INSTITUTO
DE INFORMÁTICA
UFRGS

GREEDY ALGORITHMS, PART 1: INTRODUCTION AND EXAMPLES

Marcus Ritt

CMP 601 – Algorithms and Theory of Computation —

<2020-04-14 ter>

Outline

1. Introduction

2. Examples

- There's no universal definition of a greedy algorithm
- Construct solution via series of local (myopic) decisions.
- Usually: build solution elementwise.
- Also possible: elementwise removal until solution remains.

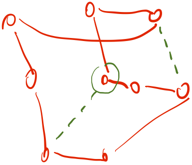
NP-c.

Greedy construction of a TSP tour: Greedy, Nearest neighbor

Greedy chiseling of a TSP tour

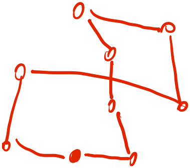
HC ∈ NP-c.

GR

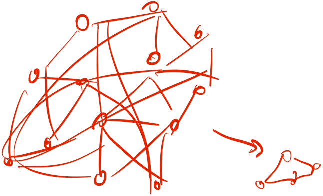


GP

NN



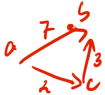
GP



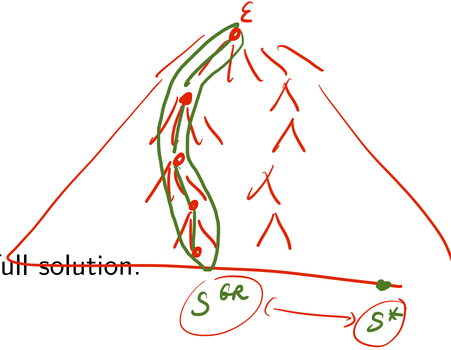
Average Percent Excess over the Held-Karp Lower Bound									
N =	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
	Random Euclidean Instances								
CHR	9.5	9.9	9.7	9.8	9.9	9.8	9.9	–	–
CW	9.2	10.7	11.3	11.8	11.9	12.0	12.1	12.1	12.2
GR	19.5	18.8	17.0	16.8	16.6	14.7	14.9	14.5	14.2
NN	25.6	26.2	26.0	25.5	24.3	24.0	23.6	23.4	23.3
	Random Distance Matrices								
GR	100	160	170	200	250	280	–	–	–
NN	130	180	240	300	360	410	–	–	–
CW	270	520	980	1800	3200	5620	–	–	–

≈ 16%
≈ 24%
≈ 100%
≈ 300%

(Johnson, McGeoch, 1995)



- Related view: search tree
- Take local, myopic best choice
- This selects a single path to a full solution.



elements
feasible subsets

$S \in \mathcal{V}$
solution

- Universe U, familiy $\mathcal{V}$ of subsets of U
- $\mathcal{V}$ closed under inclusion
- *Corresponding* optimization problem: given $w_u \geq 0$, $u \in U$, find

$$\underset{S \in \mathcal{V}}{\operatorname{argmax}}\, w(S)$$

$$w(S) = \sum_{e \in S} w_e$$

$\mathcal{U}$

- Corresponding (natural) greedy algorithm

it. = $|\mathcal{U}|$

$S := \emptyset$
while $U \neq \emptyset$
  select $u \in U$ *with* w_u *maximal* ← greedy step
  $U := U \setminus \{u\}$ remove
  if $S \cup \{u\} \in \mathcal{V}$ ← test feasibility
    $S := S \cup \{u\}$
  end if
end while
return S

Compl. $|\mathcal{U}| \cdot O($feas. test$)$

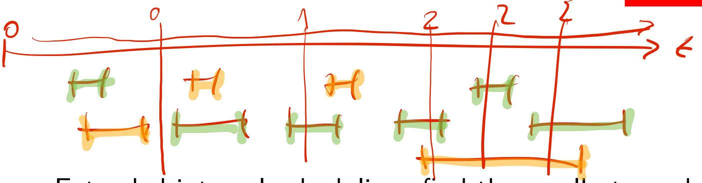
$(\mathcal{U}, \mathcal{V}) \Longleftrightarrow$ Greedy algorithm

One processor

start finish

- Intervals $[s_1, f_1], \ldots, [s_n, f_n]$
- Find largest non-overlapping set
- Solution: order intervals by non-increasing f_i, repeatedly take first, remove overlapping
- Main argument: *The greedy algorithm stays ahead.*

Maximum independent set on an interval graph



~~A processor~~ least # of proc.

Part 1

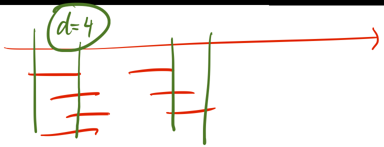
Part 2

- Extended interval scheduling: find the smallest number of processors to do all intervals
- Equivalent: find the smallest coloring of the intervals (each processor is a color)
- (Coloring = partition into independent sets.)

Minimum coloring of an interval graph

GMP-C.

Interval coloring 2

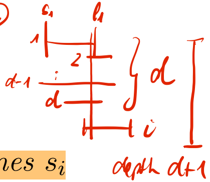


- Define *depth* as the maximum number of simultaneously overlapping intervals
- Observation: depth d is a *lower bound*: we need at least d processors.
- Now: if an algorithm that uses *at most* d it's optimal!

- Work with "colors" $1, 2, \dots, d$

$s_1\ f_1\ s_2\ s_3\ f_2\ f_3 \dots$ (2n)

O(n log n)



GIC()

O(n log n) *Sort intervals by non-decreasing start times* s_i

for $i = 1, 2, \dots, n$

O(1) { *assign the first free color to interval* i (*) }

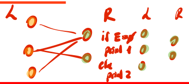
end

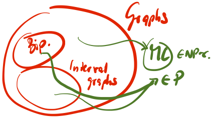
O(n)

maintain a list of free colors

- Here: color is free for i, of no overlapping predecessor uses it
- Claim: (*) never fails: otherwise we'd have d overlapping intervals with i, and thus depth $> d$.
- By construction: overlapping intervals have different colors.
- Consequence: GIC is optimal.

Ex. Hardness of HC on bipartite graphs? ∈P



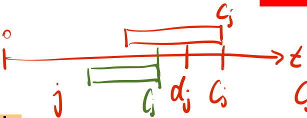


$1 \,||\, \sum w_j C_j$

- n jobs with some weights w_j and time p_j, $j \in [n]$.
- Find a schedule such that $\sum_j w_j C_j$ is minimal.
- Observation: optimal schedule is a *left-shift schedule*.
- *Exchange argument* shows: no pair i, $i+1$ has $w_i/p_i < w_{i+1}/p_{i+1}$
- Thus: optimal schedule is *free of inversions* of that kind.
- And: all inversion-free left-shift schedules have the same value.
- Thus: sorting by non-increasing w_i/p_i solves the problem.

$1 \,||\, L_{\max}$

1- processor (∅ restrictions) objective function



- Schedule n jobs on a single processor
- Each job: processing time p_j and due date d_j
- If job j completes at time C_j its lateness is $L_j = \max\{C_j - d_j, 0\}$.
- Find a schedule that minimizes the maximum lateness

$C_j - d_j > 0$

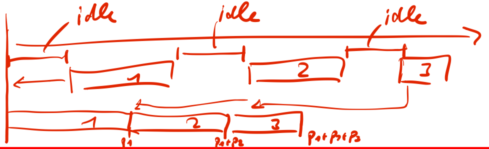
$L_j = C_j - d_j$

$C_j - d_j < 0$

$L_j = 0$

$$\text{min. } L_{\max} = \max_{j \in [n]} L_j$$

- Observation: optimal schedule is left-shift (no idle time).



$\pi = (1\ 2\ 3)$

$\pi' = (2\ 3\ 1)$

1 2 3 ; 2 1 3 EDD

- Claim: We can always order the jobs by earliest due date (EDD) $d_1 \leq d_2 \cdots \leq d_n$

$d_1 = d_2 < d_3 = d_4 = d_5 < d_6 < d_7 < d_8$

- Proof: an *exchange argument*: for contradiction, assume two consecutive jobs i and j, with $d_i \geq d_j$
- We have $L_{\max} = \max\{L, L_i, L_j\}$ for remaining latenesses L

C_j

- After swapping:

$$L'_i = \max\{0, C'_i - d_i\} \tag{1}$$
$$= \max\{0, C_j - d_i\} \leq \max\{0, C_j - d_j\} = L_j \tag{2}$$
$$L'_j \leq L_j \tag{3}$$

- Thus: $L'_{\max} = \max\{L, L'_i, L'_j\} \leq L_{\max}$.

$\exists$ opt. sched s.t. : $d_1 \le d_2 \le \dots \le d_n$ EDD

- So: an optimal schedule is left-shift, and has no inversion $d_i > d_j$.
- All ~~such~~ EDD schedules have the same maximum lateness (Proof: another simple exchange argument)
- Thus: a greedy EDD (earliest due date) order is optimal.

$O(n \log n)$