Greedy Algorithms

Thomas Schwarz, SJ

The Change Making Problem

- A given country uses a weird set of coins
  - 1, 3, 5, 8
- How do you make change with the least number of coins?
  - With these coins, it is not so obvious
  - Normally, we can just start out with the largest coin that fits, but not in this case
  - Making change for 15:
    - Use an 8, a 5 and two 1s
    - But three 5s is better

The Change Making Problem

- To solve the change making problem, we can use dynamic programming
- Some notation: v_i value of coin i, $i \in \{1,...,n\}$
  - Best number of coins for change of x is
    - Best number of coins for change of $x - v_1$ plus one
    - Best number of coins for change of $x - v_2$ plus one
    - …
    - Best number of coins for change of $x - v_n$ plus one

The Change Making Problem

- To organize the calculation
  - Create a tableau
    - For row i, column j:
      - How many coins to make change for an amount of i with coins $1, \ldots, j$

The Change Making Problem

- Example: Coins with values 1, 3, 5, 8 to make change of 15

Value	ones	threes	five	eights
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				

The Change Making Problem

- Example: Coins with values 1, 3, 5, 8 to make change of 15
- First column is easy

Value	ones	threes	five	eights
1	1			
2	2			
3	3			
4	4			
5	5			
6	6			
7	7			
8	8			
9	9			
10	10			
11	11			
12	12			
13	13			
14	14			
15	15			

The Change Making Problem

- Second column asks how many threes I should use
  - Example for value 10:
    - Can use none
      - Cost is 10
    - Can use one three
      - Cost is 1+7
    - Can use two threes
      - Cost is 2+4
    - Can use three threes
      - Cost is 3+1

Value	ones	threes	five	eights
0	0	0	0	0
1	1			
2	2			
3	3			
4	4			
5	5			
6	6			
7	7			
8	8			
9	9			
10	10	???		
11	11			
12	12			
13	13			
14	14			
15	15			

The Change Making Problem

- Second column asks how many threes I should use
  - Formula is

$$\min\{T_{i-v_j\nu,j-1}+\nu \mid \nu = 0,1,\ldots,\lfloor\frac{i}{v_i}\rfloor\}$$

$T_{i-v_j\nu,j-1}$ costs of making change of $i-\nu v_j$ with coins up to $j-1$

$+\nu$ costs of using ν coins of value v_j

Value	ones	threes	five	eights
0	0	0	0	0
1	1			
2	2			
3	3			
4	4			
5	5			
6	6			
7	7			
8	8			
9	9			
10	10	???		
11	11			
12	12			
13	13			
14	14			
15	15			

The Change Making Problem

- Our alternatives are:
  - No threes: 10
  - One three: 7+1=8
  - Two threes 4+2=6
  - Three threes 1+3=4

Value	ones	threes	five	eights
0	0	0	0	0
1	1			
2	2			
3	3			
4	4			
5	5			
6	6			
7	7			
8	8			
9	9			
10	10	4		
11	11			
12	12			
13	13			
14	14			
15	15			

The Change Making Problem

- Filling in the other values

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1		
2	2	2		
3	3	1		
4	4	2		
5	5	3		
6	6	2		
7	7	3		
8	8	4		
9	9	3		
10	10	4		
11	11	5		
12	12	4		
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - The first values are simple since we cannot use a five

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1		
2	2	2		
3	3	1		
4	4	2		
5	5	3		
6	6	2		
7	7	3		
8	8	4		
9	9	3		
10	10	4		
11	11	5		
12	12	4		
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - The first values are simple since we cannot use a five

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3		
6	6	2		
7	7	3		
8	8	4		
9	9	3		
10	10	4		
11	11	5		
12	12	4		
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - At value 5:
    - Can use a five
    - Can not use a five:
      - 3 coins according to previous column

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2		
7	7	3		
8	8	4		
9	9	3		
10	10	4		
11	11	5		
12	12	4		
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - At value 6:
    - Can use 5
      - Costs: 1+1
    - Cannot use 5
      - Costs 2

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3		
8	8	4		
9	9	3		
10	10	4		
11	11	5		
12	12	4		
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - At value 8:
    - Can use 5
      - Costs: 1+1
    - Cannot use 5
      - Costs 4

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3	3	
8	8	4	2	
9	9	3		
10	10	4		
11	11	5		
12	12	4		
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - At value 9:
    - Can use 5
      - Costs: 2+1
    - Cannot use 5
      - Costs 3

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3	3	
8	8	4	2	
9	9	3	3	
10	10	4		
11	11	5		
12	12	4		
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - At value 10:
    - Can use two 5s
    - Can use one 5
      - Costs: 3+1
    - Can use no 5s
      - Costs 3

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3	3	
8	8	4	2	
9	9	3	3	
10	10	4	2	
11	11	5		
12	12	4		
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - At value 11:
    - Can use two 5s
      - Costs 2+1
    - Can use one 5
      - Costs: 2+1
    - Can use no 5s
      - Costs 5

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3	3	
8	8	4	2	
9	9	3	3	
10	10	4	2	
11	11	5	3	
12	12	4		
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - At value 12:
    - Can use two 5s
      - Costs 2+2
    - Can use one 5
      - Costs: 3+1
    - Can use no 5s
      - Costs 4

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3	3	
8	8	4	2	
9	9	3	3	
10	10	4	2	
11	11	5	3	
12	12	4	4	
13	13	5		
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - At value 13:
    - Can use two 5s
      - Costs 2+1
    - Can use one 5
      - Costs: 4+1
    - Can use no 5s
      - Costs 5

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3	3	
8	8	4	2	
9	9	3	3	
10	10	4	2	
11	11	5	3	
12	12	4	4	
13	13	5	3	
14	14	6		
15	15	5		

The Change Making Problem

- Now on to five
  - At value 14:
    - Can use two 5s
      - Costs 2+2
    - Can use one 5
      - Costs: 3+1
    - Can use no 5s
      - Costs 6

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3	3	
8	8	4	2	
9	9	3	3	
10	10	4	2	
11	11	5	3	
12	12	4	4	
13	13	5	3	
14	14	6	4	
15	15	5		

The Change Making Problem

- Now on to five
  - At value 15:
    - Can use three 5s
      - Costs 3
    - Can use two 5s
      - Costs 2+3
    - Can use one 5
      - Costs: 4+1
    - Can use no 5s
      - Costs 5

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3	3	
8	8	4	2	
9	9	3	3	
10	10	4	2	
11	11	5	3	
12	12	4	4	
13	13	5	3	
14	14	6	3	
15	15	5	3	

The Change Making Problem

- Now on to eights
  - At value 15:
    - Can use one eight
      - Costs 1+3
    - Can use no eights
      - Costs: 3

Value	ones	threes	five	eights
0	0	0	0	0
1	1	1	1	
2	2	2	2	
3	3	1	1	
4	4	2	2	
5	5	3	1	
6	6	2	2	
7	7	3	3	
8	8	4	2	
9	9	3	3	
10	10	4	2	
11	11	5	3	
12	12	4	4	
13	13	5	3	
14	14	6	3	
15	15	5	3	3

The Change Making Problem

- Alternative: Memoization and Recursion
  - Instead of using a tableau
    - (or rather two, one to remember the best choice)
  - Can use recursion and memoization
    - Simplest form:
      - What was the last coin that was added
        - It has to be one of the coins: e.g. 1, 3, 5, or 8
        - The costs are the cost of making change for the amount minus the value of the coin plus one for the coin itself

The Change Making Problem

- Alternative: Memoization and Recursion
  - Recursion

$$c(n) = \min\{c(n - v_i) + 1\}$$

    - where the minimum is taken over all different coin values
    - We also write the coin which causes the minimum to be selected

The Change Making Problem

- For memoization in Python:
  - have a global dictionary for the costs and the best choice of coin (last_coin)
  - Also, add the values of the coins in a list

```
last_coin = {0:0}
costs = {0:0}
values = [1,3,5,7,8]
```

The Change Making Problem

- Here is very simple Python code

```
def getChange(n):
    if n in costs:
        return costs[n]

    best = 100000
    bestcoin = 0
    for x in range(len(values)):
        if values[x] > n:
            break
        alternativeCost = getChange(n-values[x])+1
        if alternativeCost < best:
            best = alternativeCost
            bestcoin = values[x]
    costs[n] = best
    last_coin[n] = bestcoin
    return best
```

The Change Making Problem

- And here is the output
  - Amount to make change for
  - Number of coins needed
  - Last coin used
- Example:
  - For 20, use a 5, left 15
  - For 15, use a 7, left 8
  - For 8, use 8

```
0 0 0
1 1 1
2 2 1
3 1 3
4 2 1
5 1 5
6 2 1
7 1 7
8 1 8
9 2 1
10 2 3
11 2 3
12 2 5
13 2 5
14 2 7
15 2 7
16 2 8
17 3 1
18 3 3
19 3 3
20 3 5
```

The Change Making Problem

- But we do not have this problem with normal coin sets
  - US$-cents: 1, 5, 10, 25, 100
  - Euro-cents: 1, 5, 10, 20, 50, 100, 200

The Change Making Problem

- Cashier's Algorithm
  - Always select the largest coin smaller or equal the current amount
  - Will not always work
    - Another example: US Postage Stamps before forever
      - 1, 5, 25, 32, 100
      - Make change for 121
        - Cashier's algorithm: 100+5+5+5+5+1
        - Better choice 32+32+32+25

The Change Making Problem

- But sometimes the Cashier's Algorithm is the best
  - Assume that we have coins of 1, 5, 10, 20, and 50
  - Proof by induction that the cashier's algorithm always give the best change
  - Represent the change as an array
    - Coefficient i of array: number of i-th coins
    - Example:

1	5	10	20	50
3	2	4	8	0

      - one way of making change for 213

The Change Making Problem

- Proof:

- Assume $C = [c_1, c_5, c_{10}, c_{20}, c_{50}]$ is the result of the cashier's algorithm for an amount of

$$c_1 + c_5 \cdot 5 + c_{10} \cdot 10 + c_{20} \cdot 20 + c_{50} \cdot 50$$

- Assume $A = [a_1, a_5, a_{10}, a_{20}, a_{50}]$ is an alternative with less coins for the same amount

$$a_1 + a_5 \cdot 5 + a_{10} \cdot 10 + a_{20} \cdot 20 + a_{50} \cdot 50$$

but

$$a_1 + a_5 + a_{10} + a_{20} + a_{50} < c_1 + c_5 + c_{10} + c_{20} + c_{50}$$

The Change Making Problem

- Proof:
- Want to show that $A = C$.

The Change Making Problem

- Proof:
  - Lemma 1: An optimal solutions has not more than four pennies
    - Otherwise replace with a 5 cent piece
  - Lemma 2: An optimal solution has not more than one 10 cent piece
    - Otherwise replace with a 20 cent piece
  - Lemma 3: An optimal solution cannot have two twenty cent pieces and one 10 cent piece
    - Otherwise replace with a 50 cent piece

The Change Making Problem

- Proof:
  - Lemma 5: Maximum number of pennies in an optimal solution is four
    - Follows from Lemma 1
  - Lemma 6: If the optimal solution has only pennies and five cents, then the amount is at most nine
    - Follows from Lemma 2 and Lemma 5

The Change Making Problem

- Lemma 7: The maximum amount for an optimal solution with only pennies, 5 cent and 10 cent pieces is 19
- Lemma 8: The maximum amount for an optimal solution with only 1 cent, 5 cent, 10 cent, and 20 cent pieces is 49

The Change Making Problem

- Proof:
  - Assume that the number of 50 cent coins in A and C differ.
  - Because of how C is defined, the number of 50 cent coins in A has to be lower $a_{50} < c_{50}$.
  - However, the difference needs to be made up with coins of smaller value
  - But an optimal solution cannot have more than 49 cents in smaller coins
  - Contradiction

The Change Making Problem

- Proof:
  - So, the number of 50 cent coins does not differ
    - If there are x 50 cent coins, then look at the best solution for amount-x coins.
    - C and A with the 50 cent coins removed are still two different solutions for the same amount
  - Now apply the same argument to the 20 cent coins.
  - Et cetera

The Change Making Problem

- We call the cashier's algorithm a *greedy algorithm*:
  - We solve the problem by going to a smaller problem
    - E.g. Making change for 134 cents.
    - Lay out 50 cents
    - Making change for 84 cents.
    - …
  - At each step, we select something optimal

Greedy Algorithms

- Many algorithms run from stage to stage
  - At each stage, they make a decision based on the information available
- A Greedy algorithm makes decisions
  - At each stage, using locally available information, the greedy algorithm makes an optimal choice

- Sometimes, greedy algorithms give an overall optimal solution
- Sometimes, greedy algorithms will not result in an optimal solution but often in one good enough

Divisible Items Knapsack Problem

- Given a set of items S
  - Each item has a weight $w(x)$
  - Each item has a value $v(x)$
- Select a subset $M \subset S$
  - Constraint: $\sum_{x \in M} w(x) < W$
  - Objective Function: $\sum_{x \in M} v(x) \longrightarrow \max$

Divisible Items Knapsack Problem

- Order all items by impact
  - $\textbf{impact}(x) = \dfrac{v(x)}{w(x)}$
- In order of impact (highest first), ask whether you want to include the item
  - And you include it if the sum of the weights of the items already selected is smaller than W

Optimal Rental

- Set of activities $S = \{a_1, a_2, \ldots, a_n\}$
  - Each activity has a start time and a finish time
    - $0 \leq s_i < f_i < \infty$
  - Each activity needs to use your facility
  - Only one activity at a time
  - Make the rental agreements that maximize the number of rentals

Optimal Rental

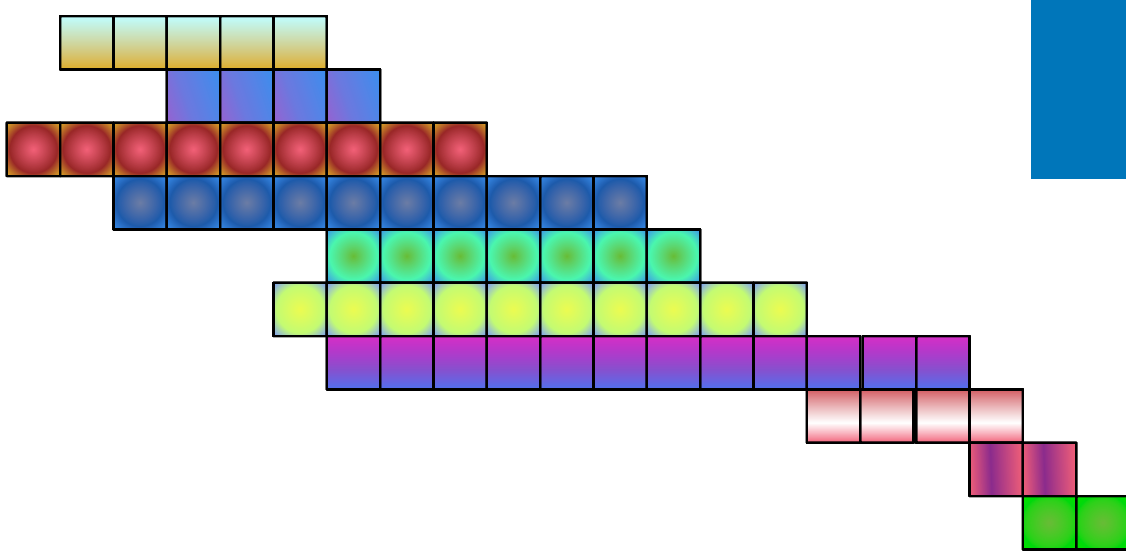
- Two activities a_i and a_j are compatible iff
  - $[s_i, f_i) \cap [s_j, f_j) = \emptyset$

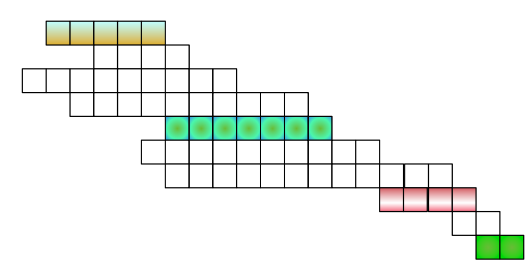
- This means that activity $i < j$ finishes before activity j

Optimal Rental

- Example:

i	1	2	3	4	5	6	7	8	9	10
si	1	3	0	2	6	5	6	15	18	19
fi	6	7	9	12	13	15	18	19	20	21





- A compatible set is $\{A_1, A_5, A_8, A_{10}\}$
- Another compatible set is $\{A_3, A_9\}$

Optimal Rental

- Optimal rental with a dynamic programming algorithm
  - Subproblems: Define S_{ik} to be the set of activities that start after a_i finishes and finish before a_k starts

i	1	2	3	4	5	6	7	8	9	10
si	1	3	0	2	6	5	6	15	18	19
fi	6	7	9	12	13	15	18	19	20	21

$$S_{1,8} = \{a_5\}$$

Optimal Rental

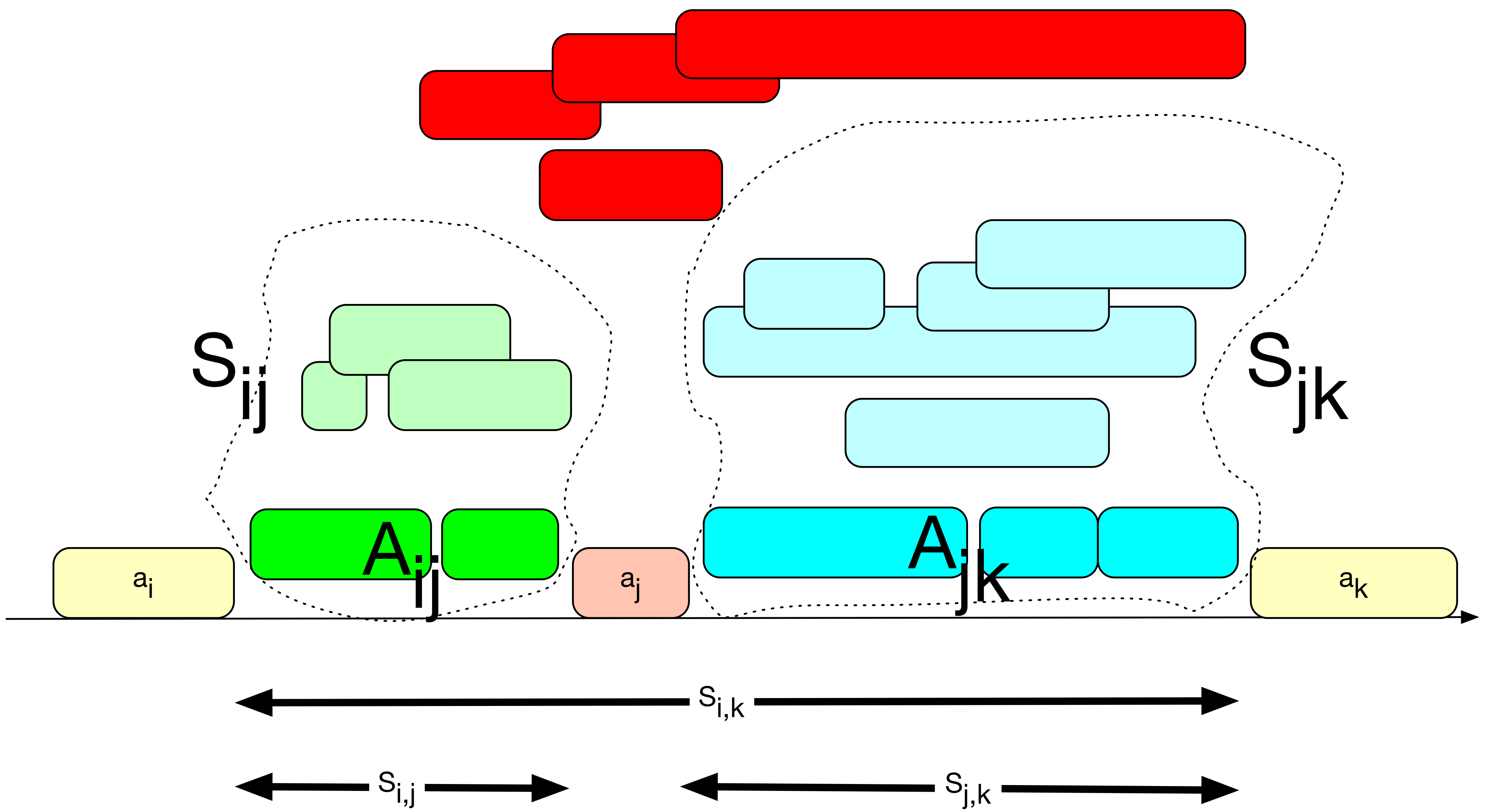
- We want to find an optimal rental plan for S_{ik}
  - Assume that there is an optimal solution that contains activity $a_j \in S_{i,k}$
  - By selecting a_j, we need to decide what to do with the time before a_j starts and after a_j finishes
  - These sets are S_{ij} and S_{jk}

Optimal Rental

- Assume that a_j is part of an optimal solution $A_{i,k}$ for $S_{i,k}$
  - Then $A_{i,k}$ is divided into the ones that end before a_j and the ones that start after a_j
  - $A_{i,j} = A_{i,k} \cap S_{i,k} \quad A_{j,k} = A_{i,k} \cap S_{j,k}$

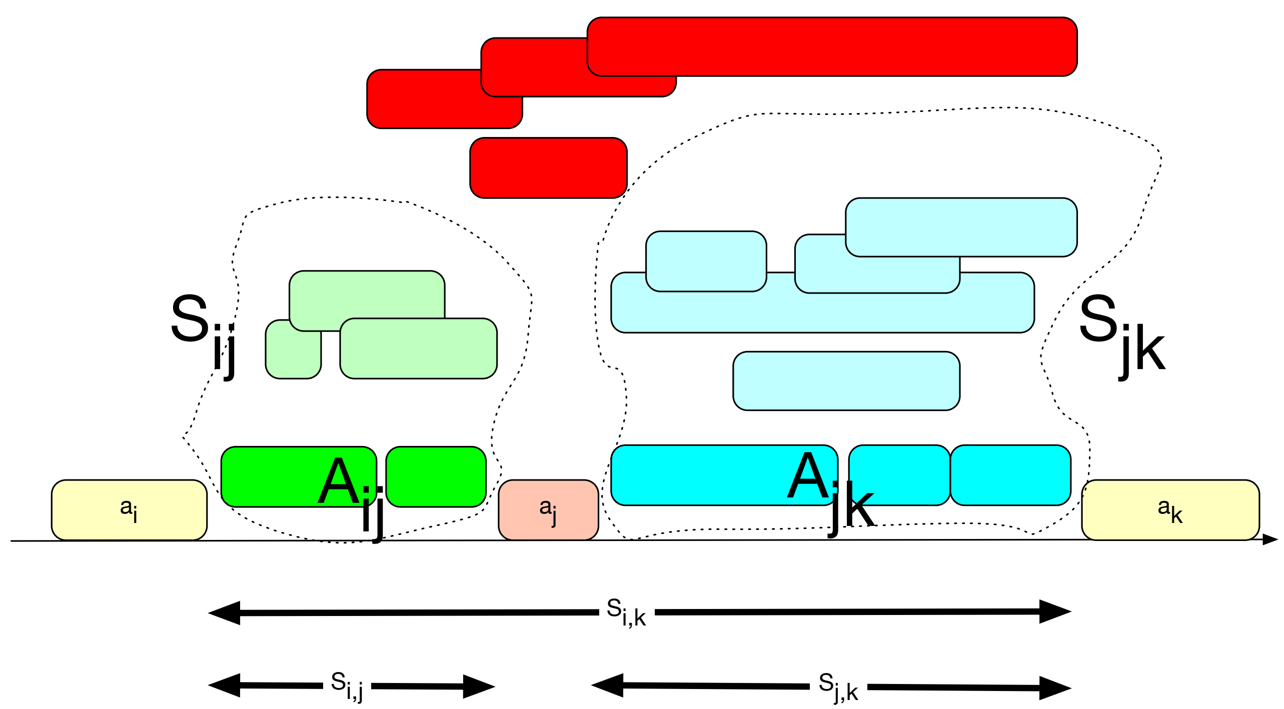
$$A_{i,k} = A_{i,j} \cup \{a_j\} \cup A_{j,k}$$

Optimal Rental



Optimal Rental

- Clearly, $A_{i,j}$ is an optimal solution for $S_{i,j}$
- $A_{j,k}$ is an optimal solution for $S_{j,k}$
- For if not, we could construct a better solution for $S_{i,k}$



Optimal Rental

- We can therefore solve recursively the problem for $S_{i,k}$ by looking at all possible activities for a_j
  - Define $C[i, k] =$ Max number of compatible activities in $S_{i,k}$
  - Then:

$$C[i, k] = \max(0, \max\left(C[i, j] + C[j, k] + 1 \,|\, a_j \in S_{i,k}\right))$$

  - The 0 is necessary because there might be no activity in $S_{i,k}$

Optimal Rental

- The recursion leads to a nice dynamic programming problem

$$C[i,k] = \max(0, \max\left(C[i,j] + C[j,k] + 1 \,|\, a_j \in S_{i,k}\right))$$

Optimal Rental

- But can we do better?

Optimal Rental

- Start out with the initial problem
  - Select the activity that finishes first
    - this would be a_1
  - This leaves most space for all other activities
    - Call S_1 the set of activities compatible with a_1
      - These are those starting after a_1
    - Similarly, call S_k the set of activities starting after a_k

Optimal Rental

- Theorem: For any non-empty problem S_k let a_m be the activity with the smallest end time. Then a_m is contained in an optimal solution

- Proof:
  - Let A_k be a solution
    - i.e. the maximum sized compatible subset in S_k
    - Let $a_1 \in A_k$ be the activity with earliest finish time
    - If $a_m = a_1$ then we are done

Optimal Rental

- Theorem: For any non-empty problem S_k let a_m be the activity with the smallest end time. Then a_m is contained in an optimal solution
- Proof:
    - Otherwise replace a_1 with a_m in A_k
        - $A'_k = A_k - \{a_1\} \cup \{a_m\}$
        - Since a_m is the first to finish, this is a set of compatible activities
        - Therefore, there exists an optimal solution with a_m

Optimal Rental

- Result of the Theorem:
    - We can find an optimal solution (but not necessarily all optimal solutions) by always picking the first one to finish.

Optimal Rental

- Example

i	1	2	3	4	5	6	7	8	9	10
si	1	3	0	2	6	5	6	15	18	19
fi	6	7	9	12	13	15	18	19	20	21

- Select a_1
- Exclude a_2, a_3, and a_4 as incompatible
- Choose a_5, a_8, and a_{10} for the complete solution

Greedy Algorithms

- Greedy algorithms
    - Determine the optimal substructure
    - Develop a recursive solution
    - Show that making the greedy choice is best
    - Show that making the greedy choice leads to a similar subproblem
    - Obtain a recursive algorithm
    - Convert the recursive algorithm to an iterative algorithm