Ford-Bellman Shortest Path Algorithm
with step-by-step execution

Fernando Michel Tavera
Student at National Autonomous University of Mexico (UNAM)
Mexico
e-mail: fernando_michel@ciencias.unam.mx

Social service project director: Dr. Patricia Esperanza Balderas Cañas
Full time professor at National Autonomous University of Mexico (UNAM)
Mexico
e-mail: balderas.patricia@gmail.com

Introduction

The Ford-Bellman Shortest Path Algorithm is a well known solution to the Shortest Paths problem, which consists in finding the shortest path (in terms of arc weights) from an initial vertex *r* to each other vertex in a directed weighted graph
In this work we utilize the definition of the Ford-Bellman algorithm given by Cook et. al. (see References) which is as follows:

"Initialize *y*, *p*;
set $i = 0$;
While $i < n$ and *y* is not a feasible potential
Replace *i* by $i + 1$;
For $e \in E$
If *e* is incorrect
Correct *e*."

where: *y* is a set of $y(v)$, the size of the shortest path found so far from *r* to *v*, for each vertex *v*;
p is a set of the 'parent vertex' $p(v)$of each vertex *v*, that is to say the vertex prior to *v* in the shortest path from *r* to *v* found so far;
Initialize means setting $y(v) = \infty$ and $p(v)$ = null for each for each vertex *v* except r, and y(r) = 0 (the value of p(r) is not important);
An arc $a=(u,v)$ with weight *w* is considered incorrect if $y(u) + w \leq y(v)$, and correct otherwise;
Correcting an arc $a=(u,v)$ means changing the value of $y(v)$ to $y(u) + w$ such that *a* becomes correct, and setting $p(v) = u$ in the process;
And *y* is a feasible potential if all arcs are correct.

This work is part of a social service project consisting in the implementation of several graph theory algorithms with step-by-step execution, intended to be used as a teaching aid in graph theory related courses.

The usage examples presented were randomly generated.

Module usage

The FordBellmanSP module contains only a single procedure definition for FordBellman(*G*, *initial*, *stepByStep*, *draw*), as follows:

Calling FordBellman(...) will attempt to calculate the shortest paths in graph *G* from *initial* to every other vertex using the Ford-Bellman Algorithm.

The parameters taken by procedure FordBellman(...) are explained below:

- *G* is an object of type Graph from Maple's *GraphTheory* library, it is the graph for which the shortest paths will be computed. *G* must be defined as directed, otherwise the procedure will terminate with an error.
  This parameter is not optional

- *initial* is a symbol representing the vertex of *G* from which the shortest paths will be calculated. If the given symbol is not in the vertex list of *G*, the procedure will terminate reporting an error, otherwise the vertex of *G* with a label matching the given symbol will be used as initial.
  This parameter is not optional.

- *stepByStep* is a true/false value. When it is set to *true*, the procedure will print a message reporting whenever an arc is corrected. When it is *false*, only the final result will be shown.
  This parameter is optional, and its default value is *false*.

- *draw* is a true/false value. When it is set to *true*, the resulting shortest paths graph will be displayed after computation finishes; if both *stepByStep* and *draw* are *true* then the graph *G* will be drawn at every step, highlighting the arcs currently in a shortest path in green and the replaced arcs in red. When *draw* is set to *false*, the graphs will not be displayed, and the procedure will print a list containing the shortest paths to each vertex, in the format:
[[v1,[route to v1],distance to v1],[v2,[route to v2],distance to v2],...,[vn,[route to vn],distance to vn]]
  This parameter is optional, and its default value is *true*.

The return value can be one of three possibilities as follows:
- If *draw* is *true*, the procedure returns a subgraph *H* of *G* containing only the arcs of *G* which are used in a shortest path.
- If *draw* is *false*, the procedure will return a list containing the shortest paths to each vertex, this is so the value reported by Maple contains more useful information.
- If *initial* is a symbol not present in the vertex list of *G*, or if *G* is not a directed graph, or if *G* contains a negative weight loop, or if there are vertices unreachable from *initial*, the procedure will return the string "ERROR".

Module definition and initialization

```
> restart:
  with(GraphTheory):
  FordBellmanSP := module()
  option package;
  export FordBellman;

  FordBellman := proc (G::Graph, initial, stepByStep::truefalse :=
  false, draw::truefalse := true)
  local H :: list, V :: list, E :: set, e :: list, g::Graph,
  finished::truefalse, replaced::set, usedArcs::Graph,
```

```
initVert::set, i::int, n::int, head::int, tail::int, j::int,
result::list, v:

#input check
if IsDirected(G)=false then
  printf("ERROR: input graph must be directed");
  return "ERROR":   #undirected graph
end if:
#variable initialization
H:={}:   #List of edges of the graph representing the shortest
paths
E:=Edges(G,weights):   #backup of G's arc list with weight
information

if initial in Vertices(G) then   #initializes y and p
  n:=0:   #number of vertices in G
  V:=[]:   #contains the values of v, y(v) and index of p(v) for
every v
  for v in Vertices(G) do:
    if v=initial then
      V:=[op(V), [v,-1,0]]:
    else
      V:=[op(V), [v,-1,infinity]]:
    end if:
    n:=n+1:
  end do:
else
  printf("ERROR: initial vertex not in graph");
  return "ERROR":   #invalid initial vertex
end if:

if draw then
  usedArcs:=Digraph(Vertices(G),{},'weighted'):   #arcs
currently in a SP, used only when drawing the graph
  if stepByStep then
    printf("key: yellow = vertices, magenta = initial vertex,
blue = original graph arcs,\n\tgreen = arcs in a SP, red =
replaced arcs.\n");
    replaced:={}:   #arcs previously in a SP replaced for
shorter arcs, used only when drawing the graph
  end if:
end if:
```

```
i:=0:   #iteration count
finished:=false:   #indicates y is a feasible potential

while i<n and finished=false do:   #continue while y is not a
feasible potential
 finished:=true:

 for e in E do:   #for each edge
  head:=-1:
  tail:=-1:
  j:=1:
  while head=-1 or tail=-1 do:   #find head and tail of e
   if V[j][1]=e[1][1] then
    tail:=j:
   end if:
   if V[j][1]=e[1][2] then
    head:=j:
   end if:
   j:=j+1:
  end do:

  if V[tail][3]<>infinity then
   if V[head][3]=infinity or V[head][3]>V[tail][3]+e[2] then
#if edge is incorrect, correct it
    if draw then
      if V[head][3]<>infinity then
        DeleteArc(usedArcs,[V[V[head][2]][1],V[head][1]]):
      end if:
      AddArc(usedArcs,e):
    end if:
    if stepByStep then
      printf("corrected arc (%a,%a)\n",e[1][1],e[1][2]);
      if draw then
       if V[head][3]<>infinity then
        replaced:=replaced union {[V[V[head][2]][1],V[head][1]]}
:
        g:=Digraph(Vertices(G),replaced):
        HighlightSubgraph(G, g, red, yellow):
       end if:
       g:=Digraph(Vertices(G),Edges(usedArcs)):
       HighlightSubgraph(G, g, green, yellow):
       HighlightVertex(G,{initial},magenta):
```

```
       print(DrawGraph(G));
      end if:
    end if:
    V[head][3]:=V[tail][3]+e[2]:
    V[head][2]:=tail:
    finished:=false:
   end if:
  else
    finished:=false:
  end if:
 end do:

 if finished=true then
  if stepByStep then
    printf("No incorrect arcs found, computation finished\n"):
  end if:
  if draw then   #if the option is set, draw the shortest path
graph
    printf("Obtained shortest paths graph:\n"):
    print(DrawGraph(usedArcs));
    return usedArcs:   #return the shortest path graph
  else
    j:=0:
    result:=[ ]:
    for v in V do:   #for each vertex, rebuild the shortest path
and store it
     if v[1]=initial then
      result:=[op(result),[initial,"is the initial vertex", 0]]:
     else
      result:=[op(result),[v[1],[v[1]], v[3]]]:
      j:=v[2]:
      while j<>-1 do:
       result[nops(result)][2]:=[V[j][1],op(result[nops(result)]
[2])]:
       j:=V[j][2]:
      end do:
     end if:
    end do:
    if stepByStep then
      printf("shortest paths found (format is [vertex, route,
distance]):\n%a\n",result):
    end if:
```

```
      return result:   #return shortest path list
    end if:
   end if:
   i:=i+1:
  end do:

  printf("ERROR: could not finish computation, graph may contain a
  negative weight loop or vertices unreachable from %a",initial);
  return "ERROR":   #bad input graph

  end proc:
  end module:

  with (FordBellmanSP);
```

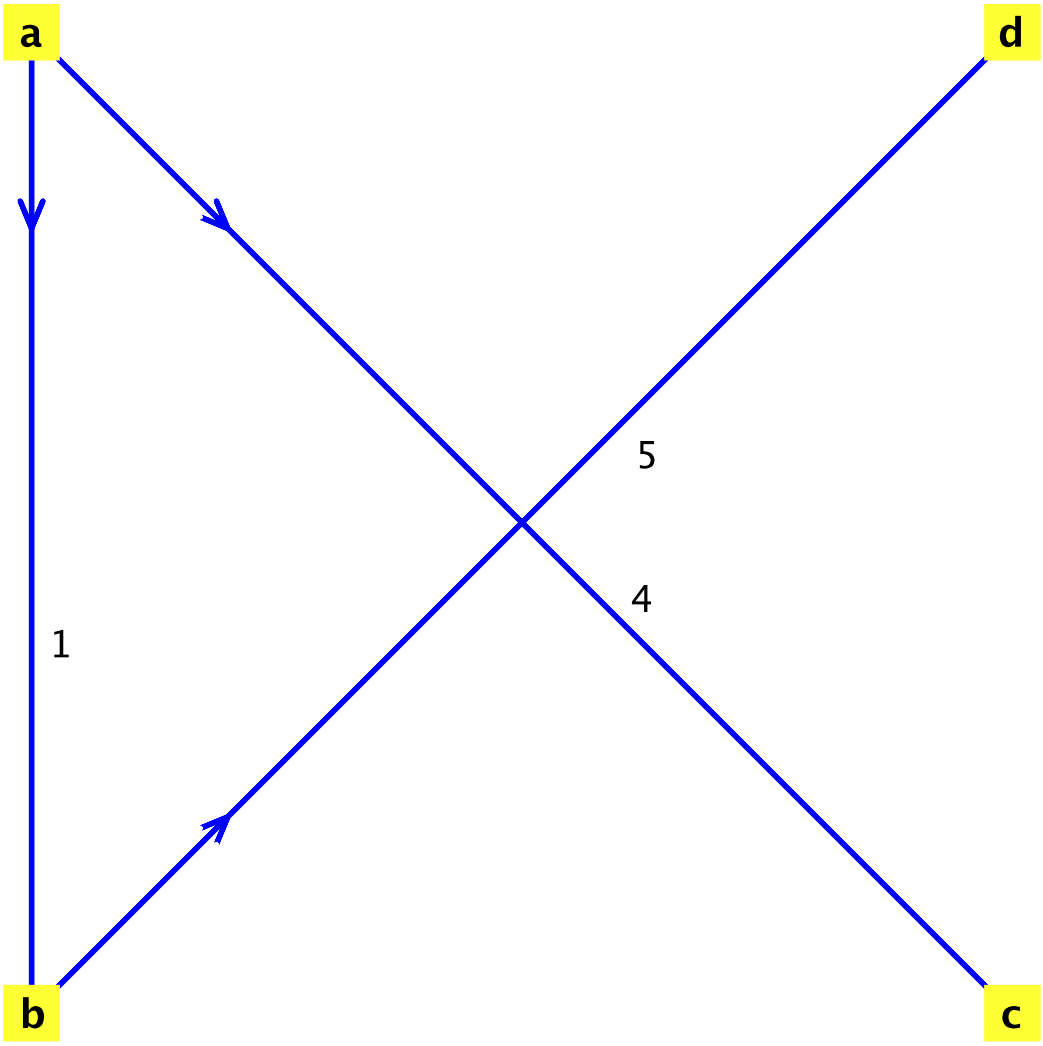
[*FordBellman*]

Usage examples

Default Behavior: print resulting graph of shortest paths, without step-by-step reports.

```
> vertices:=["a","b","c","d"]:
  arcs:={["a", "b"], 1],["a", "c"], 4],["c", "b"], 2],["b",
  "d"], 5],["c", "d"], 9]}:
  g := Digraph(vertices,arcs):
  FordBellman(g,"a");
```

Obtained shortest paths graph:



Graph 1: a directed weighted graph with 4 vertices and 3 arc(s)

Shows step-by-step reports, but doesn't print the graph

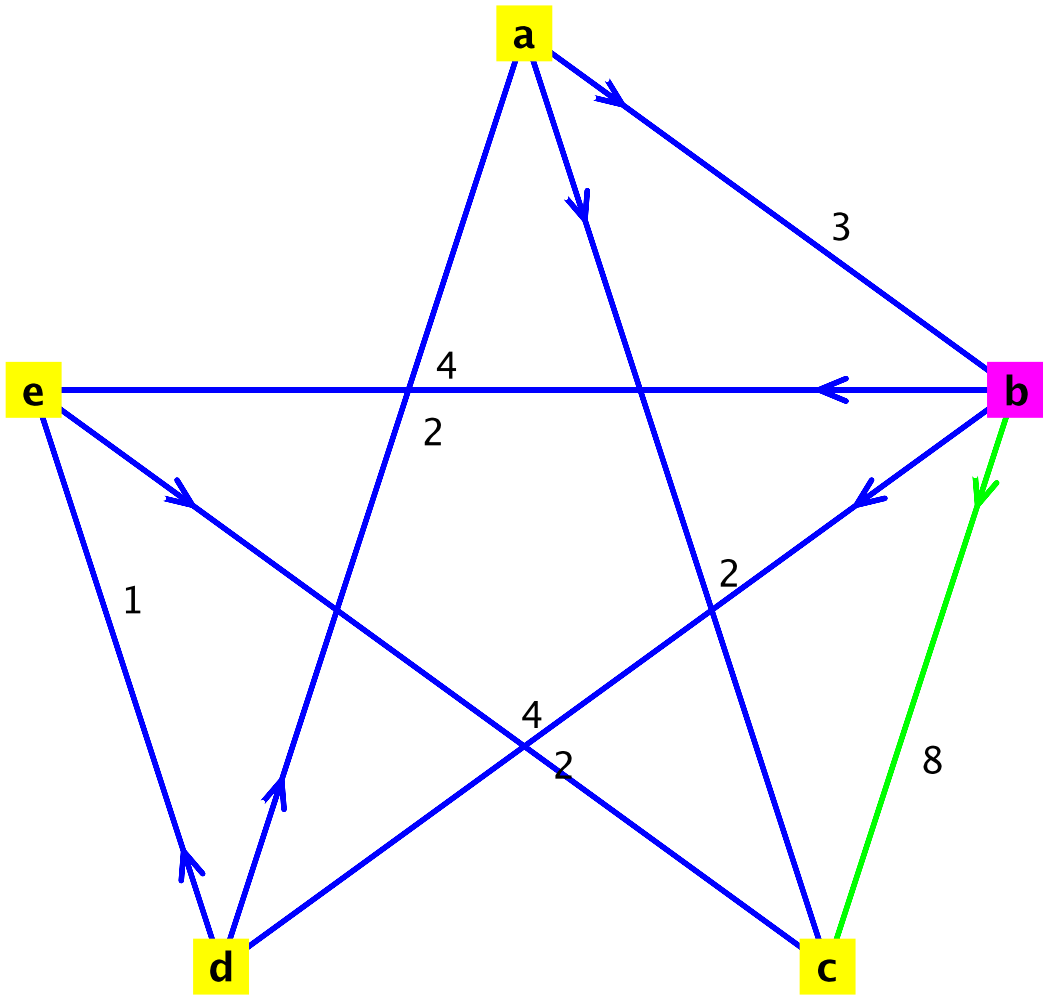
```
> vertices:=[1,2,3,4,5,6]:
  arcs:={[[1,2],6],[[1,3],2],[[4,1],5],[[2,3],6],[[2,4],4],[[2,
  5],5],[[3,4],6],[[3,5],3],[[6,4],2],[[4,5],6],[[5,6],2], [[6,
  1],1]}:
  g := Digraph(vertices,arcs):
  FordBellman(g,6,true,false):
corrected arc (6,1)
corrected arc (6,4)
corrected arc (1,2)
corrected arc (1,3)
corrected arc (2,5)
corrected arc (3,5)
No incorrect arcs found, computation finished
shortest paths found (format is [vertex, route, distance]):
```

```
[[1, [6, 1], 1], [2, [6, 1, 2], 7], [3, [6, 1, 3], 3], [4,
[6, 4], 2], [5, [6, 1, 3, 5], 6], [6, "is the initial
vertex", 0]]
```

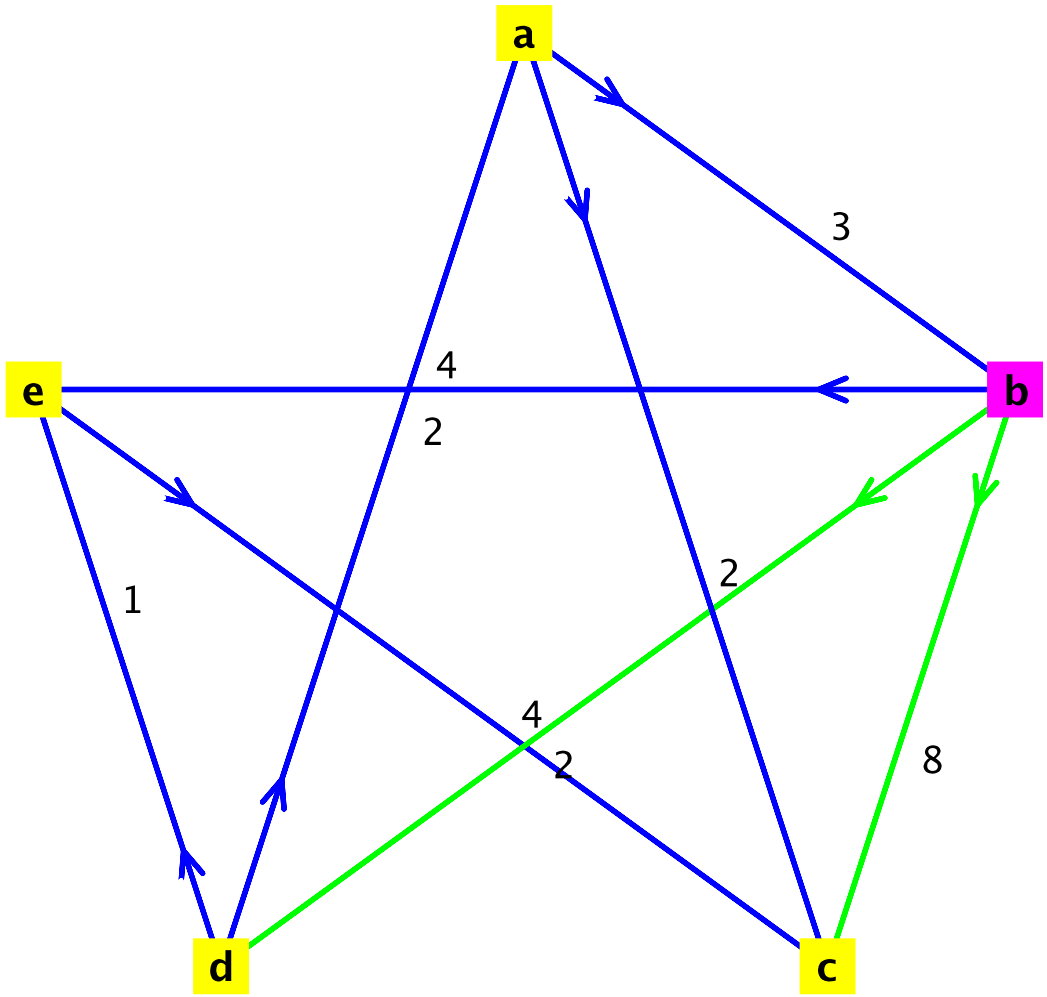
Shows step-by-step process with graphs for each step

```
> vertices:=["a","b","c","d","e"]:
  arcs:={[["a","b"],3],[["a","c"],2],[["d","a"],2],[["b","c"],
  8],[["b","d"],2],[["b","e"],4],[["e","c"],4],[["d","e"],1]}:
  g := Digraph(vertices,arcs):
  FordBellman(g,"b",true);
```

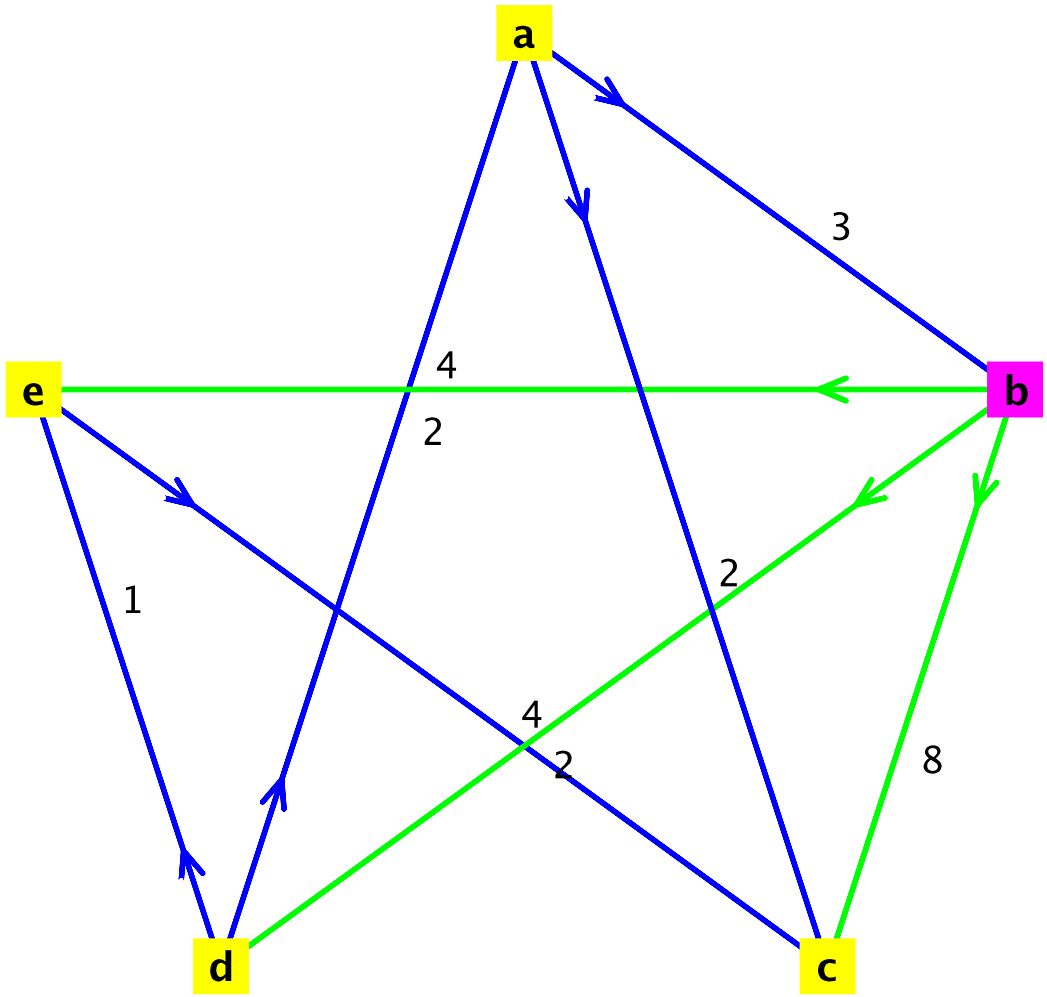
```
key: yellow = vertices, magenta = initial vertex, blue =
original graph arcs,

= arcs in a SP, red = replaced arcs.
corrected arc ("b","c")
```



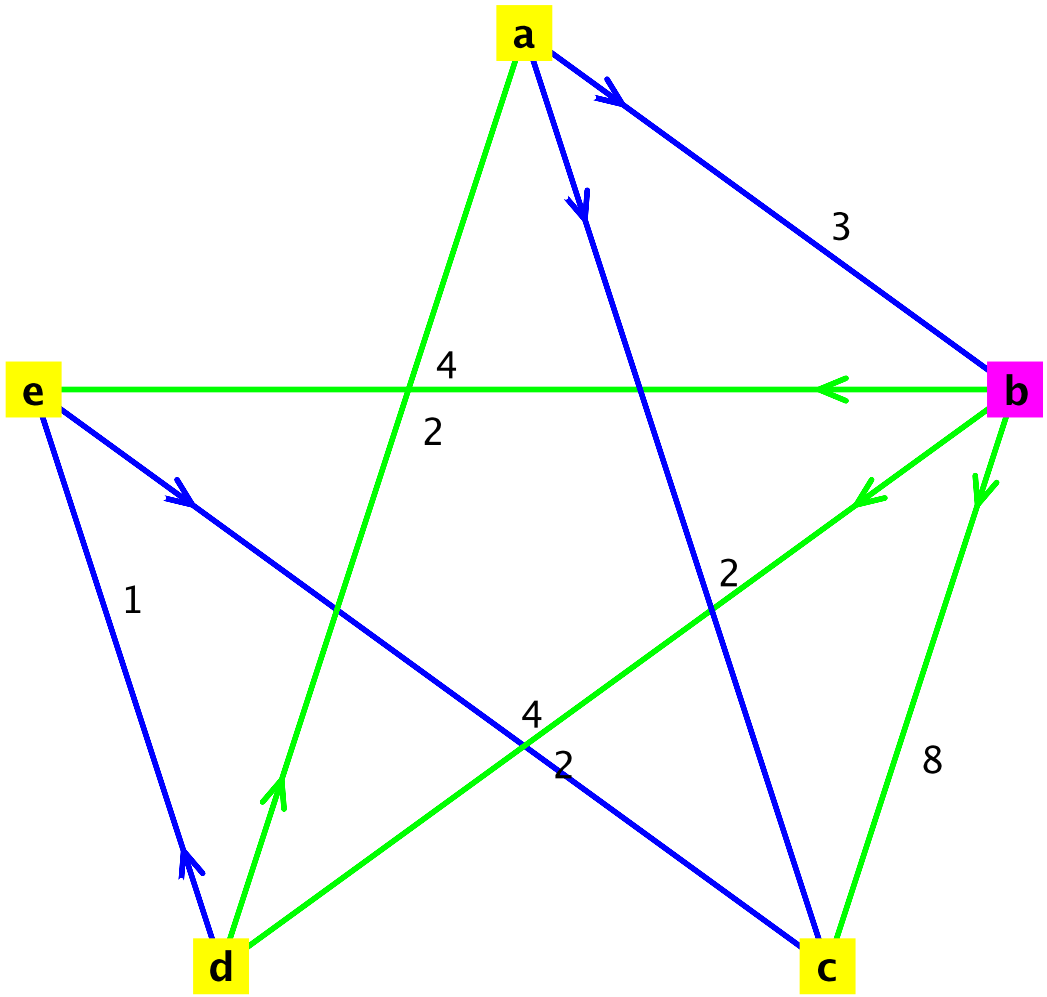

```
corrected arc ("b","d")
```



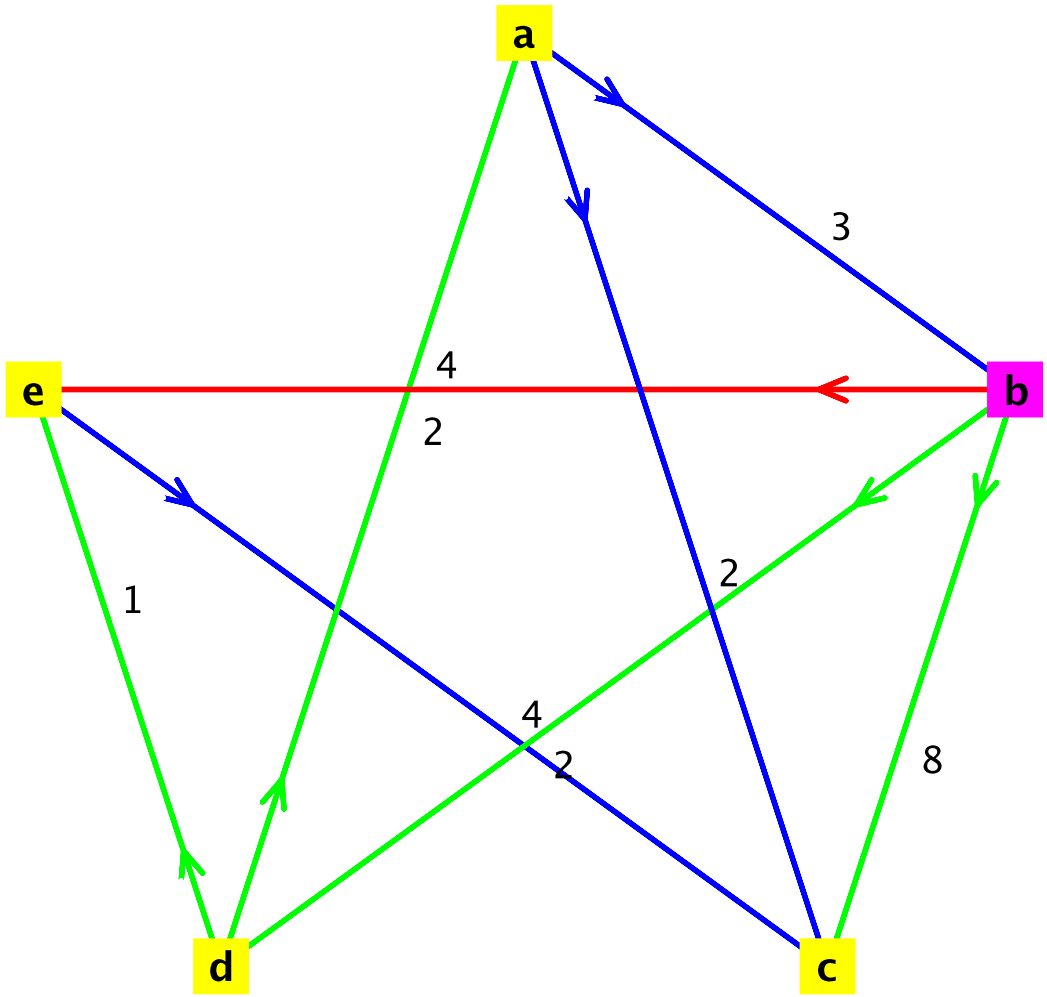
corrected arc ("b","e")



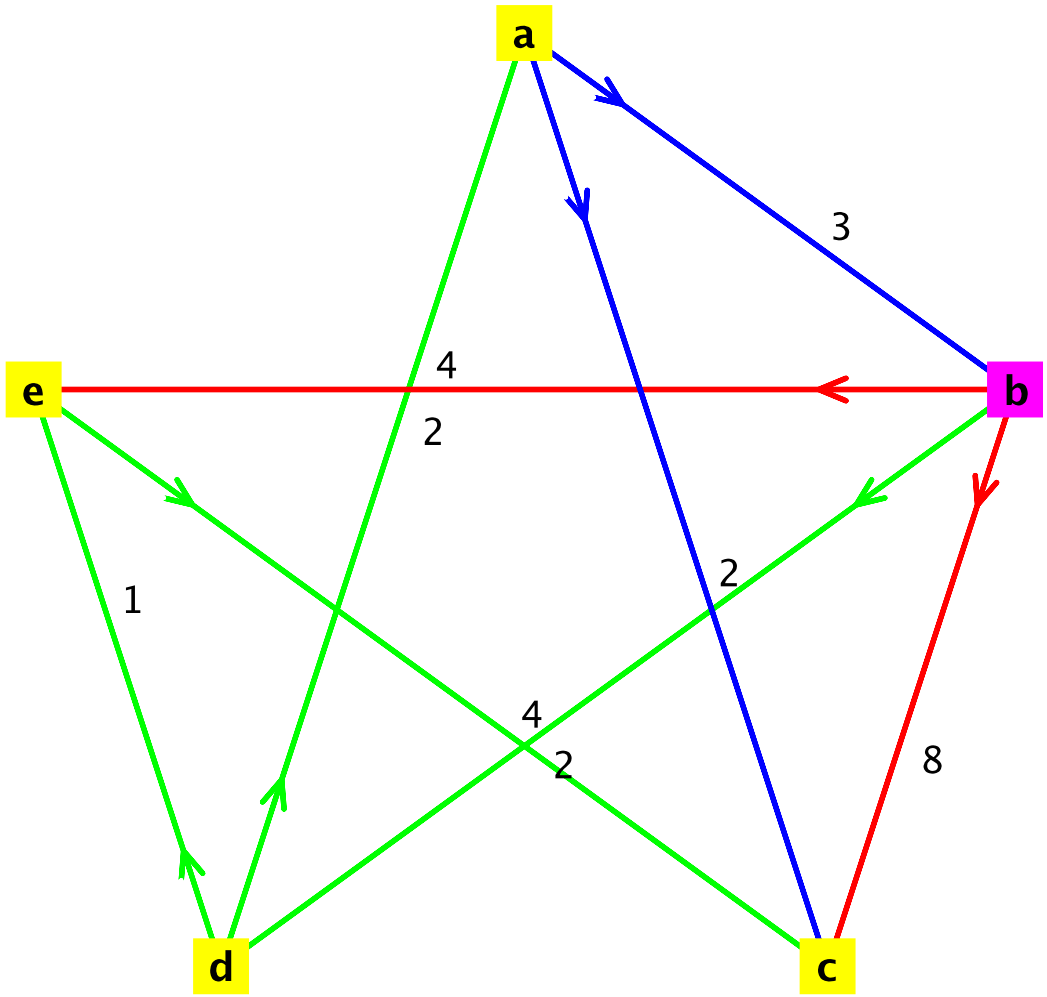

```
corrected arc ("d","a")
```



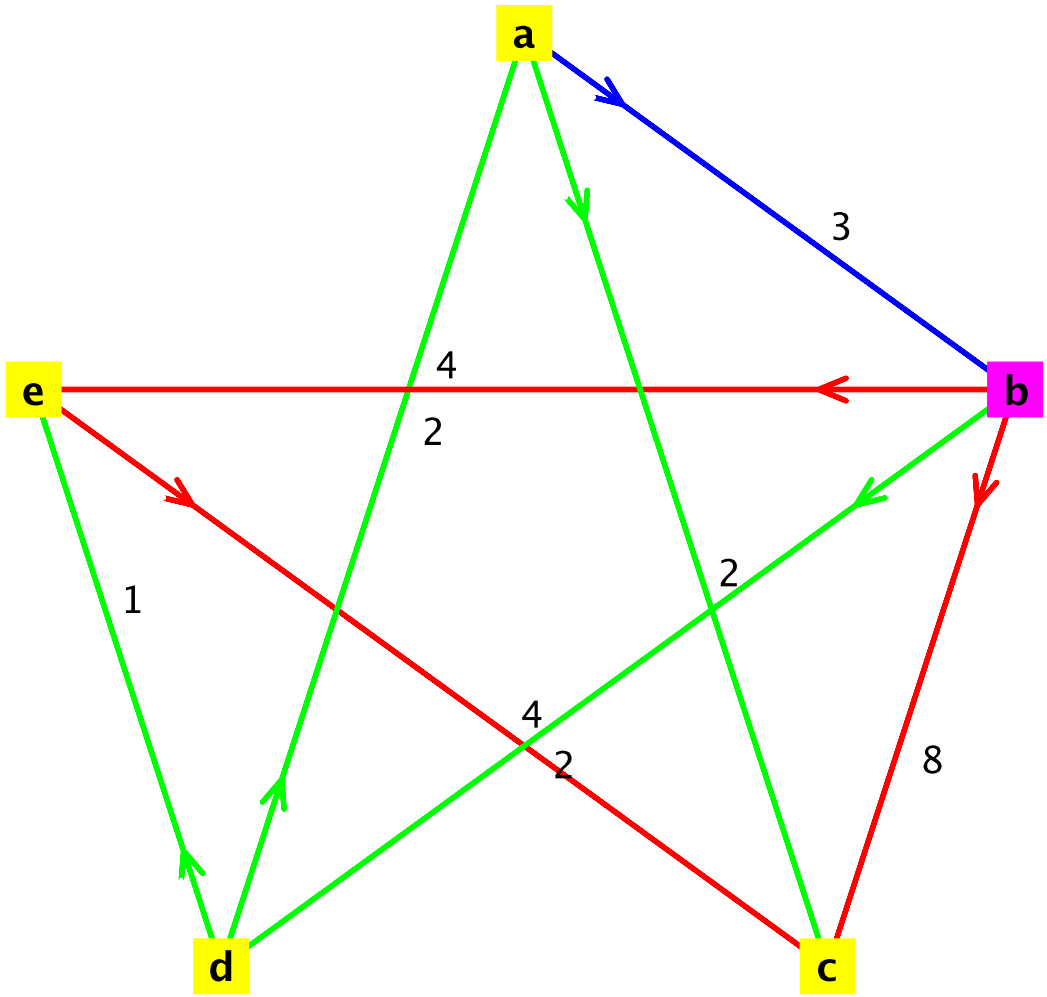
corrected arc ("d","e")



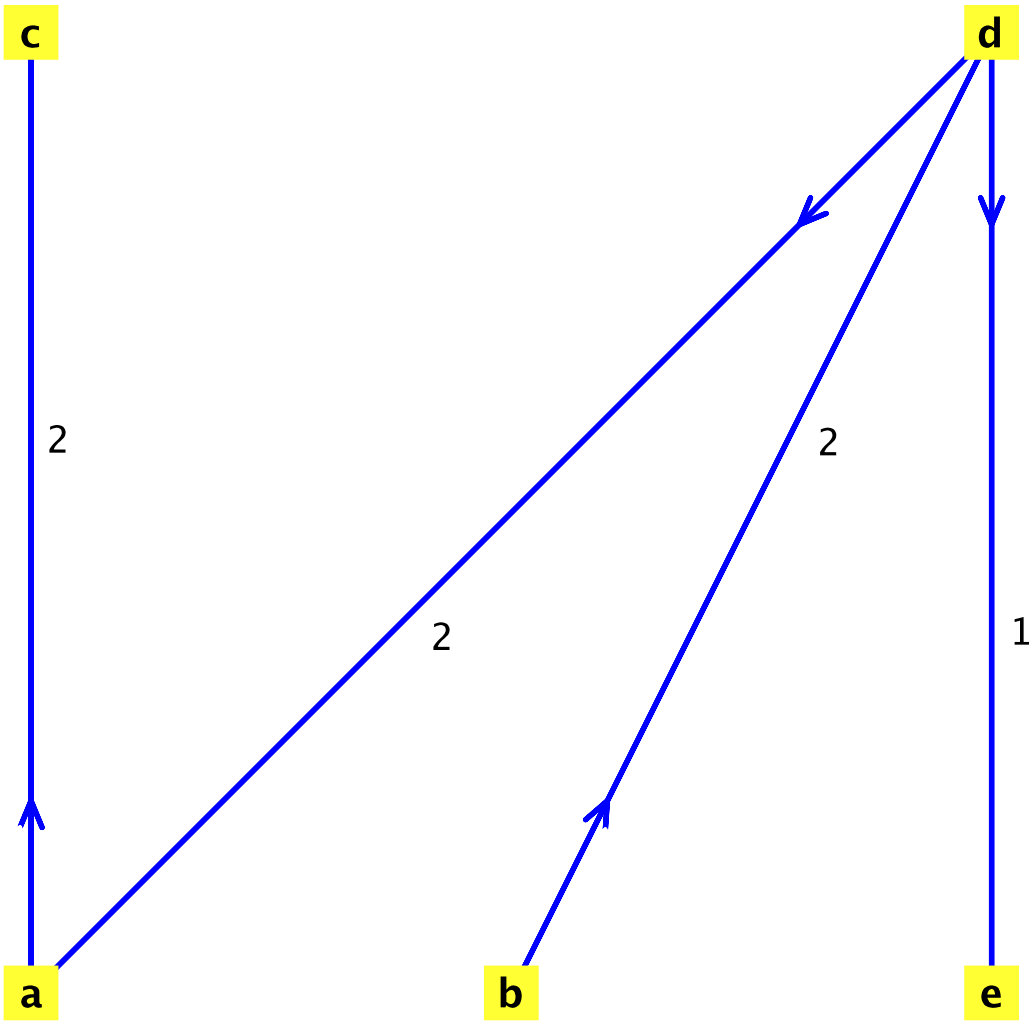
corrected arc ("e","c")




```
corrected arc ("a","c")
```



No incorrect arcs found, computation finished
Obtained shortest paths graph:



Graph 2: a directed weighted graph with 5 vertices and 4 arc(s) **(4.2.1)**

References

Cook, William J. et. al. *Combinatorial Optimization*. Wiley-Interscience, 1998. ISBN 0-471-55894-X