III Escuela de Primavera en Investigación de Operaciones

Problemas de optimización y su integración con aplicaciones empresariales desde un enfoque de desarrollo de software con el framework Timefold AI

TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO SUPERIOR DE PURÍSIMA DEL RINCÓN

José Alejandro Cornejo-Acosta

Blanca Verónica Zúñiga-Núñez

Abril 2024

Organización

- **Primera parte** 10:45 am – 12:15 pm (**1 hora y media**).
- **Receso** 12:15 pm – 12:30 pm (**15 minutos**).
- **Segunda parte** 12:30 pm – 2:00 pm (**1 hora y media**).

Contenido

1. Lenguaje de programación Java
2. Programación funcional y orientada a objetos
3. Algoritmos de optimización
    a) Heurísticas constructivas
    b) Búsqueda tabú, recocido simulado
4. Problemas de optimización/planeación
5. Framework de código abierto Timefold AI
6. Integración de modelos de optimización con microservicios

Some slides are in English.

Software necesario

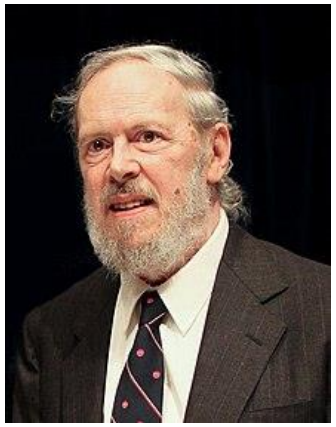
- https://shorturl.at/aqrOY
- https://github.com/alex-cornejo/EPIO2024Java

Presentación de la audiencia (15 minutos)

- Nombre y estudios.
- Experiencia con temas de investigación de operaciones.
- Experiencia con programación y desarrollo de software.

Antes de los 1990s

- Dennis Ritchie (1970s)



Bell Laboratories

AT&T

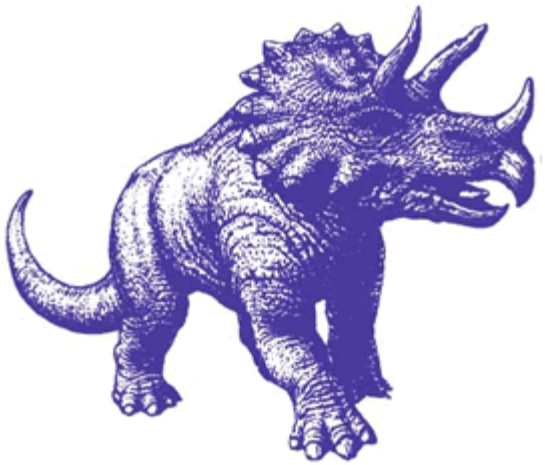
Antes de los 1990s

- Bjarne Stroustrup (1980s)

Bell Laboratories

AT&T

Antes de los 1990s



COBOL

Principios de los 1990s

- *Sun microsystems* patrocina el proyecto "***the Green Project***". Un equipo de 13 personas liderado por **James Gosling**.

Sun microsystems

Principios de los 1990s

- Como resultado se obtuvo un lenguaje de programación orientado a objetos.

Oak → Green → Java

El lenguaje Java

1990s

2005-2007

android

Google

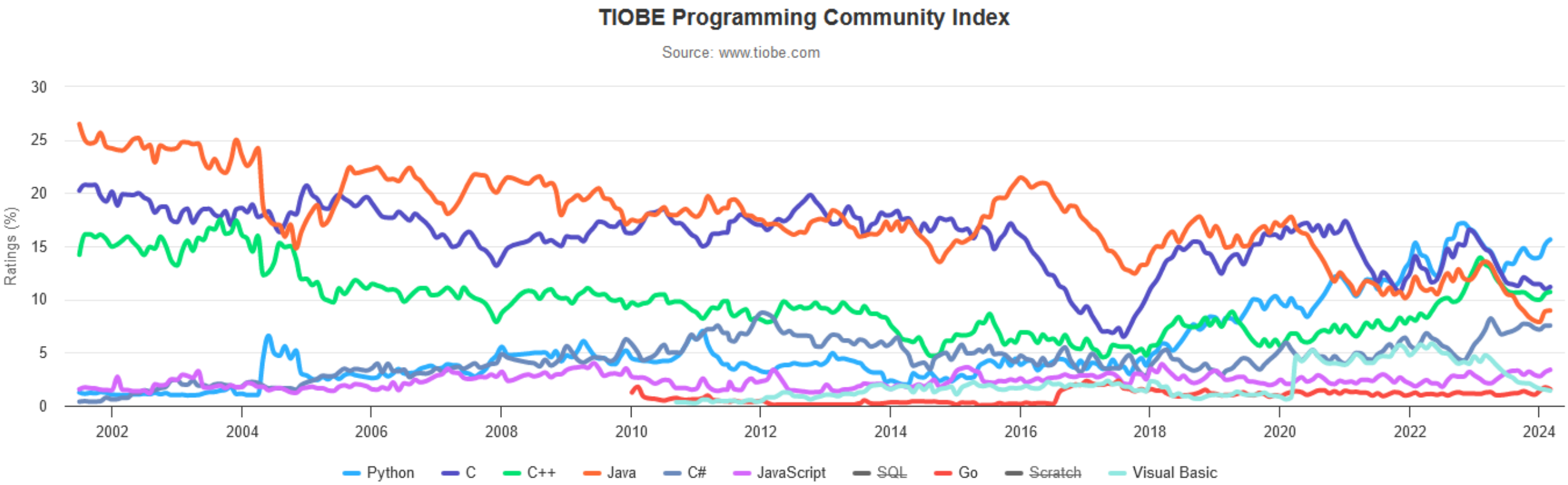
2010

ORACLE

Oracle vs Google, una historia escrita con sangre

- 2010 - Oracle **demandó** a Google por violación de patentes y derechos de autor.
- 2012 - Tribunal de Distrito de los Estados Unidos llegó a la conclusión **que Google sí vulneró la propiedad intelectual**.
- 2016 - Oracle exigió a Google un pago de 9.300 millones de dólares.
- 2016 - Tribunal de distrito llega a un nuevo **veredicto a favor de Google**.
- 2018 – **Oracle consiguió apelar** al Circuito Federal y esta vez le dieron la razón.
- 2019 - Solicita al Tribunal Supremo que revisara las decisiones previas.
- 2021 - **Google gana a Oracle**. El juicio sobre software más importante de la última década.
- https://www.xataka.com/legislacion-y-derechos/extraordinaria-noticia-para-todos-desarrolladores-software-consecuencias-sentencia-oracle-vs-google
- https://www.xataka.com/legislacion-y-derechos/oracle-google-juicio-copyright-importante-siglo-juego-esta-futuro-software

Popularidad de Java



Estatus de versiones

JDK Beta	1995
JDK 1.0	1996
JDK 1.1	1997
J2SE 1.2	1998
J2SE 1.3	2000
J2SE 1.4	2002
J2SE 5.0	2004
Java SE 6	2006
Java SE 7	2011
Java SE 8 (LTS)	2014
Java SE 9	2017
Java SE 10	2018
Java SE 11 (LTS)	2018
Java SE 12	2019
Java SE 13	2019
Java SE 14	2020
Java SE 15	2020
Java SE 16	2021
Java SE 17 (LTS)	2021
Java SE 18	2022

Legend: Old version | Older version, still maintained | **Latest version** | Future release

Rendimiento del lenguaje de programación Java

mandelbrot

source	secs	mem
Python 3	163.32	12,080
Java	4.15	69,136
C++ g++	0.84	34,780

spectral-norm

source	secs	mem
Python 3	120.99	13,424
Java	1.63	39,304
C++ g++	0.72	1,192

n-body

source	secs	mem
Python 3	567.56	8,076
Java	6.74	35,844
C++ g++	2.12	764

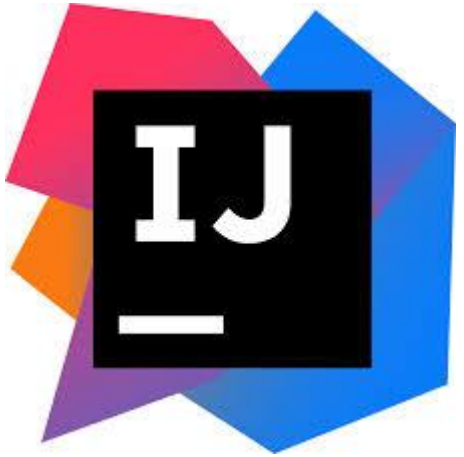
Java 16 vs Python 3.9.2 vs C++ g++ (Ubuntu 10.3.0-1ubuntu1)

https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

Desarrollo en Java

- Aplicaciones web (backend).
- Desarrollo en Android.
- Bases de datos.
- Big data y minería de datos.
- Electrodomésticos.
- Cajeros automáticos.

Desarrollo en Java



¿Vale la pena usar java?

- Java está pasado de moda…
- Java ya no se usa…
- Java es un lenguaje viejo…
- Java es verboso y confuso…
- Java es lento…

TODOS son rumores

Old Java

```java
import java.util.ArrayList;
import java.util.List;

public class Java6Example {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("Java");
        list.add("es");
        list.add("genial");

        for (String item : list) {
            System.out.println(item);
        }
    }
}
```

Modern Java is nice

```java
import java.util.List;

public class Java21Example {
    void main() {
        List<String> list = List.of("Java", "es", "genial");

        list.forEach(System.out::println);
    }
}
```

Programación orientada a objetos

Paradigma de programación que se basa en la organización del software alrededor de **"objetos"**, que son entidades que combinan datos y operaciones que actúan sobre esos datos.

Programación orientada a objetos

Paradigma de programación.

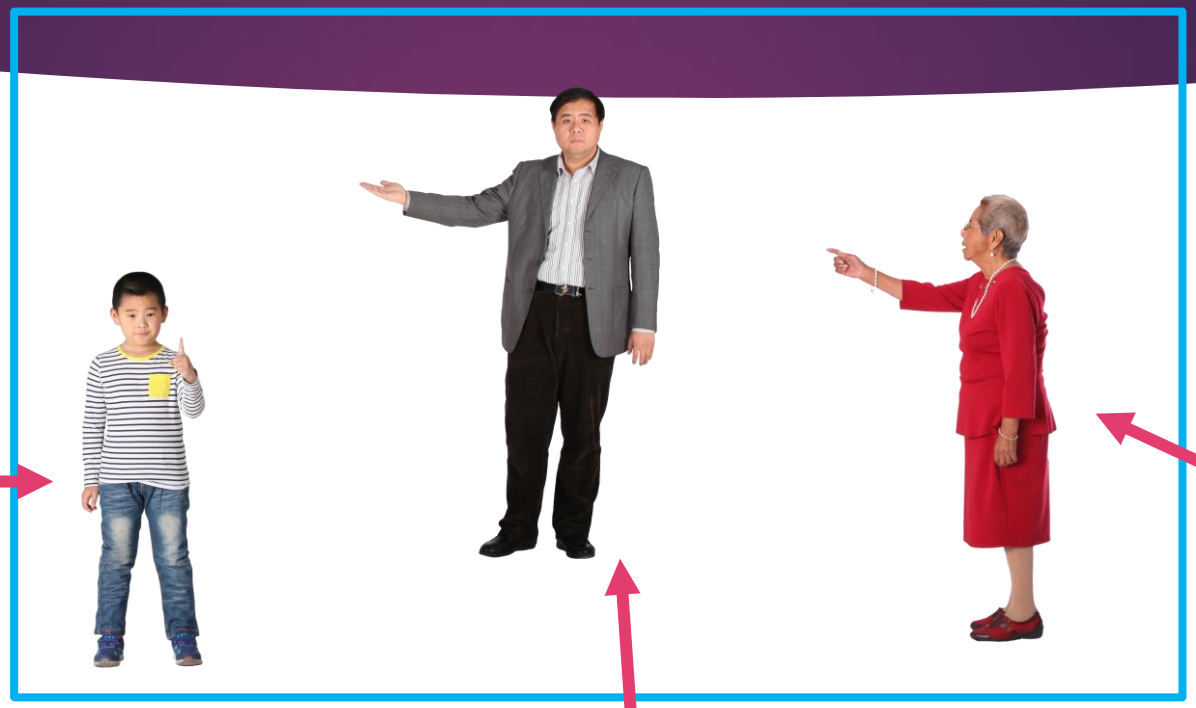
1. Abstracción
2. Herencia
3. Polimorfismo
4. Encapsulación

Clases y objetos

Clase “Persona”

Clase “Animal/Perro”

Clases y objetos



Nombre: Juan
Edad: 7 años

Nombre: Pedro
Edad: 40 años

Nombre: María
Edad: 65 años

Una clase en Java

Person.java

```
1  package com.jacaenc2021.examples;
2
3  public class Person {
4
5      private String name;
6      private int age;
7
8      public Person() {
9      }
10
11     public String getName() { return name; }
14
15     public void setName(String name) { this.name = name; }
18
19     public int getAge() { return age; }
22
23     public void setAge(int age) { this.age = age; }
26 }
27
```

Declaración de la clase

Atributos o propiedades

Constructor

Métodos de acceso

Creación de objetos en Java

```java
var person1 = new Person();
person1.setName("Juan");
person1.setAge(7);

var person2 = new Person();
person2.setName("Pedro");
person2.setAge(40);

var person3 = new Person();
person3.setName("María");
person3.setAge(65);
```

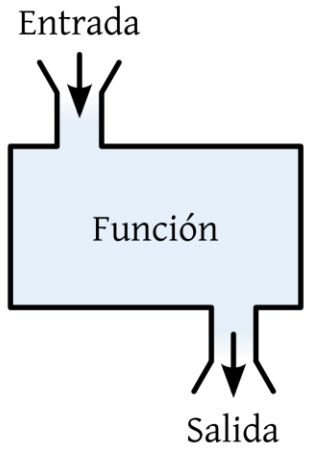
Nombre: Juan
Edad: 7 años

Nombre: Pedro
Edad: 40 años

Nombre: María
Edad: 65 años

Programación funcional

Paradigma de programación que se basa en el uso de **funciones** como elementos fundamentales. Se pasan funciones como argumentos para ser usados en otras funciones.



Programación funcional

Para ver los beneficios se revisará el siguiente caso de uso:

- Se tiene una lista de platillos, en donde cada platillo tiene varios atributos.
- De estos platillos, se quiere seleccionar aquellos que tengan pocas calorías (menos de 400).
- De los platillos seleccionados, ordenarlos ascendentemente por cantidad de calorías.
- Obtener los **nombres** platillos en una lista.

Coding time…

Coding time

- Crear la clase **Dish**.
- Agrega los atributos **name** y **calories**.
- Agregar constructor vacío y constructor con parámetros.
- Agregar getters y setters.

Programación funcional

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish dish: menu) {
    if(dish.getCalories() < 400) {
        lowCaloricDishes.add(dish);
    }
}
```

Filters the elements using an accumulator

```
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    public int compare(Dish dish1, Dish dish2) {
        return Integer.compare(dish1.getCalories(), dish2.getCalories());
    }
});
```

Sorts the dishes with an anonymous class

```
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish dish: lowCaloricDishes) {
    lowCaloricDishesName.add(dish.getName());
}
```

Processes the sorted list to select the names of dishes

Programación funcional

After (Java 8):

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
List<String> lowCaloricDishesName =
                menu.stream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dish::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```

Selects dishes that are below 400 calories

Sorts them by calories

Stores all the names in a List

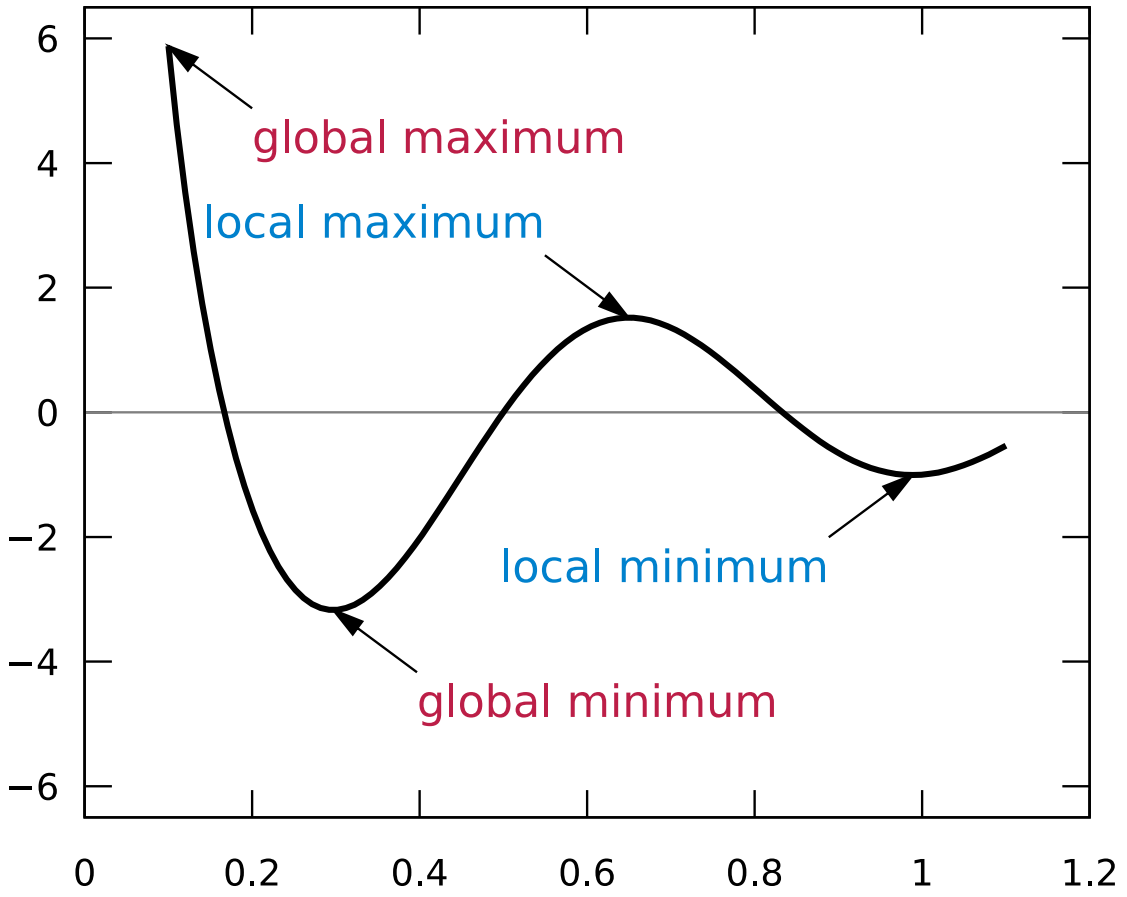
Extracts the names of these dishes

Programación funcional

Enfoque multicore (paralelismo)

```
List<String> lowCaloricDishesName =
                menu.parallelStream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dishes::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```

Optimización



Instancia de un problema de optimización

Una instancia de un problema de optimización es un par (S, f), donde S es el conjunto de todas las posibles soluciones para dicha instancia, y f es una función de costo definida como:

$$f: S \to \mathbb{R}$$

y deseamos encontrar un valor $s \in S$ tal que:

$$f(s) \leq f(y), \forall y \in S \qquad \text{(para un problema de minimización)}$$

Tal valor s es conocido como óptimo global para la instancia (S, f).

Problema de optimización

Un problema de optimización es el conjunto I de todas las posibles instancias.

Problemas de optimización

- Variables
- Función objetivo
- Restricciones

Problemas de optimización

- Optimización continua (variables continuas).
- Optimización discreta (variables discretas).

Ejemplos de problemas de optimización

The Traveling Salesman Problem (TSP

Given a **set of cities**, find the shortest tour for a salesman that visits each city exactly once, starting and ending his tour in the same city.

The Vehicle Routing Problem (VRP)

Given a **fleet of vehicles**, a **depot**, and a **set of customers**, find a set of routes that, starting and ending at the depot, visit each customer once. The distances traveled by vehicles should be minimized.

The Facility Location Problem

Given a set of **potential facility locations** and a **set of consumers** that needs to be served by the facilities. The goal is to find a subset of the potential facility locations and which consumers each should serve, such that the sum of distances from consumers to their assigned facilities is minimized.

The Task Assignment Problem

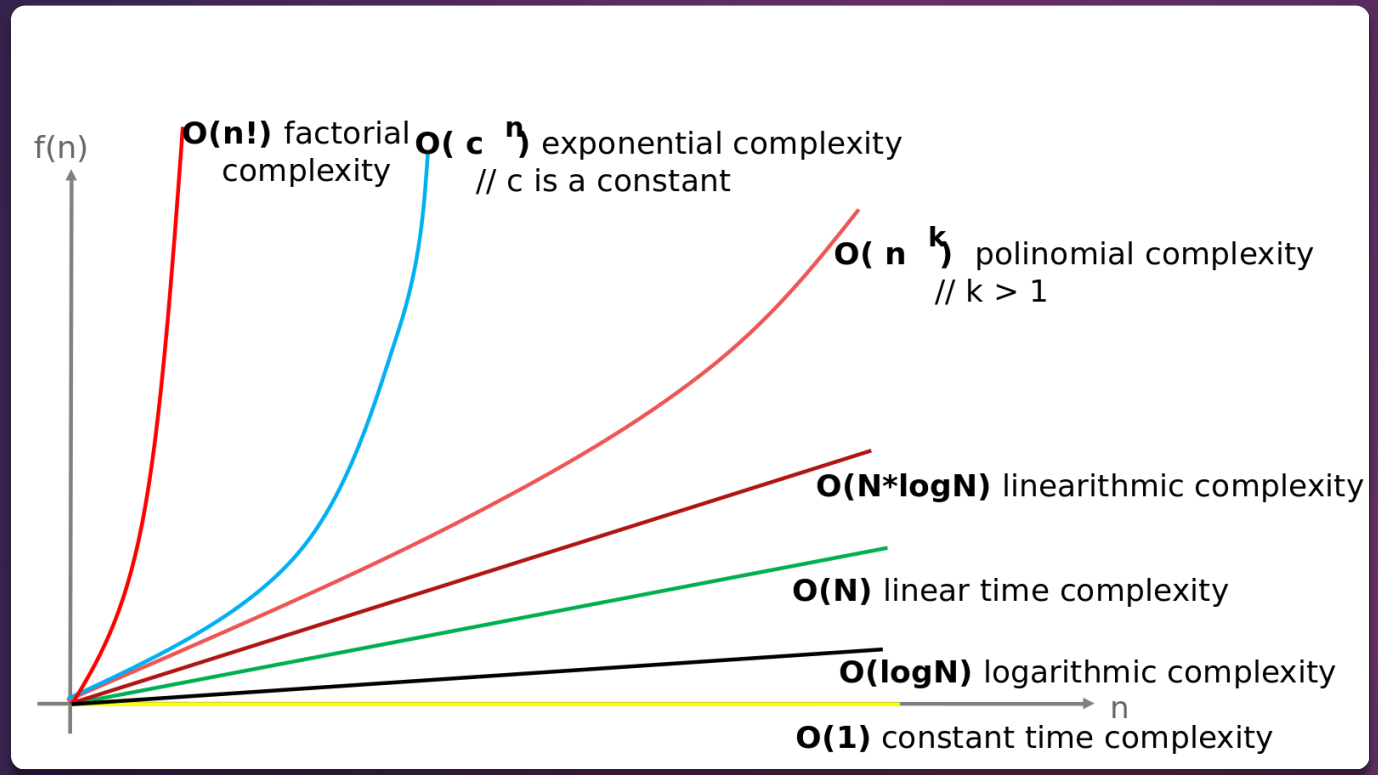
Given a **set of tasks** and a **set of employees**, assign the tasks to the employees in such a way that the time required to complete all the tasks is minimized. Each task requires one or more skills. The employee must possess all these skills.

The Timetabling Scheduling Problem

Consists of the weekly scheduling of the lectures for several **university courses** within a given number of rooms and time periods.

- A teacher must not have two lectures in the same period.
- Two lectures must not be in the same room in the same period.
- Lectures belonging to the same curriculum should be as compact as
- possible to each other (so in consecutive periods) per day.

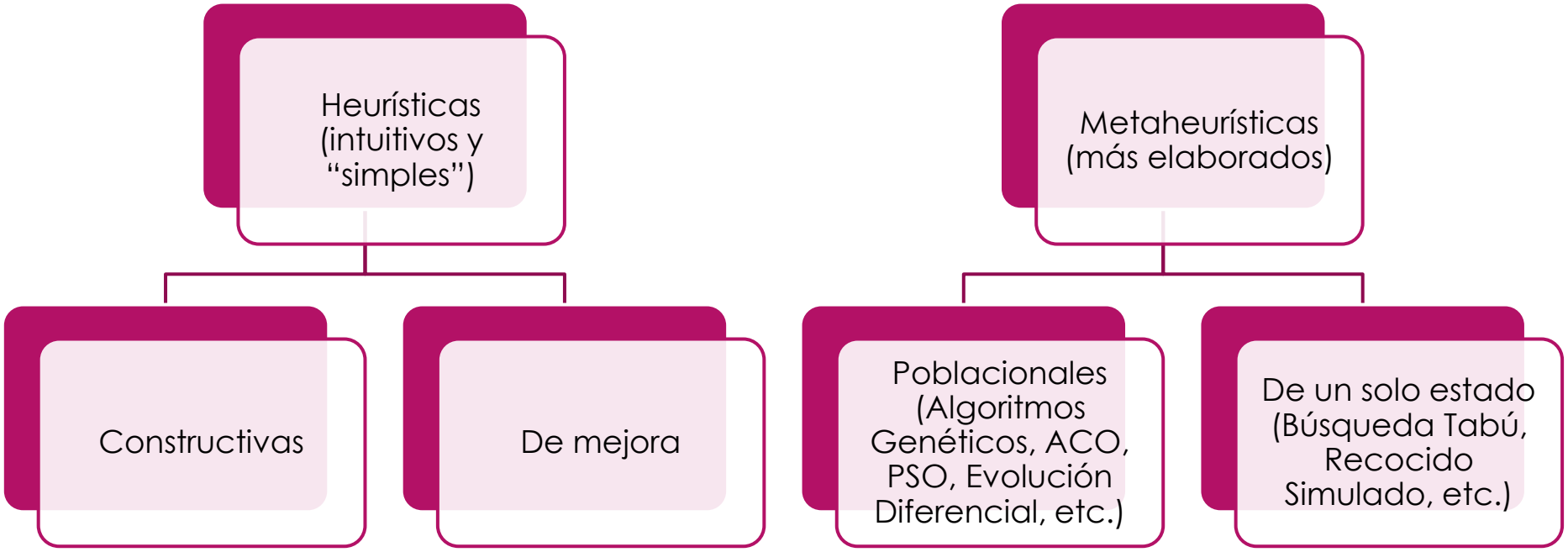
¿Qué tan difíciles son estos problemas?



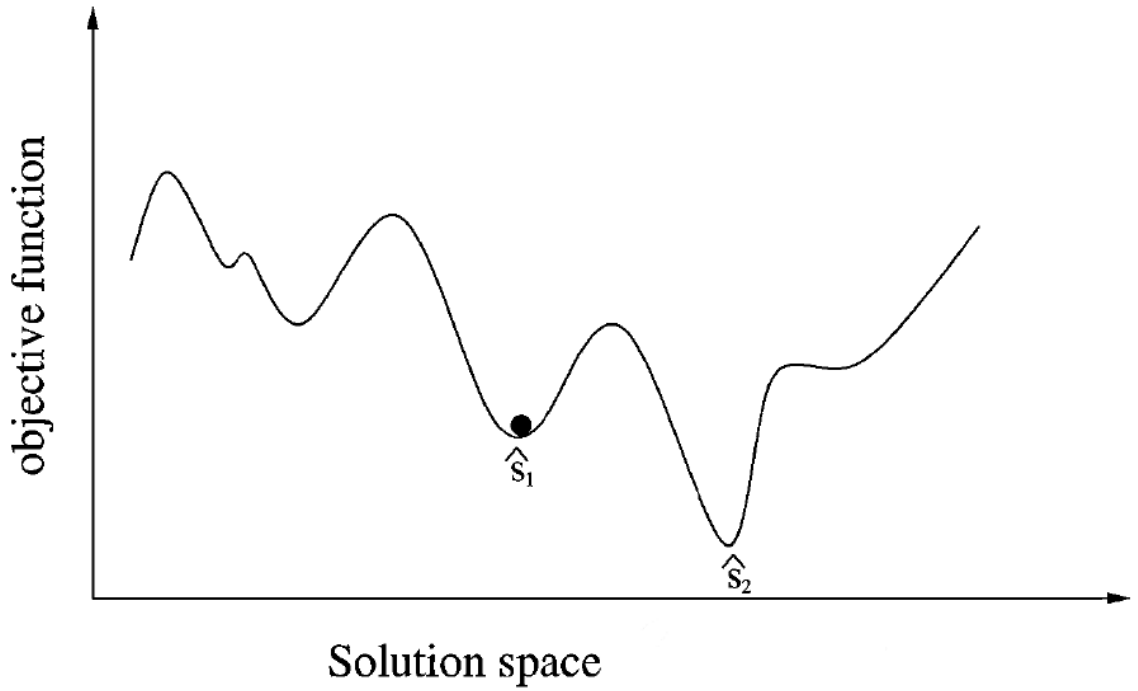
Algoritmos

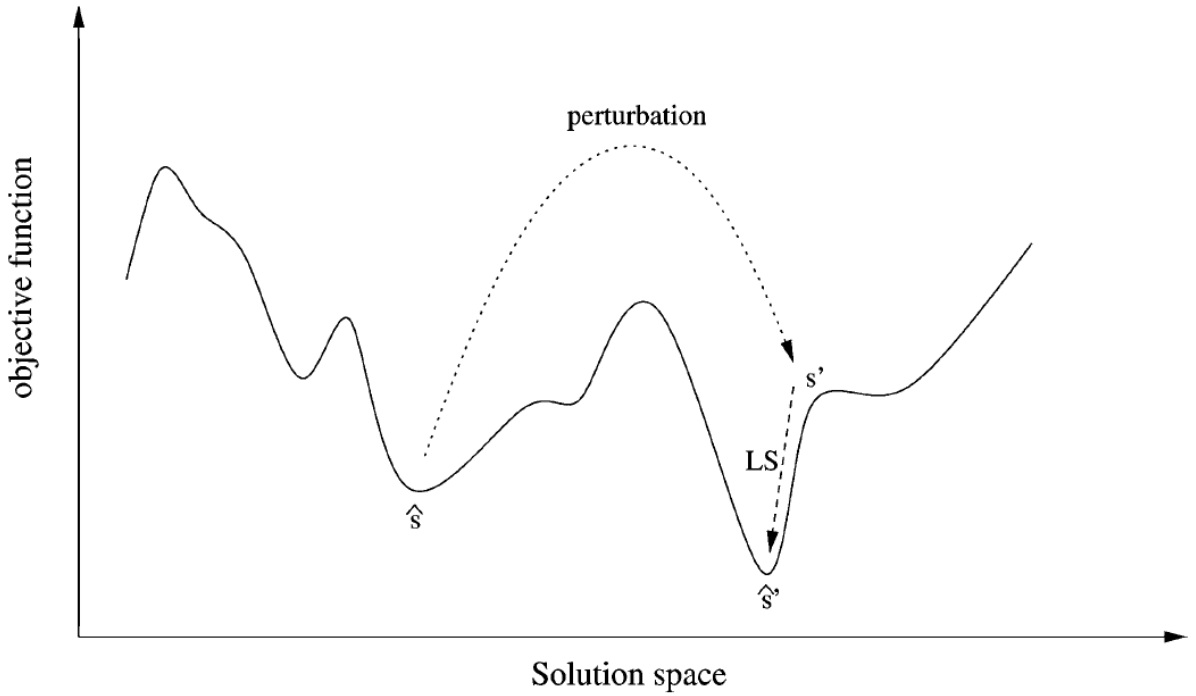
Exhaustivos/Exactos	Heurísticas/Metaheurísticas
No hay garantías en tiempos de ejecución (pueden ejecutarse por siempre).	Tiempos de ejecución “prácticos”.
Encuentran la solución óptima.	No garantizan encontrar la solución óptima.

Heurísticas/Metaheurísticas



Heurística vs Metaheurística





Metaheurística

Recocido simulado

- Propuesto en los 1980s.
- Inspirado en la metalurgia.

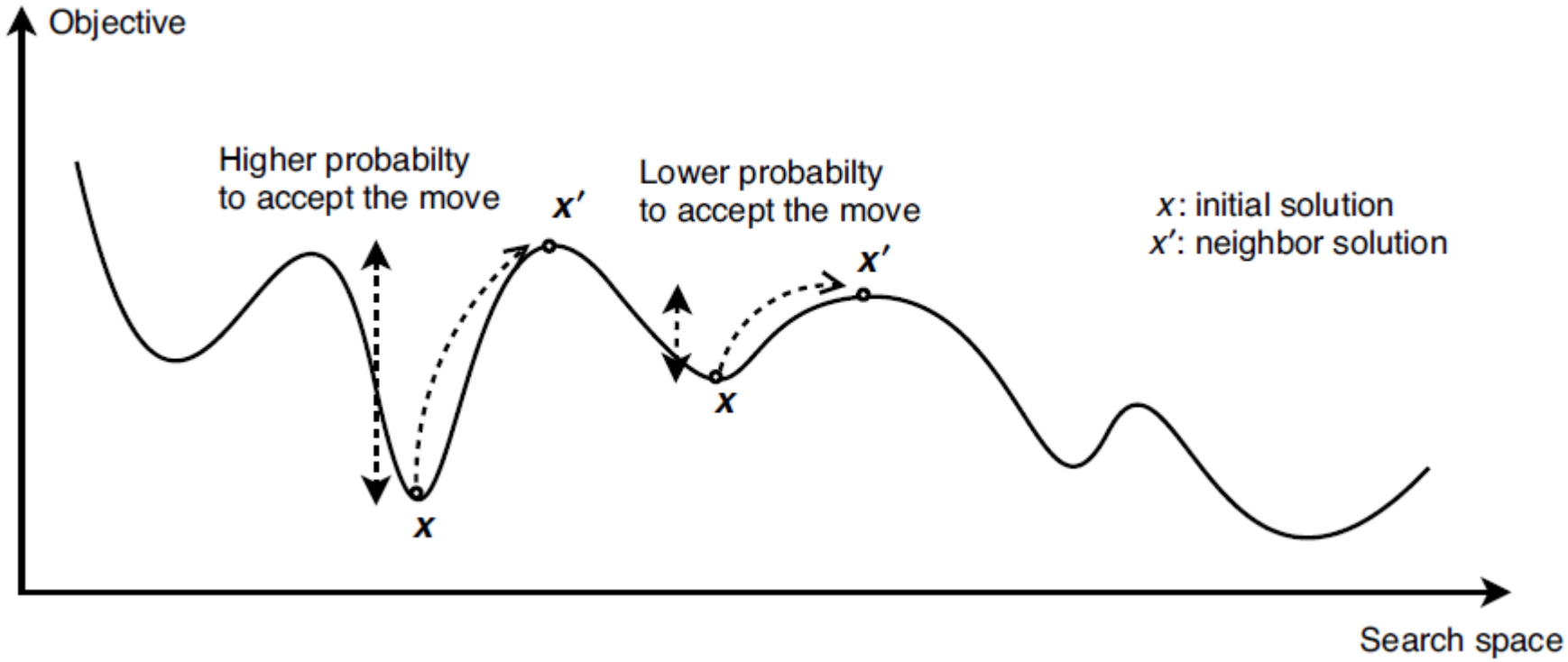
Recocido simulado

Algorithm 13 *Simulated Annealing*

1: $t \leftarrow$ temperature, initially a high number

2: $S \leftarrow$ some initial candidate solution
3: $Best \leftarrow S$
4: **repeat**
5: $\quad R \leftarrow \text{Tweak}(\text{Copy}(S))$
6: $\quad$ **if** $\text{Quality}(R) > \text{Quality}(S)$ or if a random number chosen from 0 to 1 $< e^{\frac{\text{Quality}(R)-\text{Quality}(S)}{t}}$ **then**
7: $\quad\quad S \leftarrow R$
8: $\quad$ Decrease t
9: $\quad$ **if** $\text{Quality}(S) > \text{Quality}(Best)$ **then**
10: $\quad\quad Best \leftarrow S$
11: **until** *Best* is the ideal solution, we have run out of time, or $t \leq 0$
12: **return** *Best*

Recocido simulado



Búsqueda Tabú

- Propuesto en los 1980s.
- Utiliza el concepto de “historial” y “memoria” para tomar decisiones.

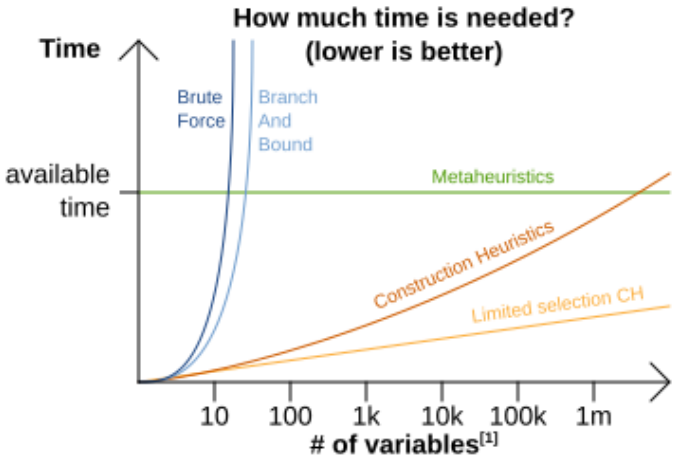
Búsqueda Tabú

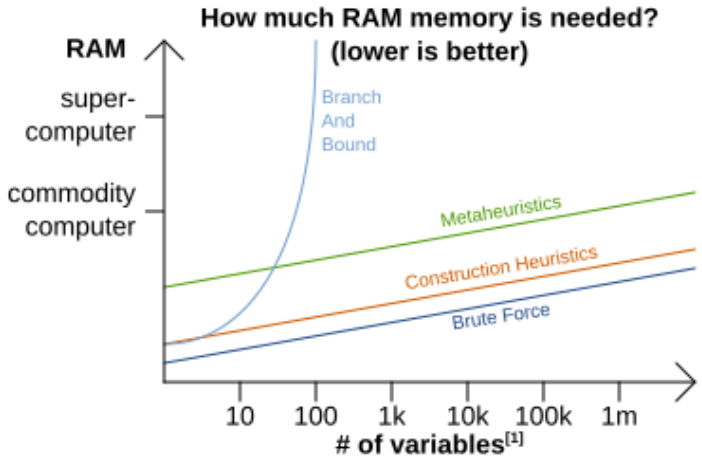
Algorithm 14 *Tabu Search*

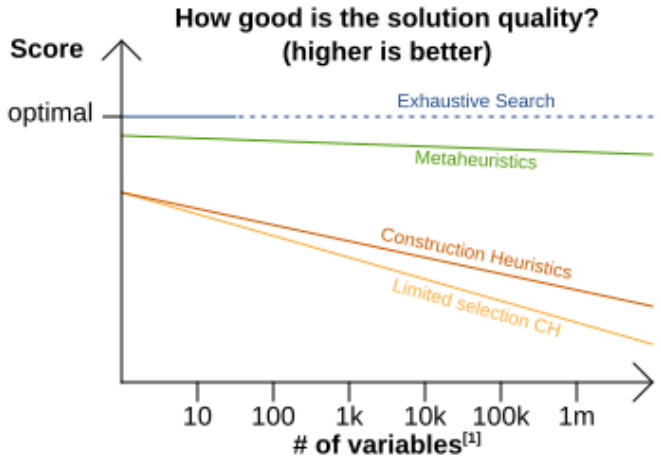
```
1: l ← Desired maximum tabu list length
2: n ← number of tweaks desired to sample the gradient

3: S ← some initial candidate solution
4: Best ← S
5: L ← {} a tabu list of maximum length l          ▷ Implemented as first in, first-out queue
6: Enqueue S into L
7: repeat
8:     if Length(L) > l then
9:         Remove oldest element from L
10:    R ← Tweak(Copy(S))
11:    for n − 1 times do
12:        W ← Tweak(Copy(S))
13:        if W ∉ L and (Quality(W) > Quality(R) or R ∈ L) then
14:            R ← W
15:    if R ∉ L then
16:        S ← R
17:        Enqueue R into L
18:    if Quality(S) > Quality(Best) then
19:        Best ← S
20: until Best is the ideal solution or we have run out of time
21: return Best
```

Scalability of optimization algorithms

When scaling out, metaheuristics deliver the best solution in reasonable time on realistic hardware.







Effects of scaling out:

Exhaustive Search delivers the optimal solution but takes forever.

Construction Heuristics (including greedy algorithms) deliver poor quality in time.

Metaheuristics deliver good quality in time.

Note: Metaheuristics include a CH to initialize.

This is a rough generalization, based on years of experience and a large number of benchmarks on realistic use cases. Results may differ per use case and per solver configuration.

[1] Vars with a large value range (binary vars scale much more)

Resumen...

¿Qué es un framework?

Conjunto de herramientas que proporciona una **base sobre la cual los desarrolladores pueden construir y desarrollar software** de manera más eficiente. **Define una arquitectura y un conjunto de reglas** que permiten a los desarrolladores crear aplicaciones de manera más rápida y con menos esfuerzo. Proporciona funcionalidades comunes y soluciones predefinidas para problemas recurrentes.

¿Qué es el código abierto?

Software que se distribuye junto con su código.

- Código diseñado para ser accesible públicamente.
- Desarrollado de forma descentralizada y colaborativa.
- Usualmente incluse una licencia.

Timefold AI

Solver que utiliza algoritmos de optimización y de inteligencia computacional para abordar problemas de optimización y satisfacción de restricciones como calendarización, ruteo, asignación de tareas, etc. Puede utilizarse con Java, Python y Kotlin.

timefold

Timefold AI

- Iniciado en 2006 bajo el nombre de *Taseree*.
- International Timetabling Competition 2007. Finished 4th.
- OptaPlanner (mantenido por Red Hat).
- Fork Timefold AI a partir de 2023.

timefold

Timefold AI

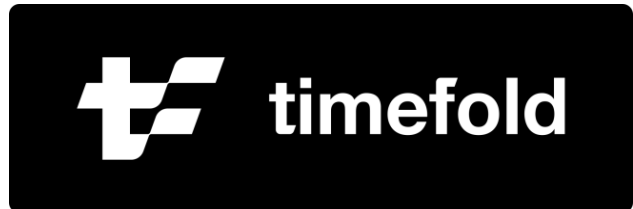
Muchos de los solvers disponibles en el mercado están diseñados para problemas académicos, como el **problema del agente viajero**, **VRP**, **Bin Packing**, **Job scheduling**, etc. En el mundo real, los problemas consideran circunstancias más complejas. Estos problemas cuentan con restricciones “duras” y “suaves”. Timefold está diseñado para abordar este tipo de problemáticas, empleado técnicas del estado del arte.

Timefold AI

- Otras opciones se centran en un lenguaje y expresiones matemáticas.
- Timefold tiene un enfoque para el desarrollo de software, para programadores.

Timefold AI

- Búsqueda exhaustiva.
- Heurísticas constructivas.
- Metaheurísticas.



Hard and soft constraints

Usually, a planning problem has at least two levels of constraints:

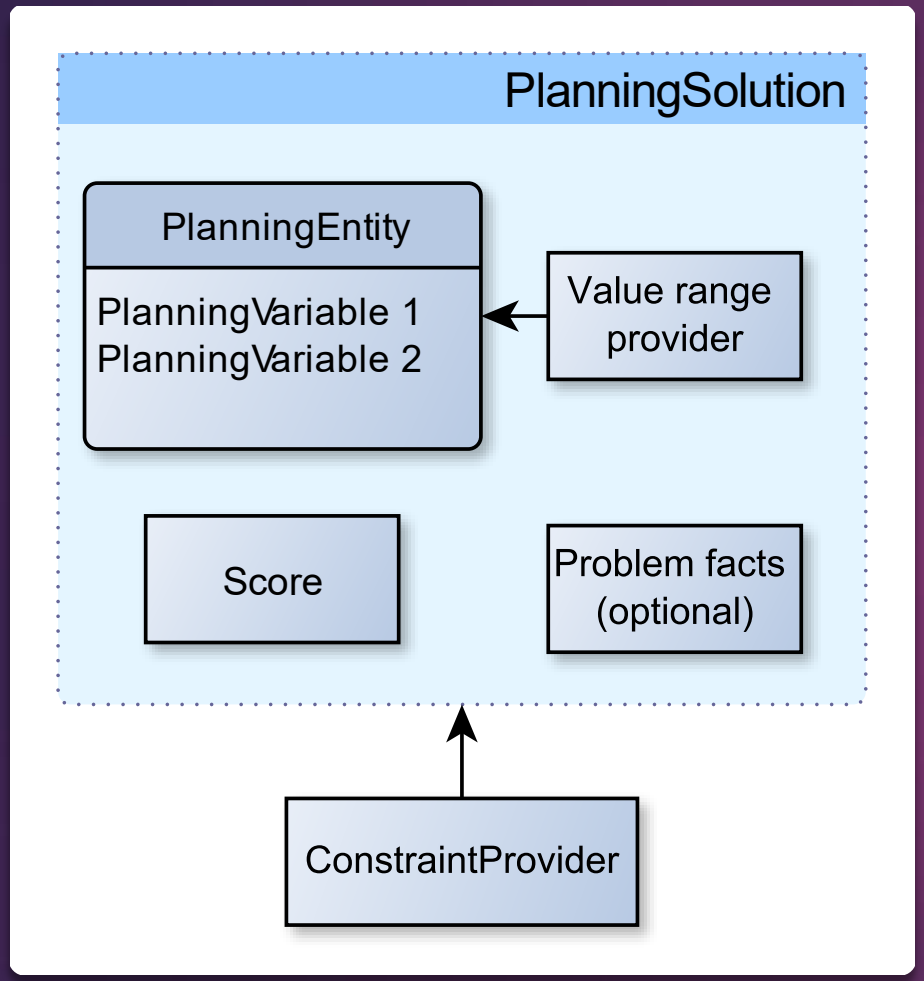
- A hard constraint must not be broken.
- A soft constraint should not be broken if it can be avoided.

Score definition

- A **score** represents the **quality** of a specific solution. The higher the better.
- Timefold AI looks for the best solution, which is the solution with the highest score found in the available time.
- The score is computed based on the constraints. Every constraint must **penalize** or **reward** the quality of a solution.

Categories of solutions

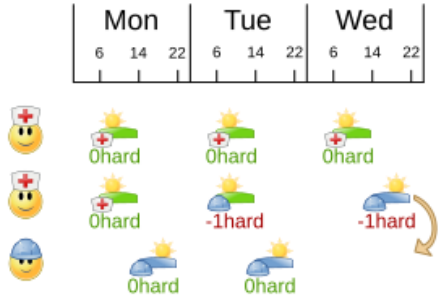
- A **possible solution** is any solution, whether or not it breaks any constraints.
- A **feasible solution** is one that does not break any hard constraints. Sometimes, there are no feasible solutions.
- An **optimal solution** is a solution with the highest score.
- The **best-found solution** is the solution with the highest score found by an implementation in a given amount of time. The best-found solution is likely to be feasible, and, given enough time, it's an optimal solution(No guarantee for heuristic/metaheuristics).


PlanningSolution
PlanningEntity
PlanningVariable 1
PlanningVariable 2
Value range provider
Score
Problem facts (optional)
ConstraintProvider

Summary

Incremental score calculation

Calculating deltas is much faster than calculating the entire solution's score.



Check every shift:

0 + 0 + 0 + 0 - 1 - 1 + 0 + 0

Required skill score: -2hard

BigO for n shifts

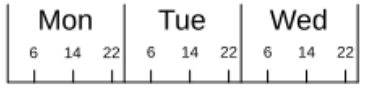
Constraint	From scratch	Incremental
Required skill	O(n)	O(1)
At most 1 shift/day	$O(n^2)$	O(n)
...	...	...

Calculation from scratch (easy java)

Check every shift again:

0 + 0 + 0 + 0 - 1 + 0 + 0 + 0

Required skill score: -1hard



Incremental calculation (java, CS)

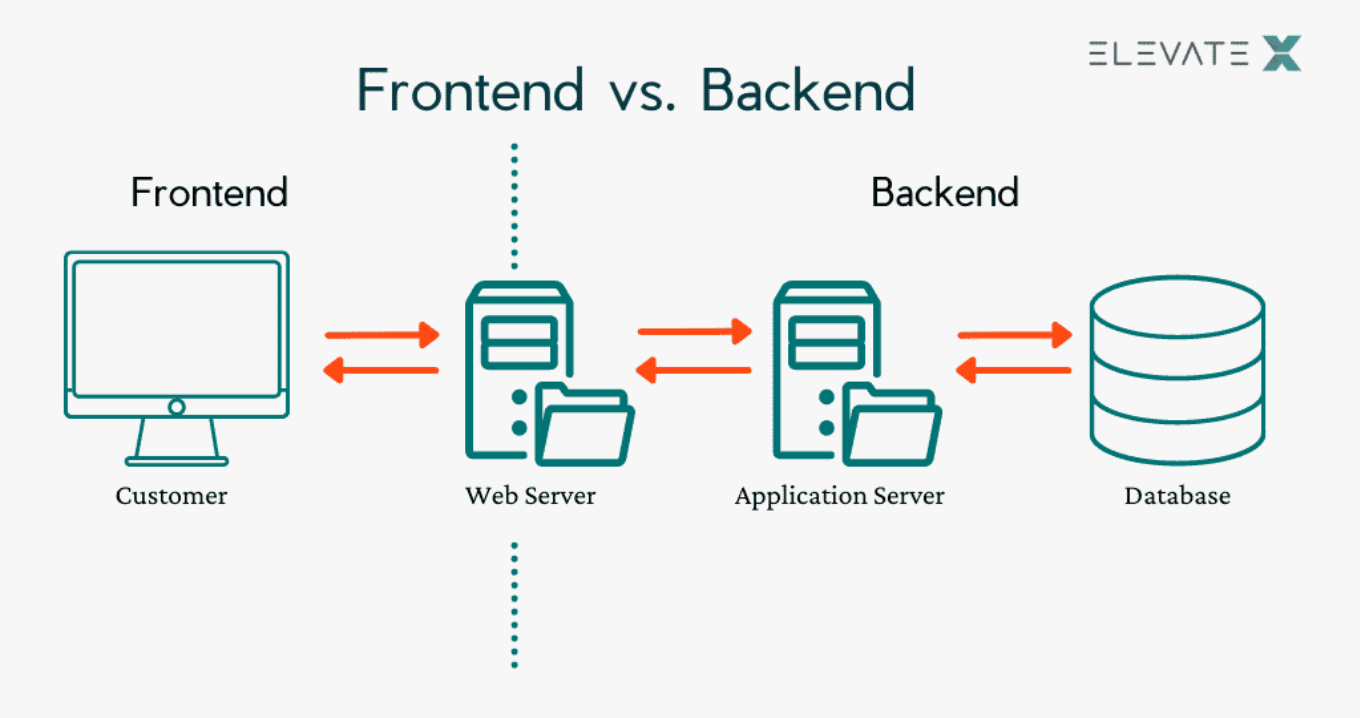
Check one shift (old & new)

-2 + 1 - 0

Required skill score: -1hard

Evaluaciones incrementales

Aplicaciones empresariales



https://elevatex.de/blog/it-insights/frontend-vs-backend-vs-fullstack-differences/

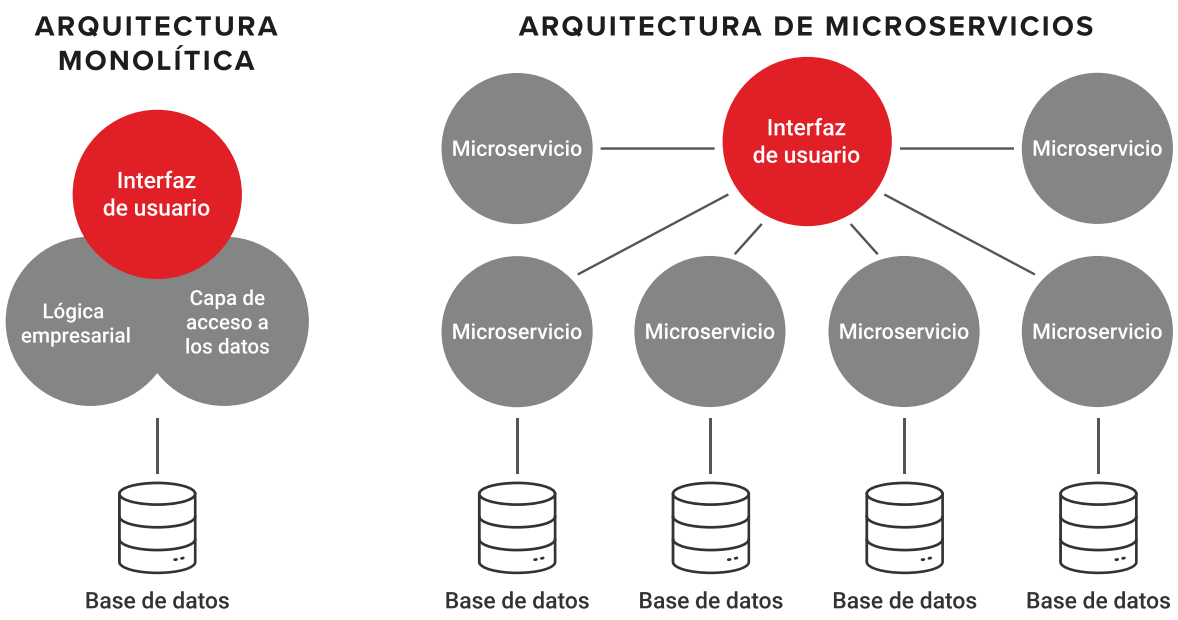
Backend

- Servidores de aplicaciones

GlassFish

- Microservicios

Arquitecturas



https://www.chakray.com/es/experiencia/microservicios/

Microservicios

"Pequeñas" piezas de software que funcionan de forma independiente y autónoma.

- Escalabilidad y mantenibilidad.
- Despliegue continuo y entrega rápida.
- Resistencia a fallos y aislamiento de errores.
- Flexibilidad tecnológica.

Appointments scheduling

Appointments

- Go to the PUB (4 hours)
- Go to the DOCTOR (1 hour)
- Go to the BARBER (70 minutes)

Start Times

- 3 pm
- 4 pm
- 5 pm
- 6 pm

Créditos a quien corresponda.

Appointments scheduling

- Go to the PUB at 3 pm – 7 pm?

Barber, Doctor?

There are overlapping.

Hard or soft constraints?

Appointments scheduling

Task: Appointment scheduling

Input:

- List of appointments with **name** and **duration**. E.g.
  - Go to the doctor, 1 hour.
- List of start times (e.g. 7am, 8 am, etc.)

Output:

- **Start time** of each appointment.

Coding time…

https://github.com/alex-cornejo/EPIO2024Java

Referencias

- Gendreau, M., & Potvin, J.-Y. (Eds.). (2019). *Handbook of Metaheuristics* (3a ed.). Springer International Publishing.
- Luke, S. (2013). *Essentials of Metaheuristics* (second). Lulu.
- Papadimitriou, C. H., & Steiglitz, K. (1998). *Combinatorial Optimization. Algorithms and Complexity* (1a ed.). Dover.
- Timefold-Team. (2024). *Timefold AI* (1.8.0) [Software]. https://timefold.ai/

Sesión de Preguntas

Gracias...