Final Exam Review

Final Exam

- Coverage: second half of the semester
- Requires familiarity with most of the concepts covered in first half
- Goal: doable in 2 hours
- Cheat sheet: you are allowed *two* 8'11" sheets, both sides

Final Exam: Study Tips

- Study tips:
  - Study each lecture
  - Study the homework and homework solutions
  - Study the midterm exams
- Re-make your previous cheat sheets

  I recommend handwriting or typing them
- Think about what you should have had on it the first time…cheat sheets is about *identifying* important concepts

- Next review of more recent topics as well as earlier topics

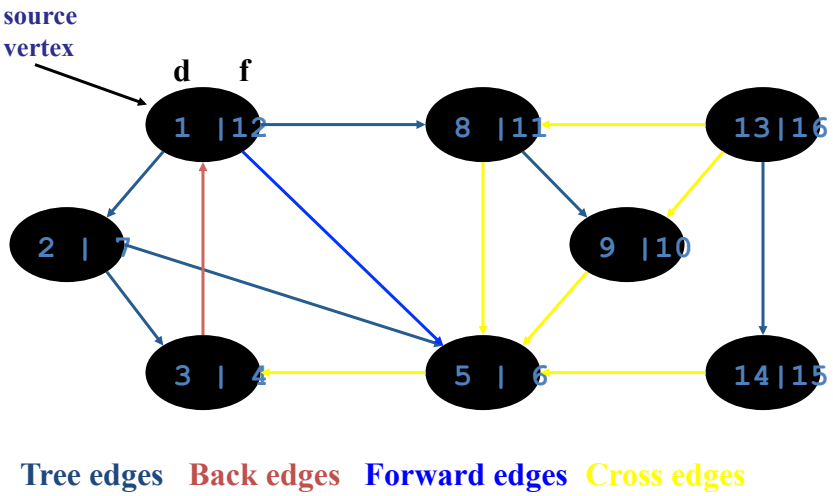
Graph Representation

- Adjacency list
- Adjacency matrix
- Tradeoffs:
  - *What makes a graph dense?*
  - *What makes a graph sparse?*
  - *What about trees ?*

Basic Graph Algorithms

- Breadth-first search
  - *What can we use BFS to calculate?*
  - A: shortest-path distance to source vertex
- Depth-first search
  - Tree edges, back edges, cross and forward edges
  - *What can we use DFS for?*
  - A: finding cycles, topological sort

DFS Example



DFS And Cycles

- *How would you modify the code to detect cycles?*

```
DFS(G)
{
   for each vertex u ∈ G->V
   {
      u->color = WHITE;
   }
   time = 0;
   for each vertex u ∈ G->V
   {
      if (u->color == WHITE)
         DFS_Visit(u);
   }
}
```

```
DFS_Visit(u)
{
   u->color = GREY;
   time = time+1;
   u->d = time;
   for each v ∈ u->Adj[]
   {
      if (v->color == WHITE)
         DFS_Visit(v);
   }
   u->color = BLACK;
   time = time+1;
   u->f = time;
}
```

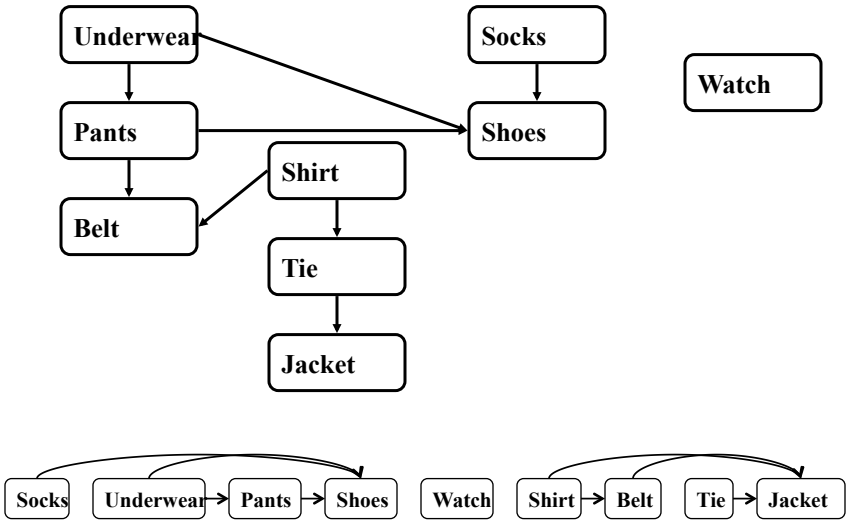
DFS And Cycles

- *What will be the running time?*
- A: O(V+E)
- We can actually determine if cycles exist in O(V) time:
  - In an undirected acyclic forest, $|E| \leq |V| - 1$
  - So count the edges: if ever see |V| distinct edges, must
  - have seen a back edge along the way

Topological Sort, MST

- Topological sort
  - Examples: getting dressed, project dependency
  - *To what kind of graph does topological sort apply?*
- Minimum spanning tree
  - Optimal substructure
  - Min edge theorem (enables greedy approach)

Getting Dressed



Topological Sort Algorithm

```
Topological-Sort()
{
   Run DFS
   When a vertex is finished, output it
   On the front of linked list
   Vertices are output in reverse topological
     order
}
```

- Time: O(V+E)
- Correctness: Want to prove that
  $(u,v) \in G \Rightarrow u\rightarrow f > v\rightarrow f$

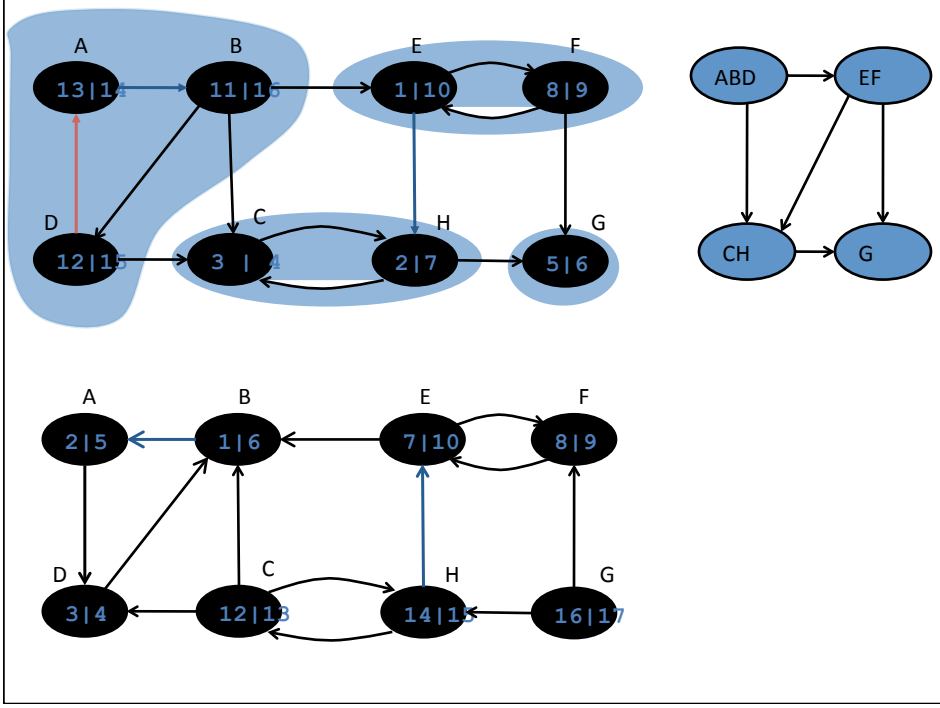
Correctness of Topological Sort

- Claim: $(u,v) \in G \Rightarrow u\rightarrow f > v\rightarrow f$
- Topological sort creates linear ordering of vertices
- Show that if there is an edge from u to v, finishing time of u is greater then v (nodes are output in reverse finish. times order – later times are output first)
- When (u,v) is explored, u is gray
  - v = gray $\Rightarrow$ (u,v) is back edge. Contradiction (*Why?*)
  - hence v cannot be gray – since there are no cycles
  - v = white $\Rightarrow$ v becomes descendent of u $\Rightarrow$ $v\rightarrow f < u\rightarrow f$
  - (since must finish v before backtracking and finishing u)
  - v = black $\Rightarrow$ v already finished $\Rightarrow$ $v\rightarrow f < u\rightarrow f$

Strongly Connected Components

- Call DFS to compute finishing times *f[u]* of each vertex
- Create transpose graph (directions of edges reversed)
- Call DFS on the transpose graph, but in the main loop of DFS, consider vertices in the decreasing order of *f[u]*
- Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

- Example



MST Algorithms

- Prim's algorithm
  - *What is the bottleneck in Prim's algorithm?*
  - A: priority queue operations
- Kruskal's algorithm
  - *What is the bottleneck in Kruskal's algorithm?*
  - Answer: depends on disjoint-set implementation
    - As covered in class, disjoint-set union operations
    - As described in book, sorting the edges

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What will be the running time?
A: Depends on queue
binary heap: O(E lg V)
Fibonacci heap: O(V lg V + E)

ExtractMin total number of calls O(V log V)

DecreaseKey total number of calls O(E log V)

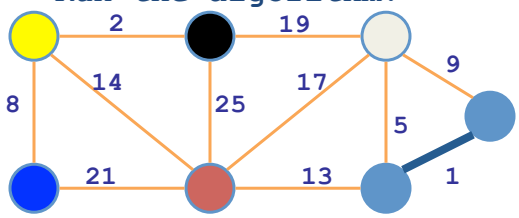
Total number of calls O(V logV +E logV) = O(E log V)

Think why we can combine things in the expression above

Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Correctness Of Kruskal's Algorithm

- Sketch of a proof that this algorithm produces an MST for *T*:
  - Assume algorithm is wrong: result is not an MST
  - Then algorithm adds a wrong edge at some point
  - If it adds a wrong edge, there must be a lower weight edge (cut and paste argument)
  - But algorithm chooses lowest weight edge at each step -> Contradiction
- Again, important to be comfortable with cut and paste arguments

Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

What will affect the running time?

1 Sort
O(V) MakeSet() calls
O(E) FindSet() calls
O(V) Union() calls
(Exactly how many Union()s?)

Kruskal's Algorithm: Running Time

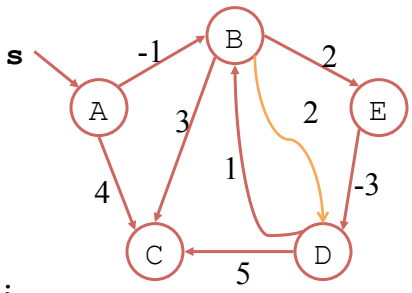
- To summarize:
  - Sort edges: *O(E lg E)*
  - *O(V)* MakeSet()'s
  - *O(E)* FindSet()'s and Union()'s
- Upshot:
  - Best disjoint-set union algorithm makes above
  - 3 operation stake $O((V+E)\cdot\alpha(V))$, α almost constant
  - (slowly growing function of V)
  - Since E >= V-1 then we have $O(E\cdot\alpha(V))$
  - Also since $\alpha(V) = O(\lg V) = O(\lg E)$
  - Overall thus *O(E lg E)*, almost linear w/o sorting

Single-Source Shortest Path

- Optimal substructure
- Key idea: relaxation of edges
- *What does the Bellman-Ford algorithm do?*
  - *What is the running time?*
- *What does Dijkstra's algorithm do?*
  - *What is the running time?*
  - *When does Dijkstra's algorithm not apply?*

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```



Ex: work on board

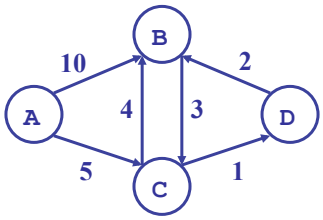
```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Dijkstra's Algorithm

```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

Note: this is really a call to Q->DecreaseKey()

Relaxation Step



Ex: run the algorithm

Dijkstra's Algorithm

```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

How many times is ExtractMin() called?

How many times is DecraseKey() called?

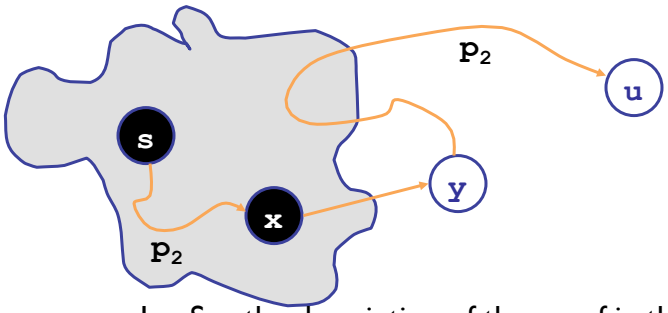
A: O(E lg V) using binary heap for Q
Can acheive O(V lg V + E) with Fibonacci heaps

Dijkstra's Algorithm

```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S U{u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

Correctness: we must show that when u is removed from Q, it has already converged

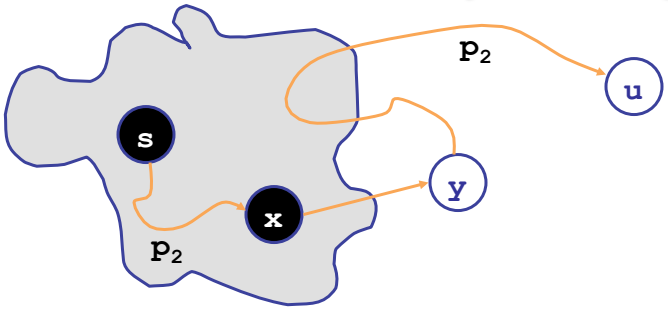
Correctness Of Dijkstra's Algorithm



1. See the description of the proof in the book

Show that Dijkstra's algorithm will terminate with
The cost of each node to be the cost of shortest path.
Idea: show that when the vertex is added to the set the cost of that vertex is the length of the shortest path
Reminder: We always add the vertex with minimal cost

Correctness Of Dijkstra's Algorithm



- Want to show that when vertex is added to set S, d[u] = δ(s,u)
- and throughout note that d[u] ≥ δ(s,u) ∀u
- Proof by contradiction d[u] is not equal to δ(s,u), when added to S
- Before u gets added, some other vertex y on that shortest path needs to be added; claim that d[y] = δ(s,y) when added.
- Know that d[x] = δ(s,x) and δ(s,y) <= δ(s,u) and d[y] = δ(s,y), so d[y] <= d[u]
- But both y and u are outside of S when is chosen so d[u] <= d[y]
- Hence d[y] = d[u] = δ(s,y) = δ(s,y)

Disjoint-Set Union

- We talked about representing sets as linked lists, every element stores pointer to list head
- *What is the cost of merging sets A and B?*
  - A: O(max(|A|, |B|))
- *What is the maximum cost of merging n 1-element sets into a single n-element set?*
  - A: $O(n^2)$
- *How did we improve this? By how much?*
  - A: always copy smaller into larger: $O(n \lg n)$

Amortized Analysis

- Idea: worst-case cost of an operation may overestimate its cost over course of algorithm
- Goal: get a tighter *amortized bound* on its cost

- Aggregate method: total cost of operation over course of algorithm divided by # operations Example: disjoint-set union
- Accounting method: "charge" a cost to each operation, accumulate unused cost in bank, never go negative

Analysis Of Dynamic Tables

- Let c_i = cost of *i*-th insert
- $c_i = i$ if i-1 is exact power of 2, 1 otherwise
- Example:
  - Operation Table Size Cost

Insert(1)	1	1
Insert(2)	2	1 + 1
Insert(3)	4	1 + 2
Insert(4)	4	1
Insert(5)	8	1 + 4
Insert(6)	8	1
Insert(7)	8	1
Insert(8)	8	1
Insert(9)	16	1 + 8

Aggregate Analysis

- *n* Insert() operations cost

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lg n} 2^j = n + (2n - 1) < 3n$$

- At most *n* operations are of cost 1 + costs of expansions
- Expansion happens only where (i-1) is power of 2
- Average cost of operation
  = (total cost)/(# operations) < 3
- Asymptotically, then, a dynamic table costs the same as a fixed-size table
- Both O(1) per *Insert()* operation

Review: The Master Theorem

- Given: a *divide and conquer* algorithm

  An algorithm that divides the problem of size *n* into *a* subproblems, each of size *n*/*b*
- Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function *f*(n)
- Then, the Master Theorem gives us a cookbook for the algorithm's running time:

Review: The Master Theorem

- if T(n) = aT(n/b) + f(n) then

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \quad \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

LCS Via Dynamic Programming

- *Longest common subsequence* (*LCS*) problem:
  - Given two sequences x[1..m] and y[1..n], find the longest subsequence which occurs in both
- Brute-force algorithm: 2^m subsequences of x to check against n elements of y: $O(n\, 2^m)$
- Define c[i,j] = length of LCS of x[1..i], y[1..j]
- Theorem:

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS Example (0)

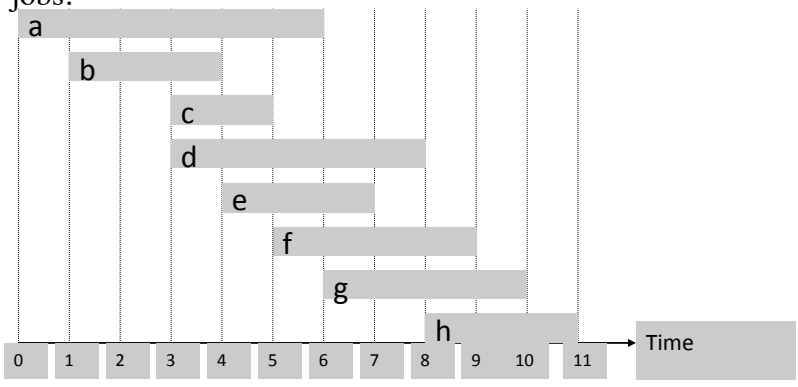
ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi							
1	**A**							
2	**B**							
3	**C**							
4	**B**							

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

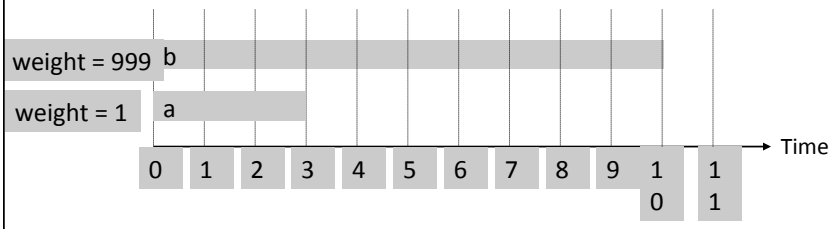
Weighted Interval Scheduling

- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.



Unweighted Interval Scheduling Review

- Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

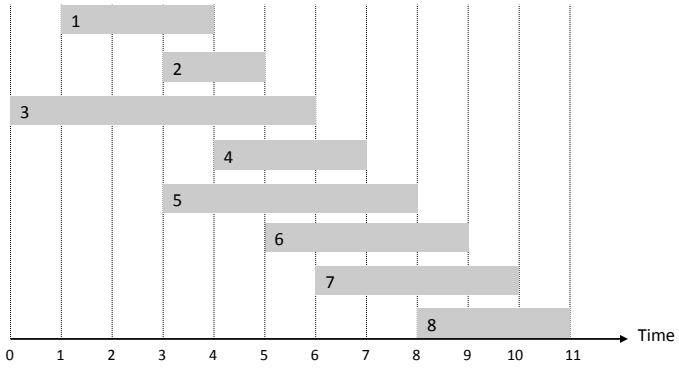


Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. p(j) = largest index i < j such that job i is compatible with j.

Ex: p(8) = 5, p(7) = 3, p(2) = 0.



Dynamic Programming: Binary Choice

• Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 } (optimal substructure)
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)
- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{\, v_j + OPT(p(j)),\ OPT(j-1) \,\} & \text{otherwise} \end{cases}$$

Matrix Chain Multiplication

- Define the cost recursively m[i,j] cost of multiplying

$$A_i \cdots A_j$$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & \text{otherwise} \end{cases}$$

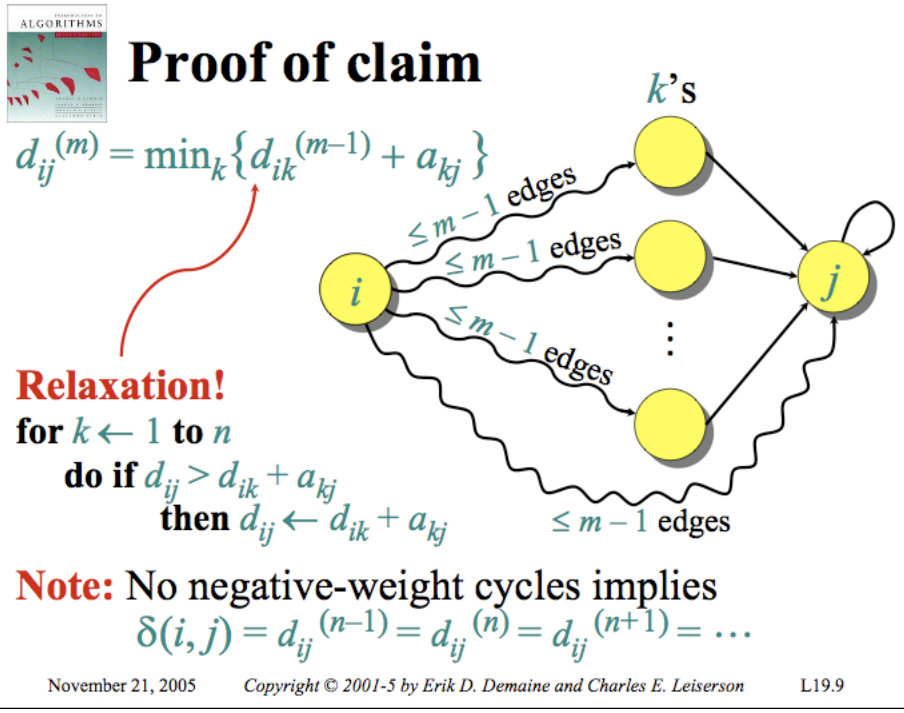
All pairs shortest path

- Final representation of the solution is in adjacency matrix
- δ(i,j) will be the length of the shortest path from *i* to *j*
- Structure of the optimal solution

$$d_{ij}^{0} = \begin{cases} 0 & \text{if } i = j \\ \infty & otherwise \end{cases}$$

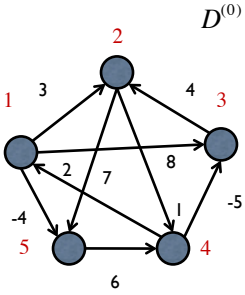
$$d_{ij}^{(m)} = \min(d_{ij}^{(m-1)}, \min_{1 \le k \le n}\{d_{ik}^{(m-1)} + w_{kj}\})$$

- Weight of the shortest path with *m-1* edges and minimum of the weight of any path consisting of at most *m* edges



Example all shortest paths

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$



$D^{(0)} = W$

$D^{(1)} = D^{(0)}W = WW$

$D^{(2)} = D^{(1)}W = D^{(0)}WW$

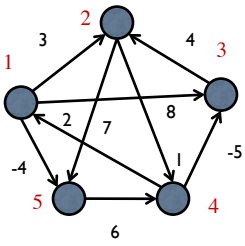
- Matrix multiplication $\Theta(n^4)$
- Repeated Squaring $\Theta(n^3 \lg n)$

$$d_{ij}^{(m)} = \min(d_{ij}^{(m-1)}, \min_{1 \le k \le n}\{d_{ik}^{(m-1)} + w_{kj}\})$$

Like matrix multiplication + => min . => +

Example all shortest paths

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix} \quad D^{(2)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad D^{(3)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$



$D^{(0)} = W$

$D^{(1)} = D^{(0)}W = WW$

$D^{(2)} = D^{(1)}W = D^{(0)}WW$

- Matrix multiplication $\Theta(n^4)$
- Repeated Squaring $\Theta(n^3 \lg n)$

Greedy Algorithms

- Indicators:
  - Optimal substructure
  - *Greedy choice property*: a locally optimal choice leads to a globally optimal solution
- Example problems:
- Activity selection: Set of activities, with start and end times. Maximize compatible set of activities.
- Fractional knapsack: sort items by $/lb, then take items in sorted order MST

Review: Dynamic Programming

- Optimization problems
- What is the structure of the sub-problem
- Common pattern:
- Optimal solution requires making a choice which leads to optimal solution
- Hard part: what is the optimal subproblem structure

How many sub-problems ?

How many choices we have which sub-problem to use ?

- Matrix chain multiplication: 2 subproblems, j-i choices
- LCS: 3 suproblems 3 choices
- Subtleties (graph examples) shortest path, longest path

Review: Greedy Algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
- The hope: a locally optimal choice will lead to a globally optimal solution
- Minimum weight spanning tree, Dijstra's algorithm (greedy)
- Dynamic programming can be overkill; greedy algorithms are easier
- Example: Activity Selection

Greedy Choice Property

- Dynamic programming? Memoize? Yes, but…
- Activity selection problem also exhibits the *greedy choice* property:
- Locally optimal choice ⇒ globally optimal sol'n
- Them 17.1: if S is an activity selection problem sorted by finish time, then ∃ optimal solution $A \subseteq S$ such that $\{1\} \in A$
- Sketch of proof: if ∃ optimal solution B that does not contain {1}, can always replace first activity in B with {1} (*Why?*). Same number of activities, thus optimal.

Review: Activity-Selection Problem

- The *activity selection problem*: get your money's worth out of a carnival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, starting and ending at different times
  - Your goal: ride as many rides as possible
- Naïve first-year CS major strategy:
  - Ride the first ride, when get off, get on the very next ride possible, repeat until carnival ends
- *What is the sophisticated third-year strategy?*

Review: Activity-Selection

- Formally:
  - Given a set S of n activities
    - s_i = start time of activity f_i = finish time of activity i
  - Find max-size subset A of compatible activities
  - Assume activities sorted by finish time
- *What is optimal substructure for this problem?*

Review: Activity-Selection

- Formally:
  - Given a set S of n activities
    - s_i = start time of activity i f_i = finish time of activity i
  - Find max-size subset A of compatible activities
  - Assume activities sorted by finish time
- *What is optimal substructure for this problem?*
  - A: If k is the activity in A with the earliest finish time, then $A - \{k\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_k\}$

Huffman coding

- Design of optimal codes
- Example (on the board)
- Idea how to design optimal code ?
- Notion of prefix code
- Greedy Algorithm for constructing optimal codes

Algorithm:
1. Keep the frequencies in Priority Queue (build heap)
2. Take two minimal elements (extract min)
3. Insert their sum to queue
4. Until queue is empty

Running time $O(n \lg n)$

Huffman coding

- What is the optimal substructure and greedy choice property ?
- Given alphabet C each character has frequency f[c]
- Suppose x and y are characters with lowest frequencies
- Then there exist an optimal code where x and y have same length and differ only in last bit.

- Optimal substructure property
- Given C and C' with the x and y removed and new symbol
- Added where f[z] = f[x]+f[y]. If we have a tree T' which represents optimal code for C' then replacing node z with two children x and y will yield optimal code for C

The Knapsack Problem

- The famous *knapsack problem*:
  - A thief breaks into a museum. Fabulous paintings,
  - sculptures, and jewels are everywhere. The thief has a good
  - eye for the value of these objects, and knows that each will
  - fetch hundreds or thousands of dollars on the clandestine art
  - collector's market. But, the thief has only brought a single
  - knapsack to the scene of the robbery, and can take away
  - only what he can carry. What items should the thief take to
  - maximize the haul?

The Knapsack Problem

- More formally, the *0-1 knapsack problem*:

- The thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
- Carrying at most W pounds, maximize value
- Note: assume v_i, w_i, and W are all integers
  - "0-1" b/c each item must be taken or left in entirety
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional
  - problem as buckets of gold dust

The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most W pounds

- *If we remove item j from the load, what do we know about the remaining load?*

- A: remainder must be the most valuable load weighing at most $W - w_j$ that thief could take from museum, excluding item j

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy

- Greedy strategy: take in order of dollars/pound
  - Example: 3 items weighing 10, 20, and 30 pounds, knapsack
  - can hold 50 pounds

    - *Suppose item 2 is worth $100. Assign values to the other items so that the greedy strategy will fail*

The Knapsack Problem: Greedy Vs. Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
- As you have seen, however, it can be solved with dynamic programming

0-1 Knapsack problem: a picture

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.
- Just another version of this problem is the "*Fractional Knapsack Problem*", where we can take fractions of items.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$

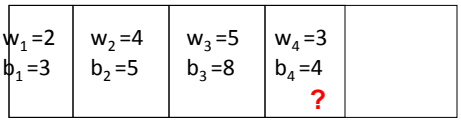
- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Defining a Subproblem

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we can't do that. Explanation follows….

Defining a Subproblem

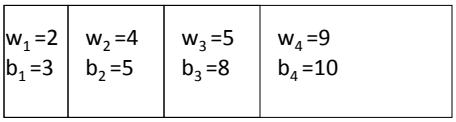


Max weight: W = 20

For S_4:

Total weight: 14;

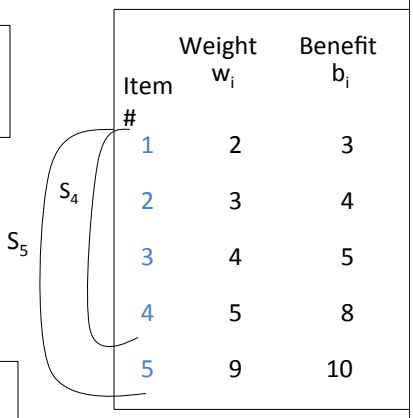
total benefit: 20



For S_5:

Total weight: 20

total benefit: 26



Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem (continued)

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: w, which will represent the exact weight for each subset of items
- The subproblem then will be to compute $B[k,w]$

Recursive Formula for subproblems

- Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- It means, that the best subset of S_k that has total weight w is one of the two:

1) the best subset of S_{k-1} that has total weight w, **or**
2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable
- Second case: $w_k <= w$. Then the item k can be in the solution, and we choose the case with greater value

0-1 Knapsack Algorithm

```
for w = 0 to W
  B[0,w] = 0
for i = 0 to n
  B[i,0] = 0
  for w = 0 to W
   if w_i <= w // item i can be part of the solution
      if b_i + B[i-1,w-w_i] > B[i-1,w]
         B[i,w] = b_i + B[i-1,w- w_i]
      else
         B[i,w] = B[i-1,w]
   else B[i,w] = B[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W
  B[0,w] = 0
for i = 0 to n
  B[i,0] = 0
  for w = 0 to W
   < the rest of the code >
```

$O(W)$

Repeat n times

$O(W)$

What is the running time of this algorithm?

$O(nW)$

Remember that the brute-force algorithm takes $O(2^n)$

68

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary
- Please see LCS algorithm from the previous lecture for the example how to extract this data from the table we built

Conclusion

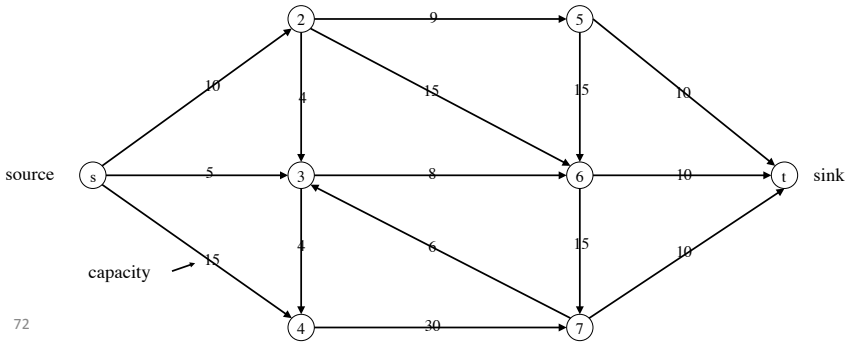
- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary
- Running time (Dynamic Programming algorithm vs. naïve algorithm):
  - LCS: **O(mn)** vs. **O(n 2^m)**
  - 0-1 Knapsack problem: **O(Wn)** vs. **O(2^n)**

- **Caveat 0-1 Knapsack is NP- complete**
- W is not polynomial in the size of the input, length of W is proportional to number of bits needed to represent that word

Minimum Cut Problem

•Flow network.

- Abstraction for material flowing through the edges.
- G = (V, E) = directed graph, no parallel edges.
- Two distinguished nodes: s = source, t = sink.
- c(e) = capacity of edge e.



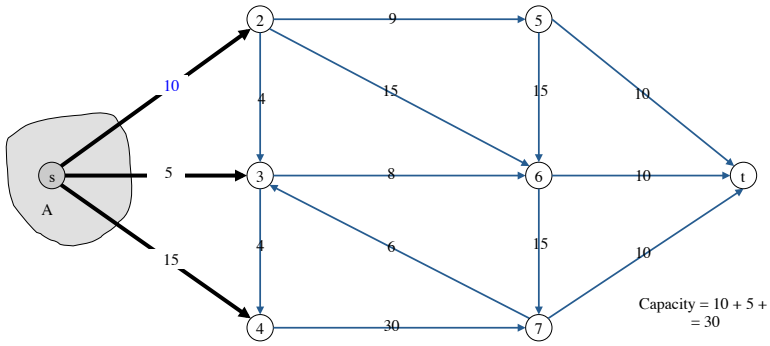
72

Cuts

- Def. An s-t cut is a partition (A, B) of V with $s \in A$ and $t \in B$.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$

- Def. The capacity of a cut (A, B) is:



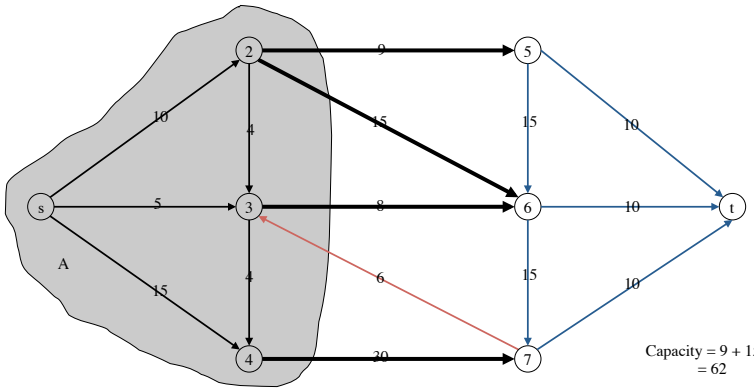
73

Cuts

- Def. An s-t cut is a partition (A, B) of V with $s \in A$ and $t \in B$.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$

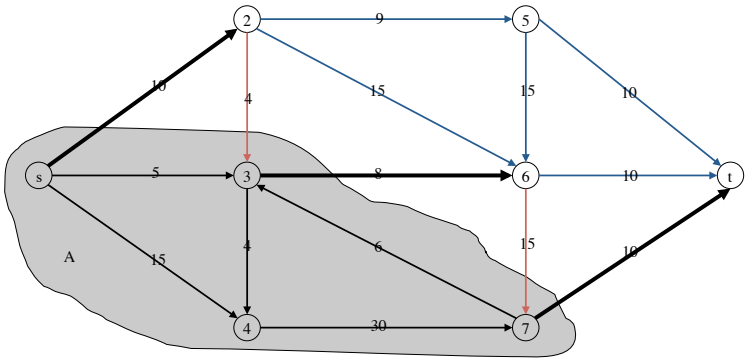
- Def. The capacity of a cut (A, B) is:



74

Minimum Cut Problem

•Min s-t cut problem. Find an s-t cut of minimum capacity.



75

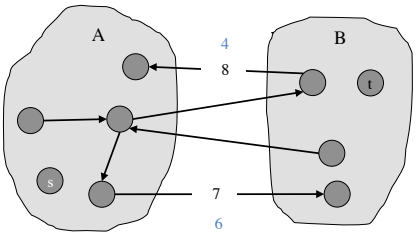
Capacity = 10 + 8 + 10
= 28

Flows and Cuts

•Weak duality. Let f be any flow. Then, for any s-t cut (A, B) we have
v(f) ≤ cap(A, B).

•Pf.

$$
\begin{aligned}
v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
&\le \sum_{e \text{ out of } A} f(e) \\
&\le \sum_{e \text{ out of } A} c(e) \\
&= \text{cap}(A, B)
\end{aligned}
$$

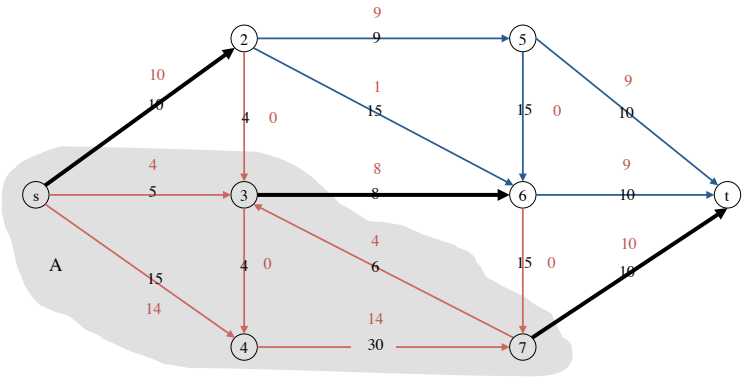


76

Certificate of Optimality

•Corollary. Let f be any flow, and let (A, B) be any cut.
If v(f) = cap(A, B), then f is a max flow and (A, B) is a min cut.

Value of flow = 28
Cut capacity = 28 ⇒ Flow value ≤ 28

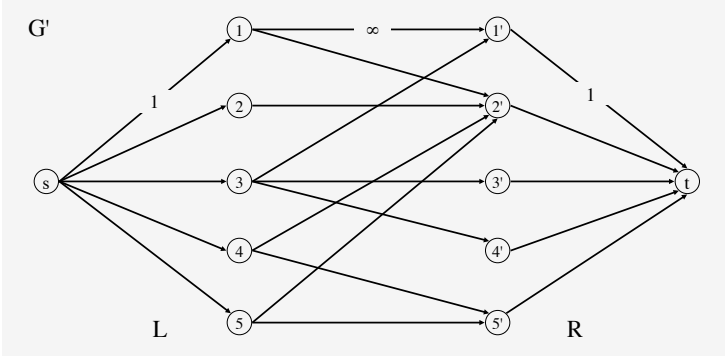


77

Bipartite Matching

•Max flow formulation.

- Create digraph G' = (L ∪ R ∪ {s, t}, E').
- Direct all edges from L to R, and assign infinite (or unit) capacity.
- Add source s, and unit capacity edges from s to each node in L.
- Add sink t, and unit capacity edges from each node in R to t.

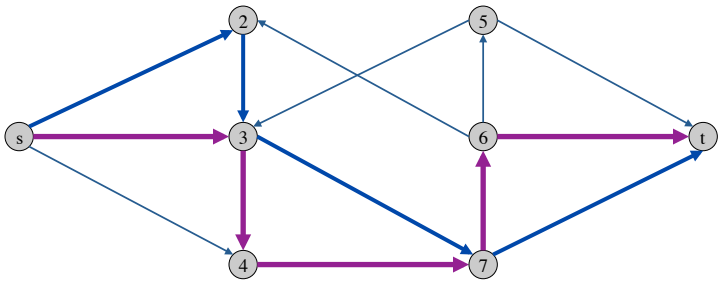


Edge Disjoint Paths

•Disjoint path problem. Given a digraph G = (V, E) and two nodes s and t, find the max number of edge-disjoint s-t paths.

•Def. Two paths are edge-disjoint if they have no edge in common.

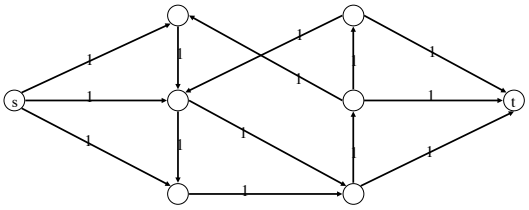
•Ex: communication networks.



79

Edge Disjoint Paths

•Max flow formulation: assign unit capacity to every edge.



•Theorem. Max number edge-disjoint s-t paths equals max flow value.

•Pf. ≤

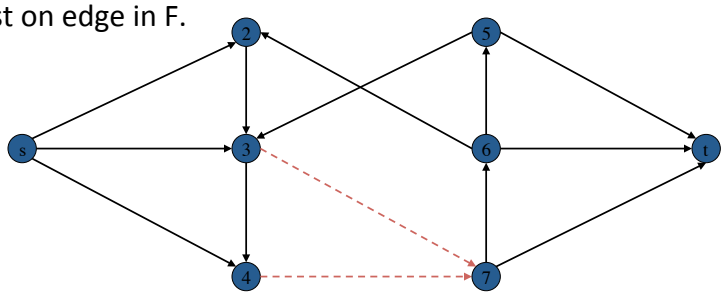
- Suppose there are k edge-disjoint paths $P_1, \ldots, P_k$.
- Set f(e) = 1 if e participates in some path P_i; else set f(e) = 0.
- Since paths are edge-disjoint, f is a flow of value k. ▪

80

Network Connectivity

•Network connectivity. Given a digraph G = (V, E) and two nodes s and t, find min number of edges whose removal disconnects t from s.

•Def. A set of edges F ⊆ E disconnects t from s if all s-t paths uses at least on edge in F.



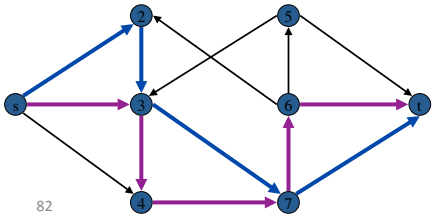
81

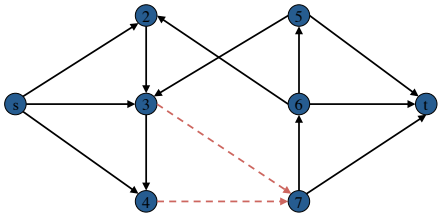
Edge Disjoint Paths and Network Connectivity

•Theorem. [Menger 1927] The max number of edge-disjoint s-t paths is equal to the min number of edges whose removal disconnects t from s.

•Pf. ≤

- Suppose the removal of F ⊆ E disconnects t from s, and | F| = k.
- All s-t paths use at least one edge of F. Hence, the number of edge-disjoint paths is at most k. ▪





82

Review: **P** and **NP**

- *What do we mean when we say a problem is in **P**?*
- *What do we mean when we say a problem is in **NP**?*
- *What is the relation between **P** and **NP**?*

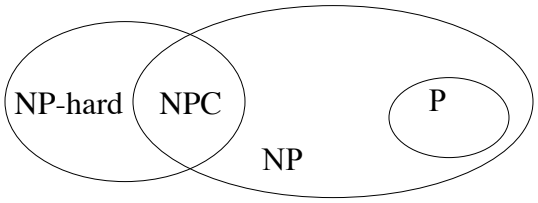
Review: **P** and **NP**

- *What do we mean when we say a problem is in **P**?*
  - A: A solution can be found in polynomial time
- *What do we mean when we say a problem is in **NP**?*
  - A: A solution can be verified in polynomial time
- *What is the relation between **P** and **NP**?*
  - A: **P** $\subseteq$ **NP**, but no one knows whether **P = NP**

Review: NP-Complete

- *What, intuitively, does it mean if we can reduce problem P to problem Q?*
  - P is "no harder than" Q
- *How do we reduce P to Q?*
  - Transform instances of P to instances of Q in polynomial time s.t. Q: "yes" iff P: "yes"
- *What does it mean if Q is NP-Hard ?*
  - Every problem P∈**NP** $\leq_p$ Q
- *What does it mean if Q is NP-Complete?*
  - Q is NP-Hard and Q ∈ **NP**

NP-Hard and NP-Complete



NP complete problems – problems which are in NP but are **as hard as** any other problem in NP

• **NP hard** which are not NP-complete, e.g. halting problem (is still possible to reduce any problem in NP to halting problem)

• Some other non-decision problems

Review: Proving Problems NP-Complete

- *What was the first problem shown to be NP-Complete?*
- A: Circuit satisfiability (*SAT*), by Cook
- *How do we usually prove that a problem R is NP-Complete?*
- A: Show R ∈**NP**, and reduce a known NP-Complete problem Q to R

Review: Reductions

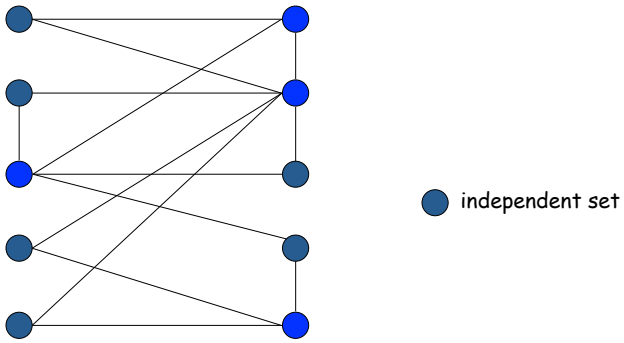
- Review the reductions we've covered:
  - Independent set <-> vertex cover
  - Directed hamiltonian cycle → undirected hamiltonian cycle
  - Undirected hamiltonian cycle → traveling salesman problem
  - 3-CNF → *k*-clique
  - *k*-clique → vertex cover

Independent Set

•INDEPENDENT SET: Given a graph G = (V, E) and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

•Ex. Is there an independent set of size ≥ 6? Yes.

•Ex. Is there an independent set of size ≥ 7? No.

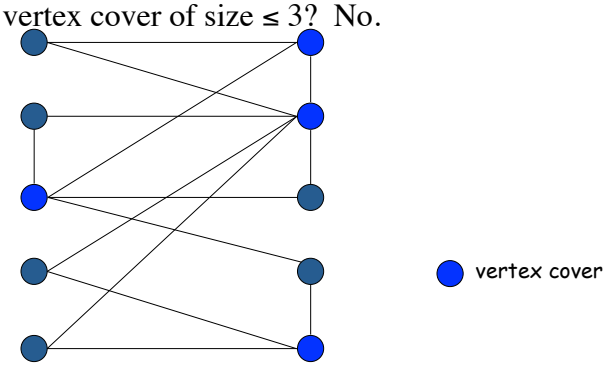


Vertex Cover

•VERTEX COVER: Given a graph G = (V, E) and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge, at least one of its endpoints is in S?

•Ex. Is there a vertex cover of size ≤ 4? Yes.

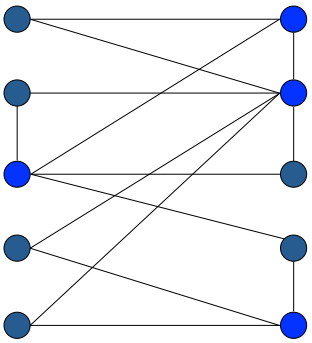
•Ex. Is there a vertex cover of size ≤ 3? No.

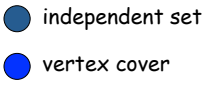


Vertex Cover and Independent Set

•Claim. VERTEX-COVER $\equiv_P$ INDEPENDENT-SET.

•Pf. We show S is an independent set iff V - S is a vertex cover.





Vertex Cover Reduces to Set Cover

•Claim. VERTEX-COVER $\leq_P$ SET-COVER.

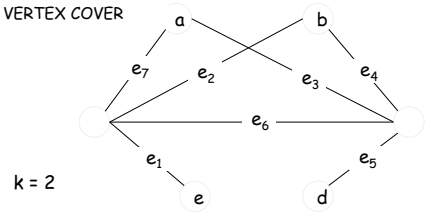
•Pf. Given a VERTEX-COVER instance G = (V, E), k, we construct a set cover instance whose size equals the size of the vertex cover instance.

•Construction.

– Create SET-COVER instance:

• k = k, U = E, $S_v = \{e \in E : e \text{ incident to } v\}$

– Set-cover of size ≤ k iff vertex cover of size ≤ k. ▪



SET COVER

U = { 1, 2, 3, 4, 5, 6, 7 }
k = 2
S_a = {3, 7}
S_b = {2, 4}
S_c = {3, 4, 5, 6}
S_d = {5}
S_e = {1}
S_f = {1, 2, 6, 7}

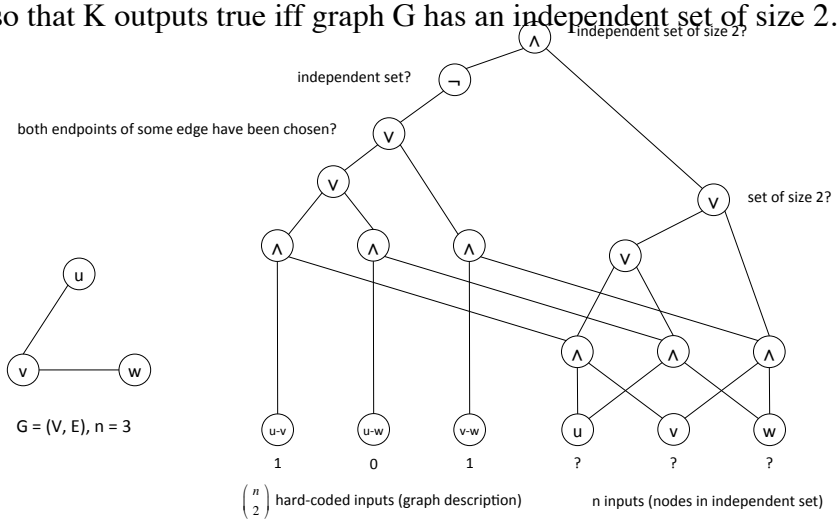
Review: Circuit Satisfiablity

- **Circuit SAT** is NP can be verified in polynomial time
  i.e. given a circuit and an input we can verify in polynomial time whether the input is a satisfying assignment.
- **Circuit SAT is NP-hard** Every problem in NP is reducible to circuit SAT; Proof:

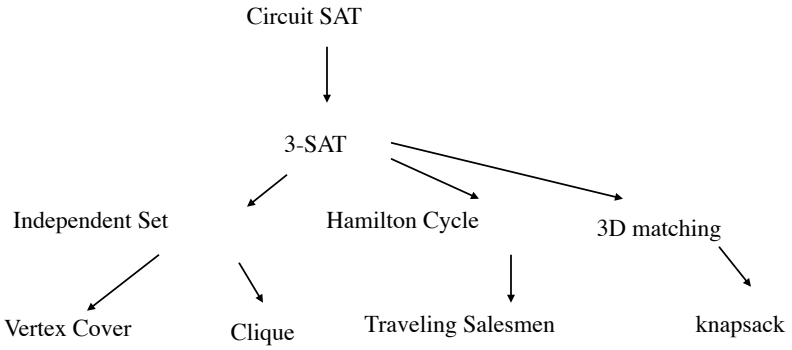
1. Problem is in NP; can be verified in polynomial time by some algorithm
2. Each step of the algorithm runs on a computer (huge boolean circuit)
3. Chaining together all circuits which correspond to the steps of the algorithm – we get large circuit which describes the run of the algorithm
4. If we plug in the input of a problem A then YES / NO answer when circuit is/is not satisfiable

Example

•Ex. Construction below creates a circuit K whose inputs can be set so that K outputs true iff graph G has an independent set of size 2.



Relationships between known NP – complete problems



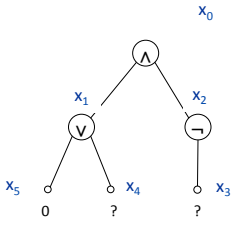
Review: Formula Satisfiability

- Show that it is easy to verify the solution
- Reduce circuit satisfiability to SAT
- Any instance of circuit satisfiability can be reduced to formula satisfiability
- Strategy: express every gate as a formula (Example).

3-SAT is NP-Complete

•Theorem. 3-SAT is NP-complete.
•Pf. Suffices to show that CIRCUIT-SAT $\leq_P$ 3-SAT since 3-SAT is in NP. Let K be any circuit.

- Create a 3-SAT variable x_i for each circuit element i.
- Make circuit compute correct values at each node:
  - $x_2 = \neg\, x_3 \Rightarrow$ add 2 clauses: $x_2 \vee x_3$, $\overline{x_2} \vee \overline{x_3}$
  - $x_1 = x_4 \vee x_5 \Rightarrow$ add 3 clauses: $x_1 \vee \overline{x_4}$, $x_1 \vee \overline{x_5}$, $\overline{x_1} \vee x_4 \vee x_5$
  - $x_0 = x_1 \wedge x_2 \Rightarrow$ add 3 clauses: $\overline{x_0} \vee x_1$, $\overline{x_0} \vee x_2$, $x_0 \vee \overline{x_1} \vee \overline{x_2}$
- Hard-coded input values and output value.
  - $x_5 = 0 \Rightarrow$ add 1 clause: $\overline{x_5}$
  - $x_0 = 1 \Rightarrow$ add 1 clause: x_0
- Final step: turn clauses of length < 3 into clauses of length exactly 3. ▪



Review: Conjunctive Normal Form

- Even if the form of the Boolean expression is simplified, the problem may be NP-Complete
  - *Literal*: an occurrence of a Boolean or its negation
  - A Boolean formula is in *conjunctive normal form*, or *CNF*, if it is an AND of clauses, each of which is an OR of literals
    - Ex: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5)$
  - *3-CNF*: each clause has exactly 3 distinct literals
    - Ex: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5 \vee x_3 \vee x_4)$
    - Notice: true if at least one literal in each clause is true

Review: The 3-CNF Problem

- Thm 36.10: Satisfiability of Boolean formulas in 3-CNF form (the *3-CNF Problem*) is NP-Complete
- The reason we care about the 3-CNF problem is that it is relatively easy to reduce to others
- Thus by proving 3-CNF NP-Complete we can prove many seemingly unrelated problems NP-Complete

Review: 3-CNF Satisfiability

- Show that it is easy to verify the solution
- Reduce Satisfiability to 3-CNF
- Strategy: Get Binary Parse Tree, introduce new variables, get clauses
- Convert Clauses to CNF form using De Morgan's Laws

3-CNF → Clique

- *What is a clique of a graph G?*
- A: a subset of vertices fully connected to each other, i.e. a complete subgraph of G
- The *clique problem*: how large is the maximum-size clique in a graph?
- *Can we turn this into a decision problem?*
- A: Yes, we call this the *k-clique problem*
- Is there a clique of size k in the graph G ?
- *Is the k-clique problem within **NP**?*
- Naïve approach ? Check all possible subsets of k vertices

Directed Hamiltonian Cycle ⇒ Undirected Hamiltonian Cycle

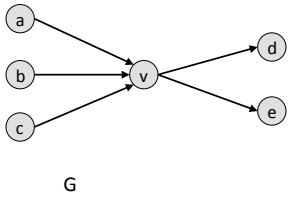
- *What was the hamiltonian cycle problem again?*
- For my next trick, I will reduce the *directed hamiltonian cycle* problem to the *undirected hamiltonian cycle* problem before your eyes
  - *Which variant am I proving NP-Complete?*
- Draw a directed example on the board
  - *What transformation do I need to effect?*

Directed Hamiltonian Cycle

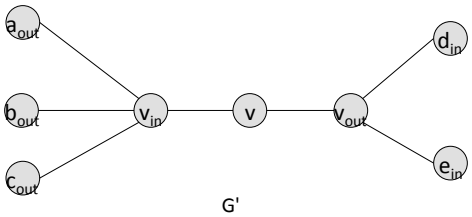
•DIR-HAM-CYCLE: given a digraph G = (V, E), does there exists a simple directed cycle Γ that contains every node in V

•Claim. DIR-HAM-CYCLE ≤ $_P$ HAM-CYCLE.

•Pf. Given a directed graph G = (V, E), construct an undirected graph G' with 3n nodes.



G

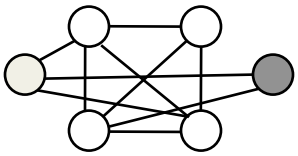


G'

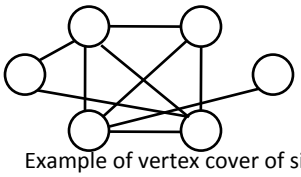
103

Clique → Vertex Cover

- A *vertex cover* for a graph G is a set of vertices incident to every edge in G
- The *vertex cover problem*: what is the minimum size vertex cover in G?
- Restated as a decision problem: does a vertex cover of size *k* exist in G?
- Thm 36.12: vertex cover is NP-Complete



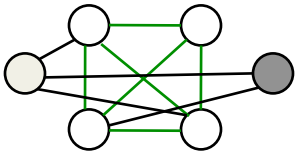
Example of vertex cover of size 2



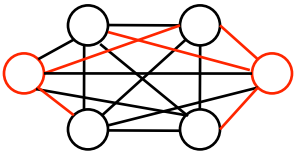
Example of vertex cover of size 2

Clique → Vertex Cover

- First, show vertex cover in **NP** (*How?*)
- How to decide whether graph G has a vertex cover of size k
- Reduce k-clique to vertex cover
- The *complement* G_C of a graph G contains exactly those edges not in G
- Compute G_C in polynomial time
- G has a clique of size k iff G_C has a vertex cover of size $|V| - k$



Clique of size 4



Vertex cover of size 2

Relationships between known NP – complete problems