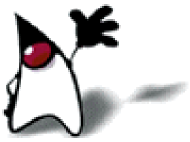
Greedy Algorithms



A short list of categories

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

Optimization problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

3

Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be: At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make $6.39, you can choose:
    - a $5 bill
    - a $1 bill, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

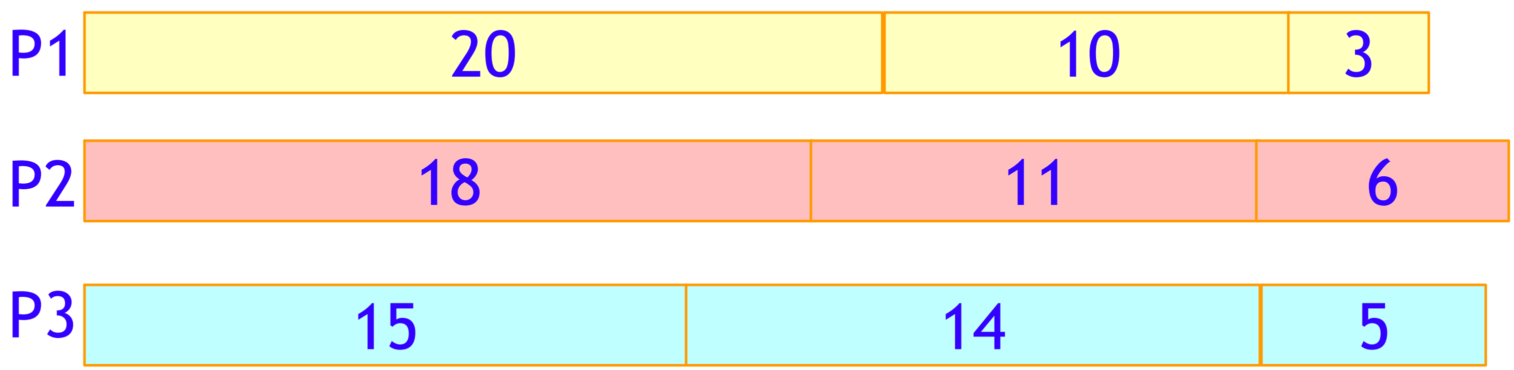
4

A failure of the greedy algorithm

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

A scheduling problem

- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- You have three processors on which you can run these jobs
- You decide to do the longest-running jobs first, on whatever processor is available

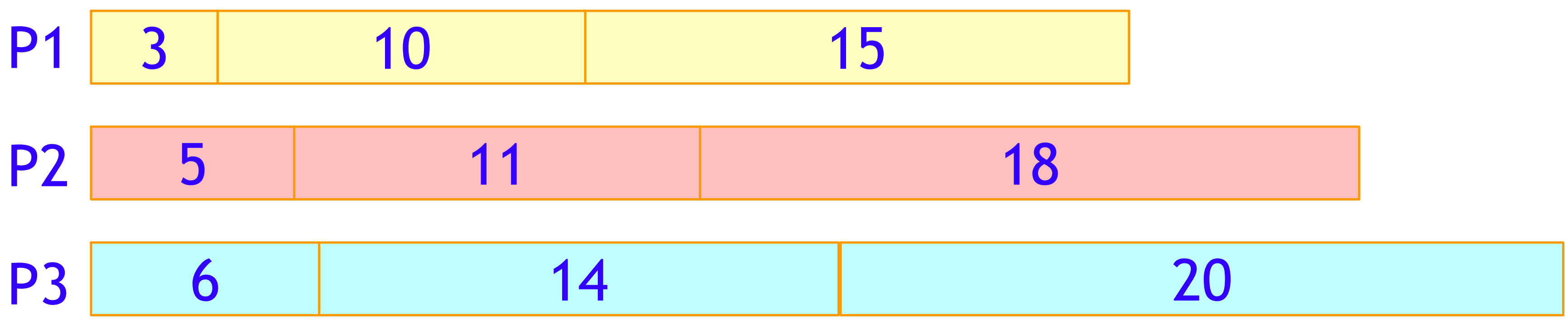


- Time to completion: 18 + 11 + 6 = 35 minutes
- This solution isn't bad, but we might be able to do better

6

Another approach

- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes

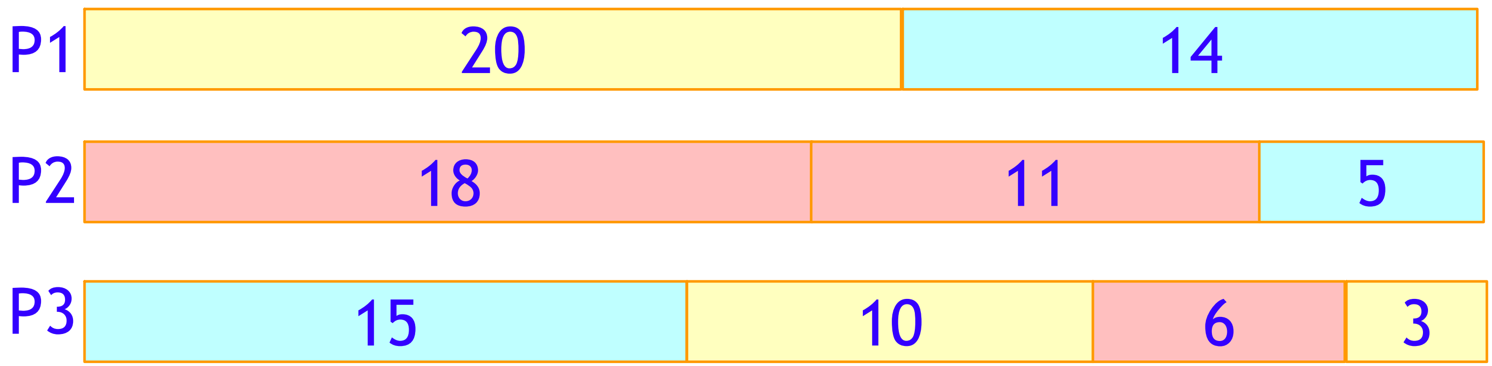


- That wasn't such a good idea; time to completion is now 6 + 14 + 20 = 40 minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum

7

An optimum solution

- Better solutions do exist:

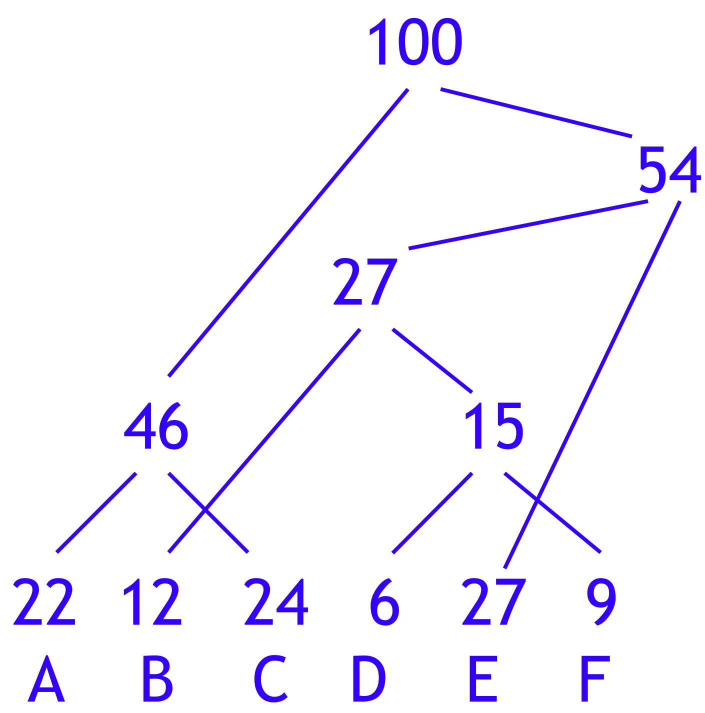


- This solution is clearly optimal (why?)
- Clearly, there are other optimal solutions (why?)
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time

8

Huffman encoding

- The Huffman encoding algorithm is a greedy algorithm
- You always pick the two smallest numbers to combine

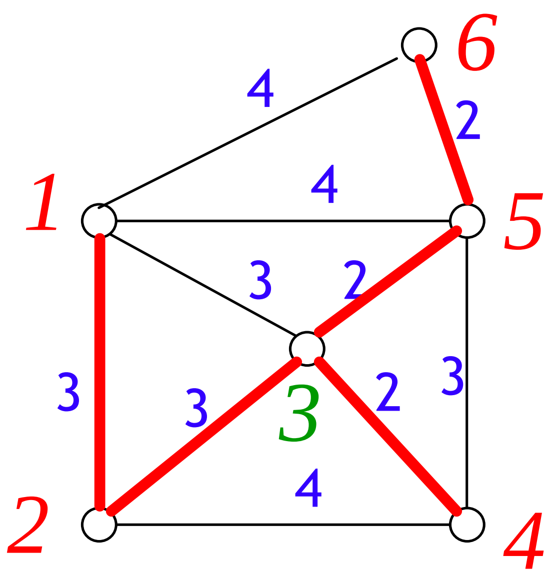


A=00
B=100
C=01
D=1010
E=11
F=1011

- Average bits/char:
  0.22*2 + 0.12*3 +
  0.24*2 + 0.06*4 +
  0.27*2 + 0.09*4
  = 2.42
- The Huffman algorithm finds an optimal solution

Minimum spanning tree

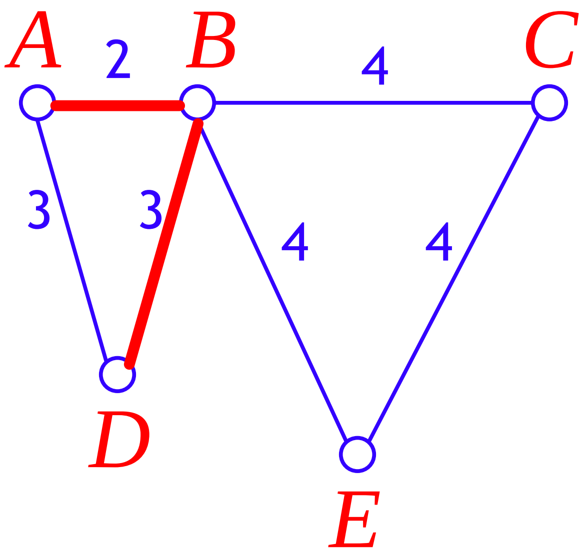
- A minimum spanning tree is a least-cost subset of the edges of a graph that connects all the nodes
  - Start by picking any node and adding it to the tree
  - Repeatedly: Pick any *least-cost* edge from a node in the tree to a node not in the tree, and add the edge and new node to the tree
  - Stop when all nodes have been added to the tree



- The result is a least-cost (3+3+2+2+2=12) spanning tree
- If you think some other edge should be in the spanning tree:
  - Try adding that edge
  - Note that the edge is part of a cycle
  - To break the cycle, you must remove the edge with the greatest cost
    - This will be the edge you just added

Traveling salesman

- A salesman must visit every city (starting from city ***A***), and wants to cover the least possible distance
  - He can revisit a city (and reuse a road) if necessary
- He does this by using a greedy algorithm: He goes to the next nearest city from wherever he is



- From ***A*** he goes to ***B***
- From ***B*** he goes to ***D***
- This is *not* going to result in a shortest path!
- The best result he can get now will be ***ABDBCE***, at a cost of 16
- An actual least-cost path from ***A*** is ***ADBCE***, at a cost of 14

Analysis

- A greedy algorithm typically makes (approximately) n choices for a problem of size n
  - (The first or last choice may be forced)
- Hence the expected running time is: O(n * O(choice(n))), where choice(n) is making a choice among n objects
  - Counting: Must find largest useable coin from among k sizes of coin (k is a constant), an O(k)=O(1) operation;
    - Therefore, coin counting is (n)
  - Huffman: Must sort n values before making n choices
    - Therefore, Huffman is O(n log n) + O(n) = O(n log n)
  - Minimum spanning tree: At each new node, must include new edges and keep them sorted, which is O(n log n) overall
    - Therefore, MST is O(n log n) + O(n) = O(n log n)

12

Other greedy algorithms

- Dijkstra's algorithm for finding the shortest path in a graph
  - Always takes the *shortest* edge connecting a known node to an unknown node
- Kruskal's algorithm for finding a minimum-cost spanning tree
  - Always tries the *lowest-cost* remaining edge
- Prim's algorithm for finding a minimum-cost spanning tree
  - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree

13

Dijkstra’s shortest-path algorithm

- Dijkstra’s algorithm finds the shortest paths from a given node to all other nodes in a graph
  - Initially,
    - Mark the given node as *known* (path length is zero)
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
  - Repeatedly (until all nodes are known),
    - Find an unknown node containing the smallest distance
    - Mark the new node as known
    - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
      - If so, also reset the predecessor of the new node

14

Analysis of Dijkstra's algorithm I

- Assume that the *average* out-degree of a node is some constant k
  - Initially,
    - Mark the given node as *known* (path length is zero)
      - This takes $O(1)$ (constant) time
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
      - If each node refers to a list of k adjacent node/edge pairs, this takes $O(k) = O(1)$ time, that is, constant time
      - Notice that this operation takes *longer* if we have to extract a list of names from a hash table

15

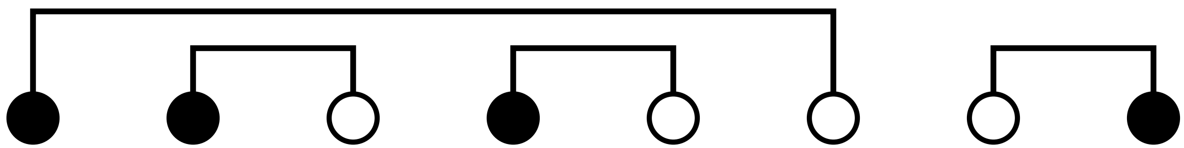
Analysis of Dijkstra's algorithm II

- Repeatedly (until all nodes are known), (n times)
  - Find an unknown node containing the smallest distance
    - Probably the best way to do this is to put the unknown nodes into a priority queue; this takes k * $O(\log n)$ time *each* time a new node is marked "known" (and this happens n times)
  - Mark the new node as known -- $O(1)$ time
  - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
    - If so, also reset the predecessor of the new node
    - There are k adjacent nodes (on average), operation requires constant time at each, therefore $O(k)$ (constant) time
  - Combining all the parts, we get:
    $O(1) + n*(k*O(\log n)+O(k))$, that is, $\underline{O(nk \log n)}$ time

16

Connecting wires

- There are n white dots and n black dots, equally spaced, in a line
- You want to connect each white dot with some one black dot, with a minimum total length of “wire”
- Example:

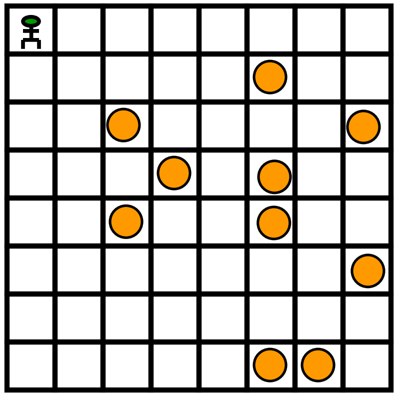


- Total wire length above is 1 + 1 + 1 + 5 = 8
- Do you see a greedy algorithm for doing this?
- Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?

17

Collecting coins

- A checkerboard has a certain number of coins on it
- A robot starts in the upper-left corner, and walks to the bottom left-hand corner
  - The robot can only move in two directions: right and down
  - The robot collects coins as it goes
- You want to collect *all* the coins using the *minimum* number of robots
- Example:



- Do you see a greedy algorithm for doing this?
- Does the algorithm guarantee an optimal solution?
  - Can you prove it?
  - Can you find a counterexample?

18

The End

19