Dijkstra's 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

Dijkstra's 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 with nonnegative weights
In this work we utilize the definition of the Dijkstra's algorithm given by Cook et. al. (see References) which is as follows:

"Initialize *y*, *p*;
set $S = V$;
While $S \neq \varnothing$
Choose $v \in S$ with $y(v)$ minimum;
Delete *v* form *S*;
Scan *v*."

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);
S is a set of vertices that have not yet been scanned, and *V* is the set of all the vertices in the graph;
Scanning a vertex *u* means verifying, for every arc $a=(u,v)$ with weight *w*, that $y(u) + w \geq y(v)$ (that is *a* is "correct"), and otherwise correct it.
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;

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 DijkstraSP module contains only a single procedure definition for Dijkstra(*G*, *initial*, *stepByStep*, *draw*), as follows:

Calling Dijkstra(...) will attempt to calculate the shortest paths in graph *G* from *initial* to every other vertex using Dijkstra's Algorithm.

The parameters taken by procedure Dijkstra(...) 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 and have nonnegative arc weights, 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 or a vertex is scanned. 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 arcs currently in a shortest path in green, replaced arcs in red, and scanned vertices in cyan. 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*, *G* is not a directed graph, *G* has negative weight arcs, or there are vertices unreachable from *initial*, the procedure will return the string "ERROR".

Module definition and initialization

```
> restart:
  with(GraphTheory):
  DijkstraSP := module()
  option package;
  export Dijkstra;
```

```
Dijkstra := proc (G::Graph, initial, stepByStep::truefalse :=
false, draw::truefalse := true)
local H :: list, V :: list, S::set, E :: set, e :: list,
g::Graph, finished::truefalse, replaced::set, usedArcs::Graph,
initVert::set, minWeight::int, minIndex::int, head::int,
tail::int, i::int, n::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
S:={op(Vertices(G))}:   #set of vertices to be scanned
E:={}:   #backup of G's arc list with weight information
for e in Edges(G,weights) do:
   if e[2]<0 then
     printf("ERROR: input graph must have nonnegative arc
weights");
     return "ERROR":   #negative weight arc
   else
     E:=E union {e}:
   end if:
end do:

if initial in S 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 S 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 = unscanned vertices, cyan = scanned
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:

while S<>{} do:   #continue while S is not empty
 minWeight := infinity:
 for i from 1 to n do: #find v with y(v) minimum
   if V[i][1] in S and V[i][3]<minWeight then
     minIndex:=i:
     minWeight:=V[i][3]:
   end if:
 end do:
 if minWeight = infinity then
   printf("ERROR: unreachable vertex");
   return "ERROR":   #unreachable vertex
 else
   S:=S minus {V[minIndex][1]}:
   if stepByStep then
     printf("scanning vertex %a:\n",V[minIndex][1]);
   end if:
 end if:

 for e in E do:   #for each edge
  if e[1][1]=V[minIndex][1] then
    head:=-1:
    i:=1:
    while head=-1 do:   #find head of e (tail is v with y(v)
minimum)
     if V[i][1]=e[1][2] then
      head:=i:
     end if:
     i:=i+1:
```

```
    end do:

   if V[head][3]=infinity or V[head][3]>V[minIndex][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, cyan):
       HighlightVertex(G,S,yellow):
       HighlightVertex(G,{initial},magenta):
       print(DrawGraph(G));
      end if:
    end if:
    V[head][3]:=V[minIndex][3]+e[2]:
    V[head][2]:=minIndex:
   end if:
  end if:
 end do:
end do:

if stepByStep then
  printf("All vertices scanned, 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
```

```
  i:=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]]]:
    i:=v[2]:
    while i<>-1 do:
     result[nops(result)][2]:=[V[i][1],op(result[nops(result)]
[2])]:
     i:=V[i][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 proc:
end module:

with (DijkstraSP);
```

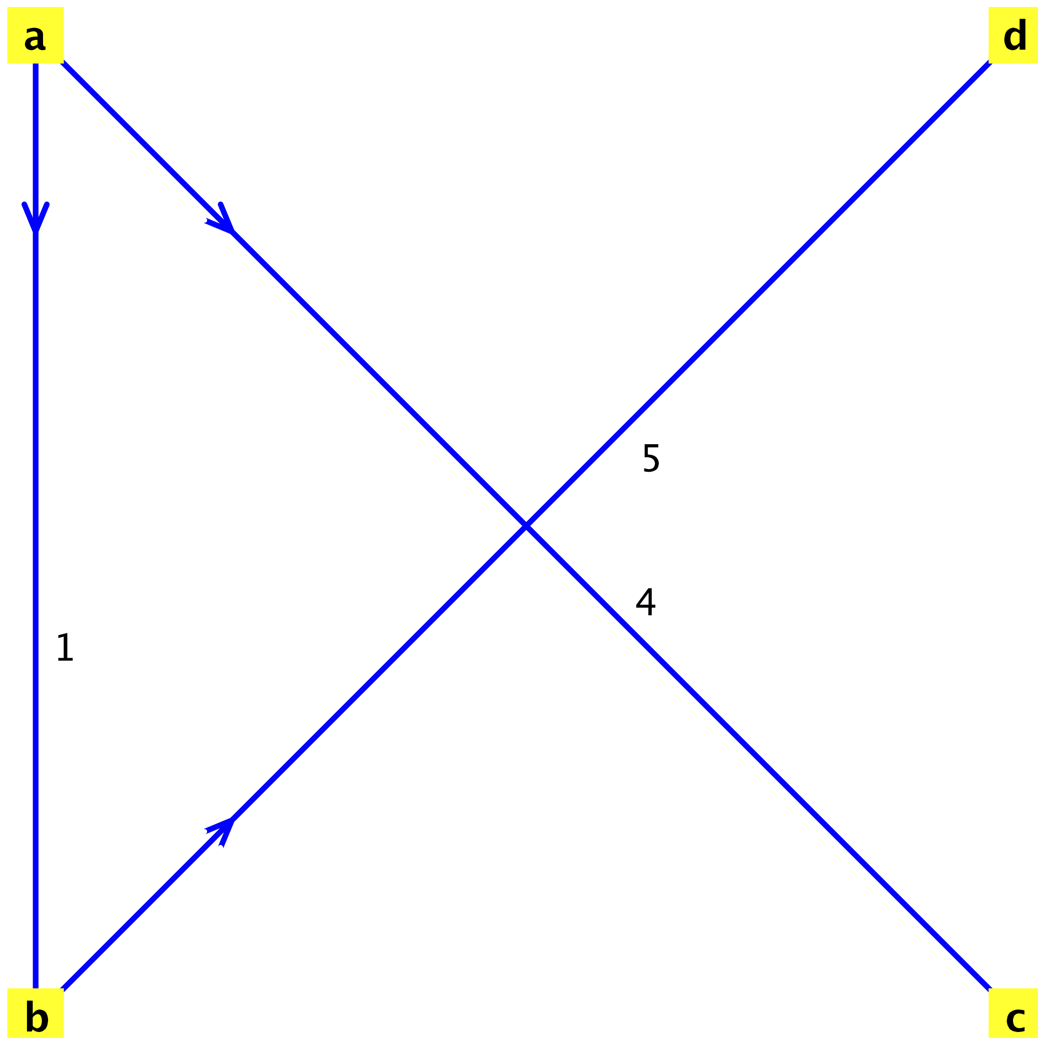
[*Dijkstra*]

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):
  Dijkstra(g,"a");
```

Obtained shortest paths graph:



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

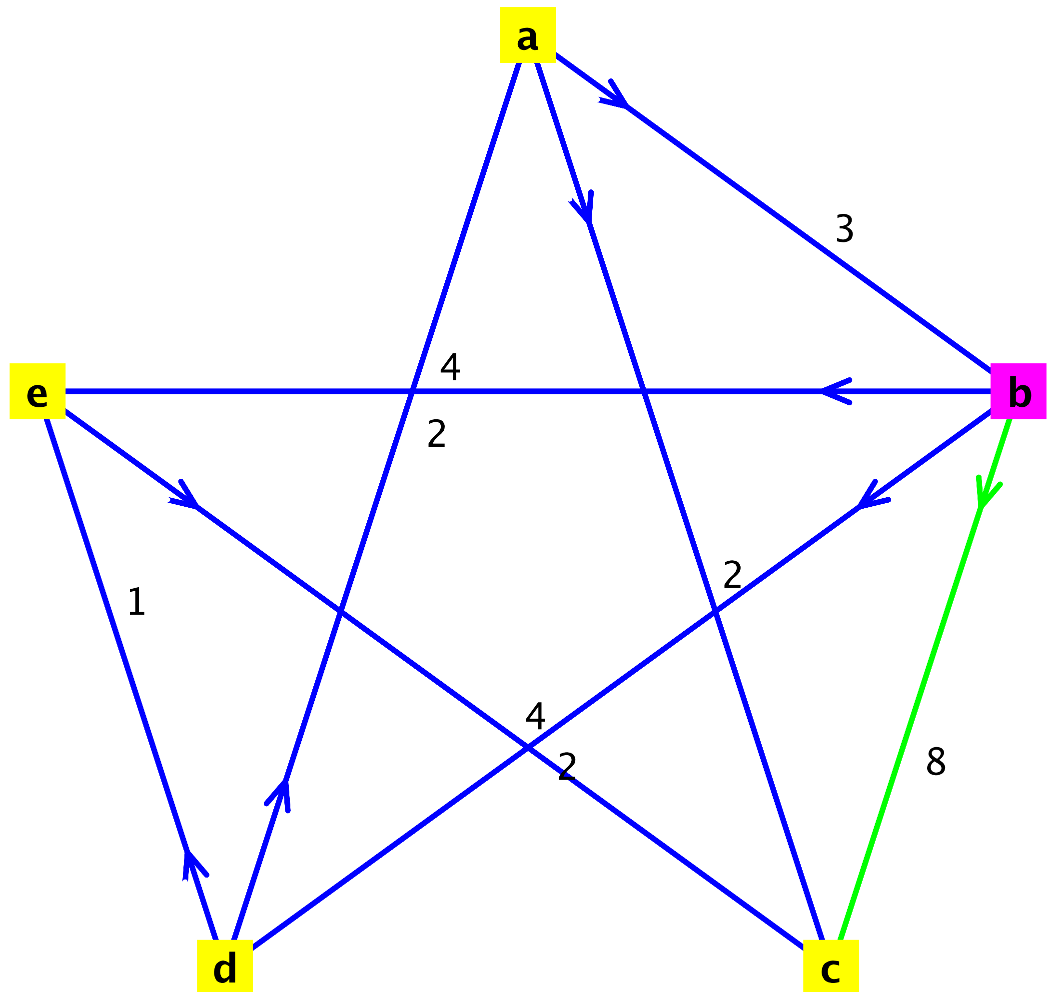
Shows step-by-step reports, but doesn't print the graph

```
> 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):
  Dijkstra(g,6,true,false):
scanning vertex 6:
corrected arc (6,1)
corrected arc (6,4)
scanning vertex 1:
corrected arc (1,2)
corrected arc (1,3)
scanning vertex 4:
corrected arc (4,5)
```

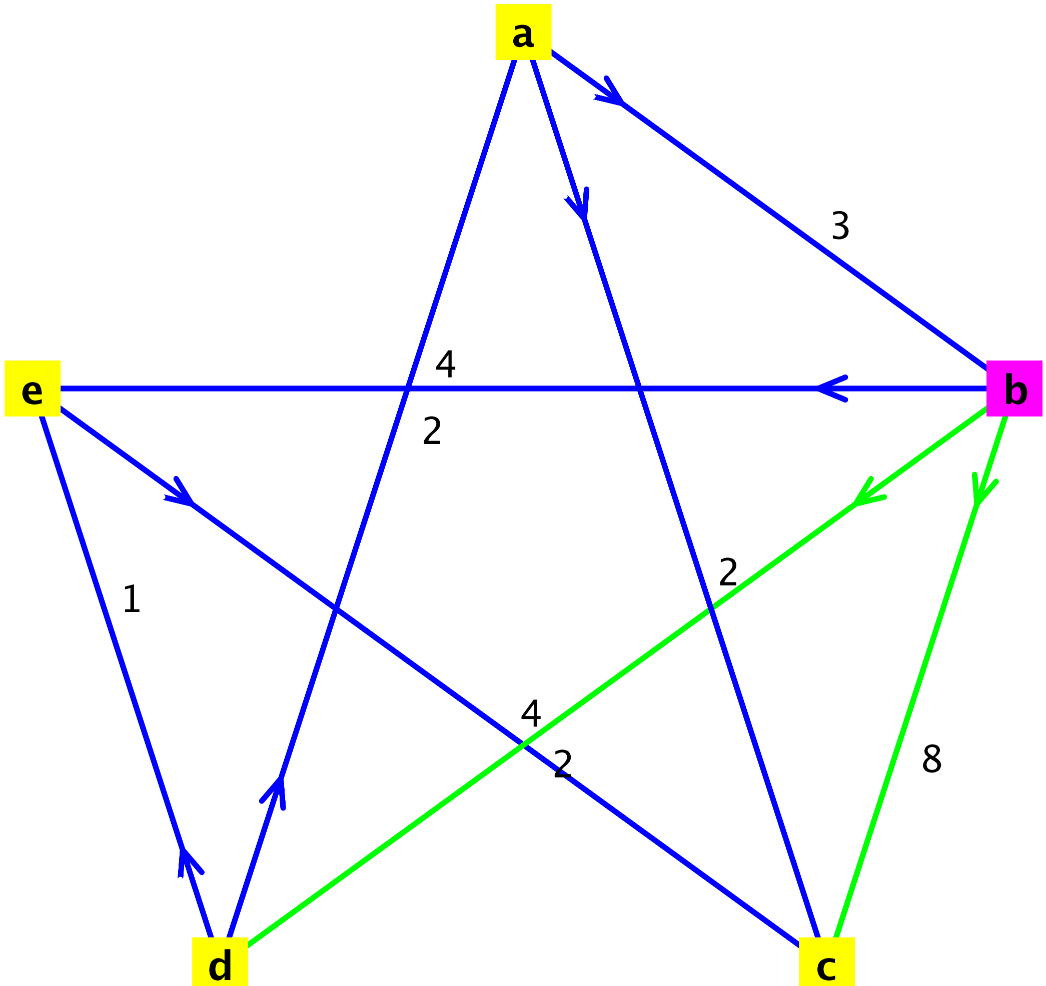
```
scanning vertex 3:
corrected arc (3,5)
scanning vertex 5:
scanning vertex 2:
All vertices scanned, computation finished
shortest paths found (format is [vertex, route, distance]):
[[1, [6, 1], 1], [2, [6, 1, 2], 7], [3, [6, 1, 3], 3], [4,
[6, 4], 2], [5, [6, 1, 3, 5], 6], [6, "is the initial
vertex", 0]]
```

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

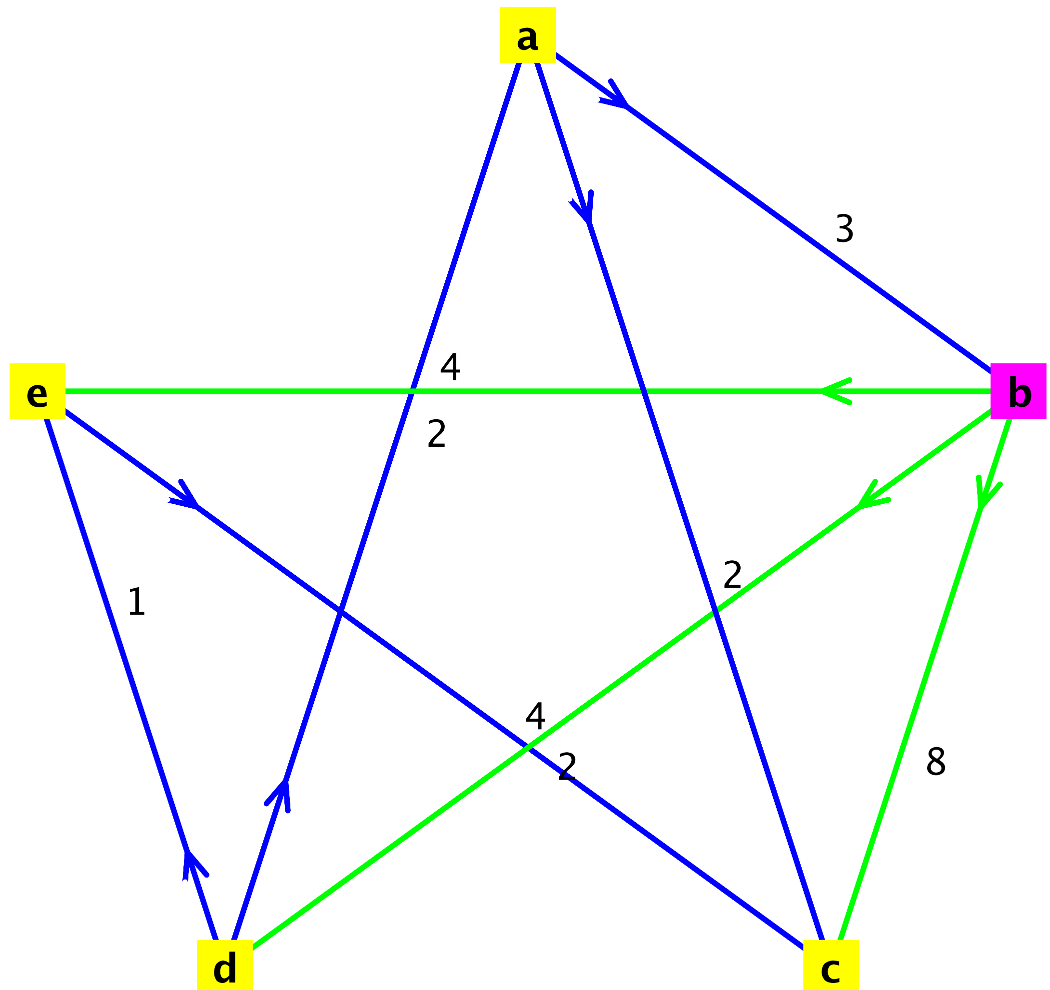
```
> vertices:=["a","b","c","d","e"]:
  arcs:={[["a","b"],3],[["a","c"],2],[["d","a"],2],[["b","c"],
  8],[["b","d"],2],[["b","e"],4],[["e","c"],4],[["d","e"],1]}:
  g := Digraph(vertices,arcs):
  Dijkstra(g,"b",true);
key: yellow = unscanned vertices, cyan = scanned vertices,
magenta = initial vertex, blue = original graph arcs,

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



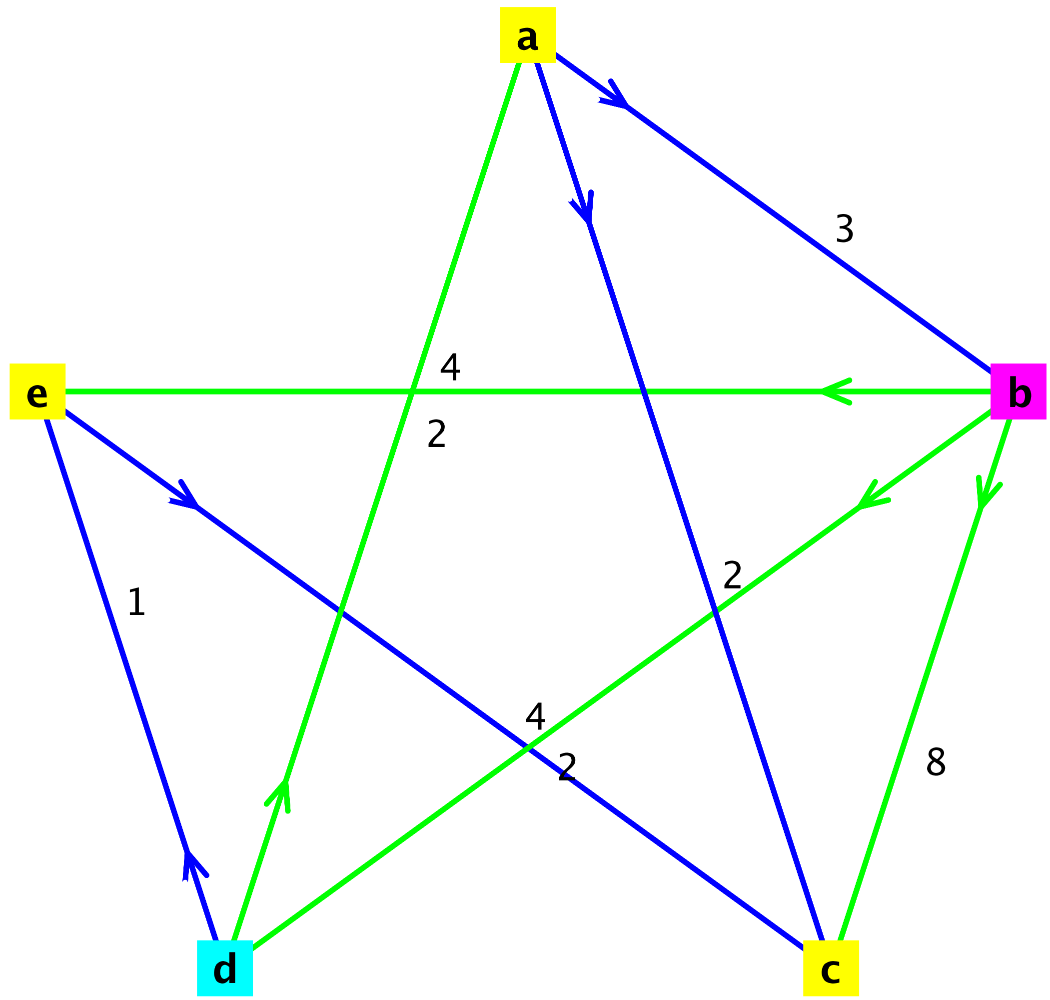
corrected arc ("b","d")



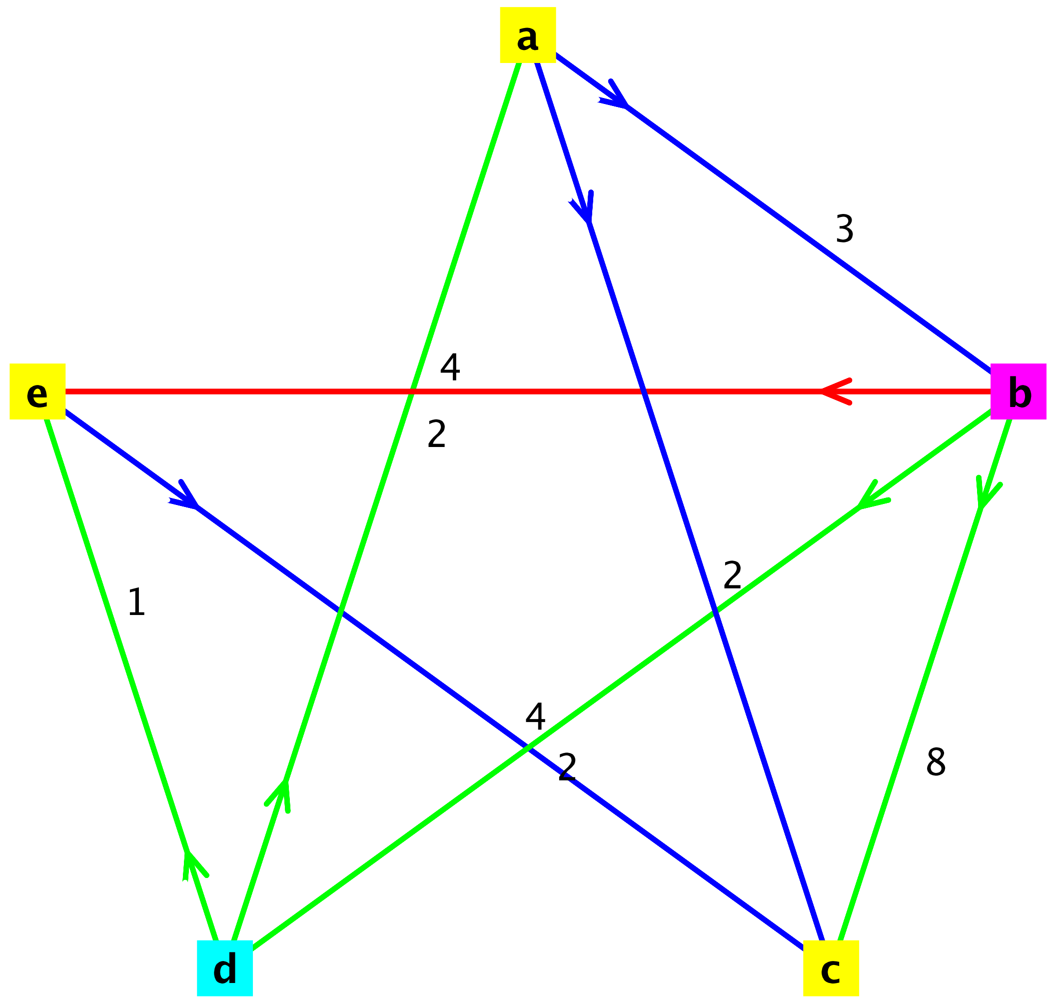
corrected arc ("b","e")



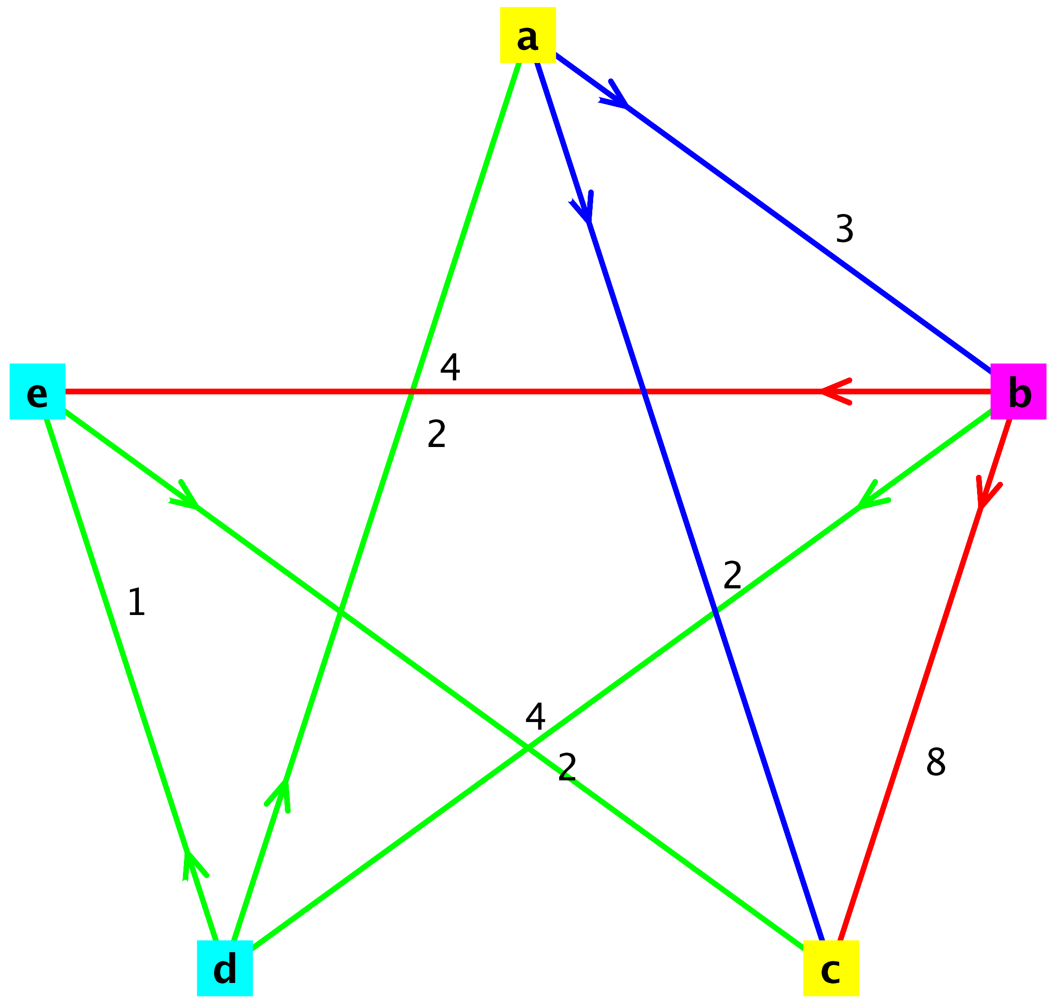
scanning vertex "d":
corrected arc ("d","a")



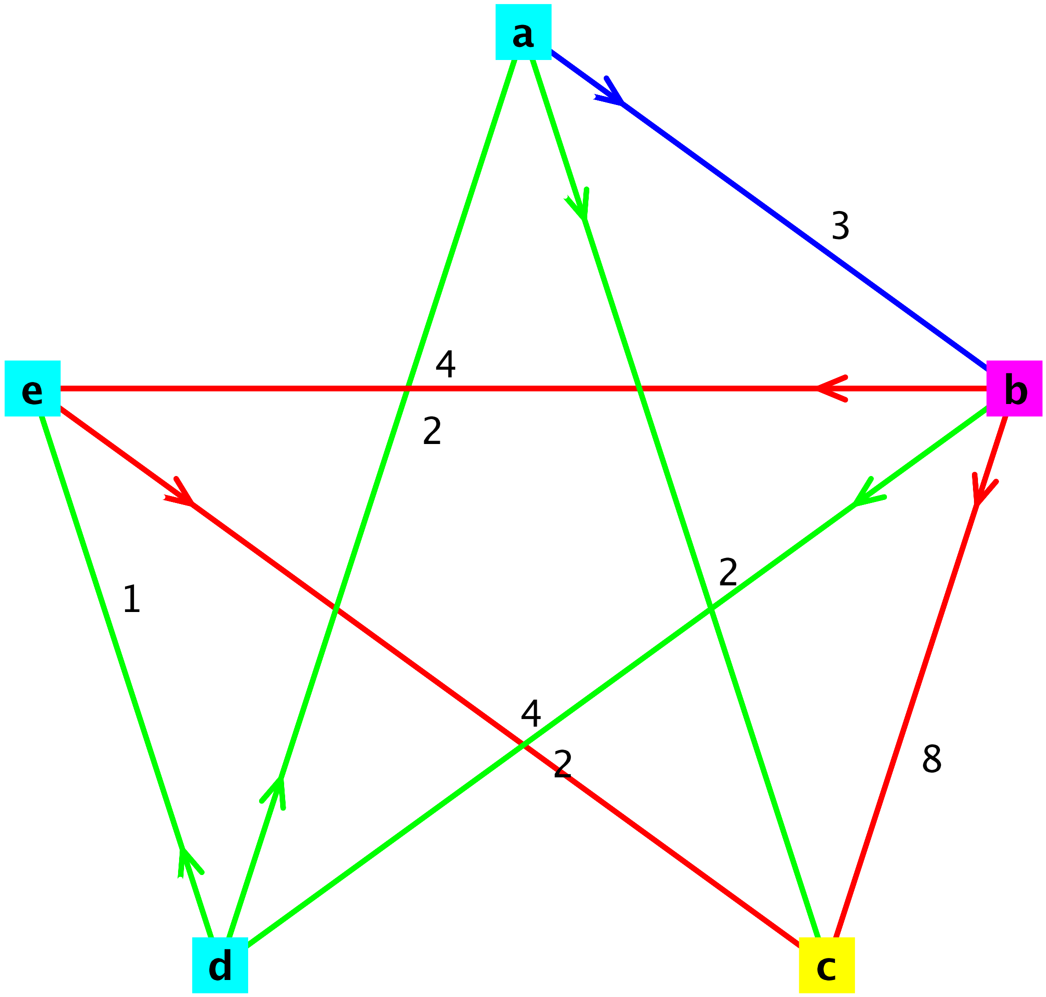
corrected arc ("d","e")



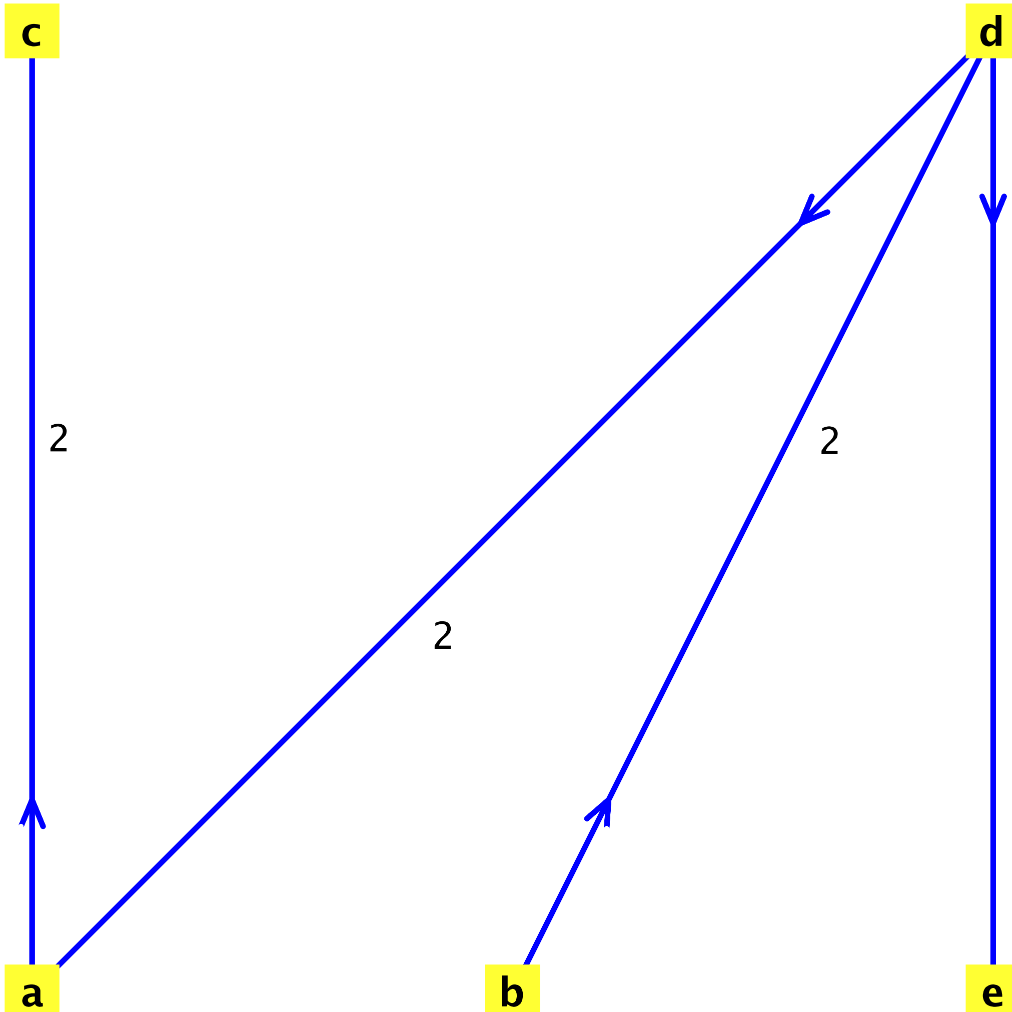

```
scanning vertex "e":
corrected arc ("e","c")
```




```
scanning vertex "a":
corrected arc ("a","c")
```




```
scanning vertex "c":
All vertices scanned, 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