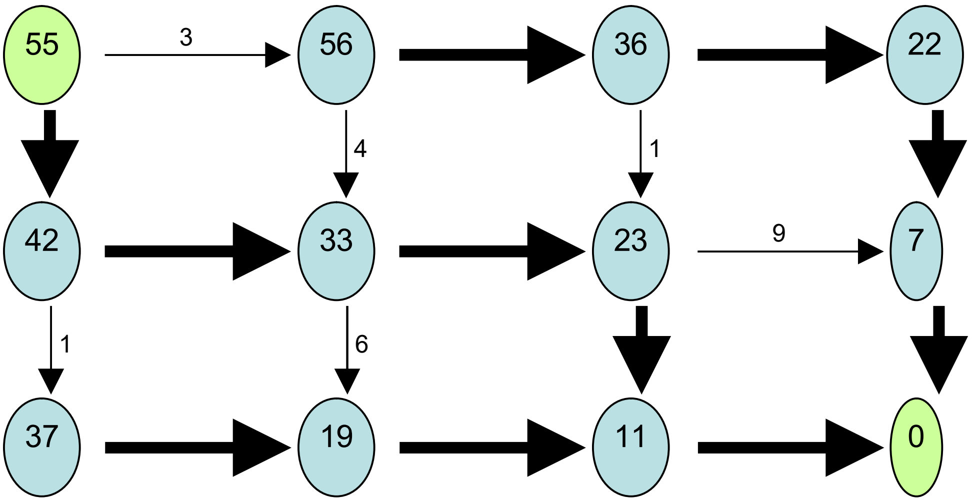
Internal storage of PathsHeap

Each path is represented uniquely by its sidetracks collection.
Sidetracks are represented here by its delta values. Rest of edges in each path are extracted from shortest path (S-D-E-F-J-T)
Empty collection (first) represents shortest path

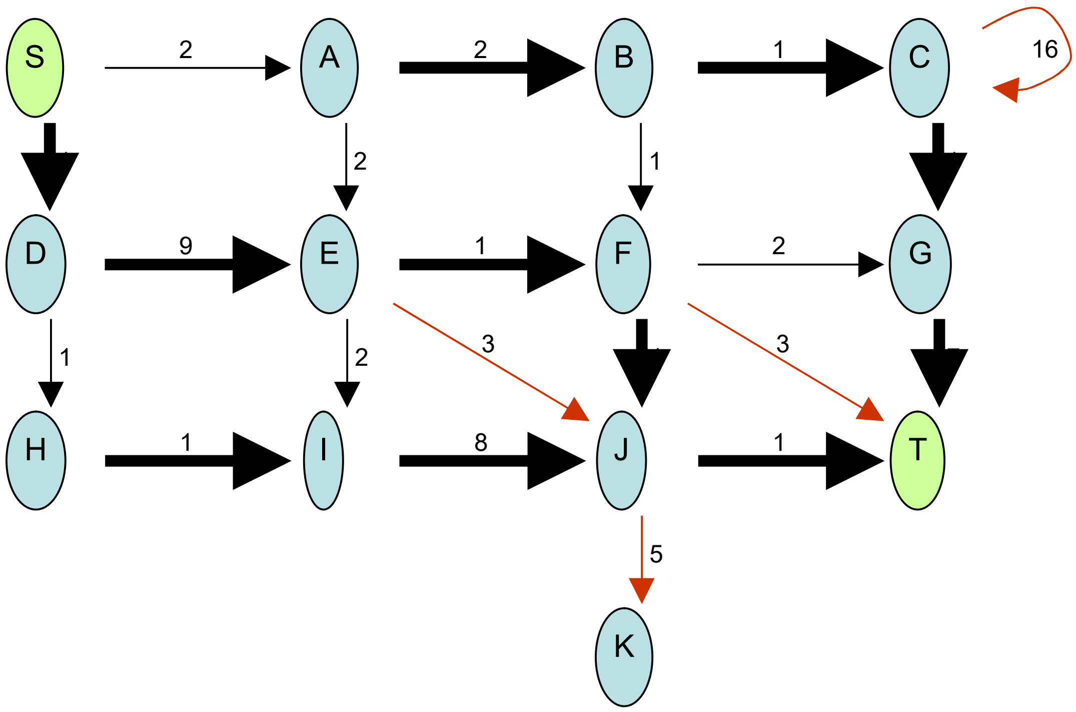
Eppstein’s original example graph



Distances to endpoint and sidetracks



Modified example


S
2
A
2
B
1
C
16
2
1
D
9
E
1
F
2
G
1
2
3
3
H
1
I
8
J
1
T
5
K