# BLG 336E – Analysis of Algorithms II

## Practice Session 3

Hasan Kivrak

Istanbul Technical University – Department of Computer Engineering

22.03.2016

## Outline

What are the most important steps in designing a greedy algorithm? Give an example of a greedy algorithm.

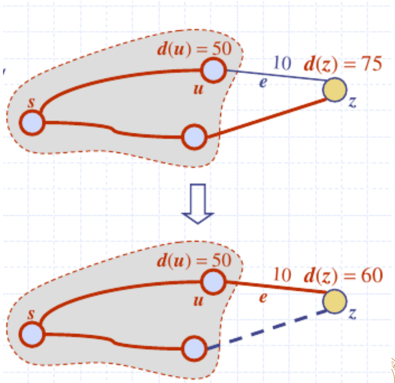
- finding a local (myopic) decision rule that improves a solution and makes it closer to the optimal solution
- showing that by following these local decision rules one reaches an optimal solution at the end.

An example greedy algorithm is **Dijkstra's algorithm** for finding the shortest path in a graph.

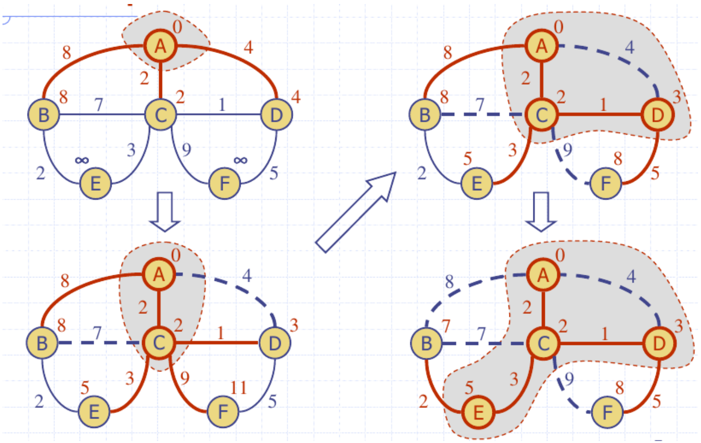
- The distance of a vertex *v* from a vertex *s* is the length of a shortest path between *s* and *v*
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex *s*
- Assumptions:
  - the graph is connected
  - the edge weights are non-negative
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex *v* a label *d(v)* representing the distance of *v* from *s* in the subgraph consisting of the cloud and its adjacent vertices
- At each step:
  - We add to the cloud the vertex *u* outside the cloud with the smallest distance label, *d(u)*
  - We update the labels of the vertices adjacent to *u*

## Edge Relaxation

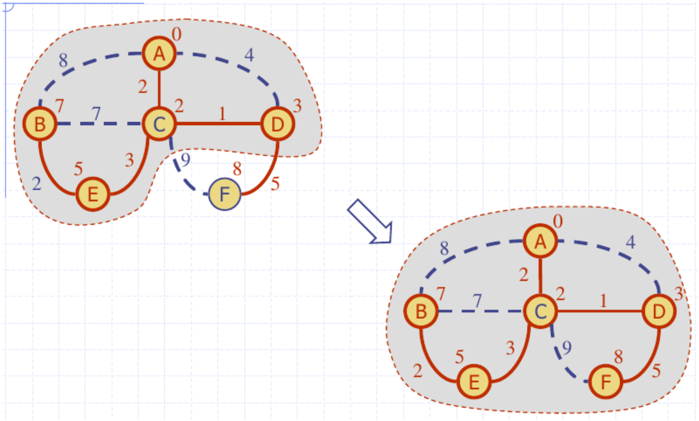
- Consider an edge $e = (u, z)$ such that
  - $u$ is the vertex most recently added to the cloud
  - $z$ is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:
  $d(z) \leftarrow min\{d(z), d(u) + weight(e)\}$



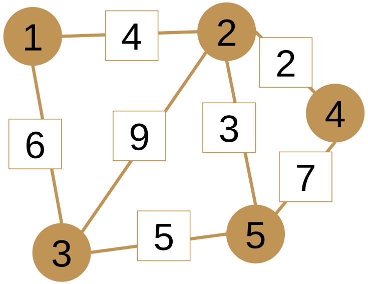
## Example :



## Example (cont.)



## Example - 2



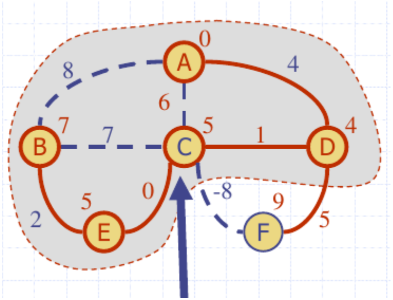
- $d(4) : \{\infty, \infty, \infty, 0, \infty\}, s : \{4\}$
- $d(4) : \{\infty, \mathbf{2}, \infty, 0, 7\}, s : \{4, 2\}$
- $d(4) : \{6, 2, 11, 0, \mathbf{5}\}, s : \{4, 2, 5\}$
- $d(4) : \{\mathbf{6}, 2, 10, 0, 5\}, s : \{4, 2, 5, 1\}$
- $d(4) : \{6, 2, \mathbf{10}, 0, 5\}, s : \{4, 2, 5, 1, 3\}$

## More Examples

- www.cs.auckland.ac.nz/software/AlgAnim/dij-op.html
- www.cs.auckland.ac.nz/software/AlgAnim/dijkstra.html

## Why It Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with $d(C) = 5$!

## Graphs with Negative-Weight Edges

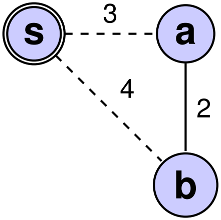
- Use Bellman-Ford algorithm

What is the difference between the Kruskal's and Prim's MST algorithms?

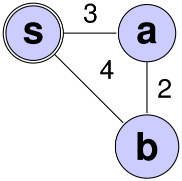
- Kruskal's algorithm. Start with $T = \textit{Empty}$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.
- Prim's algorithm. Start with some root node s and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to T that has exactly one endpoint in $T$.

- Let *G* is a graph with undirected edges and s is particular node. Prove or disprove (by giving a counter example) the following:
- *The minimum spanning tree obtanined using Prim's algorithm and startign from node s of the graph G, forms the shortest paths from node s to all the other nodes.*

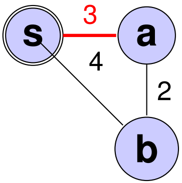
FALSE Counter Example

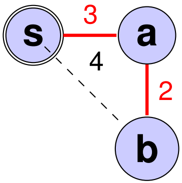


Prim's MST Alg. S:Explored nodes, Q:Priority queue of unexplored nodes.



$S = \{s\}$, $Q = \{(a, \mathbf{3}), (b, 4)\}$





$S = \{s, a\}$, $Q = \{(b, \mathbf{4})\}$
$S = \{s, a, b\}$, $Q = \{\}$

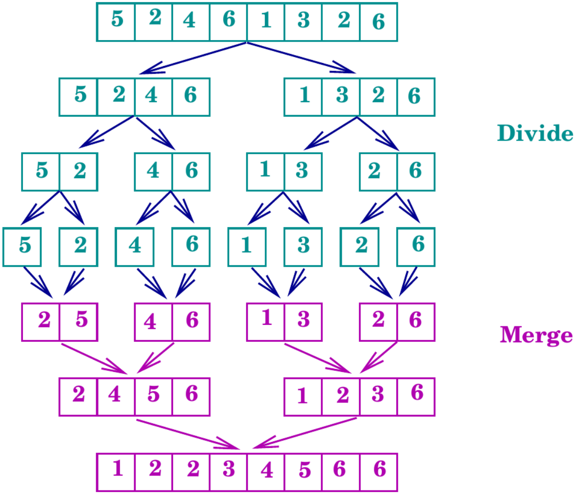
- Greedy algorithms are usually very natural.
- Divide and Conquer is a different framework. Related to induction:
  - Suppose you have a "box" that can solve problems of size $\leq k < n$
  - You use this box on some subset of the input items to get partial answers
  - You combine these partial answers to get the full answer.

But: you construct the "box" by recursively applying the same idea until the problem is small enough to be solved by brute force.

```
function mergesort($a[1 \dots n]$)
Input:  An array of numbers $a[1 \dots n]$
Output:  A sorted version of this array

if $n > 1$:
  return merge(mergesort($a[1 \dots \lfloor n/2 \rfloor]$), mergesort($a[\lfloor n/2 \rfloor + 1 \dots n]$))
else:
  return $a$
```

- In practice, you sort in-place rather than making new lists.
- Total time: $T(n) \leq 2T(n/2) + c(n)$.


5 2 4 6 1 3 2 6
5 2 4 6
1 3 2 6
5 2
4 6
1 3
2 6
5
2
4
6
1
3
2
6
Divide
2 5
4 6
1 3
2 6
Merge
2 4 5 6
1 2 3 6
1 2 2 3 4 5 6 6

How do you compute the time complexity of a divide and conquer algorithm for a certain input size?

$$T(n) = a * T(n/b) + c(n) \tag{1}$$

We need to consider the

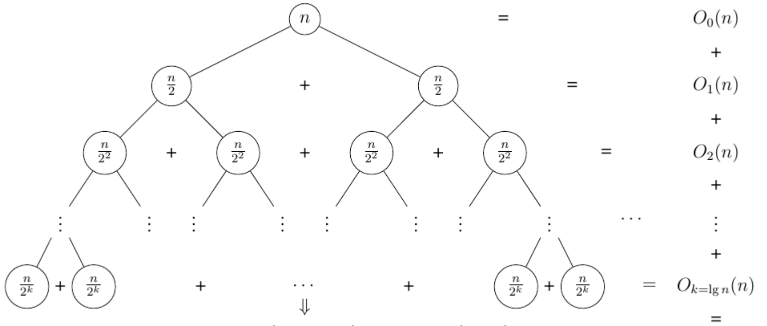
- how many subproblems need to be solved ($a$), mostly $n/2$
- what is the size of the subproblems ($n/b$)
- cost of combination $c(n)$

- Given a recurrence such as $T(n) \leq 2T(n/2) + c(n)$, we want a simple upper bound on the total running time.

- Three common ways to solve such a recurrence:
  - Unroll the recurrence and see what the pattern is.
    - Draw the recursion tree
    - Subtititon method
    - Master Theorem

## Solve a Recurrence

$$T(n) = 2(n/2) + O(n)$$



$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow O(n \cdot \log n)$$

- The root node represents the original problem.
- Every node that is not a leaf has *a* children, representing the number of subproblems it is splitting into.
- Find the size of the subproblem with the help of *b*
- The work at the leaves is $T(1)$
- The work done at each level stays consistent (in this case, it is $O(n)$ at each level)

$$\underbrace{\boxed{2^i}}_{\text{number of subproblems}} * \underbrace{\boxed{\frac{n}{2^i}}}_{\text{size of the subproblem}} = O(n) \tag{2}$$

- There are $\log(n)$ level.
- Hence O(n*log($n$))

Find the order of growth for solutions of the following recurrences.

- Merge-sort: $T(n) = 2T(n/2) + n$, $T(1) = 0$
- $T(n) = 4T(n/2) + n$, $T(1) = 1$
- $T(n) = 4T(n/2) + n^2$, $T(1) = 1$
- $T(n) = 4T(n/2) + n^3$, $T(1) = 1$

## Recurrence relations's order of growth

**Master Theorem** If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

- Merge-sort: $T(n) = 2T(n/2) + n$, $T(1) = 0$
  Here, $a = 2$, $b = 2$, and $d = 1$. Since $a = b^d$ ,
  $T(n) \in \Theta(nlogn)$.
- $T(n) = 4T(n/2) + n$.
  Here, $a = 4$, $b = 2$, and $d = 1$. Since $a > b^d$ ,
  $T(n) \in \Theta(n^{log_2^4}) = \Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$. Here, $a = 4$, $b = 2$, and $d = 2$.
  Since $a = b^d$ , $T(n) \in \Theta(n^2 logn)$.
- $T(n) = 4T(n/2) + n^3$. Here, $a = 4$, $b = 2$, and $d = 3$.
  Since $a < b^d$ , $T(n) \in \Theta(n^3)$.

- We can find items of rank 1 or n in O(n) time.

```
MINIMUM $(A)$
    $\min \leftarrow A[0]$
    **for** $i = 1$ **to** $n - 1$ **do**
        **if** $\min > A[i]$ **then** $\min \leftarrow A[i]$;
    **return** min
```

- The algorithm minimum finds the smallest (rank 1) item in O(n) time.
- A similar algorithm finds maximum item.

- Find both min and max using 3n/2 comparisons.

**MIN-MAX** $(A)$

```
$\textbf{if } |A| = 1, \textbf{ then return } \min = \max = A[0]$
$\textbf{Divide } A \textbf{ into two equal subsets } A_1, A_2$
$(\min_1, \max_1) := \textbf{MIN-MAX } (A_1)$
$(\min_2, \max_2) := \textbf{MIN-MAX } (A_2)$
$\textbf{if } \min_1 \leq \min_2 \textbf{ then return } \min = \min_1$
$\textbf{else return } \min = \min_2$
$\textbf{if } \max_1 \geq \max_2 \textbf{ then return } \max = \max_1$
$\textbf{else return } \max = \max_2$
```

1. Assuming for simplicity that $n = 2^k$.
2. We obtain the following recurrence for the number of element comparisons $C(n)$:
3. $C(n) = 2C(n/2) + 2$ for $n > 2, C(2) = 1, C(1) = 0$.
4. Solving it by **backward substitutions** for $n = 2^k$ , $k \geq 1$, yields the following:

$$\begin{aligned}
C(2^k) &= 2C(2^{k-1}) + 2 \\
&= 2[2C(2^{k-2}) + 2] + 2 = 2^2C(2^{k-2}) + 2^2 + 2 \\
&= 2^2[2C(2^{k-3}) + 2] + 2^2 + 2 = 2^3C(2^{k-3}) + 2^3 + 2^2 + 2 \\
&= ... \\
&= 2^iC(2^{k-i}) + 2^i + 2^{i-1} + ... + 2 \\
&= ... \\
&= 2^{k-1}C(2) + 2^{k-1} + ... + 2 = 2^{k-1} + 2^k - 2 = \frac{3}{2}n - 2.
\end{aligned} \tag{3}$$

- This algorithm makes about 25% fewer comparisons—$1.5n$ compared to 2n—than the brute-force algorithm.
- In fact, the algorithm is optimal in terms of the number of comparisons made.
- As a practical matter, however, it might not be faster than the brute-force algorithm because of the recursion-related overhead.