Greedy Algorithms

Sonali Gogate

Coin selection

Suppose you have unlimited supply of coins of the following denominations -

Rs. 1, 2, 5, 10, 20

And you are asked to devise a method to always pay the required amount in fewest number of coins

Rs 27

1. (5 * 5) + 2 (6 coins)
2. 10 + (5 * 3) + 2 (5 coins)
3. 10 * 2 + 5 + 2 (4 coins)
4. 20 + 5 + 2 (3 coins)

Cashier's Algorithm ($x, c_1, c_2, \ldots, c_n$)

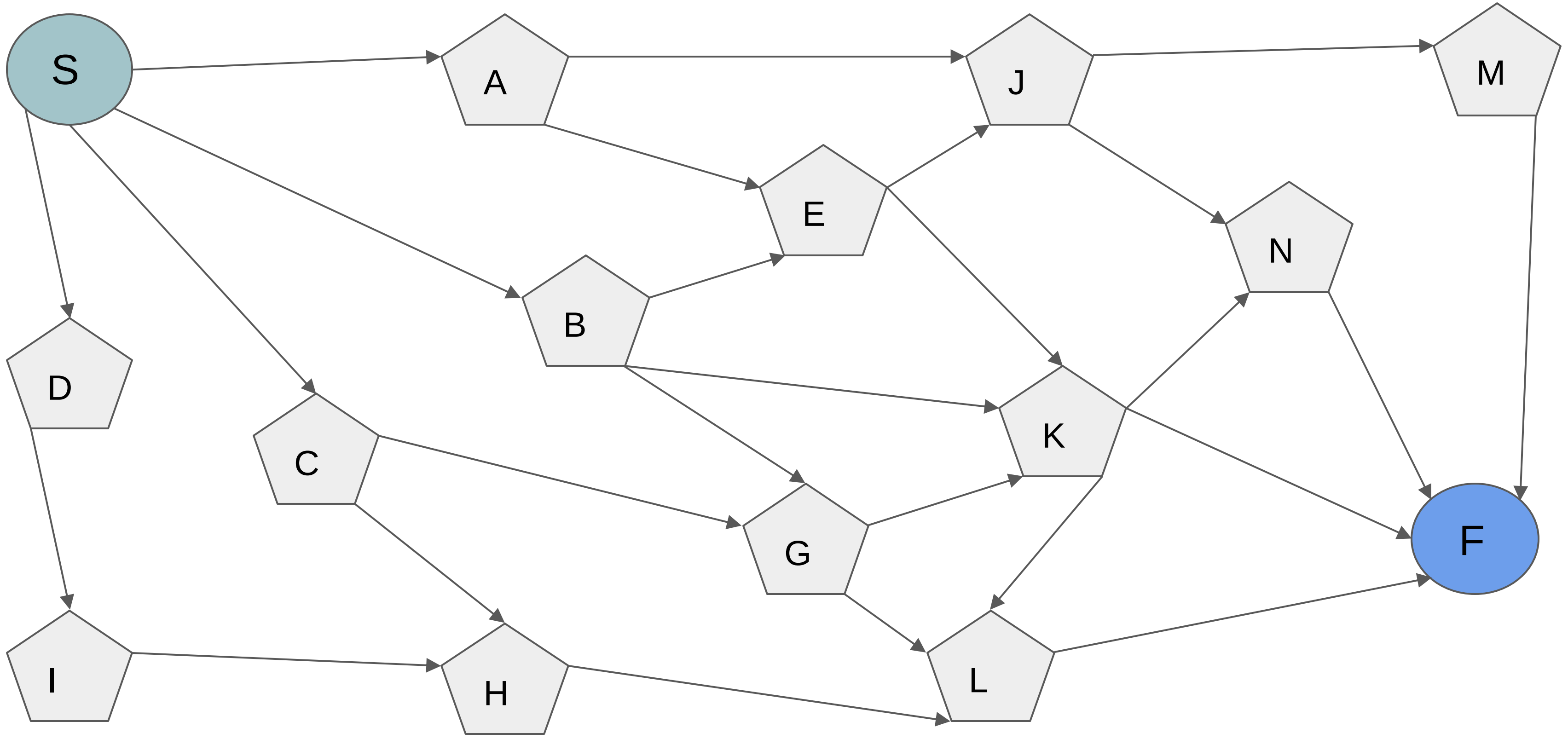
```
Sort n coin denominations so that $0 < c_1 < c_2 < \ldots < c_n$
Set  S  = {Ø};
While (x > 0)
{
    Find k, the largest denomination coin such that $c_k <= x$;
    $x = x - c_k$;
    S = S U {k}
}
Return S;
```

Minimizing the jumps for the Frog

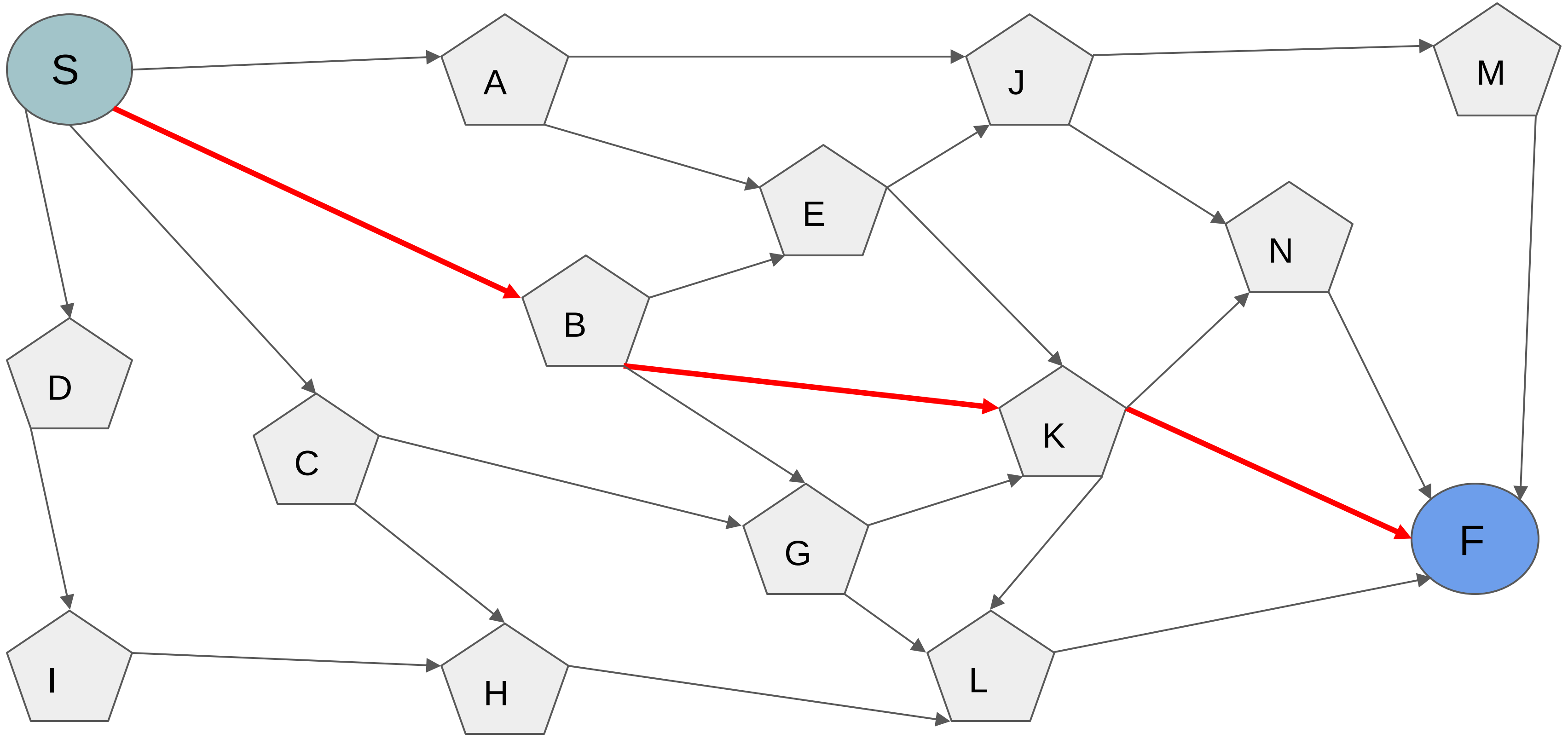
- There is a pond with lily pads
- There is a frog at one end and he wants to go to the other end with minimum number of jumps.
- Let us say where he is, is the starting position (S)
- And let us say where he wants to go is the finishing position (F)
- So, he wants to go from S to F with minimum number of jumps

Goal is the find the path the frog should take to minimize the jumps

Frog Jumping - Go from St to Fi in min number of jumps



Frog Jumping - Go from St to Fi in min number of jumps



Let us put the algorithm together

```
Let us say the set of jumps is Jp. When the frog is at S, Jp = {Ø};
Let the current position of the frog be defined by x; At the start x = S and at the finish x = F;
While the frog has not reached F,  //While x != F
{
        Find the furthest jump possible from current position x, say to y;
        Jp = Jp + (x,y);
        x = y;
}
Return Jp;
```

Greedy Algorithms

A Greedy Algorithm is one that constructs the solution one step (or one item) at a time. At each step, the algorithm chooses the locally best solution, hoping that such collection of locally best steps or items will lead to overall optimal solution.

In other words -

For an objective function that needs to be optimized (either maximized or minimized), a Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized.

Job Scheduling Problem

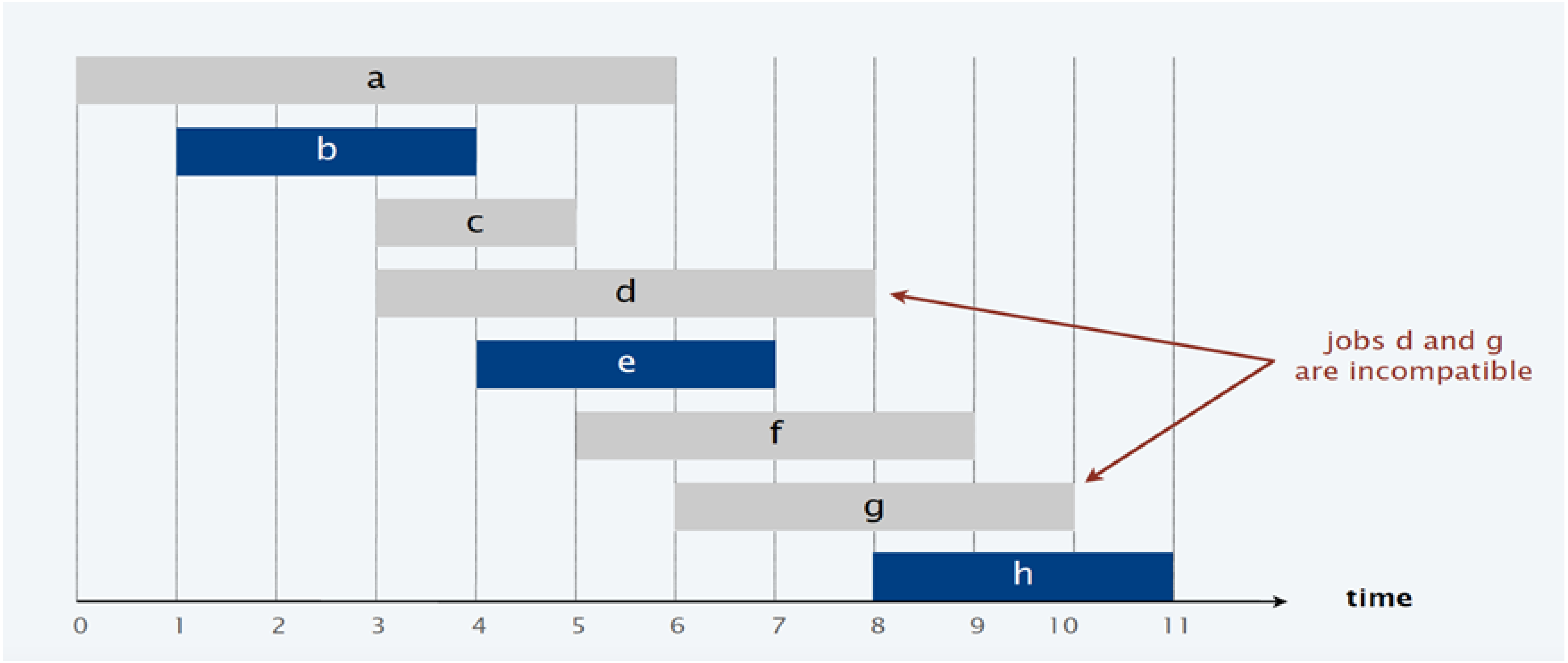
- Suppose there is a set of jobs that need be done and all of them need to use the same set of resources. (So only one job can be executed at a time.)
- These jobs could need to start at different times and could require different amount of time to complete.
- For a job j the start time and the end time are defined s_j and f_j.
- Two jobs are compatible if they don’t overlap.

Goal: find maximum subset of mutually compatible jobs.

Steps of the algorithm

1. Sort the jobs in ascending order of finishing times.
2. Set the starting set of selected jobs to be empty.
3. Then, greedily select jobs that are compatible with the current set.

Set of Jobs to be Scheduled



Job Scheduling Algorithm

```
Greedy Job Scheduling (n, $s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n$)
Sort the jobs by finish times and renumber so that $f_1 \le f_2 \le \ldots \le f_n$;
Set S = {Ø};
For ( j = 1 to n)
    If ( job j is compatible with S)
        $S \leftarrow S \cup \{ j \}$;
Return S;
```

Greedy Algorithm Time Complexity

What is the first step always?

- Sorting
- Time complexity of Sorting

What is the next step?

- Selection of jumps, items, jobs etc
- Time complexity of selection?

Overall Time complexity?

Advantages & Challenges of Greedy Algorithms

Advantages

Simplicity: Greedy Algorithms are often easy to describe and to code

Efficiency: Quite often, Greedy algorithms can be implemented efficiently

Challenges

Hard to design: Once we have got the right approach, it is easy. However, it can be very difficulty to find the right approach

Hard to verify: Showing the correctness of a Greedy algorithm is not always easy.

Knapsack Problem

Knapsack Problem

Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. (From Wikipedia)

Some interesting information about the problem -

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "knapsack problem" dates back to the early works of the mathematician Tobias Dantzig (1884–1956), and refers to the commonplace problem of packing the most valuable or useful items without overloading the luggage.

Knapsack Problem basic statement

You have set of n (1 to n) items each with a weight and a value. Weight and value of an item i are w_i and v_i respectively.

You have a weight limit for the knapsack capacity, say M.

You want to select items from the set of n such that keeping in the weight limit, you maximize the value.

Two variants of the Knapsack Problem

1. 0-1 Knapsack
   - You can either take an element or leave it.

1. Fractional Knapsack
   - You can take a fraction of the elements. So i can take 2/5 or 7/12 or whatever fraction I want to of any element if that can help in maximizing the value.

Fractional Knapsack Problem can be solved using the Greedy Method

Fractional Knapsack - a small example

There are 3 elements with following weights, values and M = 20

w_1	v_1
18	25
w_2	v_2
15	24
w_3	v_3
10	15

What happens when I take different fractions of these three

x_1	x_2	x_3	Total weight	Total Value
1/2	1/3	1/2	19	28
1	2/15	0	20	28.2
0	2/3	1	20	31
0	1	1/2	20	31.5

Goal of the Knapsack Algorithm

Given the capacity of the Knapsack AND the weights & values/profits of the available items, the goal is -

To Maximise $\sum v_i x_i$

Subject to $\sum w_i x_i <= M$

$0 <= x_i <= 1$ and $1 <= i <= n$

Greedy Algorithm for Fractional Knapsack Problem

```
Greedy Knapsack (M,n)
// P [1:n] and W[1:n] contain the profits and
weights of the n elements
//Sort the objects in descending order of $p_i/w_i$
{
    for (i = 1 to n)
        x[i] = 0;
    U = M;
    for (i = 1 to n)
    {
        If (w[i] <= U)
        {
            X[i] = 1;
            U = U - w[i];
        }
        else
            break;
    }
    if ( i <= n)
        x[i] = U/w[i];
}
```