Greedy Algorithms

Chapter 10

2016

What does “Greedy” mean ?

- Greedy algorithms are targeted for solving optimization problems.
- The solution is constructed in a sequence of steps or decisions. At each step we make the choice that looks best at the moment; in other words a Greedy algorithm makes a locally optimal choice in the hope for finding a globally optimal solution.
- Advantage: they are usually very fast because they avoid the exhaustive exploration of the solution space.
- Disadvantage: they do not always compute a globally optimal solution. In order to be sure that the computed solution is optimal, you have to prove mathematically that the locally optimal choices are appropriate for finding a globally optimal solution.
- In practice Greedy algorithms are used even if they do not guarantee the computation of a globally optimal solution, provided that this solution is “sufficiently good” or sub-optimal.

The activity selection problem

- The goal of the problem is to select a maximum-size set of mutually compatible activities. Two activities are compatible if they do not require both a resource at the same time.
- Input: a set of n activities $S = \{1, 2, \ldots, n\}$ that require or compete for the same resource. This means that only one activity at a time may use the resource. Each activity i has a start time s_i and a finish time f_i such that $s_i \leq f_i$. Activities i and j are compatible if the intervals $[s_i,f_i)$ and $[s_j,f_j)$ do not overlap, i.e. $s_i \geq f_j$ or $s_j \geq f_i$.
- Output: a maximal subset A of S consisting only of compatible activities.
- In order to give a Greedy algorithm for this problem we shall assume that activities are in order by increasing finishing times: $f_1 \leq f_2 \leq \ldots \leq f_n$

A Greedy algorithm for the activity selection problem

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1. n ← length[s]
2. A ← {1}
3. j ← 1
4. for i ← 2, n do
5.       if s_i ≥ f_j then
6.           A ← A ∪ {i}
7.           j ← i
8. return A
```

The algorithm takes *s* and *f*, the vectors of starting times and finishing times, as input and produces a maximal set of mutually compatible activities *A*.

The time complexity of this algorithm is $\Theta(n)$ where *n* is the total number of activities.

Greedy choice

- The algorithm picks up first activity 1. j specifies the most recent addition to the set A. Note that because the activities are considered in order of non-decreasing finishing times, f_j is always the maximum finishing time of any activity in A, i.e. $f_j = \max \{f_k \mid k \in A\}$.
- The algorithm selects at each step the activity with the earliest finishing time that can be legally scheduled. This is a *Greedy choice*.

Correctness

- Theorem: Greedy-ACTIVITY-SELECTOR produces solutions of maximum size to the activity selection problem.
- Proof:
  - First let us show that there is always an optimal solution that begins with the Greedy choice of activity 1. Let $A \subseteq S$ an optimal solution that starts with activity k and suppose that $k>1$. Than $B = (A \setminus \{ k \}) \cup \{ 1 \}$ is a solution with the same size as A because $f_1 \leq f_k$.
  - Next, if A is an optimal solution that begins with activity 1 then $A' = A \setminus \{1\}$ is an optimal solution for the set $S' = \{ i \in S \mid s_i \geq f_1 \}$. Let us prove this statement by refutation. Let us assume there is a solution B' of S' with more activities than A'. Then adding 1 to B' would yield a solution B for S with more activities than A. But this contradicts the optimality of A.
  - By induction on the number of choices made, making a Greedy choice at every step produces an optimal solution.

The activity selection problem – example

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

$f_1 = 4, A = \{1\}, j = 1$

$s_2 = 3 < f_1 = 4$

$s_3 = 0 < f_1 = 4$

$s_4 = 5 \geq f_1 = 4, A = \{1,4\}, j = 4$

$s_5 = 3 < f_4 = 7$

$s_6 = 5 < f_4 = 7$

$s_7 = 6 < f_4 = 7$

$s_8 = 8 \geq f_4 = 7, A = \{1,4,8\}, j = 8$

$s_9 = 8 < f_8 = 11$

$s_{10} = 2 < f_8 = 11$

$s_{11} = 12 \geq f_8 = 11, A = \{1,4,8,11\}, j = 11$

Characteristics of Greedy algorithms

- A Greedy algorithm has two ingredients:
  - The Greedy choice property
  - Optimal substructure
- The Greedy choice property says that a globally optimal solution can be arrived at by making locally optimal choices. Dynamic programming can be seen as making a sequence of choices, as well. However, there is an important difference. In dynamic programming the choice may depend on the solutions to sub-problems and sub-problems are solved bottom-up while in a Greedy algorithm choices cannot depend on any future choices, choices do not depend on the solutions to sub-problems and sub-problems are solved top-down.
- Optimal sub-structure is a common characteristic for Greedy algorithms and dynamic programming. From the correctness proof of the activity selection problem it is easy to see that this problem has the optimal sub-structure property.

The knapsack problem

- This problem comes in two flavors: discrete or 0-1 and continuous or fractional.
- *Discrete knapsack problem*: we have n items, item i is worth profit p_i and weighs w_i, $1 \leq i \leq n$. We want to pack an as valuable a load as possible in a knapsack that can hold a maximum weight of M. The name 0-1 means that an item can be either packed or not packed.
- *Fractional knapsack problem*: the same as the 0-1 problem except that we are allowed to split or pack fractions of any item.

The knapsack problem as an optimization problem

- The knapsack problem is an optimization problem with constraints. We have to maximize the profit:
  - $\Sigma_{1 \le i \le n} p_i \times x_i$
  - s.t. $\Sigma_{1 \le i \le n} w_i \times x_i \le M$.
  - For the discrete problem $x_i \in \{0,1\}$
  - For the fractional problem $x_i \in [0,1]$.
- Surprisingly, only the fractional problem can be solved using a Greedy algorithm. First we sort the items in non-increasing order of their profit per weight ratio, p_i/w_i, i.e. $p_1/w_1 \ge p_2/w_2 \ge ... \ge p_n/w_n$. Than we pack as much as possible of each item in the knapsack, considering items in this order. This algorithm takes $\Theta(n)$ time.
- Homework: Note that both problems have the optimal sub-structure property. For the case when M is a positive integer, devise an $\Theta(n \times M)$ dynamic programming algorithm for the 0-1 problem.

The knapsack problem – example

- Consider a knapsack of maximum weight $M = 50$ and three objects with profits $p = (100, 120, 60)$ and weights $w = (20, 30, 10)$. Profits per units are $p/w = (5, 4, 6)$.
- First we pack into the knapsack the most valuable object, i.e. 3. The profit gained is 60 and the remaining knapsack weight is $50 – 10 = 40$.
- Second we pack into the knapsack the next valuable object, i.e. 1. The total profit gained up to now is $60 + 100 = 160$ and the remaining knapsack weight is $50 – 10 – 20 = 20$.
- Third we pack into the knapsack a fraction of 2/3 of the last object (because it weights 30 and the remaining capacity is 20). The total profit gained is $60 + 100 + 120 \times (2/3) = 80 = 240$ and the remaining knapsack weight is 0.
- Note that if k objects were added to the knapsack then the first k-1 were added with a fraction 1 and last with a fraction less or equal than 1. In our example the fractions were 1, 1 and 2/3.

Fractional knapsack problem – proof

We assume that the sequences $(p_1, p_2, \ldots, p_n)$ and $(w_1, w_2, \ldots, w_n)$ are sorted s.t. $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$. Consider a feasible solution $(x_1, x_2, \ldots, x_n)$ of the CKP s.t. there exists $1 \leq i < j \leq n$ with $x_i \in [0, 1)$ and $x_j \in (0, 1]$. Let $\varepsilon = \frac{x_j w_j}{w_i} > 0$ and $\delta = \frac{(1-x_i)w_i}{w_j} > 0$.

We define the vector $(y_1, y_2, \ldots, y_n)$ s.t. $y_k = x_k$ for all $1 \leq k \leq n$ with $k \neq i$ and $k \neq j$ and y_i, y_j are defined according to the following rules:

a) if $\varepsilon \leq 1 - x_i$ then $y_i = x_i + \varepsilon$ and $y_j = 0$ (note that $y_i \leq 1$).

b) If $\varepsilon > 1 - x_i$ then $y_i = 1$ and $y_j = x_j - \delta$ (note that $y_j > 0$).

You can easily check that $\sum_{i=1}^{n} w_i x_i = \sum_{i=1}^{n} w_i y_i$. This shows that $(y_1, y_2, \ldots, y_n)$ is also a feasible solution.

Also it is easy check that $\sum_{i=1}^{n} p_i x_i \leq \sum_{i=1}^{n} p_i y_i$, that is $(y_1, y_2, \ldots, y_n)$ is a better feasible solution than $(x_1, x_2, \ldots, x_n)$.

The repeated application of this tranformation will produce a feasible solution $(x_1, x_2, \ldots, x_n)$ such that for all $1 \leq i < j \leq n$ we have either $x_i = 1$ or $x_j = 0$. But this is exactly the solution with at most an $x_k \in (0, 1)$, $x_i = 1$ for all $1 \leq i < k$ and $x_i = 0$ for all $k < i \leq n$.

Implementation of activity selection (I)

```
#include <stdio.h>
#define NMAX 100
void read_times(void);
void write_sol(void);
void activity_selector(void);

/* Number of activities */
int n;
/* Solution size */
int size_a;
/* Solution */
int a[NMAX];
/* Start times */
int s[NMAX];
/* Finish times, arranged in
   increasing order */
int f[NMAX];
```

```
void main() {
  scanf("%d",&n);
  while (n>0) {
    read_times();
    activity_selector();
    write_sol();
    scanf("%d",&n);
  }
}
```

Implementation of activity selection (II)

```
void activity_selector(void) {
  int j;
  int i;

  size_a = 0;
  a[++size_a] = 1;
  j = 1;
  for (i=2;i<=n;i++) {
    if (s[i] >= f[j]) {
      a[++size_a] = i;
      j = i;
    }
  }
}
```

```
void read_times(void) {
  int i;
  for (i=1;i<=n;i++)
    scanf("%d%d",&s[i],&f[i]);
}

void write_sol(void) {
  int i;
  for (i=1;i<=size_a;i++)
    printf("%3d",a[i]);
  printf("\n");
}
```

Implementation of fractional knapsack (I)

```
#include <stdio.h>
#include <stdlib.h>
#define NMAX 100
int compare(const void* p1,const void* p2);
void sort(void);
void read_input(void);
void write_sol(void);

typedef struct pw {
  float p;
  float w;
} PW;

/* Number of objects */
int n;
/* Profits and weights */
PW p_w[NMAX];
/* Knapsack weight */
float M;
/* Solution */
float x[NMAX];
/* Profit */
float p;

void main() {
  scanf("%d",&n);
  while (n>0) {
    read_input();
    pack();
    write_sol();
    scanf("%d",&n);
  }
}
```

Implementation of fractional knapsack (II)

```
int compare(const void* p1,const void* p2) {
  float x1 = ((PW*)p1)->p/((PW*)p1)->w;
  float x2 = ((PW*)p2)->p/((PW*)p2)->w;
  if (x1<x2)
    return 1;
  else if (x1 == x2)
    return 0;
  else
    return -1;
}

void sort(void) {
  qsort(&p_w[1],n,sizeof(PW),&compare);
}
```

Implementation of fractional knapsack (III)

```
void read_input(void) {
  int i;
  for (i=1;i<=n;i++)
    scanf("%f%f",&p_w[i].p,&p_w[i].w);
  scanf("%f",&M);
}
void write_sol(void) {
  int i;
  for (i=1;i<=n;i++) {
    printf("%7.3f",x[i]);
  }
  printf("\nProfit: %6.0f\n",p);
  for (i=1;i<=n;i++) {
    printf("%5.0f%5.0f%7.3f\n",
      p_w[i].p,p_w[i].w,p_w[i].p/p_w[i].w);
  }
}
```

Implementation of fractional knapsack (IV)

```
void pack(void) {
  int i;
  sort();
  p = 0;
  for (i=1;i<=n;i++) {
    if (M>0) {
      if (p_w[i].w<=M) {
        M -= p_w[i].w;
        p += p_w[i].p;
        x[i] = 1;
      } else {
        x[i] = M/p_w[i].w;
        p += p_w[i].p*x[i];
        M = 0.0;
      }
    }
    else
      x[i] = 0.0;
  }
}
```

Sample input:

```
3
100 20
120 30
60 10
50
0
```

Sample output:

```
  1.000  1.000  0.667
Profit:    240
   60   10  6.000
  100   20  5.000
  120   30  4.000
```

Homework I

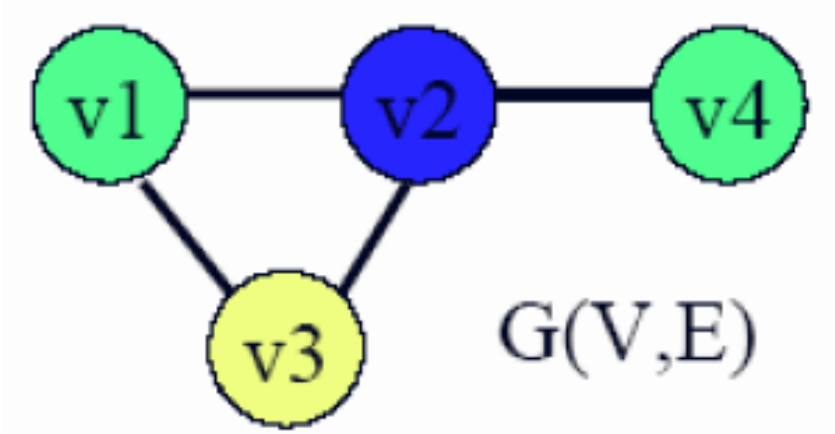
1. Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a Greedy algorithm, and prove that it yields an optimal solution. Design a Greedy algorithm based on this idea.
2. Suppose that we have a set of activities to schedule among a large number of lecture halls. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient Greedy algorithm to determine which activity should use which lecture hall.

Homework II

3. Not just any Greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities.
   a. Give an example to show that the approach of selecting the activity of least duration from those that are compatible with previously selected activities does not work.
   b. Do the same for the approaches of (i) always selecting the compatible activity that overlaps the fewest other remaining activities and (ii) always selecting the compatible remaining activity with the earliest start time.

Chromatic number

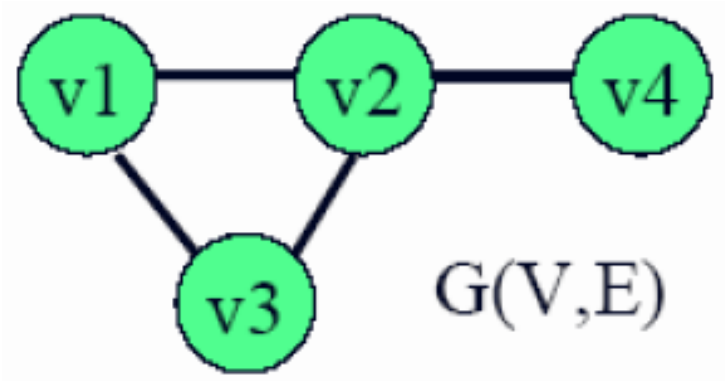
- Let $G = \langle V, E \rangle$ be an undirected graph. A *k-coloring* of G is a mapping $c : V \rightarrow C$, where C is a set of colors s.t. for all $(u,v) \in E$ we have $c(u) \neq c(v)$ and $|C| = k$. The smallest k for which G has a k-coloring is called the *chromatic number* of G, denoted as $\chi(G)$.



- This graph has chromatic number $\chi(G) = 3$.

Cliques and clique number

- A *clique* is a complete subgraph of a undirected graph $G = \langle V, E \rangle$.
- For example, the following graph has cliques with the following subsets of vertices:
  - {v1}, {v2}, {v3}, {v4}
  - {v1,v2}, {v1,v3}, {v2,v3}, {v2,v4}
  - {v1,v2,v3}



- The *clique number* $\omega(G)$ is defined as the size of the vertex set of its largest clique.
- For the example graph $\omega(G) = 3$.

Lower and upper bounds of chromatic number

- Proposition. Let $G = \langle V, E \rangle$ be an undirected graph and let $\Delta(G)$ be the value of the highest degree of its vertices. Then $\omega(G) \leq \chi(G) \leq \Delta(G)+1$.
- Proof:
- The lower bound follows from the fact that each vertex of a maximal clique must be colored with a different color.
- For the second part we consider the following Greedy algorithm for coloring the graph vertices:
  - We start with a fixed enumeration of the graph vertices: $v_1, v_2, \ldots, v_n$.
  - We consider the vertices in turn and we color each v_i with the first available color, i.e. the smallest color (positive integer) not already used to color any neighbor of v_i from the set $\{v_1, v_2, \ldots, v_{i-1}\}$.
- This algorithm clearly produces a correct coloring. It can be shown by induction on I = 1, 2, …, n, where $n = |V|$ that this coloring uses at most $\Delta(G)+1$ colors.
- Upper bound equality holds if G is complete or an odd cycle.

Interval graphs

- An *interval graph* is a graph whose vertices can be put in a one-to-one correspondence with a set of intervals such that two vertices are connected by an edge iff the corresponding intervals have non-empty intersection.
- Theorem. Any interval graph G can be optimally colored using a Greedy algorithm that considers the intervals in increasing order of their starting times – *EST strategy* (Earliest Starting Time). It is "equivalent" with Latest Finishing Time – *LFT strategy*.
- Proof sketch:
- Let d be the number of colors produced by this algorithm. Class d is used because an interval starting at s_j was incompatible with each of intervals starting before s_j. But, because we have the intervals sorted according to the starting times, it follows that we have d intervals such that any two intersect. So, we need d colors to do the coloring.
- Corollary. If G is an interval graph, then its chromatic number is equal to its clique number, i.e. $\chi(G) = \omega(G)$.

Homework III

4. Let us consider n jobs that must be scheduled by a single processor. Each job i requires t_i time to complete and it is due at time d_i. So, if i starts at time s_i then it will finish at time $f_i = s_i + t_i$. The lateness of job is defined as $l_i = \max\{0, f_i - d_i\}$. Determine a schedule of the jobs such that the maximum lateness of all jobs $L = \max_i l_i$ is minimized.
5. Let us consider a set V of n vertices and a natural number $1 \leq m \leq n(n-1)/2$. There are many possibilities of defining an undirected graph $G = \langle V, E \rangle$ s.t. $|E| = m$. Let d_i be the degree of vertex $i \in V$. The energy of graph G is defined as $E(G) = \sum_{1 \leq i \leq n} d_i^2$. Determine the maximum value of $E(G)$.