

# Progetto Solver per IHTC 2024

Questo progetto contiene un'implementazione di un solver per l'**Integrated Healthcare Timetabling Competition (IHTC) 2024**. Il codice è strutturato per caricare le istanze del problema, calcolare la validità e il costo di una data soluzione secondo tutti i vincoli hard e soft, e fornire una base per l'implementazione di un algoritmo di ottimizzazione come il **Reduced Variable Neighborhood Search (RVNS)**.

## Struttura del Progetto

Il codice è organizzato nelle seguenti cartelle principali per garantire chiarezza e modularità:

```
PROVA_PROBLEMA/
├── Data/           # Contiene i file JSON delle istanze del problema.
│   └── test01.json
├── instances/      # Contiene le classi per la gestione dei dati.
│   └── Hospital.py
├── solvers/        # Contiene la logica del solver e dei vincoli.
│   ├── constraints.py
│   └── RVNS_solver.py
└── main.py         # Entry point per eseguire il programma.
```

---

## Componenti del Codice

### 1. Cartella Data/

Questa cartella è designata a contenere tutti i file di istanza del problema in formato **JSON**. Ogni file descrive un intero scenario ospedaliero, incluse le risorse, i pazienti, gli occupanti e i pesi dei vincoli.

### 2. instances/Hospital.py

Questo file è il cuore della gestione dei dati.

## Classe Loader

La classe Loader è responsabile di leggere un file di istanza JSON e di caricare tutte le informazioni in memoria, organizzandole in strutture dati facilmente accessibili.

- **\_\_init\_\_(self, path: str):** Il costruttore prende il percorso di un file di istanza, lo legge e inizializza tutte le proprietà della classe.
- **Attributi Principali:**
  - self.data: Contiene l'intero dizionario JSON caricato dal file.
  - self.patients, self.occupants, self.nurses, self.rooms, self.surgeons, self.operating\_theaters: Liste di dizionari, ognuna contenente i dati grezzi per la relativa categoria.
  - self.patient\_dict, self.occupant\_dict, self.nurses\_dict, self.room\_dict, self.surgeon\_dict, self.operating\_theaters\_dict: Dizionari che mappano l'ID di ogni entità al suo oggetto, permettendo un accesso rapido ed efficiente (complessità  $O(1)$ ).
- **Metodi di Supporto (Helper Methods):** Contiene funzioni cruciali utilizzate dai calcolatori di vincoli per ottenere informazioni aggregate sulla soluzione.
  - get\_patient\_by\_id(): Recupera un paziente o un occupante tramite il suo ID.
  - get\_all\_patients\_in\_rooms(): Funzione fondamentale che restituisce un elenco unificato di **tutte le persone (pazienti e occupanti)** presenti in una stanza per ogni giorno della loro degenza. Questo garantisce che gli occupanti siano sempre inclusi nei calcoli dei vincoli.
  - get\_nurse\_assignments(): Crea un dizionario di facile accesso per sapere quale infermiere è assegnato a una stanza in un determinato turno.

## 3. solvers/

Questa cartella contiene la logica principale del solver.

### constraints.py

Questo file è puramente logico e contiene **l'implementazione dettagliata di ogni singolo vincolo** del problema. Ogni funzione qui è autonoma e riceve come input la soluzione corrente e l'oggetto Loader (chiamato hospital).

### RVNS\_solver.py

Questo file funge da "orchestratore". Utilizza una struttura a classi per organizzare il calcolo delle penalità in modo pulito, agendo come un **wrapper** per le funzioni logiche definite in constraints.py.

- **PenaltyWeights**: Classe base che carica i pesi delle penalità dal file di istanza.
- **PAS, NRA, SCP, GlobalPenalty**: Queste classi ereditano da PenaltyWeights e raggruppano i vincoli per categoria (es. PAS per i vincoli di ammissione dei pazienti). Ogni metodo in queste classi (es. PAS.h1\_no\_gender\_mix()) è una semplice chiamata alla funzione corrispondente in constraints.py. Questo approccio mantiene il codice ordinato e facile da leggere.
- **RVNS**: È la classe principale del solver.
  - **\_\_init\_\_**: Inizializza il solver caricando l'istanza e creando le istanze di tutte le classi di penalità.
  - **evaluate\_solution(self, solution: dict)**: Questo metodo è il cuore della valutazione. Calcola il costo totale di una soluzione chiamando tutti i metodi di penalità delle classi wrapper e sommando i risultati. Restituisce il costo totale e un dizionario con il dettaglio dei costi per ogni singolo vincolo.
  - **solve(self)**: Metodo designato a contenere la logica dell'algoritmo RVNS (attualmente un placeholder).

## 4. main.py

Questo è lo script principale per avviare il programma. Il suo ruolo è:

1. Definire il percorso del file di istanza da risolvere.
2. Inizializzare la classe RVNS.
3. Creare una **soluzione di esempio** (sample\_solution) per testare la correttezza del calcolo dei vincoli.
4. Chiamare il metodo evaluate\_solution() e stampare a schermo il costo totale e il dettaglio di ogni penalità.

---

## Come Eseguire il Codice

Per eseguire il programma e testare il calcolo dei vincoli su un'istanza:

1. **Assicurati che l'istanza sia presente**: Verifica che il file JSON dell'istanza (es. test01.json) si trovi nella cartella Data/.
2. **Configura main.py**: Apri il file main.py e assicurati che la variabile instance\_name punti al file che desideri testare.

Python

# In main.py

instance\_name = 'test01.json'

3. **Esegui da terminale:** Apri un terminale nella cartella radice del progetto (PROVA\_PROBLEMA/) ed esegui il seguente comando:  
Bash  
python main.py
  4. **Analizza l'output:** Il programma stamperà il costo totale della soluzione di esempio e un elenco dettagliato del valore di ogni singolo vincolo (H1-H7, S1-S8), permettendoti di verificare la correttezza della valutazione.  
--- Esempio di Output ---  
Initializing solver for instance: Data/test01.json  
  
--- Evaluating Sample Solution ---  
  
Total Cost of Sample Solution: 11004735  
Breakdown of costs/violations:
    - H1: 0
    - H2: 0
    - H7: 0
    - S1: 25
    - ... (e tutti gli altri vincoli)-----
- 

## Panoramica dei Vincoli

I vincoli sono divisi in due tipi principali:

- **Vincoli Hard (H):** Regole che **non devono mai essere violate**. Se una soluzione viola un vincolo hard, riceve una penalità altissima (**HARD\_VIOLATION\_PENALTY**) che la rende di fatto non valida.
- **Vincoli Soft (S):** Regole che **dovrebbero essere rispettate il più possibile**. La violazione di un vincolo soft aggiunge un costo all'obiettivo totale, calcolato in base all'entità della violazione e al peso (**weight**) definito nel file di istanza.

Tutte le funzioni ricevono due parametri principali: **solution** (la soluzione corrente da valutare) e **hospital** (l'oggetto **Loader** che contiene tutti i dati dell'istanza).

---



## Vincoli di Ammissione dei Pazienti (PAS)

Questi vincoli riguardano l'assegnazione dei pazienti alle stanze e la loro compatibilità.

### H1: `h1_no_gender_mix`

- **Scopo:** Impedire che pazienti di generi diversi condividano la stessa stanza nello stesso giorno.
- **Logica:**
  1. Usa la funzione `hospital.get_all_patients_in_rooms()` per ottenere un elenco di tutte le persone (pazienti e occupanti) presenti in ogni stanza, giorno per giorno.
  2. Per ogni coppia (`giorno`, `stanza`), crea un insieme (`set`) per memorizzare i generi dei pazienti presenti.
  3. Se la dimensione di questo insieme è maggiore di 1, significa che c'è un mix di generi, e viene contata una violazione.
- **Penalità:** `numero_di_violazioni * HARD_VIOLATION_PENALTY`

### H2: `h2_compatible_rooms`

- **Scopo:** Assicurarsi che un paziente non venga mai assegnato a una stanza che è nella sua lista di stanze incompatibili (`incompatible_room_ids`).
- **Logica:**
  1. Scorre ogni paziente nella soluzione.
  2. Controlla se la stanza assegnata (`p_sol['room']`) è presente nella lista delle stanze incompatibili del paziente.
  3. Se sì, conta una violazione.
- **Penalità:** `numero_di_violazioni * HARD_VIOLATION_PENALTY`

### H7: `h7_room_capacity`

- **Scopo:** Garantire che il numero di persone in una stanza non superi mai la sua capacità massima in nessun giorno.
- **Logica:**
  1. Utilizza `hospital.get_all_patients_in_rooms()` per creare un dizionario che conta quante persone ci sono in ogni stanza per ogni giorno.
  2. Confronta questo conteggio con la capacità della stanza (`hospital.room_dict[r_id]['capacity']`).
  3. Se il numero di occupanti supera la capacità, conta una violazione.
- **Penalità:** `numero_di_violazioni * HARD_VIOLATION_PENALTY`

### S1: `s1_mixed_age_penalty`

- **Scopo:** Minimizzare la differenza di età tra i pazienti che condividono una stanza.

- **Logica:**
    1. Usa `hospital.get_all_patients_in_rooms()` per raggruppare i pazienti per stanza e per giorno.
    2. Per ogni gruppo, converte le fasce d'età (es. "adult", "elderly") in indici numerici (0, 1, 2...).
    3. Calcola la differenza tra l'indice massimo e l'indice minimo nel gruppo. Questa differenza è il costo per quella stanza in quel giorno.
  - **Costo:** `somma_delle_differenze * peso_S1`
- 



## Vincoli di Assegnazione Infermieri (NRA)

Questi vincoli riguardano l'assegnazione degli infermieri alle stanze e la qualità delle cure.

### S2: `s2_minimum_skill_level`

- **Scopo:** Assicurarsi che l'infermiere assegnato a una stanza abbia il livello di competenza (`skill_level`) minimo richiesto dai pazienti in quella stanza, per ogni turno.
- **Logica:**
  1. Crea una mappa di facile accesso degli incarichi degli infermieri (`nurse_assignments`).
  2. Per ogni paziente/occupante in ogni stanza, per ogni giorno e ogni turno, confronta lo skill dell'infermiere assegnato con lo skill minimo richiesto dal paziente.
  3. Se lo skill dell'infermiere è inferiore, la differenza tra i due valori viene aggiunta al costo.
- **Costo:** `somma_delle_differenze_di_skill * peso_S2`

### S3: `s3_continuity_of_care`

- **Scopo:** Minimizzare il numero totale di infermieri diversi che si prendono cura di un singolo paziente durante tutta la sua degenza, per garantire la continuità delle cure.
- **Logica:**
  1. Per ogni paziente (ammesso o occupante), crea un insieme (`set`) per memorizzare gli ID unici degli infermieri che gli sono stati assegnati.
  2. Alla fine della sua degenza, la dimensione di questo insieme (il numero di infermieri unici) viene aggiunta al costo totale.
- **Costo:** `somma_del_numero_di_infermieri_unici_per_paziente * peso_S3`

### S4: `s4_maximum_workload`

- **Scopo:** Evitare che un infermiere sia sovraccaricato di lavoro in un turno.
  - **Logica:**
    1. Calcola il carico di lavoro totale per ogni infermiere in ogni specifico turno ((`infermiere`, `giorno`, `turno`)). Questo carico è la somma dei `workload_produced` di tutti i pazienti nelle stanze a lui assegnate.
    2. Confronta questo carico totale con il carico massimo (`max_load`) che l'infermiere può sostenere in quel turno.
    3. Se il carico totale supera il massimo, la differenza viene aggiunta al costo.
  - **Costo:** `somma_del_carico_in_eccesso * peso_S4`
- 



## Vincoli di Pianificazione Chirurgica (SCP)

Questi vincoli regolano l'uso delle sale operatorie e il tempo dei chirurghi.

### H3: `h3_surgeon_overtime`

- **Scopo:** Impedire che un chirurgo lavori più del suo tempo massimo giornaliero di chirurgia.
- **Logica:**
  1. Per ogni giorno, calcola il tempo totale di chirurgia per ogni chirurgo sommando la `surgery_duration` di tutti i suoi pazienti ammessi quel giorno.
  2. Se il totale supera il `max_surgery_time` del chirurgo per quel giorno, conta una violazione.
- **Penalità:** `numero_di_violazioni * HARD_VIOLATION_PENALTY`

### H4: `h4_ot_overtime`

- **Scopo:** Assicurarsi che il tempo totale degli interventi in una sala operatoria (`OT`) non superi la sua capacità giornaliera.
- **Logica:**
  1. Per ogni giorno, calcola il tempo totale di utilizzo per ogni sala operatoria.
  2. Se il totale supera la `availability` della sala per quel giorno, conta una violazione.
- **Penalità:** `numero_di_violazioni * HARD_VIOLATION_PENALTY`

### S5: `s5_open_ots`

- **Scopo:** Minimizzare il numero di sale operatorie utilizzate ogni giorno per ridurre i costi.
- **Logica:**

1. Crea un insieme (**set**) di tuple uniche (**giorno, sala\_operatoria**).
  2. La dimensione di questo insieme rappresenta il numero totale di "aperture" di sale operatorie.
- **Costo:** `numero_di_aperture * peso_S5`

## S6: **s6\_surgeon\_transfer**

- **Scopo:** Minimizzare il numero di sale operatorie diverse a cui un chirurgo deve spostarsi nello stesso giorno.
  - **Logica:**
    1. Per ogni chirurgo, per ogni giorno, crea un insieme (**set**) che contiene gli ID delle sale operatorie a cui è stato assegnato.
    2. Se la dimensione dell'insieme è maggiore di 1, significa che il chirurgo si è dovuto spostare. Il costo è (`numero_di_sale - 1`).
  - **Costo:** `somma_dei_trasferimenti * peso_S6`
- 

## **Vincoli Globali**

Questi vincoli riguardano decisioni generali sulla pianificazione dei pazienti.

## H5: **h5\_mandatory\_unscheduled**

- **Scopo:** Garantire che **tutti i pazienti obbligatori** (`mandatory: true`) siano ammessi.
- **Logica:**
  1. Crea un insieme (**set**) con gli ID di tutti i pazienti ammessi nella soluzione.
  2. Scorre la lista dei pazienti dell'istanza e, se un paziente obbligatorio non è nell'insieme degli ammessi, conta una violazione.
- **Penalità:** `numero_di_pazienti_obbligatori_non_schedulati * HARD_VIOLATION_PENALTY`

## H6: **h6\_admission\_day**

- **Scopo:** Assicurarsi che un paziente venga ammesso nella finestra temporale corretta.
- **Logica:**
  1. Per ogni paziente, controlla che il suo `admission_day` sia maggiore o uguale al `surgery_release_day`.
  2. Se il paziente è obbligatorio, controlla anche che l'ammissione sia minore o uguale al `surgery_due_day`.
  3. Ogni violazione di queste condizioni viene contata.



- **Penalità:** `numero_di_violazioni * HARD_VIOLATION_PENALTY`

### S7: `s7_admission_delay`

- **Scopo:** Minimizzare il ritardo con cui un paziente viene ammesso rispetto alla sua prima data disponibile (`surgery_release_day`).
- **Logica:**
  1. Per ogni paziente ammesso, calcola la differenza: `admission_day - surgery_release_day`.
  2. Questa differenza (se positiva) viene aggiunta al costo.
- **Costo:** `somma_dei_giorni_di_ritardo * peso_S7`

### S8: `s8_unscheduled_optional`

- **Scopo:** Minimizzare il numero di **pazienti opzionali** (`mandatory: false`) che non vengono ammessi.
- **Logica:**
  1. Simile a H5, conta quanti pazienti opzionali non compaiono nell'elenco dei pazienti ammessi.
- **Costo:** `numero_di_pazienti_opzionali_non_schedulati * peso_S8`

## L'Algoritmo RVNS: Spiegazione dei Metodi

La classe RVNS implementa la meta-euristica Reduced Variable Neighborhood Search. Ecco una descrizione delle sue funzioni chiave.

### `solve(self)`

È il metodo principale che gestisce l'intero processo di ottimizzazione.

1. **Inizializzazione:** Avvia un timer e genera una soluzione iniziale tramite `_generate_initial_solution()`.
2. **Ciclo Principale:** Continua a cercare soluzioni migliori finché non scade il tempo limite.
3. **RVNS Loop:** All'interno del ciclo, implementa la logica RVNS:
  - **Shake:** Perturba la soluzione migliore trovata finora per diversificare la ricerca e uscire da ottimi locali, utilizzando `_shake()`.
  - **Local Search:** Tenta di migliorare intensivamente la

soluzione perturbata esplorando i suoi vicinati, utilizzando `_local_search()`.

- **Accettazione:** Se la soluzione migliorata è migliore di quella globale, la aggiorna e ricomincia la ricerca dal primo vicinato ( $k=1$ ). Altrimenti, passa al vicinato successivo ( $k=k+1$ ).

4. **Restituzione:** Alla fine, restituisce la migliore soluzione trovata.

### **`_generate_initial_solution(self)`**

Crea una prima soluzione di partenza con una semplice strategia "greedy":

- Schedula tutti i pazienti **obbligator**i nella prima combinazione valida di giorno/stanza/sala che trova.
- Ignora i pazienti opzionali (saranno aggiunti successivamente dall'algoritmo).
- Assegna il primo infermiere disponibile a ogni turno che richiede personale. L'obiettivo è avere un punto di partenza, anche se di bassa qualità.

### **Le Strutture di Vicinato (`_neighborhood_...`)**

Sono il cuore dell'esplorazione. Ogni funzione definisce una "mossa" o una piccola modifica che può essere applicata a una soluzione per crearne una nuova, molto simile.

- **`_neighborhood_change_patient_room`:** Sceglie un paziente a caso e gli assegna una nuova stanza casuale (ma compatibile).
- **`_neighborhood_change_patient_day`:** Sceglie un paziente a caso e cambia il suo giorno di ammissione all'interno della sua finestra temporale valida.
- **`_neighborhood_reschedule_unscheduled`:** Cerca un paziente opzionale non ancora schedato e prova ad inserirlo nella soluzione in una posizione casuale.
- **`_neighborhood_change_nurse_assignment`:** (Attualmente un placeholder) In una versione avanzata, modificherebbe le assegnazioni degli infermieri per ottimizzare i vincoli S2, S3, S4.

### **`_shake(self, solution, k)`**

Esegue la perturbazione. Prende una soluzione e le applica la mossa

definita dal  $k$ -esimo vicinato per un numero di volte pari a  $k$ . Più alto è  $k$ , più la soluzione viene "scossa" e allontanata dallo stato attuale.

#### **`_local_search(self, solution)`**

Esegue una ricerca locale intensiva, nota come **Variable Neighborhood Descent (VND)**. Esplora sistematicamente i vicini (da  $k=1$  a  $k_{\max}$ ) e applica la prima mossa che migliora il costo della soluzione. Appena trova un miglioramento, ricomincia il processo dal primo vicinato ( $k=1$ ) per sfruttare al massimo la nuova configurazione vantaggiosa.