Greedy Algorithms

Greed, for lack of a better word, is good. Greed is right. Greed works. — Gordon Gekko

If only things were so simple.

As we will see, sometimes greed works and sometimes it doesn't.

Making change

(All variables are of type $\mathbb{N}$ or set of $\mathbb{N}$.)

Input: An amount of money t in cents. A set of coins S, each of which is a **quarter**, a **nickel**, or a **penny**.

$$\forall x \in S \cdot value(x) \in \{1, 5, 25\}$$

Solutions: Subsets C of S such that

$$\sum_{x \in C} value(x) = t$$

Optimality: We want to minimize the total number of coins output, $|C|$.

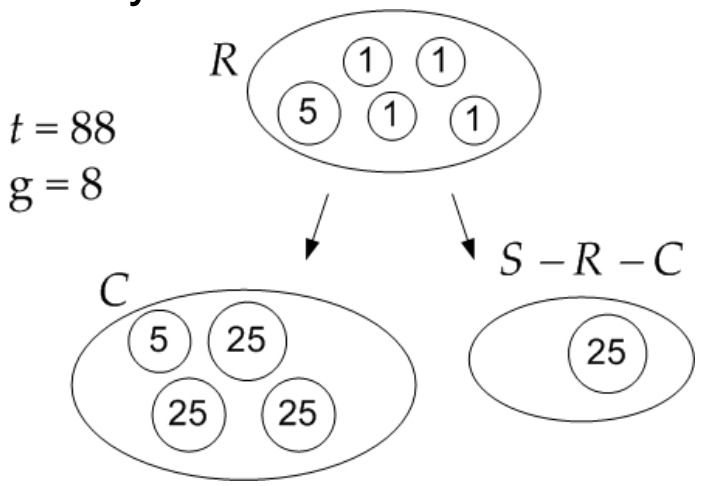
Output: An optimal solution, if there is one. Otherwise a message.

An algorithm

var $g := t$ // Remaining amount of money
var $C := \emptyset$ // The **C**ommitted set. Coins chosen
var $R := S$ // The **R**emaining set. Coins not yet chosen nor rejected
inv: $C \cup R \subseteq S \ \wedge \ C \cap R = \emptyset \ \wedge \ g = t - \sum_{x \in C} value(x)$
inv: *If a solution exists, an optimal solution is reachable*
// I.e. if a solution exists, there is an optimal solution that
// * includes all coins in C and
// * only includes coins in $C \cup R$
while R contains a coin do
  let x be the most valuable coin in R
  if $g \geq value(x)$
  then
    $C := C \cup \{x\}$ $g := g - value(x)$
  end if
  $R := R - \{x\}$
end while
{ if a solution exists, C is an optimal solution }
if $g = 0$ then output C
else output “no solution” end if

At each point, each coin is in one of three piles

- Committed. C Definitely part of the solution
- Rejected. $S - R - C$. Not part of the solution
- Remaining R. Not yet considered.



Each iteration considers a coin x and either

- moves from R to C, or
- moves it from R to $S - R - C$

Why does it work?

- As we will see, no choice made in the loop body precludes an optimal solution.
- I.e., the statement

  An optimal solution is reachable

  is preserved by the loop body.
- However we can't use that as an invariant, since we don't know *a priori* that a solution exists.
- A loop invariant

  If a solution exists, an optimal solution is reachable

Narrowing the search space.

So what does it mean that *"an optimal solution is reachable"*?

It means there is an optimal solution that

- includes all coins from C
- excludes all coins from $S - R - C$

In other words there is an optimal solution in the search space SS

$$SS = \{X \mid C \subseteq X \subseteq R \cup C\}$$

Initially $SS = \{X \mid X \subseteq S\} = 2^S$.

At termination, $R = \emptyset$ and $SS = \{C\}$.

Each decision made narrows the search space

- If an item x is kept, C gets bigger while $R \cup C$ remains the same. We remove from SS all sets that *don't* contain x

$$SS := \{X \mid C \cup \{x\} \subseteq X \subseteq R \cup C\}$$

- If an item x is rejected, $R \cup C$ gets smaller while C remains the same. We remove from SS all sets that contain x.

$$SS := \{X \mid C \subseteq X \subseteq (R \cup C) - \{x\}\}$$

(This is a kind of binary search, isn't it?)

Proving the algorithm

Establishing the invariants

All straight forward.

Proving the invariants $C \cup R \subseteq S$ and $C \cap R = \emptyset$ and $g = t - \sum_{x \in C} value(x) \geq 0$ are preserved

These are straight forward

Proving the invariant "If a solution exists, an optimal solution is reachable" is preserved.

If a solution does not exists, the invariant is preserved trivially.

Suppose a solution does exist.

- We must show that
  - If an optimal solution is reachable at the top of the loop body,
  - then an optimal solution is reachable at the bottom of the loop body.

Here we go.

- Let O be an optimal solution reachable at the top of the loop
  - $C \subseteq O$ and $O \subseteq R \cup C$
  - $g = \sum_{x \in O - C} value(x)$
- Suppose the algorithm is about to commit to a quarter x to C, then $g \geq 25$ and so $O - C$ must contain coins that add up to at least 25.
  - Case $O - C$ contains a quarter y.
    - (Note: y may, but might not, equal x).
    - Let $O' = O - \{y\} \cup \{x\}$.
      - (Note: if $x = y$, then $O' = O$.)
    - $\sum_{z \in O'} value(z) = \sum_{z \in O} value(z) = t$.
    - $|O'| = |O|$. So O' is an optimal solution.
    - $C \cup \{x\} \subseteq O'$ and
    - $O' \subseteq R \cup C = (R - \{x\}) \cup (C \cup \{x\})$.
    - So O' will be reachable at the bottom of the loop.
    - So after, committing to x, an optimal solution (O') can be reached.
  - Case $O - C$ does not contain a quarter.
    - Then $O - C$ must contain some subset of nickels and pennies that add up to 25 exactly — since any set of nickels and pennies that adds up to ≥ 25 must have a subset that adds up to 25 exactly. Furthermore this subset contains at least 5 coins. Now replace such a subset with

the quarter x to obtain a solution O''. Since O'' has fewer coins than O, O is not optimal. Contradiction. This case can't happen.

- Suppose the algorithm is about to reject a quarter x. Then $g < 25$ and so $O - C$ has a value of < 25 and must not contain a quarter. Thus $x \notin (O - C)$, and so $O \subseteq C \cup R - \{x\}$. So rejecting the quarter leaves O reachable.
- Similar arguments for nickels and pennies.

(This is what Edmonds calls a "fairy godmother argument"'. O is the fairy godmother's solution. Cormen, Leieserson, Rivest, and Stein call it a "cut and paste argument": we cut y out of the fairy godmother's solution and paste in x to obtain an optimal solution that the algorithm can still reach.)

Proving the invariants imply the postcondition

When no coins remain, the only reachable solution is C.

$$R = \emptyset \Rightarrow \{X \mid C \subseteq X \subseteq R \cup C\} = \{C\}$$

So "*if a solution exists, an optimal solution is reachable*" simplifies to

if a solution exists, C is an optimal solution. (1)

or equivalently

if C is not an optimal solution, no solution exists (2)

After the loop

Consider if $g = 0$

$g = 0$
$\Rightarrow$ By $g = t - \sum_{x \in C} value(x)$
C is a solution
$\Rightarrow$
a solution exists
$\Rightarrow$ By (1)
C is an optimal solution

Consider if $g \neq 0$

$g \neq 0$
$\Rightarrow$ By $g = t - \sum_{x \in C} value(x)$
C is not a solution
$\Rightarrow$ By (2)
no solution exists

Why is it called "greedy"?

What makes this algorithm "greedy"?

- The algorithm builds a partial solution. (In the example the C set.)
- At each point we make a 'locally optimal choice'. (Accept or reject a largest coin.)
- Earlier choices can not be reversed. (Once we have committed to 3 quarters, the solution found will contain *at least* 3 quarters.)

A Schematic Problem

Problems solved by greedy algorithms often involve making selections from a set to obtain an optimal subset that meets some criteria.

- The change making example fits this stereotype.

Schematic Problem (Optimal subset)

input: a set of *things* S and maybe some *other information*
solution: a subset of S that meets some *validity criterion* v
optimality: The solution should be least according to some *comparison operator* $\lesssim$
output: if there is a solution, an optimal solution; otherwise a message

[Aside: Formally

- There is a solution if the set $T = \{X \subseteq S \mid v(X)\}$ is nonempty.
- A solution X is optimal if $X \in T$ and $\forall Y \in T \cdot X \lesssim Y$

end of aside]

Usually the optimality operator can be expressed by an "*objective function*" f so that

$$X \lesssim Y \qquad \Leftrightarrow \qquad f(X) \leq f(Y)$$

Filling in the slots for the change making problem:

- *Things* is the set of all quarters, nickels, and pennies
- *Other information* is the goal amount t.
- The *validity criterion* is that the value of the subset equals t
- The *objective function* is the number of coins.

Filling in the slots for the single-source, shortest paths problem

- *Things* are edges in a graph
- *Other information* are the edge weights and a start node s
- The *validity criterion* is that a set of edges be a tree rooted at s that reaches every node reachable from s.
- the *comparison operator* is that tree X is better than ($\lesssim$) a tree Y if for each node u, reachable from s, the distance from s to u in X is less or equal to the distance from s to u in Y.

A schematic algorithm

Our search space will be all subsets of S that include all committed items and no rejected items..

$$\{X \mid C \subseteq X \subseteq R \cup C\}$$

Narrow the search space by taking away from R and possibly adding to C.

Schematic Algorithm

The greedy optimal subset selection algorithm scheme

var $C := \emptyset$ // Committed set
var $R := S$ // Remaining set.
inv: $C \cup R \subseteq S \ \wedge \ C \cap R = \emptyset$
inv: *If a solution exists, an optimal solution is in*

$$\{X \mid C \subseteq X \subseteq R \cup C\}$$

while there is a *suitable item* in R do
  pick a *best suitable* item x in R
  if
    committing to x does not preclude an optimal solution
  then
    $C := C \cup \{x\}$
  end if
  $R := R - \{x\}$
end while
{ a solution exists $\Rightarrow$ C is an optimal solution }
if C is a solution then output C
else output 'no solution' end if

Filling in the additional slots for the change making example

- all coins are *suitable*.
- a *best* item is one that is worth the most money

No backsies. At each stage of the algorithm, we either commit to an item x or reject it.

- Committed items C must be part of the solution.
- Rejected items $S - R - C$ will not be part of the solution

Adaptive vs fixed priority. In picking the best, we can look all available information including the commitments so far.

- In the change making example this is not required.
- For making change, the best item can be determined by consulting only R.

If only R needs to be consulted, we say that the priority is fixed and we can make the algorithm more efficient by first sorting S according to bestness.

- In the change making algorithm, the priority is fixed, so we can sort the coins at the beginning.

No lookahead. You can always make the greedy solution work, if you are willing to spend enough time picking the best item. We generally want to have some way of finding a best item that does not require very much work.

Dijkstra's algorithm as a greedy algorithm

The single-source shortest paths problem also fits this pattern

- *Things* are edges of a graph (V, E)
- *Other information* are the edge weights and a start node s
- The *validity criterion* is that a set of edges be a tree rooted at s that reaches every node reachable from s.
- The *comparison operator* is that tree X is better than ($\lesssim$) a tree Y if for each node u, reachable from s, the distance from s to u in X is less or equal to the the distance from s to u in Y.

Here is Dijkstra's algorithm as an adaptive greedy algorithm.

- Note that a solution always exists.
- *Suitable* edges are edges in R that start at nodes that are reachable from s following only committed edges.
  - Thus the algorithm will grow a tree of committed edges rooted at s
  - So each suitable edge x represents a path that
  - starts at s, follows 0 or more committed edges, and finally follows x
- Among suitable edges, the *best* is the one that represents the shortest path
- To improve efficiency, I added two tracking variables
  - H tracks vertices reachable from s following only

committed edges. That is it is the set of all v such that v is reachable from s via a path consisting only of edges in C.

* SE tracks suitable edges (Edges in R starting from H)

The *cost* of an edge x is the distance of the path starting at s, consisting of edges in C and ending with x.

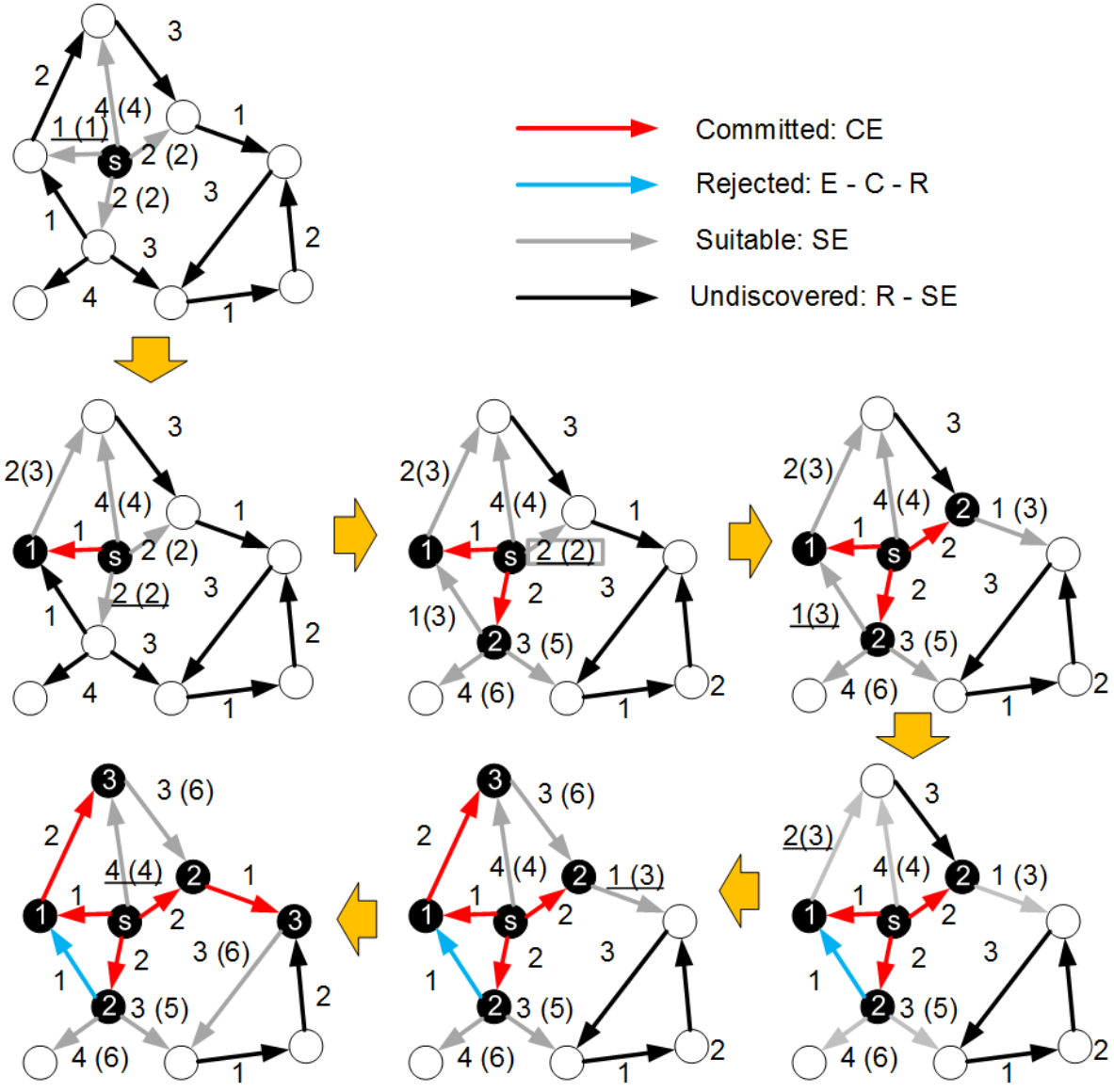
Dijkstra's shortest paths algorithm

var $C := \emptyset$ // Committed edges
var $H := \{s\}$ // Reachable from s by edges in C
// SE is the set of suitable edges
var $SE :=$ the set of edges leaving s
ghost var $R := E$ // Remaining edges.
inv: $C \cup R \subseteq E \ \wedge \ C \cap R = \emptyset$
inv: $H = \left\{ v \mid \exists p \text{ in } (V, C) \cdot s \xrightarrow{p} v \right\} \ \wedge \ SE = \{e \in R \mid \overleftarrow{e} \in H\}$
inv: there is an optimal tree in $\{X \mid C \subseteq X \subseteq R \cup C\}$
while $SE \neq \emptyset$ do
  let x be a least "cost" edge in SE
  if x ends at a vertex not in H then
    $C := C \cup \{x\}$ // Add to the tree
    $H := H \cup \{\overrightarrow{x}\}$// Add to the tree
    // Edges that leave the target of x are now suitable.
    $SE := SE \cup \{e \mid \overleftarrow{e} = \overrightarrow{x}\}$
  end if
  $R := R - \{x\}$; $SE := SE - \{x\}$
end while
output (H, C)

Edges are in one of several states.

$$(R - SE) \longrightarrow SE \begin{array}{l} \nearrow C \\ \searrow E - C - R \end{array}$$

- Undiscovered edges: $R - SE$
- Suitable edges: $SE \subseteq R$. Source in H, but not yet committed or rejected.
- Committed edges: C.
- Rejected edges: $E - C - R$



To speed the implementation, we can implement SE with a priority queue.

(This is still a bit different from the classic Dijkstra's algorithm which uses a priority queue of vertices. Here we use a priority queue of edges.)

Correctness: Assume that (H, C) is a tree that is optimal for trees build from edges considered so far $(E - R)$.

There are three possibilities

- No edge in R starts from a node in H.
  - Then (H, C) can not be extended.
  - All reachable nodes are reached.
  - The postcondition of the loop has been established, so an exit is appropriate.
- The best remaining edge x goes to a node w in H
  - The cost of reaching w by this edge can't be better than the path already in (H, C).
  - Discarding x from R preserves that the tree is optimal for trees build from edges considered so far.
- The best edge x goes to a node w not in H.
  - This is the first edge considered that reaches w so
  - any other path from s to w is represented by an edge in SE that has a priority at least as large as x's priority and thus that path is at least as long as the path represented by x.
  - Moving x from R to C restores that the tree is optimal for trees build from edges considered so far.

Efficiency of Dijkstra's shortest path algorithm:

- Implement SE as a priority queue with the priority of each edge being its cost.
- The size of SE can be as big as $|E|$ (and no bigger).
- So adding edges to and removing edges from SE is $\Theta(\log |E|)$.
- Each edge is added and removed from the queue at most once.
- In the worst case, each edge is added to the queue.
- So the whole algorithm is $\Theta(|E| \times \log |E|)$.
- Compare to $\Theta(|E| \times \log |V|)$ for the version in slide set 13.

An optimization:

- Suppose we add an edge to SE that ends at a node y and there is already an edge in SE that ends at y.
- If they have different costs, the edge with higher cost can not be part of an optimal tree.
- If they have the same costs, one is as good as the other, and we only need to keep one.
- So either way, we can immediately remove the/a one that is not cheaper than the other.
- If we do this, the size of $|SE|$ is bounded by $|V|$.
- The algorithm's complexity is $\Theta(|E| \times \log |V|)$, same as the version in slide set 13.

Scheduling a meeting room

Set of things.

- S is a set of meetings, each with a start time and an end time

Validity criterion

- There must be no overlap of meeting times

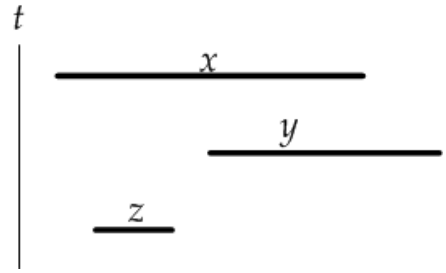
Comparison operator

- We want to schedule as many meetings as possible

Possible meanings for “best”

- Shortest meetings are best.
  - Intuitively appealing, but one short meeting might conflict with two nonconflicting long meetings
- ‘Earliest start time’ (First come—first served)
  - An early long meeting could block many short meetings.
  - Latest finish time is the same
- ‘Earliest finish time’
  - Works.
  - Earliest finish time gives the most time after for more meetings
  - Suppose that we have an optimal schedule up to time t and that
  - x is the meeting that starts at or after t which ends earliest.
  - How could an alternative y be better?

* y ends at the same time or later.
* Maybe we could squeeze in another meeting z that ends before y begins?



* No. Because then z ends before x and that contradicts that x ends at the earliest time.

The algorithm

All meetings are *suitable*.

The empty set is a solution, so we know there is a solution

The best meeting is one that ends the earliest

Greedy meeting scheduling
Input: A set of meetings S
var $C := \emptyset$ // Committed set
var $R := S$
var $t := -\infty$ //
inv: $C \cup R \subseteq S \ \wedge \ C \cap R = \emptyset$
inv: $t = \max_{w \in C} w.\mathrm{endTime}$
inv: every meeting in R ends after t
inv: There is an optimal solution in
$$\{X \mid C \subseteq X \subseteq R \cup C\}$$
while there is a meeting in R do
  pick a meeting x in R that ends the earliest
  if $x.\mathrm{startTime} \geq t$ then
    $C := C \cup \{x\}$
    $t := x.\mathrm{endTime}$
  end if
  $R := R - \{x\}$
end while
output C

A "cut and paste" proof that the 4th invariant is maintained

Suppose O is an optimal solution reachable from the current state at the top of the loop body.

I.e. $C \subseteq O \subseteq R \cup C$

Let x be the meeting the algorithm picks.

- Case: The algorithm will add x to C.
  - (We need to show that, even if O is no longer reachable after adding x to C, another optimal solution O' is.)
  - Then $x.\mathrm{startTime} \geq t = \max_{w \in C} w.\mathrm{endTime}$
  - Since $C \cup \{x\}$ is a solution, there is a solution of size $|C| + 1$.
  - Since O is optimal, $|O| \geq |C| + 1$.
  - So, as $O \subseteq R \cup C$ and $|O| \geq |C| + 1$, we know that O contains at least one item in $R - C$.
  - Of those items, let y be the one that finishes earliest.
  - (Comment: $x = y$ is possible.)
  - Let $O' = (O - \{y\}) \cup \{x\}$ (Cut and paste.)
  - From this definition and $C \subseteq O \subseteq R \cup C$, we have $C \cup \{x\} \subseteq O' \subseteq (R - \{x\}) \cup (C \cup \{x\})$ and so O' is reachable once x moves from R to C.
  - It remains to show O' is an optimal solution.
  - (If $x = y$, then $O' = O$ and so O' is an optimal solution by assumption. However, there is no need to consider the cases $x = y$ and $x \neq y$ separately; so I won't.)

  - Since it is as big as O, we only need to show O' is a solution.
  - Let z be any item in $(O - \{y\})$.
  - Since O is a solution, z can not interfere with any other meeting in O.
  - It remains to show that z can not interfere with x either.
  - Case: z in C.
    - x interferes with nothing in C (since $x.\mathrm{startTime} \geq \max_{w \in C} w.\mathrm{endTime}$).
  - Case: z not in C and not in R.
    - can't happen as z is in O and $O \subseteq R \cup C$
  - Case: z in $R - C$.
    - Then $x.\mathrm{endTime} \leq y.\mathrm{endTime}$, by the way x was chosen
    - And $y.\mathrm{endTime} \leq z.\mathrm{endTime}$, by the way y was chosen
    - But as z does not interfere with y, it must start after y finishes: $y.\mathrm{endTime} \leq z.\mathrm{startTime}$.
    - We have $x.\mathrm{endTime} \leq y.\mathrm{endTime} \leq z.\mathrm{startTime}$. No interference.

- Case: The algorithm rejects x.
  - Then $x.\mathrm{startTime} < t$
  - Since $x \in R$, it ends after t. Thus x spans t. (Which is thus finite.)
  - Since t represents the end time of some meeting in C, x interferes with a meeting in C and hence in O.

* Therefore x is not in O. Thus $C \subseteq O \subseteq R - \{x\} \cup C$. After removing x from R, O is still reachable.