

Dijkstra's routing algorithm

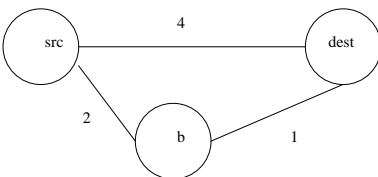
- devised by Edsger Dijkstra, is a shortest path routing algorithm
- a network can be expressed by a graph of nodes
 - each node has an arc between a neighbour and a static cost for this hop
- the algorithm is requested to find the shortest route between `src` and `dest`

Dijkstra's routing algorithm

- the algorithm is a breadth first search and it ensures that the first time a node is visited, then this will be the optimal route
 - cannot have negative costs between neighbours!

Tiny routing example

- consider in the following example we want to get from node `src` to `dest`



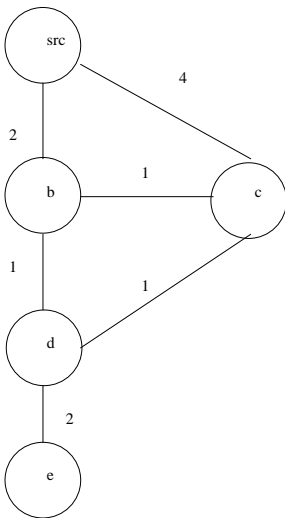
■

Tiny routing example

- starting at `src` we see that we have two choices, `b`, and `dest`
- we initially choose, `b`, as it is nearer
- now we have to examine
 - `dest` from `src`
 - `dest` from `b`
- in effect we are keeping an absolute running total to `dest` from `src` for all nodes visited
 - testing to see the next shortest hop
 - by examining each neighbour's relative cost

Slightly more complex routing example

Dijkstra's Algorithm



- keeps a record of the absolute distance of a node from the `src`
- it must be told the relative cost between nodes
 - distance separating neighbours
- maintains a list of those nodes already visited
- maintains a previous "where from?" value for each node

Dijkstra's Algorithm

Worked first graph example

```

function Dijkstra (Graph, src, dest):
  for each node n in Graph: # Initialisation
    dist[n] := infinity # currently unknown how to
    previous[n] := undefined
  dist[source] := 0 # cost of, src -> src = 0
  addToListOfChoices(source) # initially the only choice
  while list of choices is not empty:
    u := get_best_choice() # remove best node from choices
    if u == dest:
      return True # found route
    for each neighbor v of u: # where n has not yet been
      addToListOfChoices(v)
      alt = dist[u] + length(u, v) # add absolute(u) +
      # relative(u,v)
      if alt < dist[v] # have we found shorter route?
        dist[v] := alt # yes, so update absolute distance
        previous[v] := u # where did we come from?
  return False
  
```

- initially node's value of `dist` and `previous` is set to infinity and unknown respectively
- choice list is set to `src`
- the visited list is empty

Worked first graph example

choice	choiceList	Visited	notes
	{}	{}	initially
	src	{}	only choice
src	b, dest	{}	dist[c]=4, dist[b]=2
b	src	dest	chosen b as cost 2 neighbours, src and c ignore, src, as visited found better route to c dist[c] = 3
dest	src, b	{}	only choice final destination

Coursework

- download the Python Dijkstra's implementation
 - instrument the code to generate data similar to that on the previous slide
- note to do this only really involves adding print statements
- you will need to create your own local variables to count
 - number of nodes tested etc
- follow through these [exercises](http://flopsie.comp.glam.ac.uk/Glamorgan/gaius/scripting/19.html) (http://flopsie.comp.glam.ac.uk/Glamorgan/gaius/scripting/19.html) on how to generate time tables examples using Python
 - ignore anything to do with modules, concentrate on print and how to add and multiply numbers
- tutorials will be devoted to assignment support

Assignment Hints

- you need to concentrate on the function `findRoute` inside the `dijkstra.py` (<http://flopsie.comp.glam.ac.uk/Glamorgan/gaius/networks/dijkstra.py>) file
- this function matches the pseudo code given in these notes
 - you need to add various print statements to make it tell you what it is doing and what it is resolving at each step

Assignment Hints

- to print out the choices list you can
- ```
print "the choices list:", theGraph.getVisited()
```
- to print out the neighbours of node, n, you can
- ```
print u, "has neighbours", theGraph.getNode(u).getNeighbours()
```
- you can print the absolute distance currently known to a node, v, by
- ```
print "the distance is", theGraph.getNode(v).getDistanceFromSource()
```

## Assignment Hints

- some sample networks can be used to test out your modifications
  - [simplea.data](http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/networks/simplea.data) (<http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/networks/simplea.data>)
    - this network configuration is the same as used in the [Wikipedia entry](http://en.wikipedia.org/wiki/Dijkstra's_algorithm) ([http://en.wikipedia.org/wiki/Dijkstra's\\_algorithm](http://en.wikipedia.org/wiki/Dijkstra's_algorithm))
  - [simpleb.data](http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/networks/simpleb.data) (<http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/networks/simpleb.data>)
  - [simplec.data](http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/networks/simplec.data) (<http://floppsie.comp.glam.ac.uk/Glamorgan/gaius/networks/simplec.data>)
- for the final component of the coursework, you will need to create your own network description, say `simplified.data`

## simplea.data

```
■ #
simple route used by Wikipedia
#
Node neighbours[cost]
#
a b:2 c:4
b a:2 c:1
c b:1 a:4

find a -> c
answer is a -> b -> c total 3
```