CSC 421: Algorithm Design & Analysis

Spring 2019

Greedy algorithms

- greedy algorithms

  examples: optimal change, job scheduling
- Prim's algorithm (minimal spanning tree)
- Dijkstra's algorithm (shortest path)
- Huffman codes (data compression)
- applicability

1

Greedy algorithms

the greedy approach to problem solving involves making a sequence of choices/actions, each of which simply looks best at the moment

local view: choose the locally optimal option
hopefully, a sequence of locally optimal solutions leads to a globally optimal solution

example: optimal change

- given a monetary amount, make change using the fewest coins possible

  amount = 16¢ coins?

  amount = 96¢ coins?

2

Example: greedy change

while the amount remaining is not 0:

- select the largest coin that is ≤ the amount remaining
- add a coin of that type to the change
- subtract the value of that coin from the amount remaining

e.g., 96¢ = 50¢ + 25¢ + 10¢ + 10¢ + 1¢

will this greedy algorithm always yield the optimal solution?

for U.S. currency, the answer is YES

for arbitrary coin sets, the answer is NO

- suppose the U.S. Treasury added a 12¢ coin

GREEDY: 16¢ = 12¢ + 1¢ + 1¢ + 1¢ + 1¢	(5 coins)
OPTIMAL: 16¢ = 10¢ + 5¢ + 1¢	(3 coins)

3

Example: job scheduling

suppose you have a collection of jobs to execute and know their lengths

- want to schedule the jobs so as to *minimize* waiting time

Job 1:	5 minutes	Schedule 1-2-3: 0 + 5 + 15 = 20 minutes waiting
Job 2:	10 minutes	Schedule 3-2-1: 0 + 4 + 14 = 18 minutes waiting
Job 3:	4 minutes	Schedule 3-1-2: 0 + 4 + 9 = 13 minutes waiting

GREEDY ALGORITHM: do the shortest job first

i.e., while there are still jobs to execute, schedule the shortest remaining job

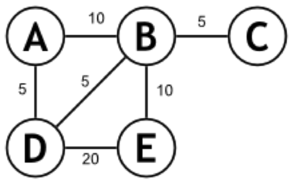
does the greedy approach guarantee the optimal schedule? efficiency?

4

Application: minimal spanning tree

consider the problem of finding a minimal spanning tree of a graph

- a *spanning tree* of a graph G is a tree (no cycles) made up of all the vertices and a subset of the edges of G
- a *minimal spanning tree* for a weighted graph G is a spanning tree with minimal total weight
- minimal spanning trees arise in many real-world applications
  e.g., wiring a network of computers; connecting rural houses with roads



spanning tree?

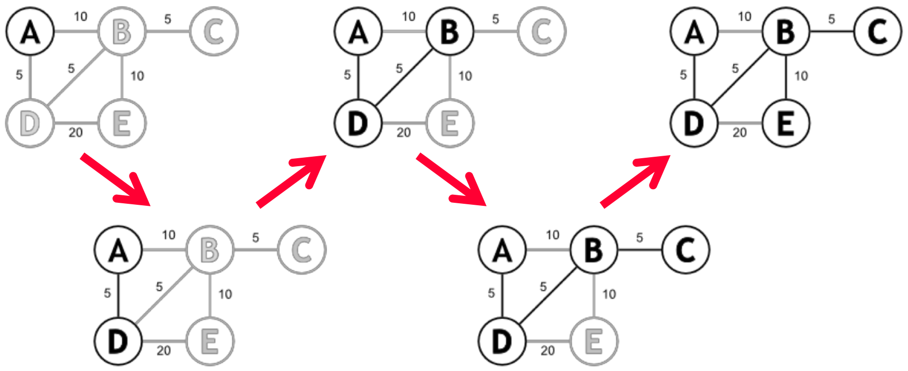
minimal spanning tree?

example from http://compprog.wordpress.com/

Prim's algorithm

to find a minimal spanning tree (MST):

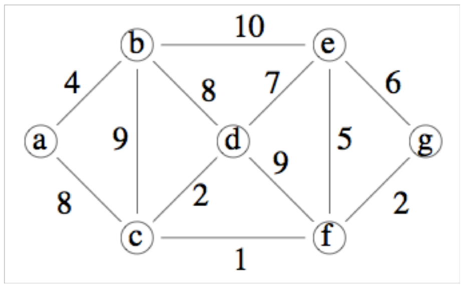
1. select any vertex as the root of the tree
2. repeatedly, until all vertices have been added:
   a) find the *lowest weight edge* with exactly one vertex in the tree
   b) select that edge and vertex and add to the tree



Prim's algorithm

to find a minimal spanning tree (MST):

1. select any vertex as the root of the tree
2. repeatedly, until all vertices have been added:
    a) find the *lowest weight edge* with exactly one vertex in the tree
    b) select that edge and vertex and add to the tree



minimal spanning tree?

is it unique?

7

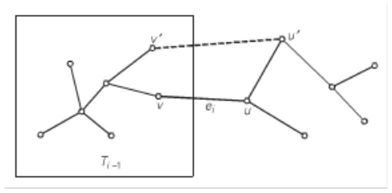
Correctness of Prim's algorithm

Proof (by induction): Each subtree T_1, T_1, …, $T_{|V|}$ in Prim's algorithm is contained in a MST. [Thus, $T_{|V|}$ is a MST.]

BASE CASE: T_1 contains a single vertex, so is contained in a MST.

ASSUME: T_1, …, T_{i-1} are contained in a MST.

STEP: Must show T_i is contained in a MST.
Assume the opposite, that T_i is not contained in a MST.
Let e_i be the new edge (i.e., minimum weight edge with exactly one vertex in T_{i-1}).
Since we assumed T_i is not part of any MST, adding e_i to a MST will yield a cycle.
That cycle must contain another edge with exactly one vertex in T_{i-1} .
Replacing that edge with e_i yields a spanning tree, and since e_i had the minimal weight of any edge with exactly one vertex in T_{i-1}, it is a MST.
Thus, T_i is contained in a MST $\rightarrow$ CONTRADICTION!



8

Efficiency of Prim's algorithm

brute force (i.e., adjacency matrix):

- simple (conservative) analysis
  for each vertex, must select the least weight edge → O(|V| * |E|)
- more careful analysis:
  note that the number of eligible edges is shrinking as the tree grows
  Σ (|V| * deg(v_i)) = O($|V|^2$ + |E|) = O($|V|^2$)

smarter implementation:

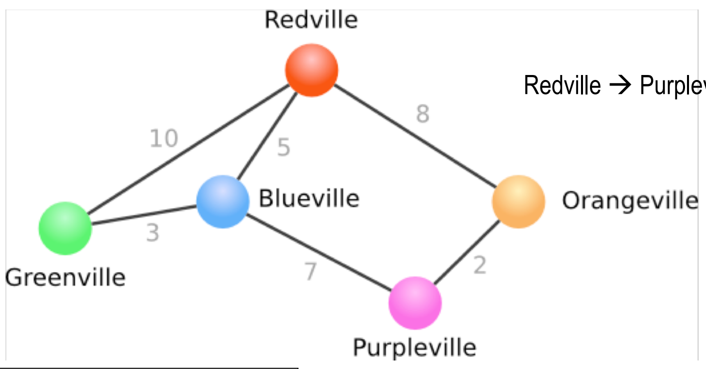
- use a priority queue (min-heap) to store vertices, along with minimal weight edge
- to select each vertex & remove from PQ → |V| * O(log |V|) = O(|V| log |V|)
- to update each adjacent vertex after removal (at most once per edge)
  →|E| * O(log |V|) = O(|E| log |V|)
- overall efficiency is O((|E|+|V|) log |V|)

9

Application: shortest path

consider the general problem of finding the shortest path between two nodes in a graph

- flight planning and word ladder are examples of this problem
  - in these cases, edges have uniform cost (shortest path = fewest edges)
- if we allow non-uniform edges, want to find lowest cost/shortest distance path



Redville → Purpleville ?

example from http://www.algolist.com/Dijkstra's_algorithm

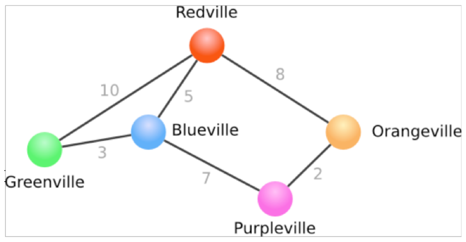
10

Modified BFS solution

we could modify the BFS approach to take cost into account

- instead of adding each newly expanded path to the end (i.e., queue), add in order of path cost (i.e., priority queue)

```
[ [Redville]:0 ]

[ [Redville, Blueville]:5,
  [Redville, Orangeville]:8,
  [Redville, Greenville]:10 ]

[ [Redville, Orangeville]:8,
  [Redville, Blueville, Greenville]:8,
  [Redville, Greenville]:10,
  [Redville, Blueville, Purpleville]:12 ]

[ [Redville, Blueville, Greenville]:8,
  [Redville, Greenville]:10,
  [Redville, Orangeville, Purpleville]:10,
  [Redville, Blueville, Purpleville]:12 ]

[ [Redville, Greenville]:10,
  [Redville, Orangeville, Purpleville]:10,
  [Redville, Blueville, Purpleville]:12 ]

[ [Redville, Orangeville, Purpleville]:10,
  [Redville, Blueville, Purpleville]:12,
  [Redville, Greenville, Blueville]:13 ]
```



note: as before, requires lots of memory to store all the paths

HOW MANY?

Dijkstra's algorithm

alternatively, there is a straightforward greedy algorithm for shortest path

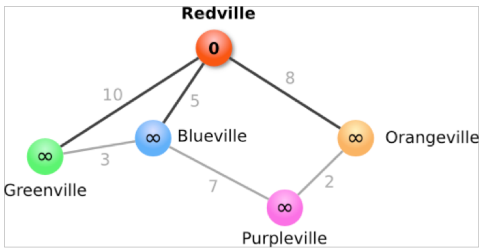
Dijkstra's algorithm

1. Begin with the start node. Set its value to 0 and the value of all other nodes to infinity. Mark all nodes as unvisited.
2. For each unvisited node that is adjacent to the current node:
   a) If (value of current node + value of edge) < (value of adjacent node), change the value of the adjacent node to this value.
   b) Otherwise leave the value as is.
3. Set the current node to visited.
4. If unvisited nodes remain, select the one with smallest value and go to step 2.
5. If there are no unvisited nodes, then DONE.

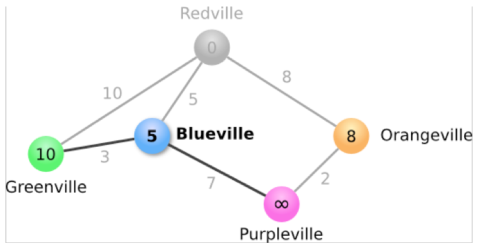
this algorithm is O(N^2), requires only O(N) additional storage

Dijkstra's algorithm: example

suppose want to find shortest path from Redville to Purpleville

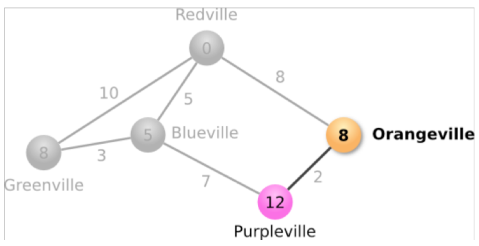


1. Begin with the start node. Set its value to 0 and the value of all other nodes to infinity. Mark all nodes as unvisited



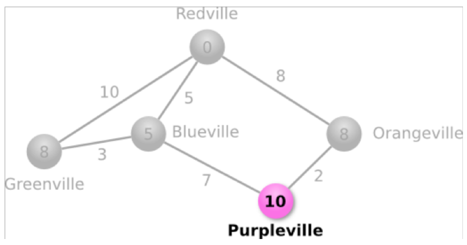
2. For each unvisited node that is adjacent to the current node:
   a) If (value of current node + value of edge) < (value of adjacent node), change the value of the adjacent node to this value.
   b) Otherwise leave the value as is.
3. Set the current node to visited.

Dijkstra's algorithm: example cont.



4. If unvisited nodes remain, select the one with smallest value and go to step 2.

Blueville: set Greenville to 8 and Purpleville to 12; mark as visited.
Greenville: no unvisited neighbors; mark as visited.



4. If unvisited nodes remain, select the one with smallest value and go to step 2.

Orangeville: set Purpleville to 10; mark as visited.
Purpleville: no unvisited neighbors; mark as visited.

5. If there are no unvisited nodes, then DONE.

With all nodes labeled, can easily construct the shortest path – HOW?

Correctness & efficiency of Dijkstra's algorithm

analysis of Dijkstra's algorithm is similar to Prim's algorithm

- can show that each greedy selection is safe, leads to shortest path

- brute force (i.e., adjacency matrix) approach
  for each vertex, need to select shortest edge → O(|V| * |E|)
  or, more carefully, Σ (|V| * deg(v_i)) = O(|V|^2 + |E|) = O(|V|^2)

- smarter implementation
  use a priority queue (min-heap) to store vertices, along with minimal weight edge

  to select each vertex & remove from PQ → |V| * O(log |V|) = O(|V| log |V|)
  to update each adjacent vertex after removal → |E| * O(log |V|) = O(|E| log |V|)

  overall efficiency is O((|E|+|V|) log |V|)

15

Another application: data compression

in a multimedia world, document sizes continue to increase

- a 6 megapixel digital picture is 2-4 MB
- an MP3 song is ~3-6 MB
- a full-length MPEG movie is ~800 MB

storing multimedia files can take up a lot of disk space

- perhaps more importantly, downloading multimedia requires significant bandwidth

it could be a lot worse!

- image/sound/video formats rely heavily on data compression to limit file size
  e.g., if no compression, 6 megapixels * 3 bytes/pixel = ~18 MB

- the JPEG format provides 10:1 to 20:1 compression without visible loss

16

Audio, video, & text compression

audio & video compression algorithms rely on domain-specific tricks

- lossless image formats (GIF, PNG) recognize repeating patterns (e.g. a sequence of white pixels) and store as a group
- lossy image formats (JPG, XPM) round pixel values and combine close values
- video formats (MPEG, AVI) take advantage of the fact that little changes from one frame to next, so store initial frame and changes in subsequent frames
- audio formats (MP3, WAV) remove sound out of hearing range, overlapping noises

what about text files?

- in the absence of domain-specific knowledge, can't do better than a fixed-width code
  e.g., ASCII code uses 8-bits for each character

```
'0': 00110000    'A': 01000001    'a': 01100001
'1': 00110001    'B': 01000010    'b': 01100010
'2': 00110010    'C': 01000011    'c': 01100011
  .                .                .
  .                .                .
  .                .                .
```

17

Fixed- vs. variable-width codes

suppose we had a document that contained only the letters a-f

- with a fixed-width code, would need 3 bits for each character

```
a   000         d   011
b   001         e   100
c   010         f   101
```

- if the document contained 100 characters, 100 * 3 = 300 bits required

however, suppose we knew the distribution of letters in the document

```
a:45,  b:13,  c:12,  d:16,  e:9,  f:5
```

- can customize a variable-width code, optimized for that specific file

```
a   0           d   111
b   101         e   1101
c   100         f   1100
```

- requires only 45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4 = 224 bits

18

Huffman codes

Huffman compression is a technique for constructing an optimal* variable-length code for text

*optimal in that it represents a *specific* file using the fewest bits (among all symbol-for-symbol codes)

Huffman codes are also known as *prefix codes*

- no individual code is a prefix of any other code

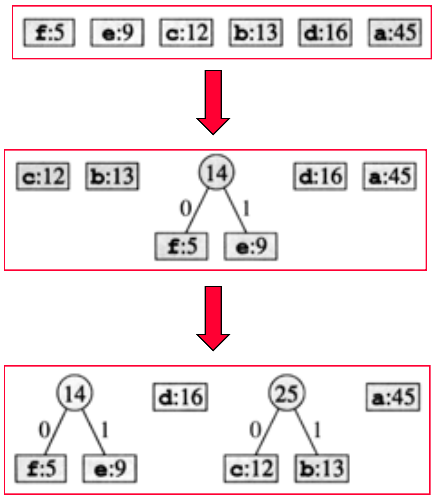
a	0	d	111
b	101	e	1101
c	100	f	1100

- this makes decompression unambiguous: `1010111110001001101`
- *note:* since the code is specific to a particular file, it must be stored along with the compressed file in order to allow for eventual decompression

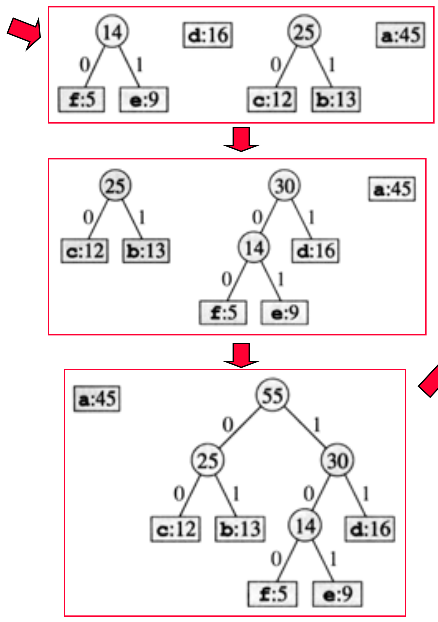
Huffman trees

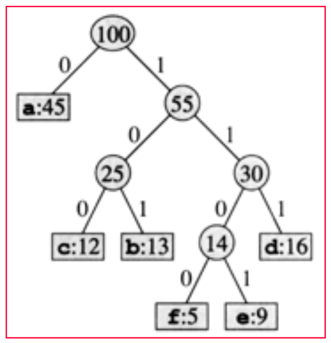
to construct a Huffman code for a specific file, utilize a greedy algorithm to construct a *Huffman tree*:

1. process the file and count the frequency for each letter in the file
2. create a single-node tree for each letter, labeled with its frequency
3. repeatedly,
   a. pick the two trees with smallest root values
   b. combine these two trees into a single tree whose root is labeled with the sum of the two subtree frequencies
4. when only one tree remains, can extract the codes from the Huffman tree by following edges from root to each leaf (left edge = 0, right edge = 1)



Huffman tree construction (cont.)





the code corresponding to each letter can be read by following the edges from the root: left edge = 0, right edge = 1

a:	0	d:	111
b:	101	e:	1101
c:	100	f:	1100

Huffman code compression

note that at each step, need to pick the two trees with smallest root values

- perfect application for a priority queue (min-heap)

- store each single-node tree in a priority queue (PQ) O(N log N)
- repeatedly, O(N) times
  remove the two min-value trees from the PQ O(log N)
  combine into a new tree with sum at root and insert back into PQ O(log N)
  ======================
  total efficiency = O(N log N)

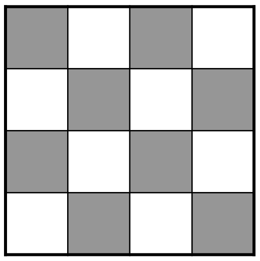
while designed for compressing text, it is interesting to note that Huffman codes are used in a variety of applications

- the last step in the JPEG algorithm, after image-specific techniques are applied, is to compress the resulting file using a Huffman code
- similarly, Huffman codes are used to compress frames in MPEG (MP4)

Greed is good?

IMPORTANT: the greedy approach is not applicable to all problems

- but when applicable, it is very effective (no planning or coordination necessary)



GREEDY approach for N-Queens: start with first row, find a valid position in current row, place a queen in that position then move on to the next row

since queen placements are not independent, local choices do not necessarily lead to a global solution

GREEDY does not work – need a more holistic approach

23