Analysis of Algorithms CECS 528

Topic 7.2. Greedy algorithm
Huffman coding, Fractional knapsack,
Dijkstra algorithm

Huffman encoding

In the MP3 audio compression scheme, a sound signal is encoded in three steps.

1. It is digitized by sampling at regular intervals, yielding a sequence of real numbers $s_1, s_2, \ldots, s_T$. For instance, at a rate of 44,100 samples per second, a 50-minute symphony would correspond to $T = 50 \times 60 \times 44{,}100 \approx 130$ million measurements.
2. Each real-valued sample s_t is quantized: approximated by a nearby number from a finite set . This set is carefully chosen to exploit human perceptual limitations, with the intention that the approximating sequence is indistinguishable from $s_1, s_2, \ldots, s_T$ by the human ear.
3. The resulting string of length T over alphabet Γ is encoded in binary.

Huffman encoding

- It is in the last step that Huffman encoding is used. To understand its role, let's look at an example in which T is 130 million and the alphabet Γ consists of just four values, denoted by the symbols A, B, C, D.
- What is the most economical way to write this long string in binary?
- The obvious choice is to use 2 bits per symbol—say codeword 00 for A, 01 for B, 10 for C , and 11 for D—in which case 260 megabits are needed in total.
- Can there possibly be a better encoding than this?

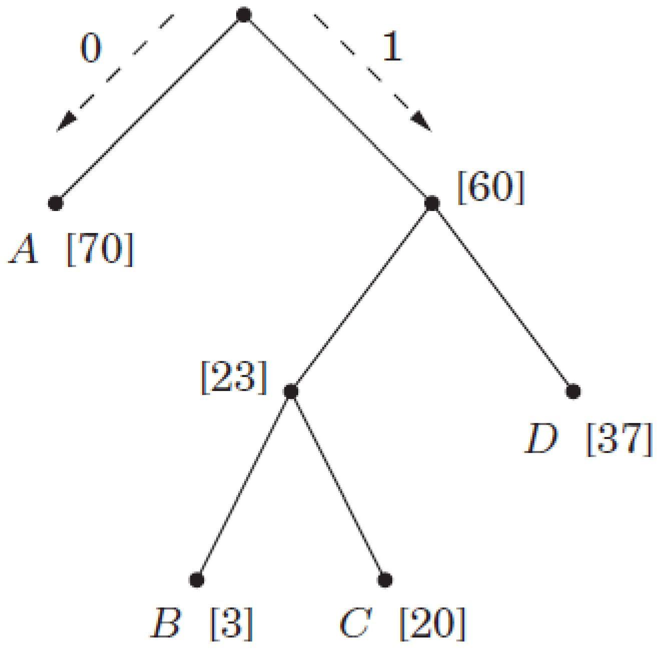
Symbols' frequency

Let's take a closer look at this particular sequence and find that the four symbols are not equally abundant.

Symbol	Frequency
A	*70 million*
B	*3 million*
C	*20 million*
D	*37 million*

Variable length encoding

Symbol	Codeword
A	0
B	100
C	101
D	11



Efficiency

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.

Suppose we have a 100,000-character data file that we wish to store compactly. The frequencies are given as

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

Only 6 different characters appear, and the character **a** occurs 45,000 times.

Variable length code

A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.

Here the 1-bit string 0 represents **a**, and the 4-bit string 1100 represents **f**.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

This code requires

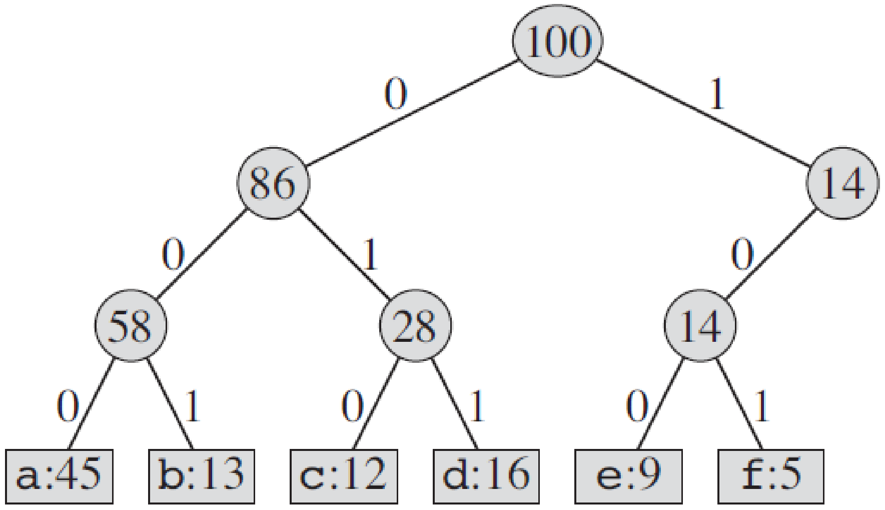
$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

to represent the file, a savings of approximately 25%.

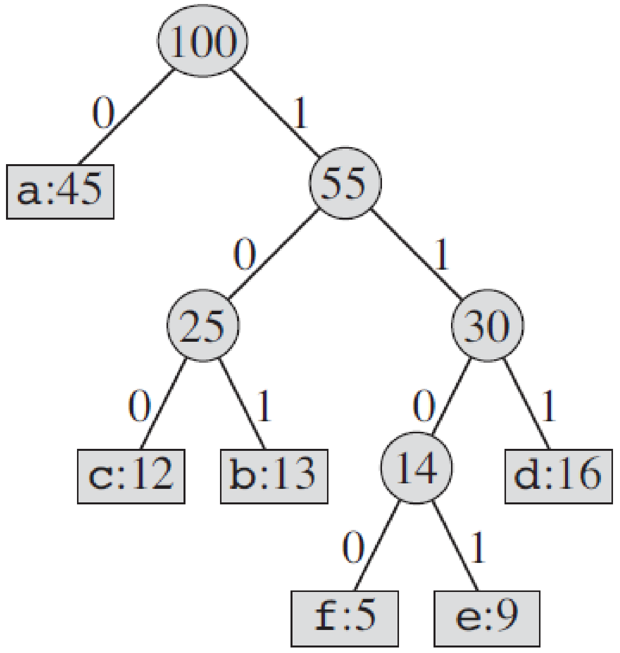
Tree of coding schemes

(a) The tree corresponding to the fixed-length code **a** = 000, . . . , **f** = 101.

(b) The tree corresponding to the optimal prefix code **a** = 0, **b** = 101, . . . , **f** = 1100.



(a)

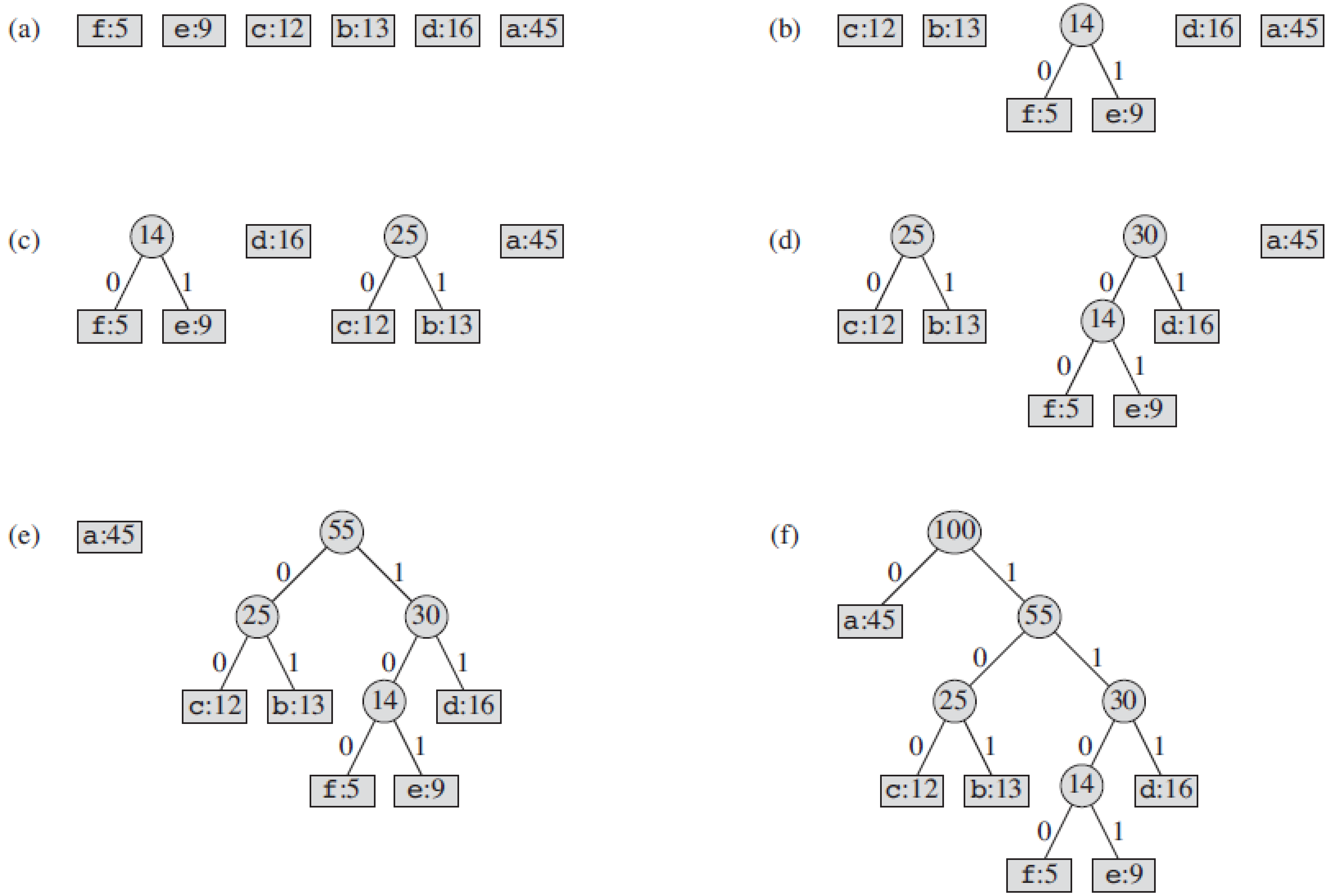


(b)

Constructing the Huffman code

```
HUFFMAN($C$)
1  $n = |C|$
2  $Q = C$
3  for $i = 1$ to $n - 1$
4      allocate a new node $z$
5      $z.left = x =$ EXTRACT-MIN($Q$)
6      $z.right = y =$ EXTRACT-MIN($Q$)
7      $z.freq = x.freq + y.freq$
8      INSERT($Q, z$)
9  return EXTRACT-MIN($Q$)      // return the root of the tree
```

The steps of Huffman code



Entropy

The total number of bits to encode a message is found by the Huffman code.

This number represents the *entropy*.

The inherent unpredictability, or *randomness, of a probability distribution can be measured* by the extent to which it is possible to compress data drawn from that distribution.

more compressible ≡ *less random* ≡ *more predictable*

Entropy

Suppose there are n possible outcomes, with probabilities $p_1, p_2, \ldots, p_n$. If a sequence of m values is drawn from the distribution, then the ith outcome will pop up roughly mp_i times (if m is large).

For simplicity, assume these are exactly the observed frequencies, and moreover that the p_i 's are all powers of 2 (that is, of the form $1/2^k$). It can be seen by induction that the number of bits needed to encode the sequence is

$$\sum_{i=1}^{n} mp_i \log(1/p_i)$$

Thus the average number of bits needed to encode a single draw from the distribution is

$$\sum_{i=1}^{n} p_i \log \frac{1}{p_i}.$$

Horn clauses

- In order to display human-level intelligence, a computer must be able to perform at least some modicum of logical reasoning.
- Horn formulas are a particular framework for doing this, for expressing logical facts and deriving conclusions.
- The most primitive object in a Horn formula is a Boolean variable, taking value either true or false.

Horn clauses

A literal is either a variable **x** or its negation ($\bar{x}$). In Horn formulas, knowledge about variables is represented by two kinds of clauses:

1. Implications, whose left-hand side is an AND of any number of positive literals and whose right-hand side is a single positive literal. These express statements of the form “if the conditions on the left hold, then the one on the right must also be true.” $(z \wedge w) \Rightarrow u$. A degenerate type of implication is the singleton “$\Rightarrow$ x,” meaning simply that x is true.
2. Pure negative clauses, consisting of an OR of any number of negative literals, as in $(\bar{u} \vee \bar{v} \vee \bar{y})$

Horn clause

- Given a set of clauses of these two types, the goal is to determine whether there is a consistent explanation: an assignment of true/false values to the variables that satisfies all the clauses. This is also called a *satisfying assignment*.
- The implications amounts to setting some of the variables to true, while the negative clauses encourage to make them false.
- The strategy for solving a Horn formula is this:
- Start with all variables false.
- Then proceed to set some of them to true, one by one, but very reluctantly, and only if we absolutely have to because an implication would otherwise be violated.
- Once we are done with this phase and all implications are satisfied, only then do we turn to the negative clauses and make sure they are all satisfied.

Algorithm

```
Input:  a Horn formula
Output: a satisfying assignment, if one exists

set all variables to false

while there is an implication that is not satisfied:
  set the right-hand variable of the implication to true

if all pure negative clauses are satisfied: return the
assignment
else: return “formula is not satisfiable”
```

Example

Suppose the formula is

$(w \wedge y \wedge z) \Rightarrow x$, $(x \wedge z) \Rightarrow w$, $x \Rightarrow y$, $\Rightarrow x$, $(x \wedge y) \Rightarrow w$, $(\bar{x} \vee \bar{y} \vee \bar{w})$, $(\bar{z})$.

Start with everything **false** and then notice that *x must be true on account of the singleton implication* $\Rightarrow$ *x.*

Then we see that y must also be true, because of $x \Rightarrow y$. And so on.

Is the algorithm correct?

To see why the algorithm is correct, notice that if it returns an assignment, this assignment satisfies both the implications and the negative clauses, and so it is indeed a satisfying truth assignment of the input Horn formula.

So we only have to convince ourselves that if the algorithm finds no satisfying assignment, then there really is none.

This is so because the greedy rule maintains the following invariant:

If a certain set of variables is set to true, then they must be true in any satisfying assignment.

Hence, if the truth assignment found after the **while** loop does not satisfy the negative clauses, there can be no satisfying truth assignment.

Example

$(w \wedge y \wedge z) \Rightarrow x$, $(x \wedge z) \Rightarrow w$, $x \Rightarrow y$, $\Rightarrow x$, $(x \wedge y) \Rightarrow w$, $(\bar{x} \vee \bar{y} \vee \bar{w})$, $(\bar{z})$.

- On the example above, Greedy-Horn first flips x to true, forced by the implication $\rightarrow x$.
- Then y gets forced to true (from $x \rightarrow y$), and similarly w is forced to true.
- Looking at the pure negative clauses, we find that the first is not satisfied, and hence we conclude the original formula had no truth assignment.

Prolog

- Horn formulas lie at the heart of Prolog ("programming by logic"), a language in which you program by specifying desired properties of the output, using simple logical expressions.
- The workhorse of Prolog interpreters is our greedy satisfiability algorithm.
- Conveniently, it can be implemented in time linear in the length of the formula.

Fractional Knapsack Problem

- A set of goods G is to be loaded into a knapsack.
- Good g weighs $w(g)$ and induces a total profit of $p(g)$. Assuming the knapsack has a weight capacity M, determine what fraction $f(g)$ of each good should be loaded onto the knapsack in order to maximize profit.

Algorithm

- Name: Fractional Knapsack Algorithm (FKA)
- Input Fractional Knapsack instance (G, w, p, M).
- Output Function $f : G \to Q$, where $f(g)$ is the fraction of good g that should be loaded into a knapsack.
- Begin Algorithm
- Initialize $f : G \to Q$ to be undefined for all $g \in G$.
- Initialize $c = M$.

  // Comment: c denotes the current capacity of the knapsack.

Algorithm

```
While $c > 0$ and $f$ is undefined for some $g \in G$,
– Choose $g$ for which $f(g)$ is undefined, and for which $p(g)/w(g)$
  is maximum.
– If $c \geq w(g)$, then define $f(g) = 1$, and set $c = c - w(g)$.
– Else define $f(g) = c/w(g)$ and set $c = 0$.
• For all $g$ for which $f(g)$ is undefined
– Set $f(g) = 0$.
• Return $f$.
• End Algorithm
```

Dijkstra’s Algorithm

For a given source node in the graph, the algorithm finds the shortest path between that vertex and every other vertex.

It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex.

In this case, the algorithm should be stopped once the shortest path to the destination vertex has been determined.

Dijkstra’s Algorithm

- Input

– weighted graph $G = (\{v_0, \ldots, v_{n-1}\}, E, c)$

– source vertex $u = v_0$.

- Output

– integer array $d[0 : n - 1]$ where $d[i]$ is the distance from u to v_i

– vertex array $p[0 : n-1]$ where $p[i]$ is the parent of v_i in some minimum cost path from u to v_i.

Algorithm

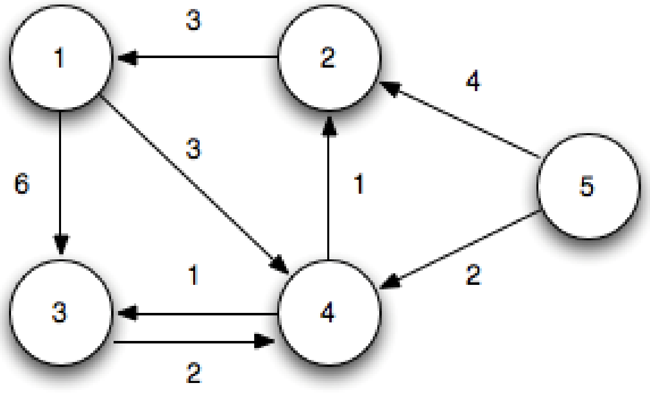
- Initialize integer array $d[0 : n-1]$ so that $d[0] = 0$ and $d[i] = \infty$ for all $1 \leq i \leq n-1$.
- Initialize vertex array $p[0 : n-1]$ so that $p[i] = \varnothing$; for all $0 \leq i \leq n-1$.
- Initialize Boolean array $f[0 : n-1]$ so that $f[0]$ = true and $f[i]$ = false for all $1 \leq i \leq n-1$.
- Comment: array f indicates if the distance from u to v_i has been found.
- Initialize $l = 0$.
- Comment: l is the index of vertex whose distance from u was last found.

Algorithm

- While there exists a vertex whose distance from *u* has not been found,
  - For every vertex v_i whose distance from *u* has not been found,
    - If $(v_l, v_i) \in E$ and $d[l] + c(v_l, v_i) < d[i]$ then assign $d[i] = d[l] + c(v_l, v_i)$.
  - Set $l = m$ where $d[m] = \min\{d[i] | f[i] = \text{false}\}$.
  - Set $f[m] = \text{true}$.
  - Set $p[m] = v_l$.
- Return *d* and *p*.

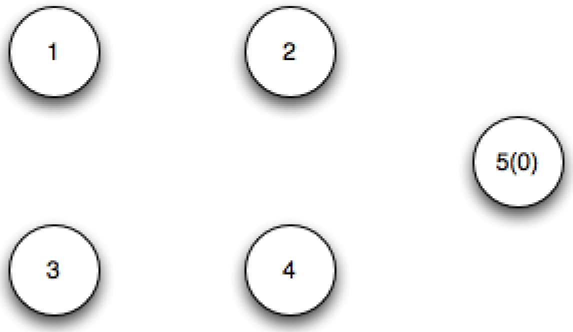
Example

Consider the directed graph:



Initial step

Using vertex 5 as the source (setting its distance to 0), we initialize all the other distances to ∞, set $S = \emptyset$, and place all the vertices in the queue (resolving ties by lowest vertex number). The predecessor node is π.



Q

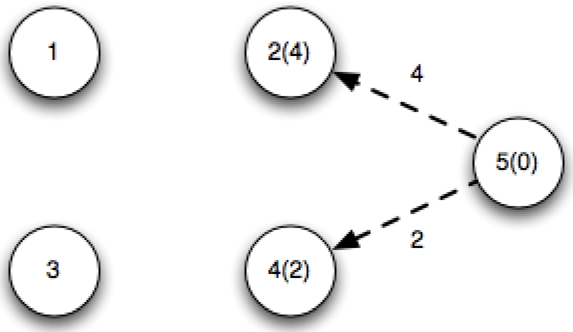
d	u	π
0	5	/
∞	1	/
∞	2	/
∞	3	/
∞	4	/

S

u	d	π

Iteration 1

Dequeue vertex 5 placing it in *S* (with a distance 0) and relaxing edges (u_5,u_2) and (u_5,u_4) then reprioritizing the queue



Q

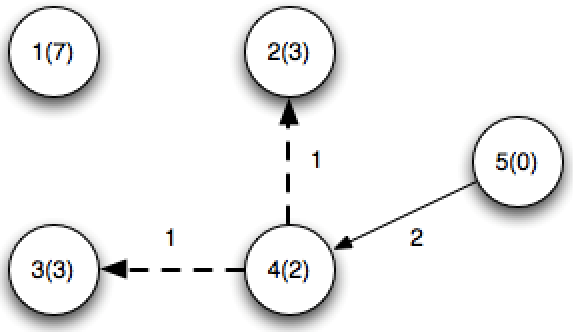
d	u	π
2	4	**5**
4	2	**5**
∞	1	/
∞	3	/

S

u	d	π
5	**0**	**/**

Iteration 2

Dequeue vertex 4 placing it in *S* (with a distance 2) and relaxing edges (u_4,u_2) and (u_4,u_3) then reprioritizing the queue. Note edge (u_4,u_2) finds a shorter path to vertex 2 by going through vertex 4



Q

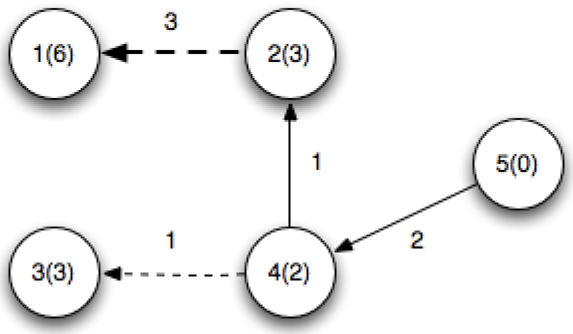
d	u	π
3	2	**4**
3	3	**4**
∞	1	/

S

u	d	π
5	0	/
4	**2**	**5**

Iteration 3

Dequeue vertex 2 placing it in *S* (with a distance 3) and relaxing edge (u_2,u_1) then reprioritizing the queue.



Q

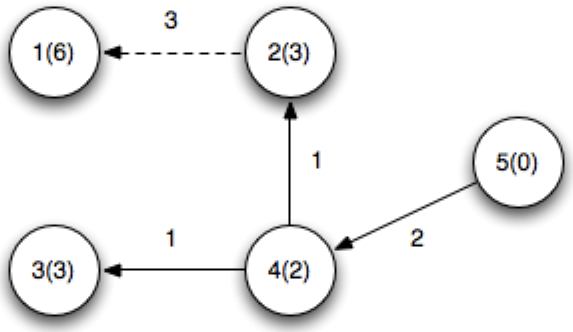
d	u	π
3	3	4
6	1	**2**

S

u	d	π
5	0	/
4	2	5
2	**3**	**4**

Iteration 4

Dequeue vertex 3 placing it in *S* (with a distance 3) and relaxing no edges then reprioritizing the queue.



Q

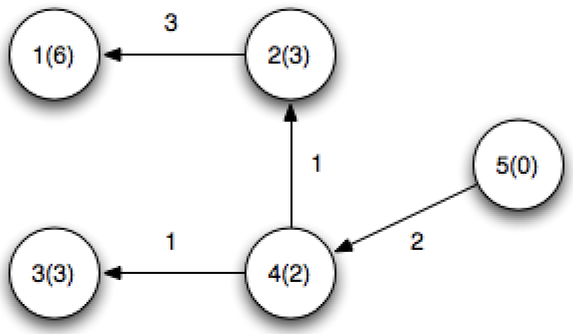
d	u	π
6	1	2

S

u	d	π
5	0	/
4	2	5
2	3	4
3	**3**	**4**

Iteration 5

Dequeue vertex 1 placing it in *S* (with a distance 6) and relaxing no edges.



Q

d	u	π

S

u	d	π
5	0	/
4	2	5
2	3	4
3	3	4
1	**6**	**2**

Theorem

For every $v_i \in V$, Dijkstra's Algorithm yields the correct distance from *u* to *v*.

For simplicity we assume that $G = (V,E)$ is a directed simple weighted graph.

However, the algorithm holds for all directed and undirected general graphs with nonnegative cost function c.

Proof's Idea

- Induction.
- We claim that every time a vertex x *is marked finished,* it contains the length of the shortest path from v *to* x *as its value.*
- Clearly, this is true for v. Assume it is true up to the point where we are about to mark x finished, but assume that there is a strictly shorter path from v to x that exists somewhere in the graph.
- There must be at least one unfinished vertex along this path; otherwise, the value on vertex x is incorrect.
- Let x' be the first unfinished vertex along this path. Notice that the value on the vertex x' must be strictly less than the value on vertex x by the inductive hypothesis.
- Thus, Why are we finishing x rather than x'?