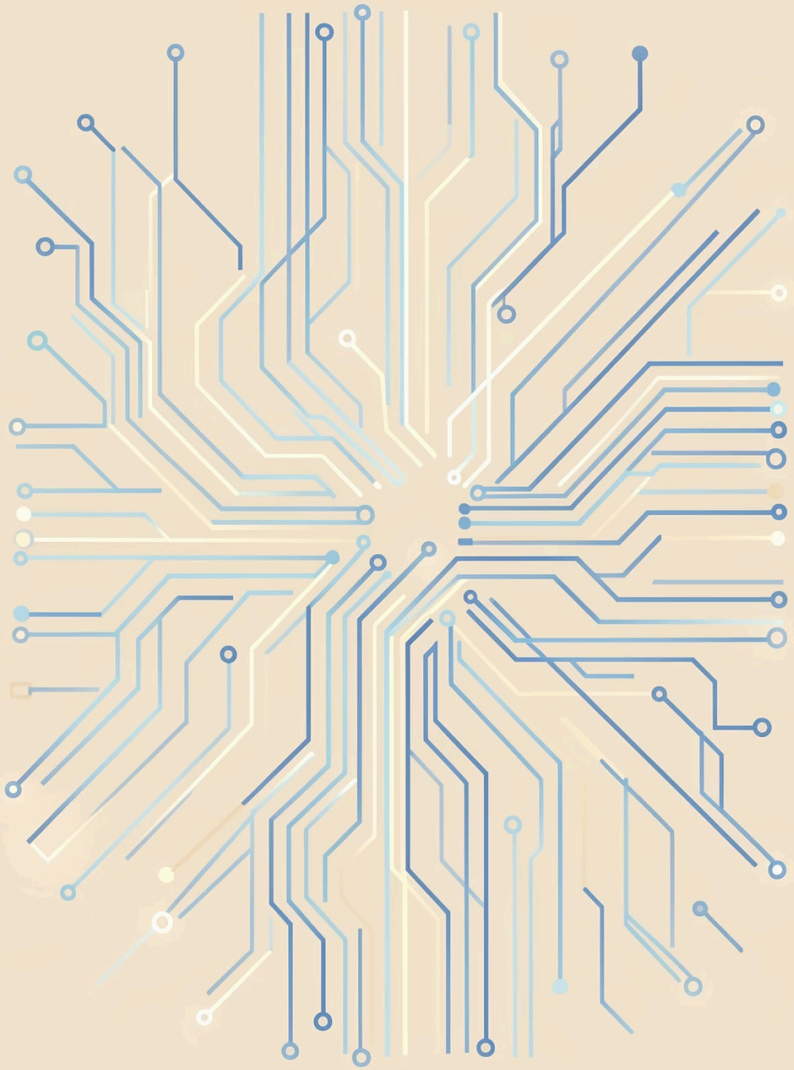


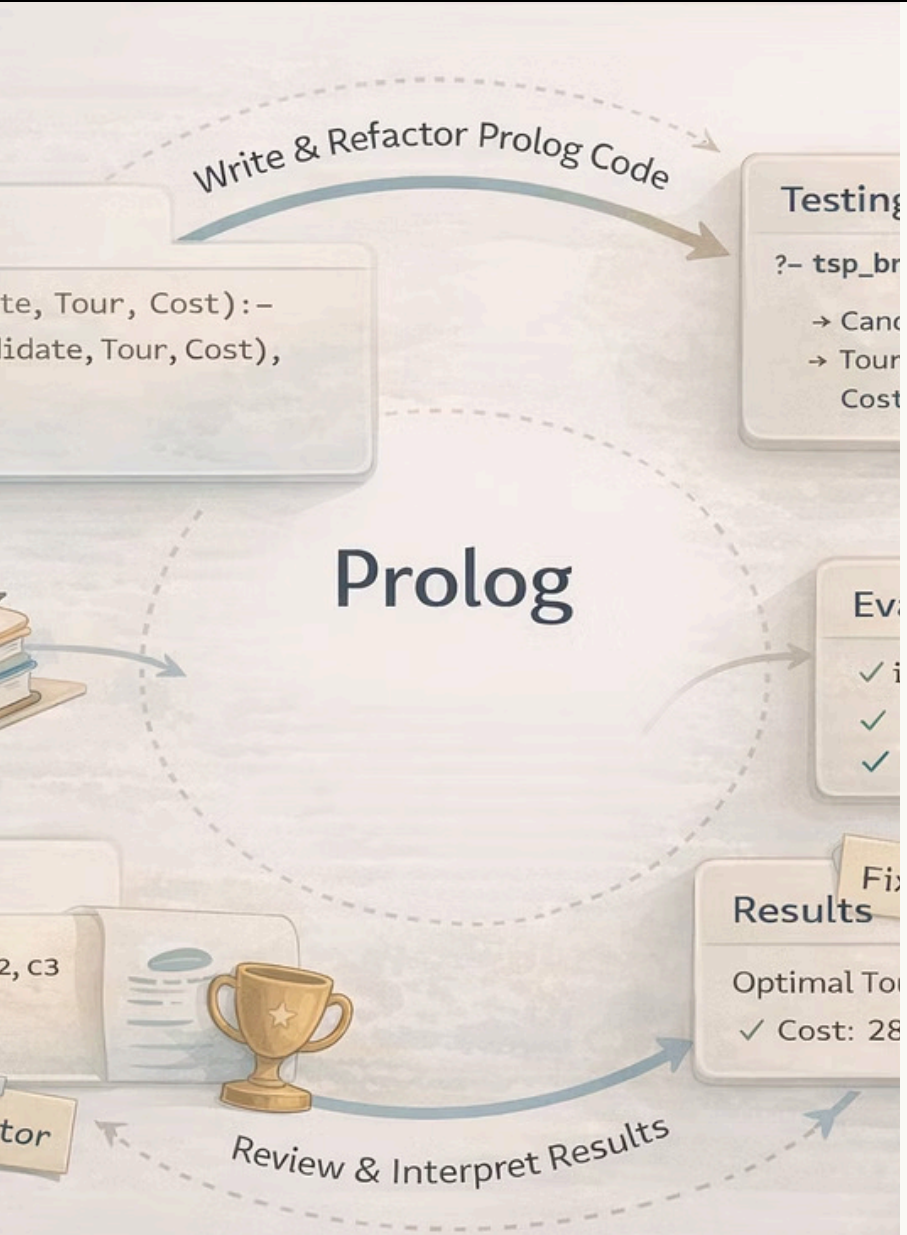
ЛОГІЧНЕ ПРОГРАМУВАННЯ ФУНКЦІОНАЛЬНІ МОВИ

Задача комівояжера (TSP)

Індивідуальне завдання: Реалізація та порівняння підходів до розв'язання TSP.

- **ПІБ:** Yermolovych Zakhar Maksymovych
- **Формат вводу:** `*.tsp` (матриця відстаней)
- **Формат виводу:** `cost=<int> tour=[1,...,1]`
- **Репозиторій: GitHub**

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)



Огляд роботи

У цій роботі продемонстровано комплексний підхід до вирішення Задачі комівояжера.

Коректна постановка

Єдиний формат I/O для Prolog і Haskell, що забезпечує узгодженість.

Детальні реалізації

Prolog: bruteforce (перестановки) та CLP(FD) (обмеження + оптимізація).

Haskell: bruteforce (permutations) як контрольний базис.

Відтворюваність результатів

CLI-команди запуску, тестові інстанси у `data/instances/`, smoke-тест `scripts/test_all.sh`.

Аналітична оцінка

Бенчмарки для N=5, 8, 10, 12 з короткими висновками щодо продуктивності.

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)

Постановка TSP

Задача комівояжера полягає у знаходженні найкоротшого маршруту, що відвідує кожне місто рівно один раз і повертається у початкове місто.

Вхідні дані

- N міст
- Матриця відстаней N×N:
  - Цілі невід'ємні значення
  - Діагональ рівна 0 (відстань від міста до себе)

Вихідні дані

- Замкнутий тур: `[1,...,1]`
- Мінімальна вартість: `cost` (сума ребер)

> **Ціль:** мінімізувати сумарну вартість маршруту.
>
> Індексація міст у проєкті: 1..N.

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)

Дані та I/O

Формат інстансу *.tsp

Файли *.tsp використовуються для опису вхідних даних задачі.

- Рядки, що починаються з #, ігноруються як коментарі.
- Перший значущий рядок містить ціле число N (кількість міст).
- Далі N рядків, кожен з яких містить N цілих чисел, розділених пробілами або табуляціями, що представляють матрицю відстаней.

Валідація: Програма перевіряє розмірність N×N, невід'ємність значень та рівність 0 на діагоналі.

Приклад (n5_demo.tsp)

```
# tsp-matrix v1
5
0 2 9 10 7
2 0 6 4 3
...
```

Формат виводу

Результат роботи програми виводиться у стандартизованому форматі:

```
cost=21 tour=[1,2,4,5,3,1]
```

Цей формат є єдиним для всіх реалізацій (Prolog та Haskell) і полегшує автоматизоване тестування.

Складність і мотивація підходів

Задача комівояжера є класичною NP-складною проблемою, що демонструє експоненціальне зростання складності при збільшенні кількості міст.

Bruteforce підхід

Заснований на переборі всіх можливих перестановок міст від 2 до N. Кількість таких турів зростає факторіально `~(N-1)!`, що швидко робить повний перебір непрактичним для великих `N`.

CLP(FD) підхід

Використовує програмування в логіці обмежень над скінченними доменами (Constraint Logic Programming over Finite Domains). Задача формулюється як набір обмежень, а потім застосовуються евристики та алгоритми мінімізації для ефективного пошуку оптимального рішення.

Haskell Bruteforce

Реалізований як базовий bruteforce для забезпечення контрольованої точки порівняння продуктивності з Prolog-версіями.

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)

Prolog bruteforce: generate & test

Prolog-реалізація bruteforce використовує парадигму "generate and test" для знаходження оптимального маршруту.

Ідея алгоритму

- Фіксація старту та фінішу туру у місті 1 для уникнення дублікатів і спрощення перебору.
- Генерація всіх перестановок міст `[2..N]` за допомогою `permutation(Cities, Perm)`.
- Побудова повного туру: `[1|Perm] ++ [1]`.
- Обчислення вартості кожного туру за допомогою предикату `tour_cost/3`.

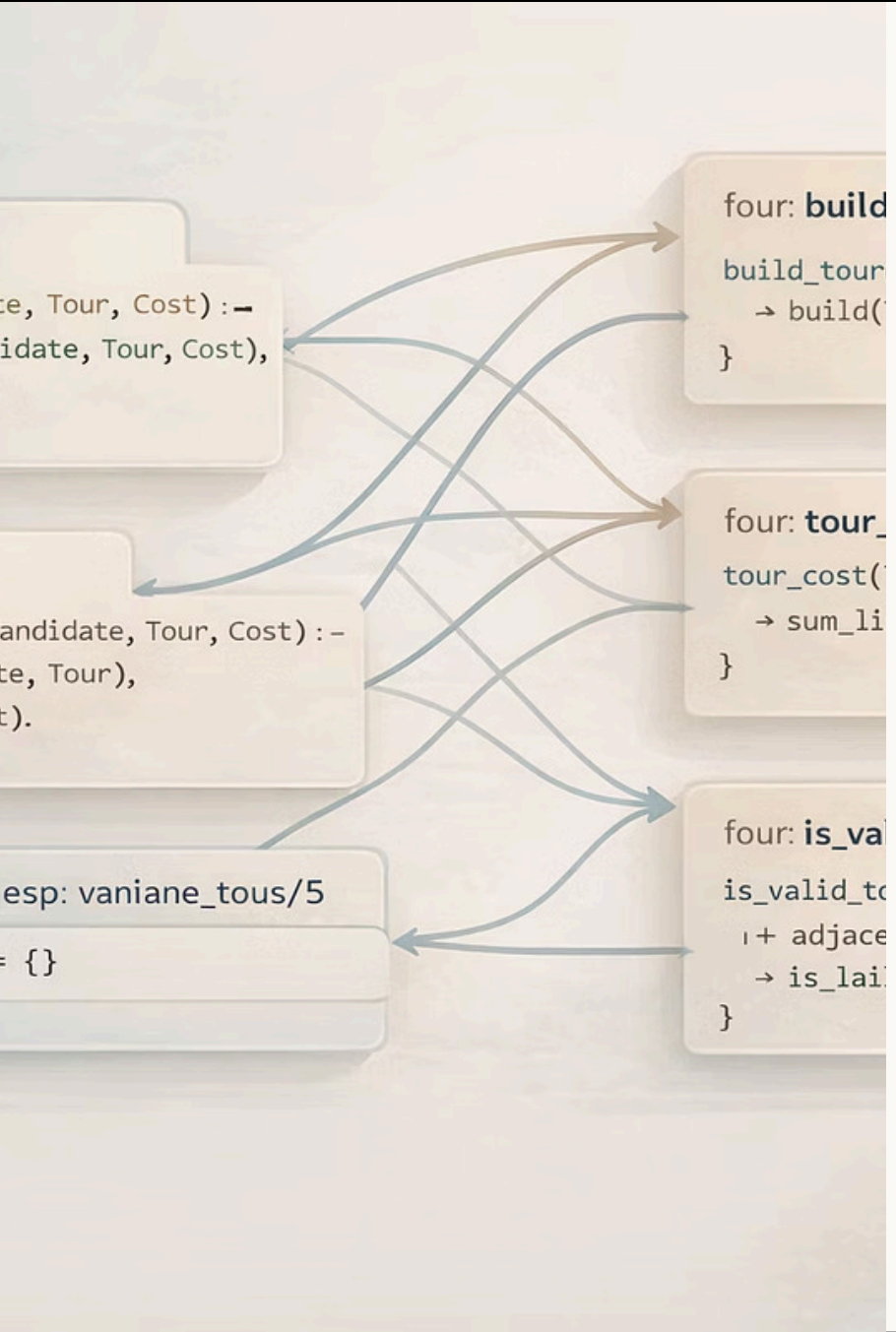
Вибір оптимуму

- Для вибору туру з мінімальною вартістю використовується агрегатний предикат `aggregate_all(min(Cost,Tour), Goal, min(...))`.
- Цей предикат ефективно повертає один оптимальний тур серед усіх згенерованих кандидатів.

Скорочений фрагмент коду (ідея кандидата)

```
permutation(Cities, Perm),
build_tour(Perm, Tour),
tour_cost(Matrix, Tour, Cost).
```

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)



Bruteforce: ключові предикати та модулі

Prolog-реалізація bruteforce базується на кількох ключових предикатах, розподілених між модулями для кращої організації коду.

1

SRC/PROLOG/TSP_BRUTEFORCE_YERMOLOVYCH.PL

- `tsp_bruteforce/3`: Головний предикат, що ініціює пошук та знаходить оптимальний тур через `aggregate_all/3`.
- `candidate_tour/4`: Недетермінований генератор кандидатів турів.
- `build_tour/2`: Забезпечує коректне замикання туру `[1,...,1]`.

2

SRC/PROLOG/TSP_COMMON_YERMOLOVYCH.PL

- `tour_cost/3`: Обчислює сумарну вартість заданого туру на основі матриці відстаней.
- `is_valid_tour/2`: Перевіряє валідність туру (використовується переважно для внутрішніх тестів та налагодження).

Цей поділ дозволяє легко розширювати та підтримувати код, забезпечуючи чітке розмежування відповідальності.

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)

Prolog CLP(FD): постановка через обмеження

Підхід CLP(FD) перетворює TSP на задачу задоволення обмежень, де пошук рішення відбувається значно ефективніше.

Змінні Succs[1..N]

Кожна змінна Succs[I] представляє наступне місто після міста I. Домени цих змінних обмежуються діапазоном 1..N.

circuit/1 обмеження

Обмеження circuit(Succs) гарантує, що значення Succs формують єдиний гамільтонів цикл, який відвідує кожне місто рівно один раз.

Обчислення вартості

Вартість маршруту розраховується за допомогою element/3 для отримання вартості ребер з матриці відстаней, яка попередньо сплющується у Flat список. Потім sum/3 агрегує ці вартості.

Оптимізація пошуку

Предикат once(labeling([ffc, min(Cost)], Succs)) використовується для пошуку рішення. ffc (first-fail-first-choose) є евристикою вибору змінної, а min(Cost) активує пошук оптимального рішення з мінімальною вартістю.

Представлення туру

Хоча Succs представляє тур як функцію наступників, для виведення він конвертується у список міст [1,...,1].

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)

CLP(FD): Зв'язок `Succs <=> Cost` через `element/3`

Ключовим моментом у CLP(FD) є ефективне зв'язування змінних-наступників із загальною вартістю маршруту. Це досягається за допомогою предикату element/3.

- Сплющення матриці: Матриця відстаней Matrix (N×N) сплющується в одномірний список Flat довжиною N*N. Це дозволяє звертатися до вартості ребра за індексом.
- Визначення індексу: Для кожного міста I та його наступника Si (тобто Succs[I]) обчислюється індекс Idx в списку Flat за формулою Idx #= (I−1)*N + Si.
- Зв'язок Di з Flat: Предикат element(Idx, Flat, Di) встановлює, що Di є вартістю ребра від міста I до міста Si.
- Сумарна вартість: Всі окремі вартості ребер Ds підсумовуються за допомогою sum(Ds, #=, Cost), що задає загальну вартість туру Cost.

Фрагмент обмежень (скорочено)

```
nth1(I, Succs, Si),
Idx #= (I-1)*N + Si,
element(Idx, Flat, Di).
sum(Ds, #=, Cost).
```

> ### Де читати деталі
>
> - docs/clpfd_explained.md
> - tsp_clpfd_Yermolovych.pl (коментарі + приклади)

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)

CLP(FD): Відновлення туру з successor-подання

Після того, як CLP(FD) знаходить оптимальний набір змінних `Succs`, що описує наступне місто для кожного міста, необхідно відновити повний список туру у форматі `[1, ..., 1]`.

- `Succs` **як функція:** Список `Succs` можна інтерпретувати як функцію, де `Succs[Current]` дає наступне місто.
- **Початок з міста 1:** Процес відновлення туру починається з міста 1, оскільки це фіксована точка старту/фінішу.
- **Покрокове відстеження:** Виконується `N` кроків, де на кожному кроці визначається `Next = Succs[Current]`, поки не будуть відвідані всі міста.
- **Формування туру:** Результатом є список `Tour = [1, ..., 1]` довжини `N+1`, що представляє замкнутий маршрут.
- **Замикання туру:** Обмеження `circuit/1` гарантує, що тур автоматично замкнеться, повертаючись до початкового міста 1.

Ідея предикату `succs_to_tour/2` (скорочено)

```
succs_to_tour(Succs, Tour) :-
    walk(1, N, Succs, Visited),
    Tour = [1|Visited].
```

Предикат `walk/4` відповідає за ітеративне проходження містами, використовуючи масив `Succs` для визначення наступного кроку, та збирає список відвіданих міст.

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)

Відтворюваність та автоматичне тестування

Для забезпечення надійності та коректності роботи проєкту впроваджено систему автоматичного тестування.

Автоматизована перевірка

Проєкт підтримує перевірку через скрипт `./scripts/test_all.sh`.

Коректність модулів

Перевіряється правильність завантаження всіх Prolog-модулів.

Запуск інстансів

Виконується запуск `main` для різних інстансів (bruteforce, clpfd).

Збіг вартості

Контролюється збіг вартості, отриманої bruteforce та CLP(FD) підходами.

Перевірка документації

Виконується груба перевірка стилю на відсутність помилок у документації.

Результат тесту

При успішному проходженні тестів виводиться `[ALL OK]`.

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)

Висновки

Алгоритмічний підхід

Brute Force

Факторіальна складність перебору.

Декларативний підхід

CLP(FD)

Опис задачі через **обмеження**.

Circuit

`circuit/1` гарантує **гамільтонів цикл**.

Оптимізація

`min(Cost)` знаходить **оптимум**.

Тестування

Автоматичне тестування підтверджує **коректність** та **відтворюваність**.

> CLP(FD) демонструє перевагу декларативного підходу над явним перебором.

TSP · SWI-Prolog (bruteforce + CLP(FD)) · Haskell (bruteforce)