IFT 2125 Study Guide for Intra

Sean Grogan

# Asymptotic notation

Algorithms are classified according to their time complexity in the worst case, the average or best case

In the course, we became interested in complexity in the worst case

## Theoretical analysis

To calculate the time complexity of an algorithm, we count the number of elementary operations that must execute in the worst case

Asymptotic notation is used to express the complexity of an algorithm

# Big

Let there be a function. We define the order of as as the set of bounded functions above by:

Threshold rule: Given two strictly positive functions then

The Big O relation is reflexive and transitive

If and then

# Big

Let there be a function . We define as the set of functions bounded below by :

Duality rule:

# Big

We say that is in the exact order of , denoted as if

This is equivalent to

# Limit Laws

If then or

If then and

If then and

# Conditional Asymptotic notation

There exists and

In a similar way, we can define

In a similar way we can define

# Smoothness Rules

A function is eventually non decreasing or END, if such that

A function is b-smooth for if

1. It is END

A function is smooth if it is b smooth for all

## Smoothness Rule:

Let be a smooth function, if it exists and if it exists is an END Function, then

# Solving homogeneous linear recurrences with constant coefficients

For the recurrence

Use these steps to solve:

1. Find the characteristic polynomial of the recurrence
2. Find the roots of

If the roots are distinct

1. The general solution is in the form
2. Solve the system of linear equations with the initial conditions to find the values of
3. Write the solution according to the constants

For the recurrence

Use these steps to solve:

1. Find the characteristic polynomial of the recurrence
2. Find the roots of

If the roots are NOT distinct

1. The general solution is in the form where we have roots of of multiplicity
2. Solve the system of linear equations with the initial conditions to find the values of
3. Write the solution according to the constants

# Solving of non-homogeneous linear recurrences with constant coefficients (special case):

For the recurrence

Where is a constant

Follow these steps to solve

1. We begin to transform the recurrence to a homogenous recurrence  
   for this particular case, we multiply the recurrence by and replace with in the obtained equation  
   Subtracting the initial recurrence, this new recurrence, we obtain a homogeneous recurrence
2. Solve (Homogenous case) with the as usual, but by ensuring that the initial conditions also satisfy the original equation:

# Changing of variables

Sometimes it is easier to start with a change of variables when we want to solve a recurrence

For example, if . By the change of variables we obtain the linear, non-homogenous recurrence

This recurrence is solved in the usual way and then the solution obtained is expressed in in the function of utilizing the substitution and therefore

# Greedy Algorithms

Easy to develop

A local optimum is selected regardless of the effects in the future (not in the reverse)

We would like this local strategy leads to a global optimum

We must prove the optimality to show that greedy algorithm finds the optimal solution

Examples in class:

* Money Change
* Minimal Tree Cut (Kruskal – Prim)
* Shortest Path (Dijkstra)
* Backpack
* Simple Waiting Queue

## Money Change

Problem: It has an unlimited number of coins of different values. We want to make money in the amount so that returns the fewest pieces

Greedy Solution: We begin by giving the maximum coins of greatest value (local optimum) and then complete the amount n with parts smaller values

Proof of Optimality: The optimality here depends on the value of coins in our possession and the fact that we assume an unlimited number of each piece.

## Minimum Spanning Tree

Problem: We are given a connected, non-orientated graph a cost function, we want to find such that is a tree such that the sum

Is minimal

Greedy solution:

1. Start with an empty set of edges at each step and select the smallest edge cost that has not yet been chosen or rejected (regardless of where it is in the graph) (Kruskal)
2. Start vertex in a graph and construct a tree from the top of each step by selecting the edge which adds minimal cost to the existing new tree node (Prim)

### Kruskal

The algorithm maintains a forest of trees

At each iteration, we choose the edge with the minimal cost

This edge is accepted, if it reads two different sub-trees, otherwise it is rejected (could form a cycle)

The algorithm is completed when a single tree is left

### Prim

We choose a random vertex in a "cloud" and the minimum spanning tree constructed by enlarging the "cloud" one vertex at a time.

We keeps track each vertex , a label , which here is equal to the minimum weight among the weights of the edges connecting v to a vertex inside the cloud.

At each step

* We add to the cloud the outer vertex with the smallest weight
* We update the labels of vertices adjacent to

### Optimality

#### Property partition

* Consider a partition of the vertices of G into two sets U and V
* Let e ​​be an edge of minimum weight between U and V
* So, there exists a minimum spanning tree of G containing e

#### Proof:

* Let T be a minimum spanning tree of G
* If T does not contain e, C is the cycle formed by adding e to the tree T and let f be an edge between U and V
* Through the domain of cycles, we have that
* As we took e minimum weight, if there is where and then we obtain another MST replacing f by e

The optimality of both algorithms follows from the property of partition of MSTs

#### Kruskal

The graph of the partition is considered here, since an edge (u, v) of minimum cost, on the one hand all the vertices belonging to the connected component of the other of u and all other vertices.

If u and v are not part of the same connected component, the property partition guaranteed that the edge (u, v) is part of a MST

#### Prim

Here, the partition is considered cloud / non-cloud

## Shortest Path

Problem: Let be a connected, non-orientated (or orientated) graph and a cost function. Given a source vertex, we want to find the shortest paths from the source to all other vertices.

Greedy Solution: Dijkstra algorithm

### Dijkstra Algorithms

* The distance between a vertex v and a vertex s is the minimum total weight of a path between 'v' and 's'
* Dijkstra's algorithm is a greedy algorithm that calculated the distance between a vertex 'v' and all the other vertices of a graph
* Here we assume
  + The graph is connected
  + The weight of the edges are non-negatives
* We're going to grow a "cloud" of vertices, initially containing 'v' and eventually covering all vertices
* We will give a label and 'd ​​(u)' at each vertex, representing the distance between 'v' and 'u' in the subgraph is vertices in the cloud and adjacent edges
* At each step:
  + We add a vertex u to the cloud that is outside the could that has the smallest label ‘d(u)’
  + We update the labels of vertices adjacent to u

Problem: Let be a connected, non-orientated (or orientated) graph and a cost function. Given a source vertex, we want to find the shortest paths from the source to all other vertices.

Greedy Solution: Dijkstra algorithm

Proof of Optimality: comes from the following proposition

Proposal: With Dijkstra, whenever a vertex u is included in the cloud, D (u) is the minimum cost path between u and the source and vertex path is entirely within the cloud

## Backpack Problem

Problem: we have available objects of positive weights and have positive values . Our backpack has a capacity (maximum weight) of .

Our goal is to fill the bag so to maximize the value of the objects in the bag while respecting the constraint weight.

To be able to solve problems with a greedy algorithm, we suppose we can bring a fraction of each object for each object.

Goal:

Greedy solution:

Select each object in turn in a certain order, to the largest possible fraction of this object in the bag (without exceeding the weight capacity of the bag) and stop when the bag is full.

We select the object in the value per unit weight is maximum

Proof of optimality: comes from the following theorem

Theory: If objects are selected in descending order of then this strategy returns an optimal solution

## Simple waiting queue

We have a server with clients

Each client needs a service time

If the client must wait, its waiting time is denoted

Problem:

Theorem: If customers are served in ascending order of time of service requested, the greedy algorithm to find an optimal solution.

Proof of Optimality: Assume that an optimal solution exists where the order of customer service is different from the ascending time of service requested and derives a contradiction