Artificial Intelligence II

Lesson 4 - Optimization

First Code Academy

Today’s Plan

Teach Back	00 - 5 min
Greedy Algorithms	10 - 15 min
Dynamic Programming	15 - 20 min
Quiz	20-25 min
Break	25 - 28 min
Project - Edit Distance	28 - 55 min
Lesson Recap	55 - 60 min

Teach Back

_____ refer to conditions that limit our possible choices

The range of values our variables can take is called their _____.

With _____ we make a decision and abandon it if it does not lead to a solution.

Key Terms

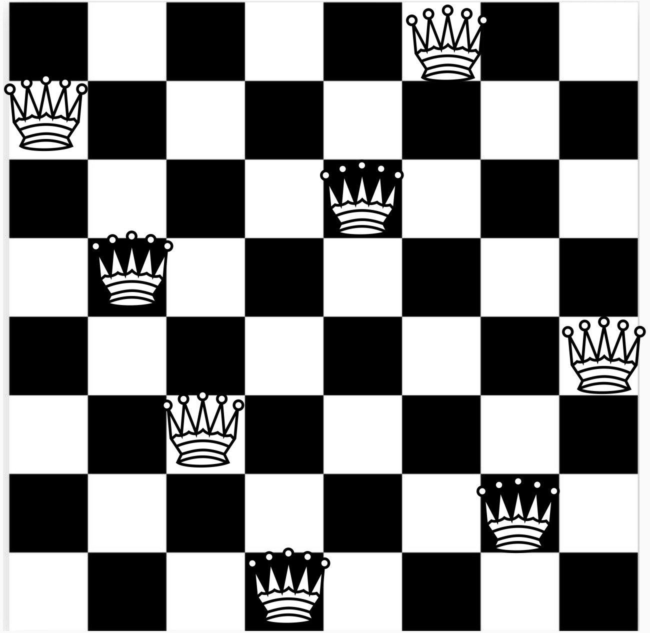
Greedy Algorithms

Dynamic Programming

What did we learn last time?

N-Queens

How can we find an arrangement of queens on a chessboard such that none of them attack each other?



Greedy Algorithms

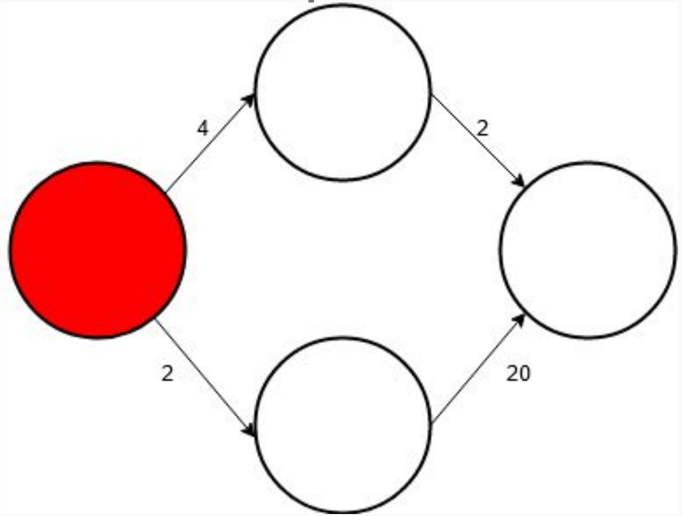
Greedy Algorithms

A greedy algorithm always makes the **locally** best choice (take the lowest-cost path, etc...)

Does this approach always lead to the best solution?

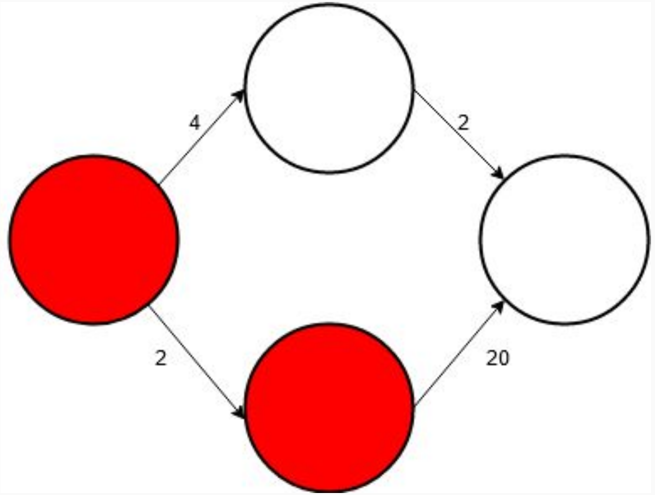
Greedy Algorithms

Do not always lead to an optimal solution



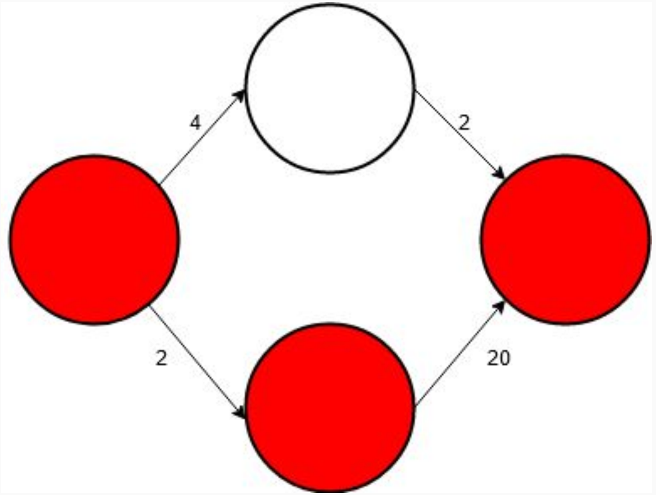
Greedy Algorithms

Do not always lead to an optimal solution



Greedy Algorithms

Do not always lead to an optimal solution



Greedy Algorithms

Although they don't always find the optimal solution, they sometimes give a good-enough approximation.

Dynamic Programming

Dynamic Programming

A way of solving problems which are made up of smaller versions of the same problem.

These smaller problems are called **subproblems.**

Example

The **Fibonacci** sequence of numbers is made by adding the previous two numbers in the sequence.

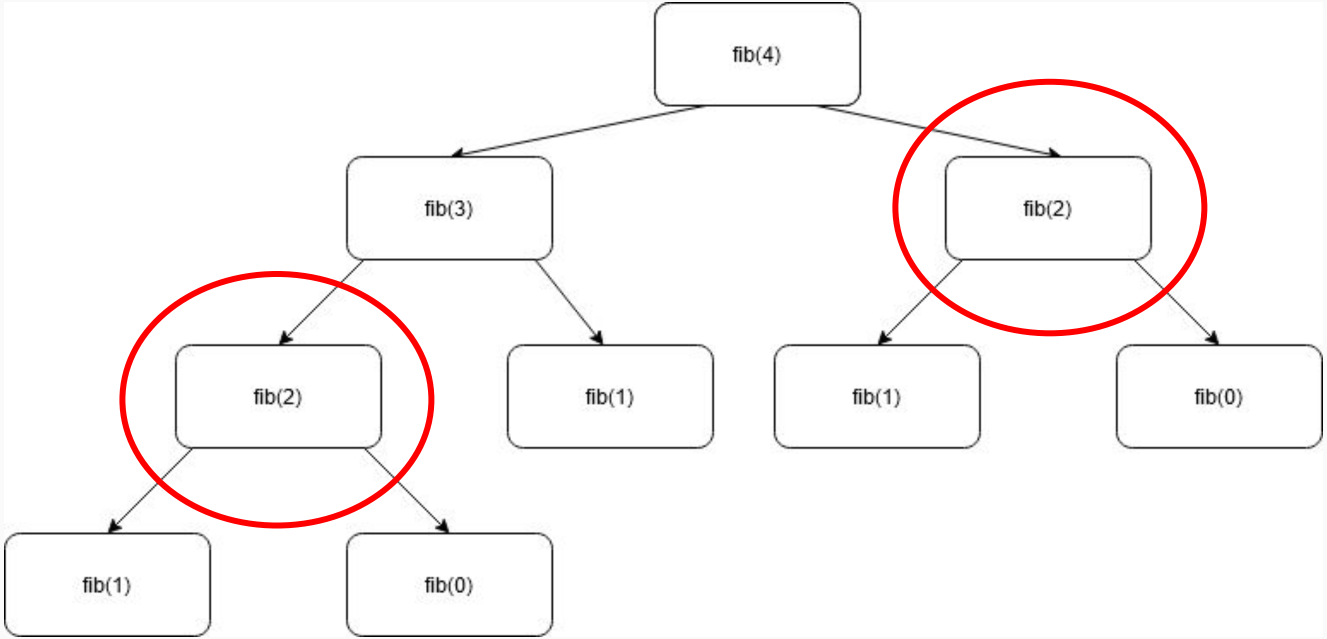
```
Fibonacci = [0, 1, 1, 2, 3, 5, 8, 13,...]
```

The n^{th} fibonacci number is then:

```
fib(n) = fib(n-1) + fib(n-2)
```

Example

However, we end up solving many subproblems multiple times



Strategy

A general approach to dynamic programming problems is:

```
If we already stored the solution:
    return solution
Else:
    calculate and store current solution
```

This simple `if` statement saves us much trouble!

Example

We can store the solution to each subproblem and then retrieve it when we need the subproblem

Try for yourself!

http://bit.ly/FCA_AI2_Fib

Quiz: https://bit.ly/FCA_Quiz_AI

Project: Edit Distance

Get the starter file

http://bit.ly/FCA_AI2_Starter

Edit Distance

Given two strings, what is the minimum number of edits to turn one string into another?

`rat -> rut` ← Can just substitute the “u” for the “a”

`eating -> reading` ← Can add “r” and change “t” to “d”

Edit Distance

We have three choices:

1. Insert a character.
2. Substitute one character with another
3. Delete a character

Base Cases

To go from an empty string to "red", we have to insert 3 characters

$$\begin{array}{cc} & \begin{matrix} \_ & r & e & a & d & i & n & g \end{matrix} \\ \begin{matrix} \_ \\ r \\ e \\ d \end{matrix} & \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & & & & & & & \\ 2 & & & & & & & \\ 3 & & & & & & & \end{bmatrix} \end{array}$$

Let’s code the naive solution!

Naive Solution

1. If the current characters we are comparing are the same, don't make an edit.
2. For the three possible actions, recurse and pick the one with minimum cost.

Naive Solution

If we’re checking the empty string, we just return the length of the other string

```python
def edit_distance_naive(source, target, source_char, target_char):

    # We can just insert len(target)
    if source_char == 0:
        return target_char

    # If target's length is 0, we could just remove all the
    # letters from source string
    if target_char  == 0:
        return source_char
```

Naive Solution

If the characters we’re comparing are the same, don’t make an edit

```
    # If the characters we're comparing are the same, just
    # continue recursing
    if source[source_char - 1] == target[target_char - 1]:
        return edit_distance_naive(source,
                                   target,
                                   source_char - 1,
                                   target_char -1)
```

Naive Solution

Recurse on the possible moves and choose the one with minimum cost

```
        return 1 + min(
                edit_distance_naive(source, target, source_char-1, target_char),
                edit_distance_naive(source, target, source_char, target_char-1),
                edit_distance_naive(source, target, source_char-1, target_char-1)
            )
```

Let’s code the better solution!

Dynamic strategy

Make an n x m matrix, where n and m are the lengths of the two strings, respectively

See how many edits we have to make to substrings of length i and j

Choose the operation with the minimum cost

Dynamic Solution

We create a solution matrix and store the solution to all our subproblems. This will make the recursion faster!

```python
def edit_distance(source, target):
    # Matrix to store our solution
    sol = [[0 for i in range(len(source)+1)] for j in range(len(target)+1)]

```

Dynamic Solution

Do some **preprocessing** before the main loop

```
    # If source string is empty, delete source_char characters
    for i in range(1, len(source) + 1):
        sol[0][i] = sol[0][i-1] + 1

    # If source string is empty, also can insert target_char characters
    for j in range(1, len(target) + 1):
        sol[j][0] = sol[j-1][0] + 1
```

Dynamic Solution

If the characters we're comparing are the same, just pass it along!

```python
    for i in range(1, len(target)+1):
        for j in range(1, len(source)+1):

            # If the two letters are the same, use the same answer
            if source[j-1] == target[i-1]:
                sol[i][j] = sol[i-1][j-1]

```

Dynamic Solution

Instead of recursing, we can just look inside the matrix for the solution to the subproblems

```
# Take the operation with minimum cost
else:
    sol[i][j] = 1 + min(sol[i][j-1],
                        sol[i-1][j],
                        sol[i-1][j-1])
```

Dynamic Solution

Now we just return the appropriate cell in our matrix!

```
        return sol[len(target) - 1][len(source) - 1]
```

Try it out!

```
C:\Users\Andres Ponce\FCA_AI\FCA_AI2\L4>python L4.py

Naive or dynamic solution: naive
Enter two strings separated by a space: algorithm astronaut
Going from algorithm to astronaut takes 8 edits.
```

Challenge

Try and time how long the algorithms take to compare their difference! (it will be more noticeable for longer words)

Challenge - Answer

We start a timer before the function call and stop it when we return. Have to modify the `edit()` method a bit.

```python
def edit(method, source, target):

    start = timer()
    if method == 'naive':
        ans = edit_distance_naive(source, target, len(source), len(target))
    elif method == 'dynamic':
        ans = edit_distance(source, target)
    end = timer()

    print(f'\tAlgorithm took {end - start} seconds.')
    return ans
```

Key Terms - Review

Greedy Algorithms

Dynamic Programming

That’s it for today!

Artificial Intelligence II

Lesson 4 - Optimization

First Code
Academy