Colorado State University

CS 320
Algorithms: Theory and Practice
Introduction 01.2
Representative Problems

Sanjay Rajopadhye
(inspired by Wim Bohm)
Colorado State University
Aug 2023

Problem solving paradigm

1. Formulate the problem with precision (usually using mathematical concepts, such as sets, relations, and graphs, costs, benefits, optimization criteria)
2. (Re)design an algorithm
3. Prove its correctness
4. Analyze its complexity
5. Implement it *respecting the derived complexity*

- Steps 2-5 are often repeated, to improve efficiency

  The first algorithm for Stable Matching was exponential,

  The second was polynomial (quadratic)

Bipartite Matching

- Stable matching was defined as matching elements of two disjoint sets.
- We can express this in terms of graphs.
- A graph is *bipartite* if its nodes can be partitioned in two sets X and Y, such that the edges go from an
- x in X to a y in Y

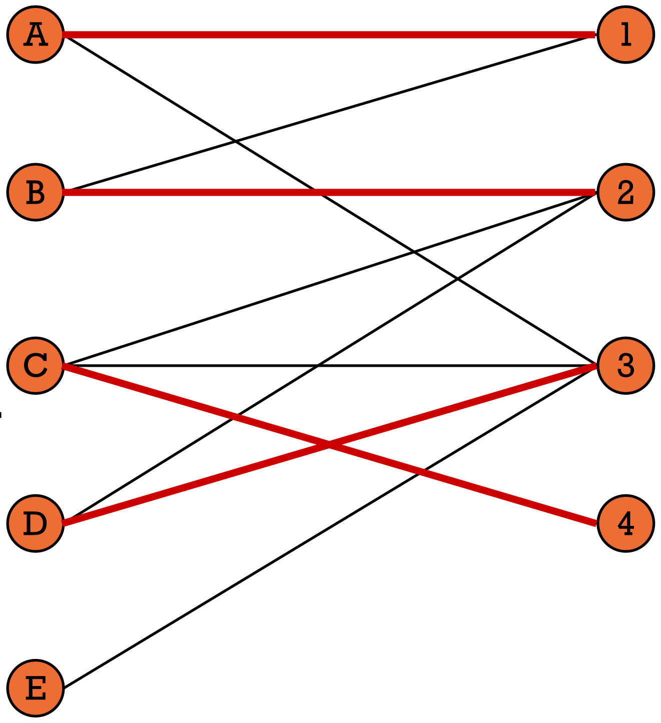
Bipartite Matching

- Input. Bipartite graph.
- Goal. Find *maximum cardinality* matching.

Matching can model assignment problems, e.g., assigning jobs to machines, where an edge between a job j and a machine m indicates that m can do job j, or professors and courses.

Same as stable matching problem?

- Not perfect
- No preferences, less information



Colorado State University

Interval Scheduling

- You have a resource (hotel room, printer, lecture room, telescope, manufacturing facility, professor...)
- There are requests to use the resource in the form of start time s_i and finish time f_i, such that $s_i < f_i$

Objective: grant as many requests as possible.

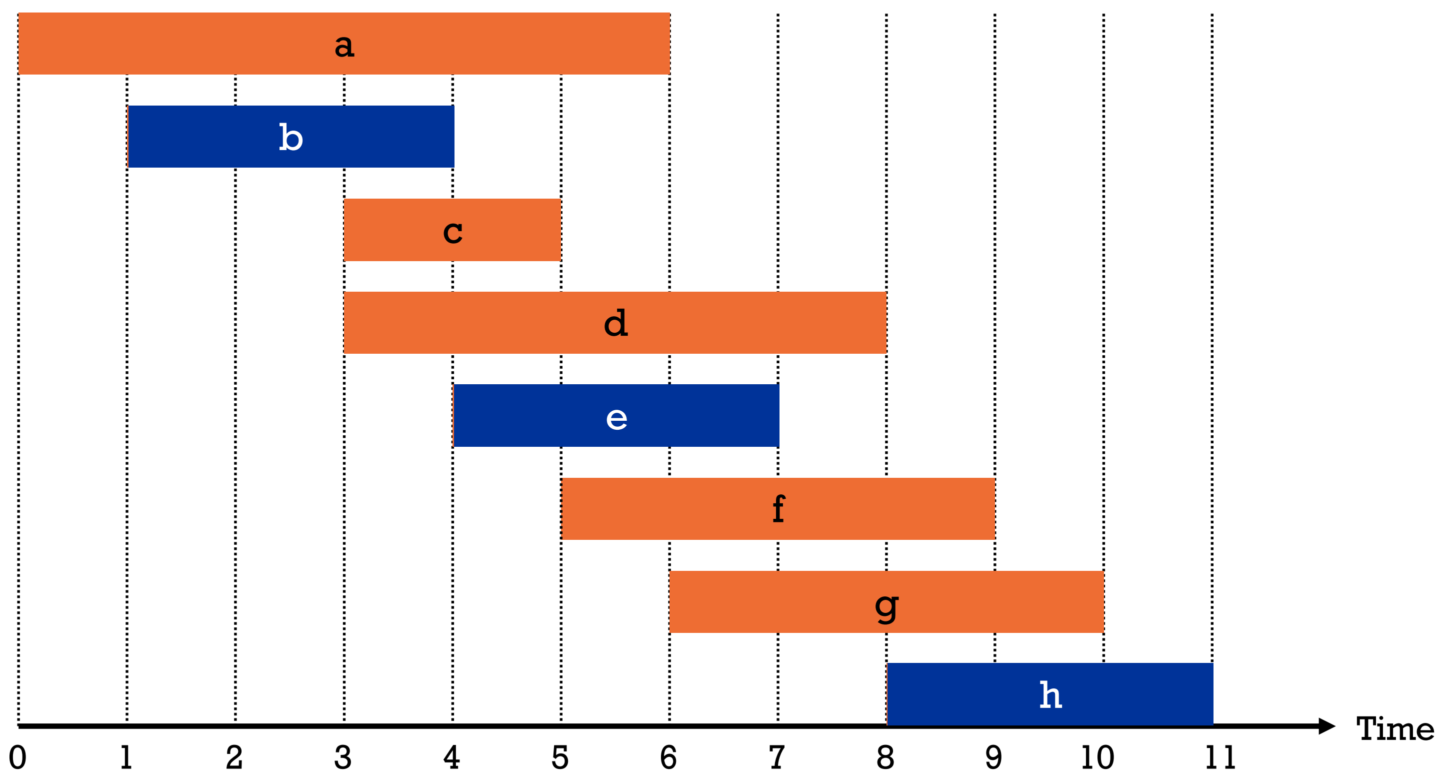
- Two requests i and j are *compatible* if they don't overlap, i.e.,

$$f_i \leq s_j \text{ or } f_j \leq s_i$$

Colorado State University

Interval Scheduling

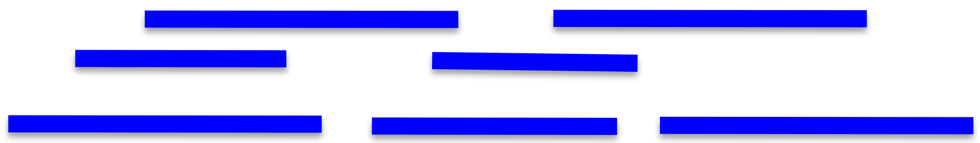
- Input. Set of jobs with start times and finish times.
- Goal. Find *maximum cardinality* subset of compatible jobs.



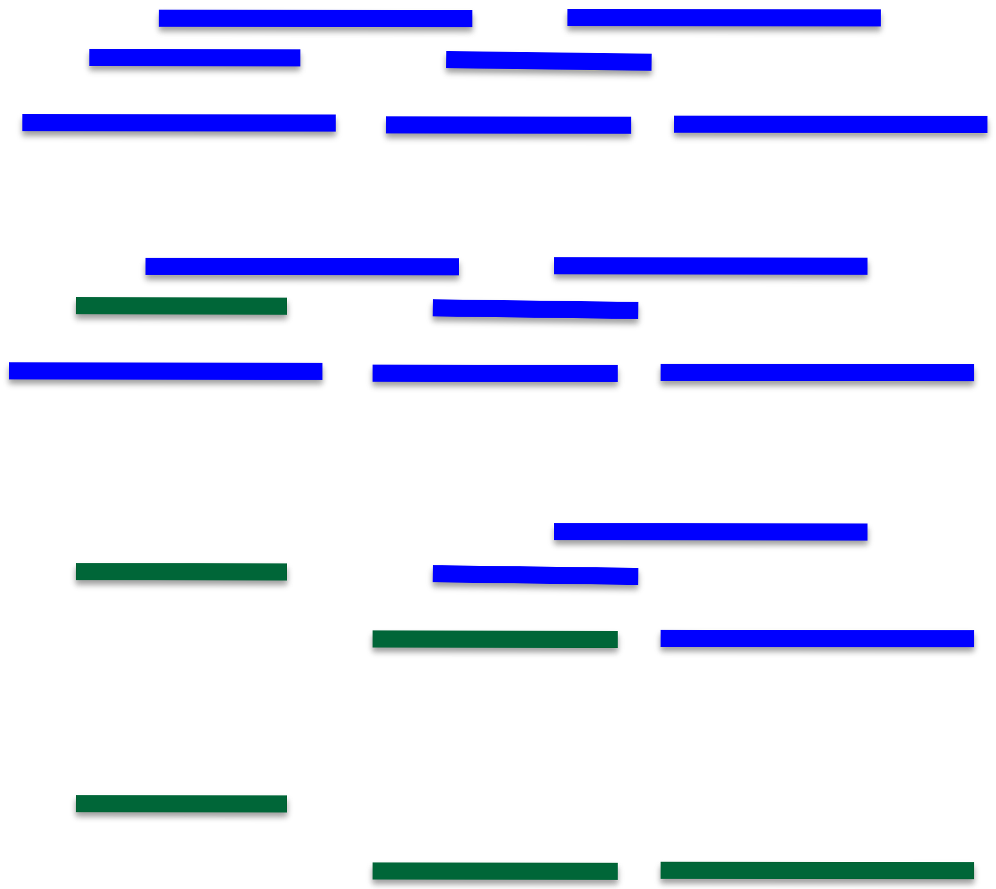
- What happens if you pick the first starting (a)?,
- the smallest (c)? What is the optimum?

Algorithmic Approach

- The interval scheduling problem is amenable to a very simple solution.
- Now that you know this, can you think of it using this example?
- Hint: Think how to pick a first interval while preserving the longest possible free time…

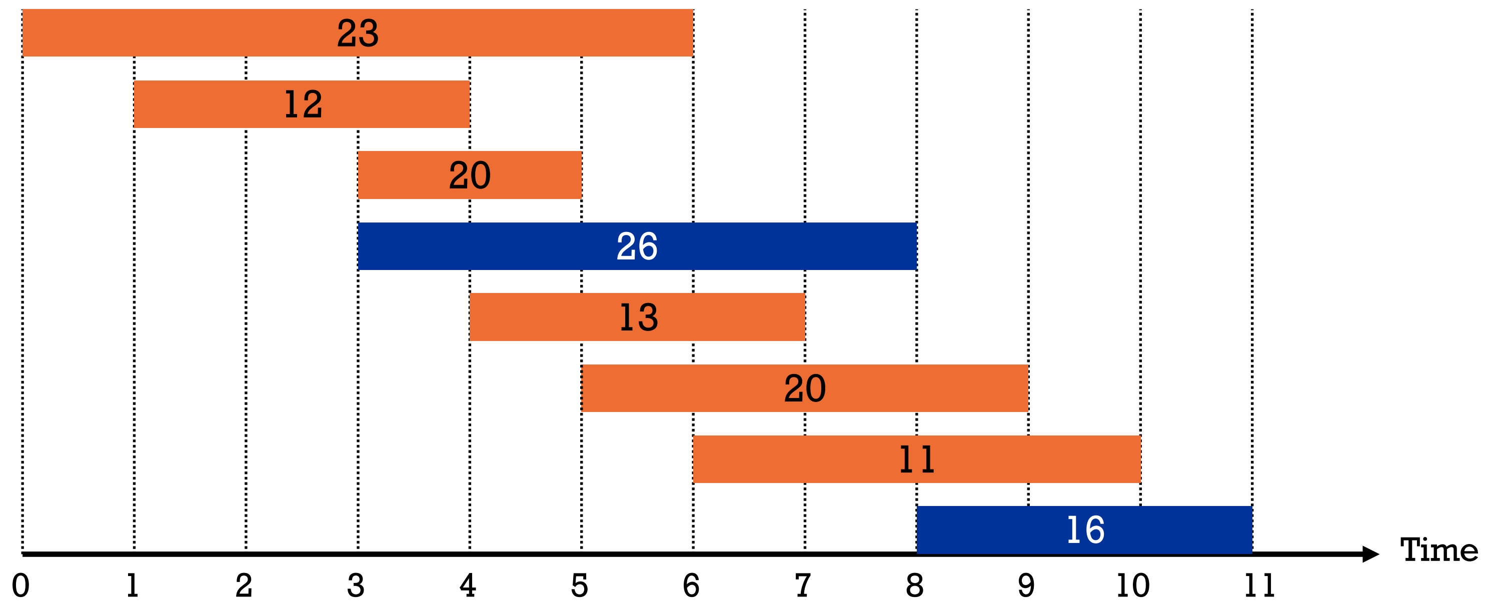


Colorado State University



Weighted Interval Scheduling

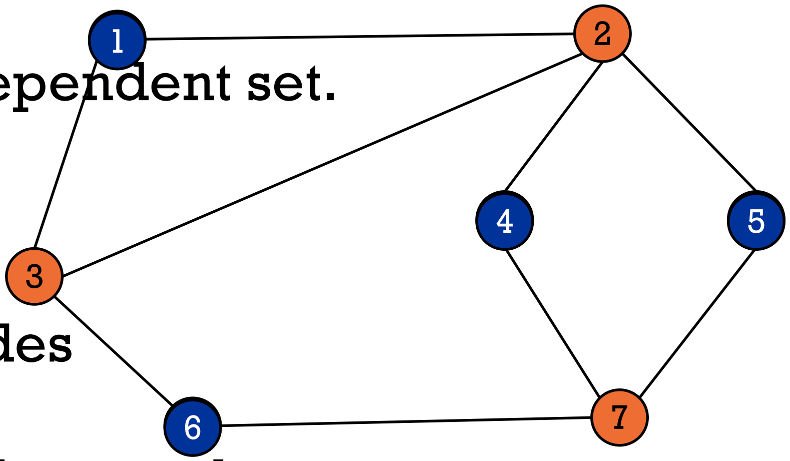
- Input: Set of jobs with start times, finish times, and profits.
- Goal: Find *maximum profit* subset of compatible jobs.



Independent set

- Input: Graph.
- Goal: Find *maximum cardinality* independent set.

Definition: two nodes are independent if there is no edge between them. An *independent set* is a subset of pairwise-independent nodes



Any relation to interval scheduling?

~~They are the same problem. An algorithm to solve one can be used to solve the other – aka reduction.~~

The interval scheduling problem can be formulates as an *instance of* the independent set problem. Build a graph where nodes correspond to the interval, and there is an edge between any pair of nodes whose intervals are not compatible (what is the complexity of this construction?)

Colorado State University

Independent set

- There is no known *efficient algorithm* to solve the independent set problem
- But we just said that we can formulate the interval scheduling problem as an independent set problem
  - And the former has a *very efficient algorithm*
- What gives?
- And what is an *efficient algorithm*?
  - One where the only option is to try all the subsets and find the largest one.
  - Q: How many subsets does a set of cardinality n have?
  - A: 2^n (recall counting from CS 220)

Representative problems and their complexities

Looking ahead

- Interval scheduling has an $n \log n$ greedy algorithm
- Weighted interval scheduling has an $n \log n$ dynamic programming algorithm
- Independent set is in NP – no known polynomial time algorithm exists

Algorithm

- An algorithm is an *effective* procedure
  - To map the input to the output
  - effective = unambiguous, executable
  - Like a Turing machine
- Is there an effective algorithm for every possible problem?
  - No, the problem must the effectively specified (e.g., "*how many angels can dance on the head of a pin?*" is not
  - Even if it is effectively specified, there is not always an algorithm to solve it
  - Often occurs in *analyzing programs*

Ulam's problem

Steps in running f(n) for a few values of n:

1

2, 1

3, 10, 5, 16, 8, 4, 2, 1

4, 2, 1

5, 16, 8, 4, 2, 1

6, 3, 10, 5, 16, 8, 4, 2, 1

7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

8, 4, 2, 1

9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

10, 5, 16, 8, 4, 2, 1

```
def f(n) :
  if (n==1) return 1
        elif (odd(n)) return
  f(3*n+1)
        else return f(n/2)
```

Ulam's problem

Question:

Does f(n) always stop?

- Nobody has found a proof that f(n) stops for all n
- Nobody has found a counter-example – a value of n for which the algorithm loops for ever

```
def f(n) :
    if (n==1) return 1
        elif (odd(n)) return f(3*n+1)
        else return f(n/2)
```

- A generalization of this has been proved to be *undecidable*.
  - A problem P is *decidable* if there is an algorithm that produces P(x) for every possible input x
  - It is *undecidable* if we can prove that no such algorithm exists

The Halting Problem is undecidable

- A generalization of Ulam's problem
- Given a program P(x) for and input x, will P(x) stop on input x?

We can prove (CS420) that:

The halting problem is undecidable

i.e. there is no algorithm Halt(P,x) that decides whether P stops on input x (note that P and x are parameters to Halt, so it must work correctly on any P and x.

But for some "nice" programs, we can prove they halt, e.g.:

```
for i in range(100): print(i)
```

Colorado State University

Intractability

- Suppose we have a program,
  - does it execute a in a reasonable time?
  - e.g., towers of Hanoi (CS 220).

Three pegs, one with n smaller and smaller disks, move (1 disk at a time) to another peg without ever placing a
larger disk on a smaller

Monk: before a tower of Hanoi of size 100 is moved, the world will have vanished

hanoi

```
# pegs are numbers, via is computed
# empty base case
def hanoi(n, from, to):
  if (n>0) :
    via = 6 - from - to
   hanoi(n-1,from, via)
    print "move disk", n,  " from", from, " to ",  to
    hanoi(n-1,via,to);
```

Colorado State University

f(n): #moves in hanoi

f(n) = # moves for tower of size n

$f(n) = 2f(n-1) + 1$, $f(1)=1$

$f(1) = 1$, $f(2) = 3$, $f(3) = 7$, $f(4) = 15$

- Claim $f(n) = 2^n-1$
  - How can you show that?
  - By induction (CS 220)
- Was the monk right?
- 2^{100} moves, say 1 per second.....
- How many years?

$2^{100} \sim 10^{30} \sim 10^{25}$ days $\sim 3.10^{22}$ years

- *more than the age of the universe*

Colorado State University

Is there a better algorithm?

THE ONE MILLION DOLLAR QUESTION IN THIS CLASS

Is there a better algorithm for towers of hanoi?

Pile(n-1) must be

Off peg 1, and

completely on one other peg

before disk n can be moved to its destination

- so all moves are necessary

Algorithm complexity

- Measures in units of *time* and *space*
- Linear Search X in dictionary D

```
i=1
    while not at end and X!= D[i]:
        i=i+1
```

- CS220: We don't know if X is in D, and we don't know where it is, so we can only give *worst* or *average* time bounds
- We don't know the time for atomic actions, so we only determine *Orders of Magnitude*

Linear Search: time and space complexity

Space: n locations in D plus some local variables

Time:

- In *the worst case* we search all of D, so the loop body is executed n times
- In *average case* analysis we compute the *expected* number of steps: i.e., we sum the products of the probability of each option and the time cost of that option. In the average case the loop body is executed about n/2 times

$$\sum_{i=1}^{n} 1/n * i = 1/n \sum_{i=1}^{n} i = (n(n+1)/2)/n \approx n/2$$

Colorado State University