Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme

Programming, Data Structures and Algorithms using Python

Summary of weeks 10 & 11

Content

- String matching
  - Boyer-Moore Algorithm
  - Rabin-Karp Algorithm
  - Automata
  - Knuth-Morris-Pratt Algorithm
  - Tries
  - Regular Expressions
- Linear programming
- Network flows
  - Ford-Fulkerson algorithm
- Reductions
- Checking algorithms
- P, NP and NP-Complete

String matching

- Searching for a pattern is a fundamental problem when dealing with text
- Formal definition:
  - A ***text*** string t of length n
  - A ***pattern*** string p of length m
  - Both t and p are drawn from an ***alphabet*** of valid letters, denoted by $\sum$
  - Find every position i in t such that $t[i : i + m] == p$
- Complexity of naïve algorithm: $O(nm)$

Boyer-Moore Algorithm

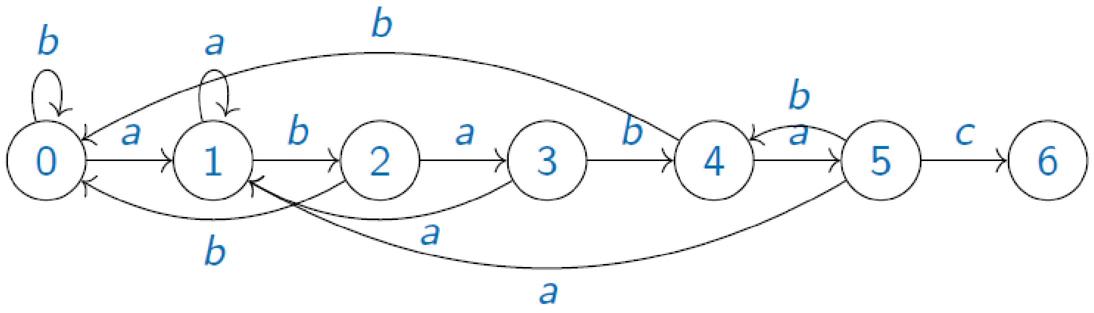
- For each starting position i in t, compare t[i : i + m] with p
  - Scan t[i : i + m] right to left
- If a letter c in t that does not appear in p
  - Shift the next scan to position after mismatched letter c
- If a letter c in t that does appear somewhere in p
  - Align rightmost occurrence of c in p with t
- Use a dictionary last
  - For each c in p, last[c] records right most position of c in p
- Without dictionary, computing last is a bottleneck, complexity is $O(|\Sigma|)$
- The algorithm works well in practice but the worst case complexity remains $O(nm)$

Rabin-Karp Algorithm

- Any string over $\sum$ can be thought of as a number in base 10
- Pattern p is an m digit number n_p
- Each substring of length m in the text t is again an m digit number
- Scan t and compare the number n_b generated by each block of m letters with the pattern number n_p
- Whenever $n_b = n_p$, scan and validate like brute force
  - It can be a false positive (spurious hit)
- In practice number of spurious matches will be small but the worst case complexity remains $O(nm)$
- If $|\sum|$ is small enough to not require modulo arithmetic, overall time is $O(n + m)$ or $O(n)$, since $m \ll n$

Automata

- It is used to keep track of longest prefix that we have matched
- It is a special type of graph where nodes are states and edges describe how to extend the match
- Using this automaton, we can do string matching in $O(n)$
- Bottleneck is precomputing the automaton
  - Overall complexity: $O(m^3.|\sum|)$



Processing *abababac*

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{c} 6$

Knuth-Morris-Pratt Algorithm

- It is very similar to what we did in automaton
- Precomputing step can be handled effectively using fail()
- The Knuth-Morris-Pratt algorithm efficiently computes the automaton describing prefix matches in the pattern p
- Complexity of preprocessing the fail() is $O(m)$
- After preprocessing, can check matches in the text t in $O(n)$
- Overall, KMP algorithm works in time $O(m+n)$
- However, the Boyer-Moore algorithm can be faster in practice, skipping many positions

Tries

- A ***trie*** is a special kind of tree deriver from “information re***trie***val”
- Rooted tree
  - Other than root, each node labelled by a letter from Σ
  - Children of a node have distinct labels
- Each maximal path is a word
  - One word should not be a prefix of another
  - Add special end of word symbol $\$$
- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w, follow its path
  - If the node we reach has $\$$ as a successor, $w \in S$
  - If we cannot complete the path, $w \notin S$
- Using a suffix trie we can answer the following:
  - Is w a substring of s
  - How many times does w occur as a substring in s
  - What is the longest repeated substring in s

Regular Expressions

- The automaton we built had a linear structure, with a single path from start to finish
- In general, automata can follow multiple paths and accept multiple words
- The set of words that automata can accept are called regular sets
- Each pattern p describes a set of words, those that it matches
- The sets we can describe using patterns are exactly the same as those that can be accepted by automata
- Those patterns are called regular expressions

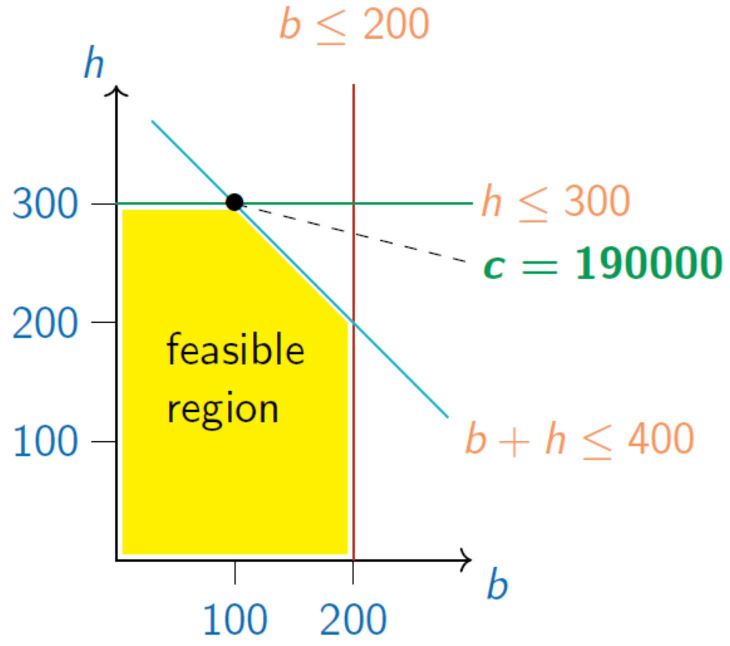
- For every automaton, we can construct a pattern p that matches exactly the words that the automaton accepts
- For every pattern p, we can construct an automaton that accepts all words that match p
- We can extend string matching to pattern matching by building an automaton for a pattern p and processing the text through this automaton
- Python provides a library for matching regular expressions

Linear programming

- Profit for each box of barfis is Rs.100
- Profit for each box of halwa is Rs.600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Constrains:

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$



Objective:

- Maximize $100b + 600h$

Linear programming

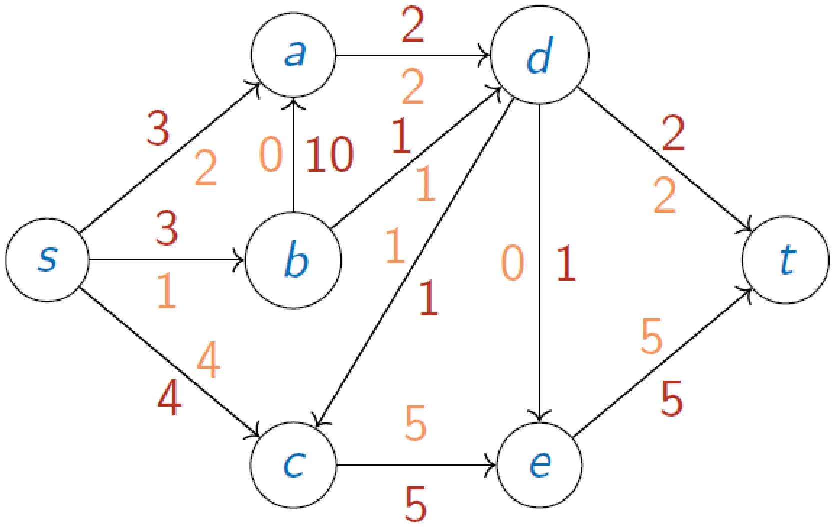
- Constraints and objective to be optimized are linear functions
  - Constrains: $a_1x_1 + a_2x_2 + \ldots + a_mx_m \leq K$, $b_1x_1 + b_2x_2 + \ldots + b_mx_m \geq L$
  - Objective: $c_1x_1 + c_2x_2 + \ldots + c_mx_m$

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbors, stop
- Can be exponential, but efficient in practice

- Feasible region is convex
- May be empty – constraints are unsatisfiable, no solutions
- May be unbounded – no upper/lower limit on objective

Network flows

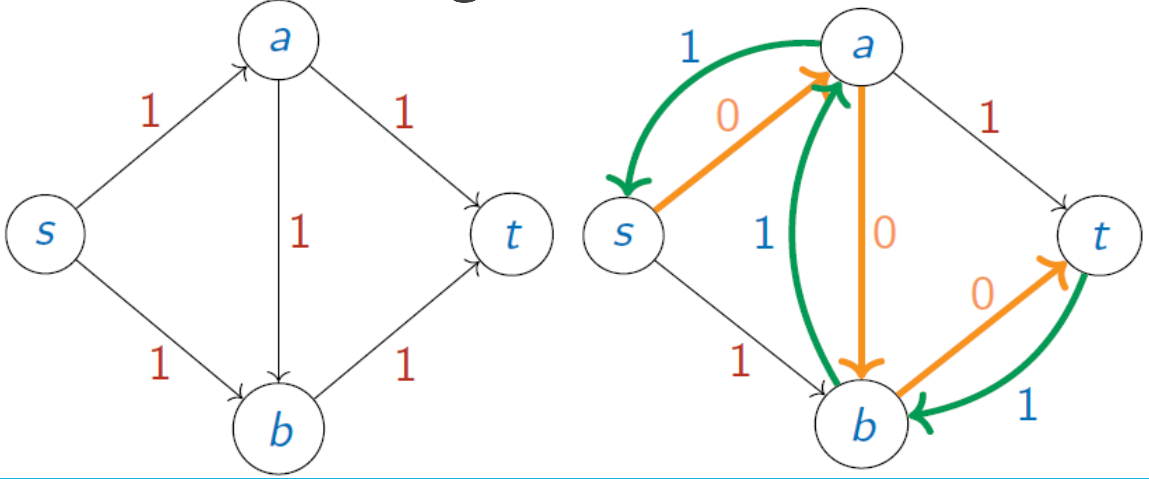
- Network: graph G = (V, E)
- Special nodes: s (source), t (sink)
- Each edge e has capacity c_e
- Flow: f_e for each edge e
  - $f_e \leq c_e$
  - At each node, except s and t, sum of incoming flows equal sum of outgoing flows
- Total volume of flow is sum of outgoing flow from s



Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Network given in the example has max flow 2
- What if one chooses a bad flow to begin with?
- Add reverse edges to undo flow from previous steps

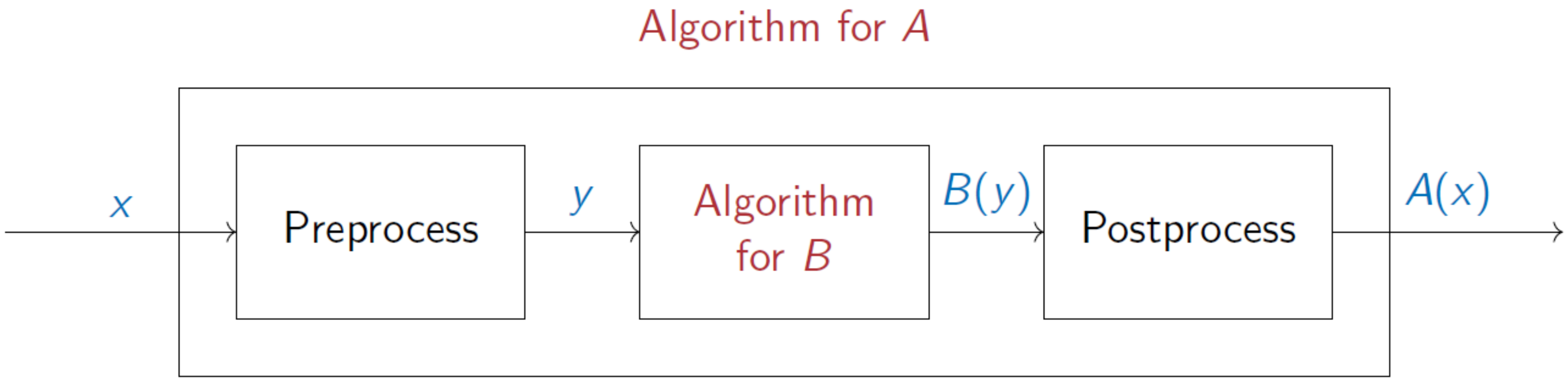
- Residual graph: for each edge e with capacity c_e and current flow f_e
  - Reduce capacity to $c_e - f_e$
  - Add reverse edge with capacity f_e
- Use BFS to find augmenting path with fewest edges



Reductions

- We want to solve problem A
- We know how to solve problem B
- Convert input for A into input for B
- Interpret output of B as output of A
- A reduces to B

- Can transfer efficient solution from B to A
- But preprocessing and postprocessing must also be efficient
- Typically, both should be polynomial time



Reductions

- Bipartite matching reduces to max flow
- Max flow reduces to LP
- Number of variables, constraints is linear in the size of the graph
- Reverse interpretation is also useful
- If A is known to be intractable and A reduces to B, then B must also be intractable
- Otherwise, efficient solution for B will yield efficient solution for A

Checking algorithms

- Factorize a large number that is the product of two primes
- Generate a solution
  - Given a large number N, find p and q such that $pq = N$
- Check a solution
  - Given a solution p and q, verify that $pq = N$
- Examples: satisfiability, travelling salesman, vertex cover, independent set, etc.

- Checking algorithm C for problem P
- Takes an input instance I for P and a solution “certificate” S for I
- C outputs yes if S represents a valid solution for I, no otherwise
- For factorization
  - I is N
  - S is $\{p, q\}$
  - C involves verifying that $pq = N$

P, NP and NP-Complete

- P (**P**olynomial) is the class of problems with regular polynomial time algorithms (worst-case complexity)
- NP (**N**on-deterministic **P**olynomial) is the class of problems with checking algorithms
- An algorithm A is NP-Complete if it satisfies two conditions:
  - A is in NP
  - Every algorithm in NP is polynomial time reducible to A