Datastructures and Algorithms

Dynamic Programming

Adel Gavranović — ETH Zürich — 2025

Overview

n.ethz.ch/~agavranovic

Material

Webpage

Mail

1. Follow-up

Follow-up from last session

(Slide 29) "Is it possible to reconstruct the (entire) original flow network given its residual network?"

- *Probably* yes, but *probably* needs the assumption that s only has outgoing flows and t only has incoming flows.

2. Feedback regarding **code** expert

General things regarding **code** expert

Any questions regarding **code** expert on your part?

3. Learning Objectives

Objectives

- ☐ Understand what Dynamic Programming is all about
- ☐ Be able to solve a problem using the concepts from Dynamic Programming

4. Summary

Getting on the same page

- What did you see in the lectures up to now?

5. Dynamic Programming

5.1. Recap Theory

Dynamic Programming: Idea

1. Divide a complex problem into a reasonable number of sub-problems; Partial solutions are combined to more complex ones
   = Top-down recursion ("assume the subproblems")
2. Identical problems will be computed only once
   = Memoization
   - The idea is to simply **store the results of subproblems** so that we do not have to re-compute them when needed later.
3. Eliminate recursion
   = Bottom-up algorithms ("combine the subproblems")

- Optionally, not always possible: Save space by storing as little as possible in the DP table

Dynamic Programming: Idea

Question: Which of the following Fibonacci implementations would perform better?

```
int fib(int n) {
  if (n <= 1) {
    return n;
  }
  return fib(n - 1) +
         fib(n - 2);
}
```

```
int fib2(int n) {
  std::vector<int> f(n+1);
  f[0] = 0;
  f[1] = 1;
  for(int i=2;i<=n;++i){
    f[i] = f[i-1]+f[i-2];
  }
  return f[n];
}
```

```
int fib3(int n) {
  if (n <= 1) {
    return n;
  }
  int a = 0;
  int b = 1;
  for(int i=2;i<=n;++i){
    int a_old = a;
    a = b;
    b += a_old;
  }
  return b;
```

Dynamic Programming = Divide-And-Conquer ?

- In both cases the original problem can be solved (more easily) by utilizing the solutions of sub-problems. The problem provides **optimal substructure**.
- Divide-And-Conquer algorithms (such as Mergesort): sub-problems are independent; their solutions are required only once in the algorithm.
- DP: sub-problems are dependent. The problem is said to have **overlapping sub-problems** that are required multiple-times in the algorithm.
- In order to avoid redundant computations, results are tabulated. For **sub-problems there must not be any circular dependencies**.

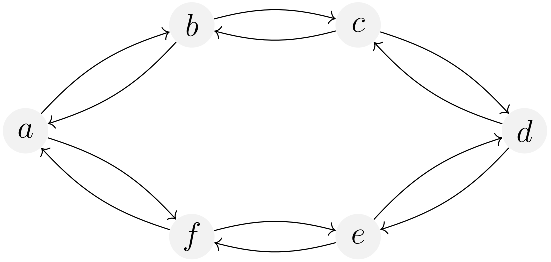
Memoization vs. Dynamic Programming

- **Memoization:**
  - Top-down approach
  - Recursion with caching of results
  - Lazily computes values on-demand
  - Can be more efficient if only a few values are needed
- **Dynamic Programming:**
  - Iterative bottom-up approach
  - Builds solutions from smaller subproblems
  - Computes all values in a predefined order
  - Can be more efficient if all values are needed

Problem Without Optimal Substructure

Question: Problem Without Optimal Substructure?

Example: Longest (simple) path



$$\begin{tikzcd}
 & b \arrow[r, bend left] \arrow[dl, bend left] & c \arrow[l, bend left] \arrow[dr, bend left] & \\
a \arrow[ur, bend left] \arrow[dr, bend left] & & & d \arrow[ul, bend left] \arrow[dl, bend left] \\
 & f \arrow[ul, bend left] \arrow[r, bend left] & e \arrow[l, bend left] \arrow[ur, bend left] &
\end{tikzcd}$$

Problem Without Optimal Substructure: Longest Path

$$\begin{tikzcd}
& b \arrow[r, bend left] \arrow[dl, bend left] & c \arrow[l, bend left] \arrow[dr, bend left] & \\
a \arrow[ur, bend left] \arrow[dr, bend left] & & & d \arrow[ul, bend left] \arrow[dl, bend left] \\
& f \arrow[ul, bend left] \arrow[r, bend left] & e \arrow[l, bend left] \arrow[ur, bend left] &
\end{tikzcd}$$

- Longest path from, e.g. a to e is a, b, c, d, e, i.e. via c
- But the longest path from a to c is *not* a, b, c (and analogously for c to e)

$\Rightarrow$ Combining optimal subsolutions does not yield an optimal overall solution

$\Rightarrow$ The problem does not have optimal substructure

Memoization vs. Dynamic Programming

Question

In which of the following cases might memoization be significantly more efficient than dynamic programming?

1. When all values are required for the final result
2. When only a few values are required for the final result
3. When the problem has overlapping subproblems
4. When the problem can be solved iteratively

Dynamic Programming

A complete description of a dynamic program **always** consists of the following aspects:

- **Definition of the subproblems / the DP table**: What are the dimensions of the table? What is the meaning of each entry?
- **Recursion: Computation of an entry**: How can an entry be computed from the values of other entries? Which entries do not depend on others?
- **Computation order (topological order)**: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
- **Solution and Running Time**: How can the final solution be extracted once the table has been filled? Running time of the DP algorithm.

Review

Choose which characteristics a problem needs to have for a dynamic programming approach to be appropriate:

- Optimal substructure
- Problem solving for Real-time problems
- Independent sub-problems
- Memory-efficient solution
- Recursive structure
- Overlapping sub-problems
- Circular dependencies
- Tabulation or memoization potential
- Small state space

Example: Coin Change Problem

Definition

Given a set of coin denominations and a target amount, find the minimum number of coins needed to make the target amount. Note that the same coin denomination can be used more than once.

Example

Given coins = [1, 2, 4] and target amount = 8, the solution is 2 (4 + 4).

Remark

When the problem does not have a solution, the algorithm returns -1.

Coin Change Problem

Task

Design a recursive algorithm to solve the task.

Coin Change: Recursive Solution

```
int coin_change(const std::vector<int>& coins, int amount) {
    if (amount == 0) {
        return 0;
    }
    int min_coins = INT_MAX;
    for (int coin : coins) {
        if (amount - coin >= 0) {
            int temp = coin_change(coins, amount - coin);
            if (temp != -1) {
                min_coins = std::min(min_coins, temp + 1);
            }
        }
    }
    return min_coins == INT_MAX ? -1 : min_coins;
}
```

Coin Change Problem

Task

Design a DP algorithm to solve the task.

Coin Change: Dynamic Programming

We can use dynamic programming to solve this problem by building a one-dimensional array where `dp[i]` represents the minimum number of coins required to make the amount i:

- Set each element in `dp` to a value larger than the maximum possible number of coins.
- Set `dp[0] = 0`.
- For each coin `c`, iterate through the array and update `dp[i]` if `dp[i-c]+1` has a lower value.

Coin Change: DP Solution

```cpp
int coin_change(const std::vector<int>& coins, int amount) {
    std::vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;
    for (int coin : coins) {
        for (int i = coin; i <= amount; ++i) {
            dp[i] = std::min(dp[i], dp[i - coin] + 1);
        }
    }
    return dp[amount] <= amount ? dp[amount] : -1;
}
```

Coin Change: DP Visualisation

Coins: [1, 4, 5]   Target: 8

`i`	0	1	2	3	4	5	6	7	8
`dp[i]`	0	1	2	3	1	1	2	3	2

After processing the third and last coin. Answer: `dp[8] = 2`.

Coin Change: Time Complexity

Task

Compare the time complexity of the DP algorithm with that of the naive recursive algorithm

Naive Algorithm

The naive algorithm has an exponential time complexity of $\mathcal{O}(c^n)$, where c is the number of coin denominations and n is the target amount.

Dynamic Programming Algorithm

The dynamic programming algorithm has a polynomial time complexity of $\mathcal{O}(c \cdot n)$, where c is the number of coin denominations and n is the target amount.

5.2. Example: Longest Common Subsequence

DP Example: Longest Common Subsequence

Definition

A *subsequence* of a sequence is generated by removing some or none of the elements of the original sequence. For example, "AC" is a subsequence of "ABC".

Problem

Given two sequences X and Y, find the length of the longest common subsequence of X and Y.

Concrete Problem Instance

Example

X: PROGRAM

Y: ARMOR

X: PROGRAM

Y: ARMOR

Answer?

length 3: ROR

Subproblems?

String X of length m and string Y of length n:
Which subproblems are there?

- if last character matches: $+1$ and make both strings shorter by one
- make X shorter by one, Y the same
- make Y shorter by one, X the same

Recursive Solution

```cpp
int lcs(const std::string& X, const std::string& Y, int m, int n) {
    if (m == 0 || n == 0) {
        return 0;
    }
    if (X[m - 1] == Y[n - 1]) {
        return 1 + lcs(X, Y, m - 1, n - 1);
    } else {
        return std::max(lcs(X, Y, m - 1, n),
                        lcs(X, Y, m, n - 1));
    }
}
```

Dynamic Programming

Instead, we can use dynamic programming to solve this problem by building a table to store the lengths of the longest common subsequences of the prefixes of X and Y:

- Update the table values from the top left to the bottom right.
- If the characters at the current position match, set the current cell value to the diagonal cell value incremented by one, or one if it doesn't exist.
- If they don't match, set the current cell value to the maximum of the left and top cell values, or zero if they don't exist.

DP Table

X/Y	P	R	O	G	R	A	M
A							
R							
M							
O							
R							

Solution Reconstruction

find LCS (reconstruct solution)?

Time Complexity

Question

How does the time complexity of the DP algorithm compare to the naive recursive algorithm?

5.3. Example: Palindromes

DP Example: Palindromes

A *palindrome* is a word that reads the same way in either forward or reverse direction. Example: RACECAR.

Formally: $\langle a_1, \ldots, a_n \rangle$ is a palindrome $\iff$

- either $n = 1$, or
- $a_1 = a_n$ and $\langle a_2, \ldots, a_{n-1} \rangle$ is a palindrome [1]

We use an array $A[1..n]$ to store a string of length n. A subarray $A[i..j]$ is called *palindrome in* A if it is a palindrome. Examples:

- `[L, A, R, A]` contains palindromes A (2x), R, L and ARA
- `[A, N, N, A]` contains palindromes A (2x), N (2x), NN and $ANNA$

[1]for $n = 2$ we only require $a_1 = a_2$

DP Example: Palindromes

Task 1.1: Describe an efficient dynamic programming algorithm that finds all pairs (i, j) where $A[i] \dots A[j]$ is a palindrome.

Examples:

- `[L, A, R, A]` $\longrightarrow (1,1), (2,2), (3,3), (4,4), (2,4)$
- `[A, N, N, A]` $\longrightarrow (1,1), (2,2), (3,3), (4,4), (2,3), (1,4)$

Task 1.2: What is the running time of your solution?

- Try to find a DP algorithm!
- How does the table look like?
- How do we traverse the table?
- How do we compute an entry?

Palindromes Task 1.1: Solution

	R	A	C	E	C	A	R
R	1	0	0	0	0	0	1
A	-	1	0	0	0	1	0
C	-	-	1	0	1	0	0
E	-	-	-	1	0	0	0
C	-	-	-	-	1	0	0
A	-	-	-	-	-	1	0
R	-	-	-	-	-	-	1

Palindromes Task 1.1: Solution

Definition of the DP table: We use an $n \times n$ table T with entries that are 0 or 1. For $1 \leq i \leq j \leq n$ let $T[i, j] = 1 \iff \langle A[i], \ldots, A[j]\rangle$ is a palindrome.

Computation of an entry: We distinguish three cases.

1. $1 \leq i = j \leq n$: $A[i]$ is a palindrome of length 1, thus we set
$$T[i, j] = T[i, i] = 1$$
2. $1 \leq i \leq n$, $j = i + 1 \leq n$: We consider palindromes of length 2, and set
$$T[i, i + 1] = 1 \iff A[i] = A[i + 1]$$
3. $1 \leq i \leq n$, $i + 1 < j \leq n$: Let $\langle A[i], \ldots, A[j]\rangle$ be the considered sequence. By definition it is a palindrome if $A[i] = A[j]$ and additionally, $\langle A[i + 1], \ldots, A[j - 1]\rangle$ is a palindrome. Thus we set
$$T[i, j] = 1 \iff A[i] = A[j] \text{ and } T[i + 1, j - 1] = 1$$

Palindromes Task 1.1: Solution

Example: $A =$ RACEC***E***R is not a palindrome, but contains non-trivial palindromes CEC and ECE.

	R	A	C	E	C	E	R
R	1	0	0	0	0	0	0
A	-	1	0	0	0	0	0
C	-	-	1	0	1	0	0
E	-	-	-	1	0	1	0
C	-	-	-	-	1	0	0
E	-	-	-	-	-	1	0
R	-	-	-	-	-	-	1

Palindromes: Solution

Task 1.2: What is the running time of the algorithm?

- The table has n^2 entries. We must effecively fill $\frac{n(n+1)}{2} \in \Theta(n^2)$ of these.
- Each table entry can be computed in time $\mathcal{O}(1)$.
- Hence, filling the table is done in $\mathcal{O}(n^2)$ steps.

Task 2.1: Describe how a longest palindrome in A can be extracted from the DP table constructed before.

Traverse table in opposite order of filling, starting from the entry $T[1,n]$. If $T[i,j] = 1$, then $A[i] \ldots A[j]$ is a palindrome. The first such entry found is a longest palindrome.

Task 2.2: What is the running time of the reconstruction?

Same as before: $\mathcal{O}(n^2)$.

6. Summary

Recursive Problem-Solving Strategies

From the Lecture

Brute Force Enumeration	**Backtracking**	**Divide and Conquer**	**Dynamic Programming**
Recursive Enumerability	Constraint Satisfaction, Partial Validation	Optimal Substructure	Optimal Substructure, Overlapping Subproblems
DFS, BFS, all Permutations, Tree Traversal	n-Queen, Sudoku, m-Coloring, SAT-Solving, naive TSP	Binary Search, Mergesort, Quicksort, Hanoi Towers, FFT	Bellman Ford, Warshall, Rod-Cutting, LAS, Editing Distance, Knapsack Problem DP

7. Code-Expert Exercise

Code-Example

"Exam Q: Maximum sum increasing subsequence (DP)" on Code-Expert

8. From the previous week

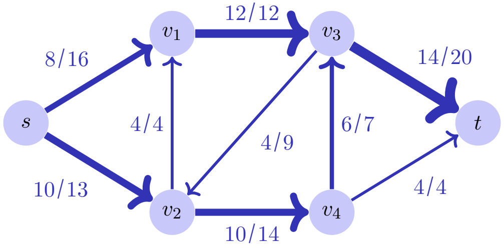
8.1. Maxflow Theory Recap

Flow

A **Flow** $f : V \times V \to \mathbb{R}$ fulfills the following conditions:

- **Bounded Capacity**:
  For all $u, v \in V$: $f(u,v) \leq c(u,v)$.
- **Conservation of flow**:
  For all $u \in V \setminus \{s,t\}$:

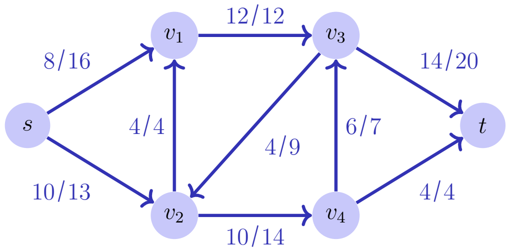
$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v).$$

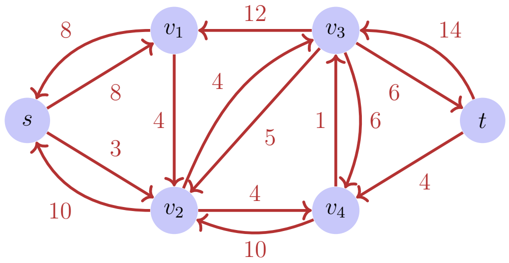


Value of the flow:
$|f| = \sum_{v \in V} f(s,v)$.
Here $|f| = 18$.

Residual Network

The **Residual network** G_f is provided by the edges with positive residual capacity. What does it look like for this flow network?





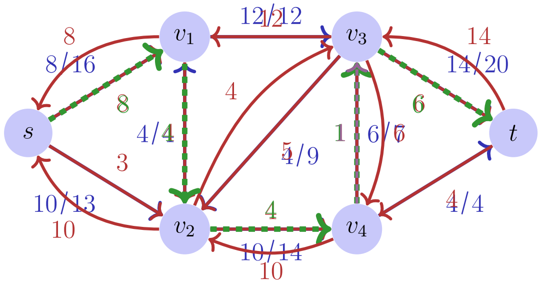
Residual networks provide the same kind of properties as flow networks with the exception of permitting antiparallel edges

Augmenting Paths

Augmenting Path p: simple path from s to t in the residual network G_f.

Residual Capacity $c_f(p)$: the least capacity along the augmenting path p

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ edge in } p\}$$



Note: There can be multiple augmenting paths!

Algorithm Ford-Fulkerson(G, s, t)

Input: Flow network $G = (V, E, c)$
Output: Maximal flow f.

```
for $(u, v) \in E$ do
    $f(u, v) \leftarrow 0$
while Exists path $p : s \rightsquigarrow t$ in residual network $G_f$ do
    $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$
    foreach $(u, v) \in p$ do
        if $(u, v) \in E$ then
            $f(u, v) \leftarrow f(u, v) + c_f(p)$
        else
            $f(v, u) \leftarrow f(v, u) - c_f(p)$
```

Edmonds-Karp Algorithm

Choose in the Ford-Fulkerson-Method for finding a path in G_f the augmenting path of shortest possible length (e.g. with BFS)

Theorem 1

When the Edmonds-Karp algorithm is applied to some integer valued flow network $G = (V, E)$ with source s and sink t then the number of flow increases applied by the algorithm is in $\mathcal{O}(|V| \cdot |E|)$

$\Rightarrow$ ***Overall asymptotic runtime:*** $\mathcal{O}(|V| \cdot |E|^2)$

Max-Flow Min-Cut Theorem

Theorem 2

Let f be a flow in a flow network $G = (V, E, c)$ with source s and sink t. The following statements are equivalent:

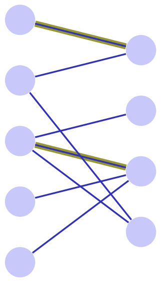
1. *f is a maximal flow in G*
2. *The residual network G_f does not provide any augmenting paths*
3. *It holds that $|f| = c(S, T)$ for a cut (S, T) of G.*

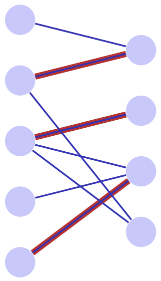
Application: maximal bipartite matching

Given: bipartite undirected graph $G = (V, E)$.

Matching M: $M \subseteq E$ such that $|\{m \in M : v \in m\}| \leq 1$ for all $v \in V$.

Maximal Matching M: Matching M, such that $|M| \geq |M'|$ for each matching M'.





8.2. TSP

Travelling Salesperson Problem

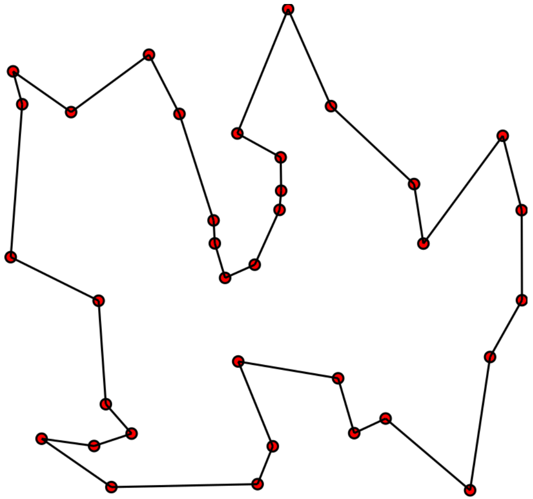
Problem

Given a map and list of cities, what is the shortest possible route that visits each city once and returns at the original city?

Mathematical model

On an undirected, weighted graph G, which cycle containing all of G's vertices has the lowest weight sum?

Travelling Salesperson Problem



Travelling Salesperson Problem

- The problem has no known polynomial-time solution.
- Many heuristic algorithms exists. They do not always return the optimal solution.

Travelling Salesperson Problem

- The heuristic algorithm that you are asked to implement on CodeExpert (*The Travelling Student*) on CodeExpert uses an MST:
    1. Compute the minimum spanning tree M
    2. Make a depth first search on M
- The algorithm is 2-approximate, meaning that the solution it generates has at most twice the cost of the optimal solution.
- The algorithm assumes a complete graph $G = (V, E, c)$ that satisfies the triangle inequality: $\forall v, w, x \in V : c(v, w) \leq c(v, x) + c(x, w)$

9. Outro

General Questions?

See you next time!

Have a nice week!