Example Algorithms

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington

Examples of Algorithms

- Union-Find.
- Binary Search.
- Selection Sort.
- What each of these algorithms does is the next topic we will cover.

Connectivity: An Example

- Suppose that we have a large number of computers, with no connectivity.
  - No computer is connected to any other computer.
- We start establishing direct computer-to-computer links.
- We define connectivity(A, B) as follows:
  - If A and B are directly linked, they are connected.
  - If A and B are connected, and B and C are connected, then A and C are connected.
- Connectivity is *transitive*.

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
    - How do we tell the computer? What do we need to provide?

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
    - How do we tell the computer? What do we need to provide?
    - Answer: we need to provide two integers, specifying the two computers that are getting linked.

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
    - What does it mean that "connectivity changed"?

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
    - What does it mean that "connectivity changed"?
    - It means that there exist at least two computers X and Y that were not connected before the new link was in place, but are connected now.

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
    - Can you come up with an example where the new link does not change connectivity?

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
    - Can you come up with an example where the new link does not change connectivity?
    - Suppose we have computers 1, 2, 3, 4. Suppose 1 and 2 are connected, and 2 and 3 are connected. Then, directly linking 1 to 3 does not add connectivity.

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
  - How do we do that?

A Useful Connectivity Property

- Suppose we have N computers.
- At each point (as we establish links), these N computers will be divided into separate networks.
  - All computers within a network are connected.
  - If computers A and B belong to different networks, they are not connected.
- Each of these networks is called a **connected component.**

Initial Connectivity

- Suppose we have N computers.
- Before we have established any links, how many connected components do we have?

Initial Connectivity

- Suppose we have N computers.
- Before we have established any links, how many connected components do we have?
  - N components: each computer is its own connected component.

Labeling Connected Components

- Suppose we have N computers.
- Suppose we have already established some links, and we have K connected components.
- How can we keep track, for each computer, what connected component it belongs to?

Labeling Connected Components

- Suppose we have N computers.
- Suppose we have already established some links, and we have K connected components.
- How can we keep track, for each computer, what connected component it belongs to?
  - Answer: maintain an array **id** of N integers.
  - **id[p]** will be the ID of the connected component of computer p (where p is an integer).
  - For convenience, we can establish the convention that the ID of a connected component X is just some integer **p** such that computer **p** belongs to X.

The Union-Find Problem

- We want a program that behaves as follows:
  - Each computer is represented as a number.
  - We start our program.
  - Every time we establish a link between two computers, we tell our program about that link.
  - We want the program to tell us if the new link has changed connectivity or not.
  - How do we do that?

Union-Find: First Solution

- It is rather straightforward to come up with a brute force method:
- Every time we establish a link between **p** and **q**:
  - The new link means **p** and **q** are connected.
  - If they were already connected, we do not need to do anything.
  - How can we check if they were already connected?

Union-Find: First Solution

- It is rather straightforward to come up with a brute force method:
- Every time we establish a link between **p** and **q**:
  - The new link means **p** and **q** are connected.
  - If they were already connected, we do not need to do anything.
  - How can we check if they were already connected?
    - Answer: **id[p] == id[q]**

Union-Find: First Solution

- It is rather straightforward to come up with a brute force method:
- Every time we establish a link between **p** and **q**:
  - The new link means **p** and **q** are connected.
  - If they were not already connected, then the connected components of **p** and **q** need to be merged.

Union-Find: First Solution

- It is rather straightforward to come up with a brute force method:
- Every time we establish a link between **p** and **q**:
  - The new link means **p** and **q** are connected.
  - If they were not already connected, then the connected components of **p** and **q** need to be merged.
  - We can go through each computer **i** in the network, and if **id[i] == id[p]**, we set **id[i] = id[q]**.

Union-Find: First Solution

```
#include <stdio.h>
#define N 10000
main()
  { int i, p, q, t, id[N];
    for (i = 0; i < N; i++) id[i] = i;
    while (scanf("%d %d\n", &p, &q) == 2)
      {
        if (id[p] == id[q]) continue;
        for (t = id[p], i = 0; i < N; i++)
          if (id[i] == t) id[i] = id[q];
        printf(" %d %d\n", p, q);
      }
 }
```

Time Analysis

- The first solution to the Union-Find problem takes at least M*N instructions, where:
  - N is the number of objects.
  - M is the number of union operations.
- What is the best case, that will lead to faster execution?

Time Analysis

- The first solution to the Union-Find problem takes at least M*N instructions, where:
  - N is the number of objects.
  - M is the number of union operations.
- What is the best case, that will lead to faster execution?
  - Best case: all links are identical, we only need to do one union. Then, we need at least N instructions.

Time Analysis

- The first solution to the Union-Find problem takes at least M*N instructions, where:
  - N is the number of objects.
  - M is the number of union operations.
- What is the worst case, that will lead to the slowest execution?

Time Analysis

- The first solution to the Union-Find problem takes at least M*N instructions, where:
  - N is the number of objects.
  - M is the number of union operations.
- What is the worst case, that will lead to the slowest execution?
  - Worst case: each link requires a new union operation. Then, we need at least N*L instructions, where L is the number of links.

Time Analysis

- The first solution to the Union-Find problem takes at least M*N instructions, where:
  - N is the number of objects.
  - M is the number of union operations.
  - L is the number of links.
- Source of variance: M. In the best case, M = ???. In the worst case, M = ???.

Time Analysis

- The first solution to the Union-Find problem takes at least M*N instructions, where:
  - N is the number of objects.
  - M is the number of union operations.
  - L is the number of links.
- Source of variance: M. In the best case, M = 1. In the worst case, M = L.

The Find and Union Operations

- **find**: given an object **p**, find out what set it belongs to.
- **union**: given two objects **p** and **q**, unite their two sets.
- Time complexity of **find** in our first solution:
  – ???
- Time complexity of **union** in our first solution:
  – ???

The Find and Union Operations

- **find**: given an object **p**, find out what set it belongs to.
- **union**: given two objects **p** and **q**, unite their two sets.
- Time complexity of **find** in our first solution:
  - Just checking **id[p]**.
  - One instruction in C, a **constant** number of instructions on the CPU.
- Time complexity of **union** in our first solution:
  - At least N instructions, if **p** and **q** belong to different sets.

Rewriting First Solution With Functions - Part 1

```
#include <stdio.h>
#define N 10  /* Made N smaller, so we can print all ids */

/* returns the set id of the object. */
int find(int object, int id[])
{
  return id[object];
}

/* unites the two sets specified by set_id1 and set_id2*/
void set_union(int set_id1, int set_id2, int id[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    if (id[i] == set_id1) id[i] = set_id2;
}
```

Rewriting First Solution With Functions - Part 2

```
main()
{ int p, q, i, id[N], p_id, q_id;
  for (i = 0; i < N; i++) id[i] = i;
  while (scanf("%d %d", &p, &q) == 2)
  {
    p_id = find(p, id); q_id = find(q, id);
    if (p_id == q_id)
    {
      printf(" %d and %d were on the same set\n", p, q);
      continue;
    }
    set_union(p_id, q_id, id, N);
    printf(" %d %d link led to set union\n", p, q);
    for (i = 0; i < N; i++)
      printf("    id[%d] = %d\n", i, id[i]);
    }
  }
}
```

Why Rewrite?

- The rewritten code makes the **find** and **union** operations explicit.
- We can replace **find** and **union** as we wish, and we can keep the main function unchanged.
- Note: **union** is called **set_union** in the code, because **union** is a reserved keywords in C.

- Next: try different versions of **find** and **union**, to make the code more efficient.

Next Version

- **id[p]** will not point to the set_id of p.
  - It will point to just another element of the same set.
  - Thus, **id[p]** initiates a sequence of elements:
  - **id[p] = p2, id[p2] = p3, …, id[pn] = pn**
- This sequence of elements ends when we find an element **pn** such that **id[pn] = pn**.
- We will call this **pn** the id of the set.
- This sequence is not allowed to contain cycles.
- We re-implement **find** and **union** to follow these rules.

Second Version

```
int find(int object, int id[])
{ int next_object;
  next_object = id[object];

  while (next_object != id[next_object])
    next_object = id[next_object];

  return next_object;
}

/* unites the two sets specified by set_id1 and set_id2 */
void set_union(int set_id1, int set_id2, int id[], int size)
{
  id[set_id1] = set_id2;
}
```

id Array Defines Trees of Pointers

- By drawing out what points to what in the **id** array, we obtain trees.
  - Each connected component corresponds to a tree.
  - Each object **p** corresponds to a node whose parent is **id[p]**.
  - Exception: if **id[p] == p**, then p is the **root** of a tree.
- In first version of Union-Find, a connected component of two or more objects corresponded to a tree with two levels.
- Now, a connected component of **n** objects (**n** >= 2) can have anywhere from 2 to **n** levels.
- See textbook figures 1.4, 1.5 (pages 13-14).

Time Analysis of Second Version

- How much time does **union** take?
- How much time does **find** take?

Time Analysis of Second Version

- How much time does **union** take?
  - a constant number of operations (which is the best result we could hope for).
- How much time does **find** take?
  - **find(p)** needs to find the root of the tree that **p** belongs to. This needs at least as many instructions as the distance from **p** to the root of the tree.

Time Analysis of Second Version

- Worst case?

Time Analysis of Second Version

- Worst case: we process M links in this order:
  - 1 0
  - 2 1
  - 3 2
  - …
  - M M-1
- Then, how will the ids look after we process each link?

Time Analysis of Second Version

- Worst case: we process M links in this order:
  - 1 0
  - 2 1
  - 3 2
  - …
  - M M-1
- Then, how will the ids look after we process the m-th link?
  - id[m] = m-1, id[m-1] = m-2, id[m-2] = m-3, …

Time Analysis of Second Version

- Worst case: we process links in this order:
  - 1 0, 2 1, 3 2, …, M M-1.
- Then, how will the ids look after we process each link?
  - id[m] = m-1, id[m-1] = m-2, id[m-2] = m-3, …
- How many instructions will **find** take?

Time Analysis of Second Version

- Worst case: we process links in this order:
  - 1 0, 2 1, 3 2, …, M M-1.
- Then, how will the ids look after we process each link?
  - id[m] = m-1, id[m-1] = m-2, id[m-2] = m-3, …
- How many instructions will **find** take?
  - at least m instructions for the m-th link.
- Total?

Time Analysis of Second Version

- Worst case: we process links in this order:
  - 1 0, 2 1, 3 2, …, M M-1.
- Then, how will the ids look after we process each link?
  - id[m] = m-1, id[m-1] = m-2, id[m-2] = m-3, …
- How many instructions will **find** take?
  - at least m instructions for the m-th link.
- Total? 1 + 2 + 3 + … + M = 0.5 * M * (M+1). So, at least 0.5 * M^2 instructions. **Quadratic in M.**
- Compare to first version: M*N. Which is better?

Time Analysis of Second Version

- Worst case: we process links in this order:
  - 1 0, 2 1, 3 2, …, M M-1.
- Then, how will the ids look after we process each link?
  - id[m] = m-1, id[m-1] = m-2, id[m-2] = m-3, …
- How many instructions will **find** take?
  - at least m instructions for the m-th link.
- Total? 1 + 2 + 3 + … + M = 0.5 * M * (M+1). So, at least 0.5 * M^2 instructions. **Quadratic in M.**
- Compare to first version: M*N. Which is better?
  - The new version, if M < N.

Time Analysis of Second Version

- Worst case: we process links in this order:
  - 1 0, 2 1, 3 2, …, M M-1.
- Then, how will the ids look after we process each link?
  - id[m] = m-1, id[m-1] = m-2, id[m-2] = m-3, …
- What if M > N?
- Then the number of instructions is: 1+2+3+…+N+N+…+N.
- Still better than first version (where we need M*N instructions). Compare:
  1+2+3+…+N+N+…+N (second version)
  N+N+N+…+N+N+…+N (first version)

Third Version

- **find**: same as in second version.
- **union**: always change the id of the smaller set to that of the larger one.
  - How do we know which set is smaller?

Third Version

- **find**: same as in second version.
- **union**: always change the id of the smaller set to that of the larger one.
  - How do we know which set is smaller?
  - Use a new array, that keeps track of the size of each set.

Third Version

- **find**: same as in second version.
- **union**: always change the id of the smaller set to that of the larger one.

```
void set_union(int set_id1, int set_id2, int id[], int sz[])
{ if (sz[set_id1] < sz[set_id2])
   {
     id[set_id1] = set_id2;
     sz[set_id2] += sz[set_id1];
   }
  else
   {
     id[set_id2] = set_id1;
     sz[set_id1] += sz[set_id2];
   }
}
```

Third Version

```
main()
{ int p, q, i, id[N], sz[n], p_id, q_id;
  for (i = 0; i < N; i++)
    { id[i] = i; sz[i] = 1; }
  while (scanf("%d %d", &p, &q) == 2)
  { p_id = find(p, id); q_id = find(q, id);
    if (p_id == q_id)
    {
      printf(" %d and %d were on the same set\n", p, q);
      continue;
    }
    set_union(p_id, q_id, id, sz);
    printf(" %d %d link led to set union\n", p, q);
    for (i = 0; i < N; i++)
    { printf("    id[%d] = %d\n", i, id[i]);  }
  }
}
```

Time Analysis of Third Version

- What is the key effect of considering the size of the two sets?

Time Analysis of Third Version

- What is the key effect of considering the size of the two sets?
- We get flatter trees. When we merge two trees, we avoid creating long chains.
- How does that improve running time?

Time Analysis of Third Version

- What is the key effect of considering the size of the two sets?
- We get flatter trees. When we merge two trees, we avoid creating long chains.
- How does that improve running time?
- For a connected component of **n** objects, **find** will need at most log n operations.
  - Remember, log is always base 2.
- Thus, now we need how many steps in total, for all the **find** operations in the program?

Time Analysis of Third Version

- What is the key effect of considering the size of the two sets?
- We get flatter trees. When we merge two trees, we avoid creating long chains.
- How does that improve running time?
- For a connected component of **n** objects, **find** will need at most log n operations.
  - Remember, log is always base 2.
- Thus, now we need at most M * log N steps in total.

Optional: Fourth Version

- As we go through a tree during a **find** operation, flatten the tree at the same time.

```
int find(int object, int id[])
{
  int next_object;
  next_object = id[object];

  while (next_object != id[next_object])
  {
    id[next_object] = id[id[next_object]];
    next_object = id[next_object];
  }
  return next_object;
}
```

Optional: Fourth Version

- After repeated **find** operations, trees get flatter and flatter, and closer to the best case (two levels).

```
int find(int object, int id[])
{
  int next_object;
  next_object = id[object];

  while (next_object != id[next_object])
  {
    id[next_object] = id[id[next_object]];
    next_object = id[next_object];
  }
  return next_object;
}
```

Optional: Fourth Version

- When all trees are flat (2 levels), how many operations does a single **find** take?

Optional: Fourth Version

- When all trees are flat (2 levels), how many operations does a single **find** take?
- It just needs to check **id[p]**. The number of operations does not depend on the size of the connected component, or the total number of objects.
- When the number of operations does not depend on any variables, we say that the number of operations is **constant**.
- A constant number of operations is algorithmically the best case we can ever hope for.

Next Problem: Membership Search

- We have a set **S** of **N** objects.
- Given an object **v**, we want to determine if **v** is an element of **S**.
- For simplicity, now we will only handle the case where objects are integers.
  - It will become apparent later in the course that the solution actually works for much more general types of objects.
- Can anyone think of a simple solution for this problem?

Sequential Search

- We have a set **S** of **N** objects.
- Given an object **v**, we want to determine if **v** is an element of **S**.
- Sequential search:
  - Compare **v** with every element of **S**.
- How long does this take?

Sequential Search

- We have a set **S** of **N** objects.
- Given an object **v**, we want to determine if **v** is an element of **S**.
- Sequential search:
  - Compare **v** with every element of **S**.
- How long does this take?
  - If **v** is in **S**, we need on average to compare **v** with |**S**|/2 objects.
  - If **v** is not in **S**, we need compare **v** with all |**S**| objects.

Sequential Search - Version 2

- Assume that **S** is sorted in ascending order (this is an assumption that we did not make before).
- Sequential search, version 2:
  - Compare **v** with every element of **S**, till we find the first element **s** such that **s** >= **v.**
  - Then, if **s** != **v** we can safely say that **v** is not in **S**.
- How long does this take?

Sequential Search - Version 2

- Assume that **S** is sorted in ascending order (this is an assumption that we did not make before).
- Sequential search, version 2:
  - Compare **v** with every element of **S**, till we find the first element **s** such that **s** >= **v.**
  - Then, if **s** != **v** we can safely say that **v** is not in **S**.
- How long does this take?
  - We need on average to compare **v** with |**S**|/2 objects, regardless of whether **v** is in **S** or not.
- A little bit better than when **S** was not sorted, but only by a factor of 2, only when **v** is not in **S**.

Binary Search

- Again, assume that **S** is sorted in ascending order.
- At first, if **v** is in **S**, **v** can appear in any position, from 0 to **N**-1 (where **N** is the size of **S**).
- Let's call **left** the leftmost position where **v** may be, and **right** the rightmost position where **v** may be.
- Initially:
  - **left = 0**
  - **right = N - 1**
- Now, suppose we compare **v** with **S[N/2].**
  - Note: if **N/2** is not an integer, round it down.
  - What can we say about **left** and **right**?

Binary Search

- Initially:
  - **left** = 0
  - **right** = **N** - 1
- Now, suppose we compare **v** with **S**[**N**/2**]**.
  - What can we say about **left** and **right**?
- If **v == S[N/2]**, we found **v**, so we are done.
- If **v < S[N/2]**, then **right = N/2 - 1**.
- If **v > S[N/2]**, then **left = N/2 + 1**.
- Importance: We have reduced our search range in half, with a single comparison.

Binary Search - Code

```
/* Determines if v is an element of S.
   If yes, it returns the position of v in a.
   If not, it returns -1.
   N is the size of S.
*/
int search(int S[], int N, int v)
{
  int left, right;
  left = 0; right = N-1;
  while (right >= left)
  { int m = (left+right)/2;
    if (v == S[m]) return m;
    if (v < S[m]) right = m-1; else left = m+1;
  }
  return -1;
}
```

Time Analysis of Binary Search

- How many elements do we need to compare **v** with, if **S** contains **N** objects?
- At most log(N).
- This is what we call **logarithmic time complexity**.
- While **constant time** is the best we can hope, we are usually very happy with logarithmic time.

Next Problem - Sorting

- Suppose that we have an array of items (numbers, strings, etc.), that we want to sort.
- Why would we want to sort?

Next Problem - Sorting

- Suppose that we have an array of items (numbers, strings, etc.), that we want to sort.
- Why would we want to sort?
  - To use in binary search.
  - To compute rankings, statistics (top-10, top-100, median).
- Sorting is one of the most common operations in software.
- In this course we will do several different sorting algorithms, with different properties.
- Today we will look at one of the simplest: Selection Sort.

Selection Sort

- First step: find the smallest element, and exchange it with element at position 0.
- Second step: find the second smallest element, and exchange it with element at position 1.
- n-th step: find the n-th smallest element, and exchange it with element at position n-1.
- If we do this |S| times, then S will be sorted.

Selection Sort - Code

- For simplicity, we only handle the case where the items are integers.

```
/* sort array S in ascending order.
   N is the number of elements in S. */
void selection(int S[], int N)
{ int i, j, temp;
  for (i = 0; i < N; i++)
  { int min = i;
    for (j = i+1; j < N; j++)
      if (S[j] < S[min]) min = j;
    temp = S[min]; S[min] = S[i]; S[i] = temp;
  }
}
```

Selection Sort - Time Analysis

- First step: find the smallest element, and exchange it with element at position 0.
  - We need N-1 comparisons.
- Second step: find the second smallest element, and exchange it with element at position 1.
  - We need N-2 comparisons.
- n-th step: find the n-th smallest element, and exchange it with element at position n-1.
  - We need N-n comparisons.
- Total: (N-1) + (N-2) + (N-3) + … + 1 = about $0.5 * N^2$ comparisons.

Selection Sort - Time Analysis

- Total: (N-1) + (N-2) + (N-3) + … + 1 = about 0.5 * N^2 comparisons.
- **Quadratic time complexity**.
- Commonly used sorting algorithms are a bit more complicated, but have N * log(N) time complexity, which is much better (as N gets large).