Факултет техничких наука

Универзитет у Новом Саду

Паралелне и дистрибуиране архитектуре и језици

Проблем трговачког путника у Раст програмском језику

Аутор:
Алекса Ђукић

Индекс:
E2 84/2024

5. јул 2025.

Сажетак

У овом раду разматра се проблем трговачког путника и његово решавање применом *Held–Karp* алгоритма заснованог на динамичком програмирању. Алгоритам је имплементиран у програмском језику Раст, како у секвенцијалној, тако и у паралелној верзији. Посебна пажња посвећена је изазовима паралелизације и управљања нитима. Експерименти су спроведени на скуповима различитих величина, а резултати показују да паралелна имплементација постиже значајна убрзања код већих инстанци проблема, док код мањих утицај паралелизације остаје занемарљив.

Садржај

Списак изворних кодова

Списак слика

Списак табела

1 Увод

Проблем трговачког путника је класичан проблем у комбинаторној оптимизацији, који поставља питање: "Ако имамо листу градова и удаљеност између сваког пара градова, који ја најкраћи пут којим можемо обићи све градове и вратити се у град из којег смо кренули?". Овај, проблем је добро познат у теорији рачунарских наука и спада у класу *NP-hard* проблема [3]. Постоје бројна алгоритамска решења овог проблема, од којих већина нуди апроксимативна решења у разумним временима извршавања, док су егзактна решења погодна за коришћење само за ограничен број градова.

У овом раду као метод решавања биће примењен *Held-Karp* алгоритам [1], који постиже егзактно решење уз бољу временску сложеност у односу на наивни приступ, а такође оставља простор за скраћивање времена извршавања техником паралелизације. Имплементација алгоритма изведена је у Раст програмском језику, који представља изузетно ефикасан алат за развијање конкурентних програма.

2 *Held-Karp* алгоритам

Held-Karp или *Bellman-Held-Karp* је алгоритам заснован на динамичком програмирању представљен 1962. године као решење за проблем трговачког путника. Основна идеја заснива се на проналажењу најкраћих путања за подскупове градова растуће величине. Решења за мање подскупове користе се као основ за израчунавање најкраћих путања за веће подскупове, све док се не обради комплетан скуп градова и не добије оптимална путања која обухвата све градове и враћа се у почетни.

2.1 Опис и мотивација алгоритма

Градови су нумерисани са $1, 2, ...n$, где ће 1 бити произвољно изабран као почетни град (будући да је решење кружна путања неће бити битно одакле се креће). Алгоритам почиње тако што се за сваки подскуп $S \subseteq \{2, ..., n\}$ и сваки град $e \in S$, рачуна најкраћа путања од града 1 до града e пролазећи кроз све градове из скупа S. Ова дистанца се обележава са $g(S, e)$, а $d(u, v)$ представља раздаљину између градова u и v.

За подскупе S са мање од 2 елемента, одређивање најкраће дистанце је тривијално. Подскупи са 2 или више елемената захтевају евалуацију више различитих путања како би се дошло до најкраће.

Узмимо случај где знамо да је путања $1 \to 2 \to 3 \to 4$ краћа од $1 \to 3 \to 2 \to 4$. У том случају путања $1 \to 2 \to 3 \to 4 \to 5$ мора бити краћа од $1 \to 3 \to 2 \to 4 \to 5$. Уколико нам је задатак да дођемо најкраћим путем од града 1 до града 6, пролазећи кроз градове $\{2, 3, 4, 5\}$, а знамо да ће се ова путања завршити са $5 \to 6$, онда сигурно знамо да је тражена путања управо $1 \to 2 \to 3 \to 4 \to 5 \to 6$. Наравно, алгоритам не може само да претпостави да ће се путања завршити са $5 \to 6$, већ је потребно поставити сваки елемент из скупа $\{2, 3, 4, 5\}$, као претпоследњи на путањи, и израчунати колико је дугачак пут за сваки од ових случајева, а затим сачувати најкраћу путању. Ово значи да ћемо приликом рачунања најкраћих путања кроз 7 различитих градова, на располагању имати податак $g(\{2, 3, 4, 5, 6\}, 6)$.

Алгоритам ће итеративно проналазити најкраће путање кроз подскупове градова, почевши од $|S| = \emptyset$ па до $|S| = n$. Када се пронађу све најкраће путање, потребно је пронаћи ону која је најкраћа након повратка у град 1.

Цео алгоритам описан је изворним кодом 1.

2.2 Алгоритамска сложеност

Held-Karp алгоритам има експоненцијалну временску сложеност $\theta(2^n n^2)$, што је значајно боље од суперекспоненцијалног *brute-force* алгоритма $\theta(n!)$. Са друге стране, *Held-Karp* захтева $\theta(n^2)$ простора за складиштење израчунатих путања, док

function algorithm TSP(G, n)
    for k := 2 **to** n **do**
        $g(\{k\}, k) := d(1, k)$
    end for

    for s := 2 **to** n-1 **do**
        for all $S \subseteq \{2, ..., n\}, |S| = s$ **do**
            for all $k \in S$ **do**
                $g(S, k) := min_{m \neq k, m \in S}[g(S \backslash \{k\}, m) + d(m, k)]$
            end for
        end for
    end for

    $opt := min_{k \neq 1}[g(\{2, 3, ..., n\}, k) + d(k, 1)]$
    return (opt)
end function

Изворни код 1: Псеудокод за *Held-Karp* алгоритам

наивни алгоритам захтева само $\theta(n^2)$ за чување података о растојањима између градова.

3 Имплементација

У овом поглављу описана је имплементација *Held-Karp* алгоритма у Раст програмском језику. Прва имплементација је секвенцијална и служи као полазна тачка за анализу перформанси. Након тога, примењене су Раст-ове конструкције за конкурентно програмирање, како би се алгоритам адаптирао за рад у вишенитном режиму. Циљ ове адаптације је смањење укупног времена извршавања и ефикасније коришћење рачунарских ресурса.

3.1 Секвенцијална имплементација

Основна логика алгоритма имплементирана је кроз функцију `held_karp_seq` која као параметре прима матрицу растојања између градова `distances`, број градова `n` і помоћну мапу `cities` за потребе мапирања индекса у матрици на назив града.

У изворном коду 2 на линији 6, дефинисана је променљива `shortest_paths` која чува парове кључ-вредност, где су и кључ и вредност по две вредности типа `usize`. Кључ чине два елемента: први је подскуп градова кроз које се пролази, представљен као битмаска, а други је индекс града у којем се та путања завршава. Вредност повезана са овим кључем садржи два податка: дужину најкраће путање која креће из града 0, пролази кроз дате градове и завршава се у наведеном граду, као и индекс града који је претходио последњем. Тај индекс се користи при реконструкцији целе путање након завршетка алгоритма.

Мапа се првобитно попуњава свим растојањима између града 0 и осталих градова, а затим се итеративно формирају све могуће комбинације од $2, 3, ..., n$ градова, и најкраће путање траже и попуњавају у мапу помоћу функције `evaluate_subset_seq`. Њен код се види у изворном коду 3

Ова функција бира, редом, по један град из подскупа као дестинацију, а затим проналази најкраћу путању до њега и смешта резултат у мапу.

3.2 Паралелна имплементација

3.2.1 Почетна запажања

Основна идеја паралелизације извршавања овог алгоритма заснива се на паралелном израчунавању најкраћих путања за различите подскупове градова исте величине. Оваква стратегија је могућа јер је за израчунавање путање кроз k градова потребно претходно израчунати све најкраће путање кроз све подскупове величине $k-1$. Управо ова зависност представља и ограничење – неопходно је да сва израчунавања за подскупове одређене величине буду завршена пре него што започне обрада

```rust
fn held_karp_seq(
    distances: &Grid<usize>,
    n: usize,
    cities: &HashMap<usize, String>,
) -> (usize, String) {
    let mut shortest_paths: HashMap<(usize, usize), (usize, usize)> = HashMap::new();
    for i in 1..n {
        shortest_paths.insert((1 << i - 1, i), (distances[(0, i)], 0));
    }

    for subset_size in 2..n {
        for subset in get_combinations(subset_size, n - 1) {
            evaluate_subset_seq(distances, &mut shortest_paths, subset);
        }
    }

    let (min_cost, path) = find_best_cost_path(distances, n, cities, shortest_paths);

    (min_cost, path)
}
```

Изворни код 2: Главна секвенцијална функција

```rust
fn evaluate_subset_seq(
    distances: &Grid<usize>,
    shortest_paths: &mut HashMap<(usize, usize), (usize, usize)>,
    subset: usize,
) {
    indices_of_set_bits(subset)
        .into_iter()
        .for_each(|(idx, mask)| {
            shortest_paths.insert(
                (subset, idx),
                find_min_cost_path_via_subset_seq(&distances, &shortest_paths, subset ^ mask, idx),
            );
        })
}

fn find_min_cost_path_via_subset_seq(
    distances: &Grid<usize>,
    shortest_paths: &HashMap<(usize, usize), (usize, usize)>,
    intermediate_nodes: usize,
    dest: usize,
) -> (usize, usize) {
    indices_of_set_bits(intermediate_nodes)
        .into_iter()
        .map(|(idx, _)| {
            let cost = shortest_paths[&(intermediate_nodes, idx)].0 + distances[(idx, dest)];
            (cost, idx)
        })
        .min()
        .unwrap()
}
```

Изворни код 3: Помоћне секвенцијалне функције

већих подскупова. Анализом алгоритма приказаног у изворном коду 2, може се уочити да је могуће паралелизовати извршавање петље која се налази у реду 12, док петља из реда 11 мора остати секвенцијална.

3.2.2 Наиван приступ

Узимајући у обзир наведена ограничења, најинтуитивнији приступ паралелизацији био би покретање засебне нити за сваки подскуп исте величине, при чему би се чекало да се све нити заврше пре него што се приступи обради подскупова већих димензија. Међутим, овакав приступ није практичан, јер би довео до креирања изузетно великог броја нити, што би преоптеретило системске ресурсе. На слици 1 може се видети да само за подскупове од 9 или 10 елемената, из скупа од 19 градова, постоји више од 90.000 различитих комбинација, што јасно илуструје неизводљивост овог приступа у пракси.

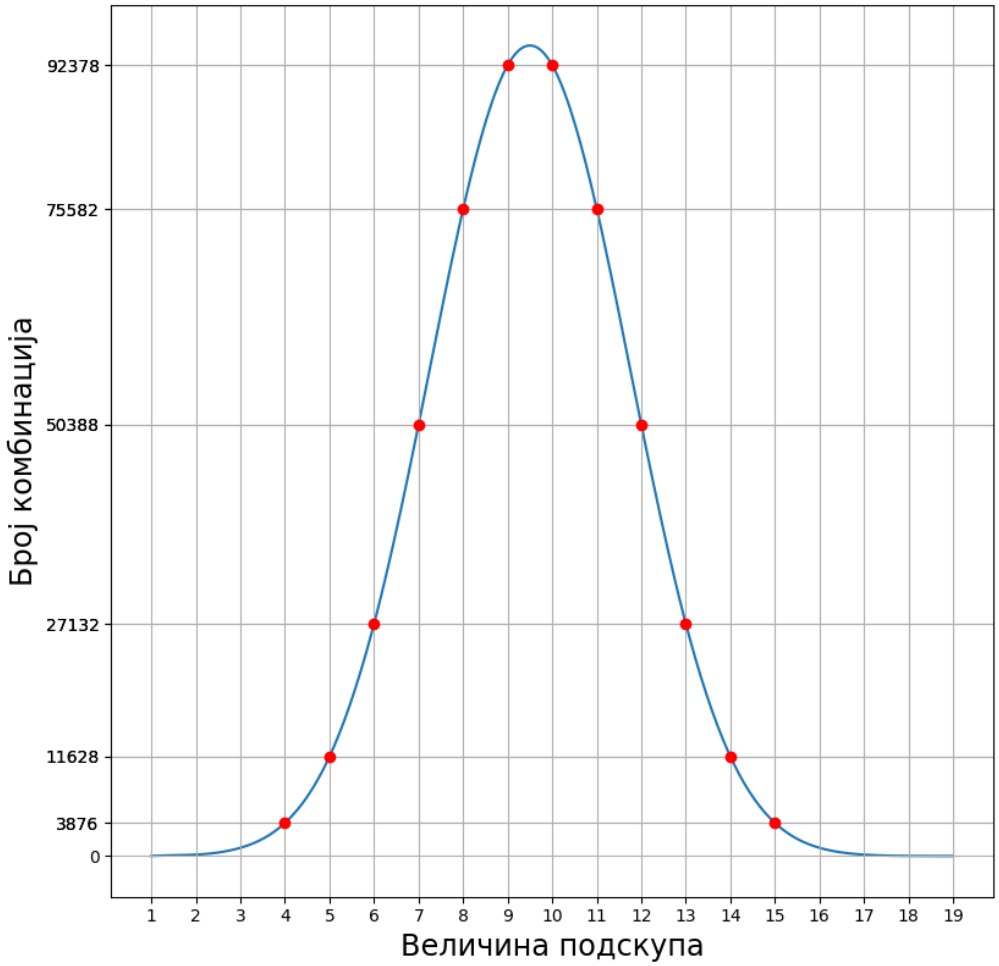
```rust
let mut channel_num = 0;
for subset_size in 2..n {
    let shortest_paths_arc =
     ↪  Arc::new(shortest_paths.clone());
    let threads = (0..num_threads)
        .map(|_| {
            let distances_clone = distances_arc.clone();
            let shortest_paths_clone =
             ↪  shortest_paths_arc.clone();
            let (sender, receiver) = channel();

            (
                thread::spawn(move || {
                    let mut result = HashMap::new();
                   while let Ok(subset) = receiver.recv() {
                        result.extend(evaluate_subset_par(
                            &distances_clone,
                            &shortest_paths_clone,
                            subset,
                        ));
                    }
                    result
                }),
                sender,
            )
        })
        .collect::<Vec<_>>();

    for subset in get_combinations(subset_size, n - 1) {
        threads[channel_num].1.send(subset).unwrap();
        channel_num = (channel_num + 1) % num_threads;
    }

    for thread in threads {
        drop(thread.1);
        shortest_paths.extend(thread.0.join().unwrap());
    }
}
```

Изворни код 4: Главна паралелна функција



Слика 1: Однос величине подскупа и броја могућих комбинација

3.2.3 Базен нити

Значајно практичнији приступ проблему било би креирање базена нити (engl. *Thread pool*) где би био креиран одређен број нити (конкретно онолико колико има логичких језгара процесора), а са нитима и по један канал који би служио за примање задатака. Начин на који је ово одрађено види се на линијама 4-25 изворног кода 4. Главна петља би и даље итерирала кроз све могуће комбинације подскупова и додељивала сваки подскуп једној од нити, *round robin* методом распоређивања. [2]

3.2.4 Синхронизација нити

У линијама 32–35 изворног кода 4 јасно је приказано да, након расподеле посла за све подскупове одређене величине, главна нит мора да сачека да све остале нити заврше своје задатке пре него што настави са обрадом већих подскупова, као што је већ истакнуто у поглављу 3.2.1.

Имплементација користи метод *join* за синхронизацију нити, што подразумева да се приликом сваког повећања величине подскупа мора креирати нови базен нити. Поставља се питање: да ли је могуће избећи поновно стварање базена нити у свакој итерацији?

Одговор на ово питање није једноставан. Током имплементације испитане су различите стратегије синхронизације. Један од приступа заснивао се на употреби дељене бројачке променљиве, која се инкрементирала при свакој додели задатка нити, а декрементирала по завршетку обраде. Овај механизам је омогућавао да се сачека завршетак свих задатака пре покретања нове групе. Ипак, у пракси се показало да овај приступ доводи до значајног успоравања, због континуиране борбе нити за приступ заједничкој променљивој — без обзира на то да ли је синхронизација реализована помоћу *mutex*-а или атомичких операција.

У конкретном случају, где број задатака може достићи и преко 90.000 по итерацији, ови трошкови постају значајни. Испоставило се да је трошак поновног стварања нити занемарљив у односу на трошкове конкурентног приступа заједничкој променљивој. Стога, упркос интуитивној привлачности задржавања истих нити током више итерација, боље перформансе се у пракси постижу поновним креирањем нити у свакој итерацији петље.

3.2.5 Библиотеке за конкурентно програмирање

Важно је напоменути да у програмском језику Rust постоје бројне библиотеке као што су *rayon*, *tokio*, или *crossbeam*, које пружају напредне механизме за рад са нитима, као што су базени нити, паралелне итерације и ефикасне структуре за синхронизацију. Коришћењем тих библиотека могуће је значајно поједноставити имплементацију и побољшати перформансе.

Ипак, ова имплементација је намерно реализована без икаквих зависности од спољних библиотека — као таква, пружа јасну слику о основним изазовима конкурентног програмирања. Овакав приступ омогућава детаљно испитивање различитих стратегија синхронизације и управљања нитима, као и боље разумевање трошкова који се јављају у реалним условима.

4 Резултати

Алгоритам *Held–Karp* гарантује проналажење оптималне путање која обилази све дате градове тачно једном и враћа се у почетни град. У циљу евалуације тачности и перформанси, алгоритам је тестиран на скуповима од 8, 12, 16 и 20 градова. У наставку су приказани добијени резултати, укључујући оптималне путање, као и поређење времена извршавања између секвенцијалне и паралелне имплементације.

С обзиром на то да је алгоритам егзактне природе, резултати обе имплементације — секвенцијалне и паралелне — биће идентични, па се поређење односи искључиво на утицај паралелизације на перформансе извршавања.

4.1 Решења

Подаци коришћени за тестирање алгоритма обухватају 20 европских градова, са растојањима између сваког пара градова. Пре почетка рада алгоритма, ови подаци организовани су у 20×20 матрицу, из које је могуће бирати подскупове градова различите величине за потребе тестирања.

4.1.1 Скуп од 8 градова

Први тестни скуп садржи градове: Париз, Берлин, Праг, Беч, Будимпешту, Загреб, Венецију и Милано. Најкраћа путања кроз ове градове може се видети на слици 2, и износи 3943km.

Слика 2: Најкраћа путања кроз 8 градова

4.1.2 Скуп од 12 градова

Наредни скуп проширује претходни са 4 града: Цирих, Минхен, Франкфурт и Амстердам. На слици 3 види се путања која износи 4583km.

Слика 3: Најкраћа путања кроз 12 градова

4.1.3 Скуп од 16 градова

Скуп од 16 градова добија се додавањем Брисела, Лондона, Мадрида и Барселоне на претходни скуп. Оптимална путања дугачка је 6918km и видљива је на слици 4.

Слика 4: Најкраћа путања кроз 16 градова

4.1.4 Скуп од 20 градова

Највећи скуп на којем је алгоритам тестиран добија се проширењем претходног скупа са још 4 града: Лион, Женева, Рим и Фиренца. Решење овог проблема, на слици 5 је 7868km.

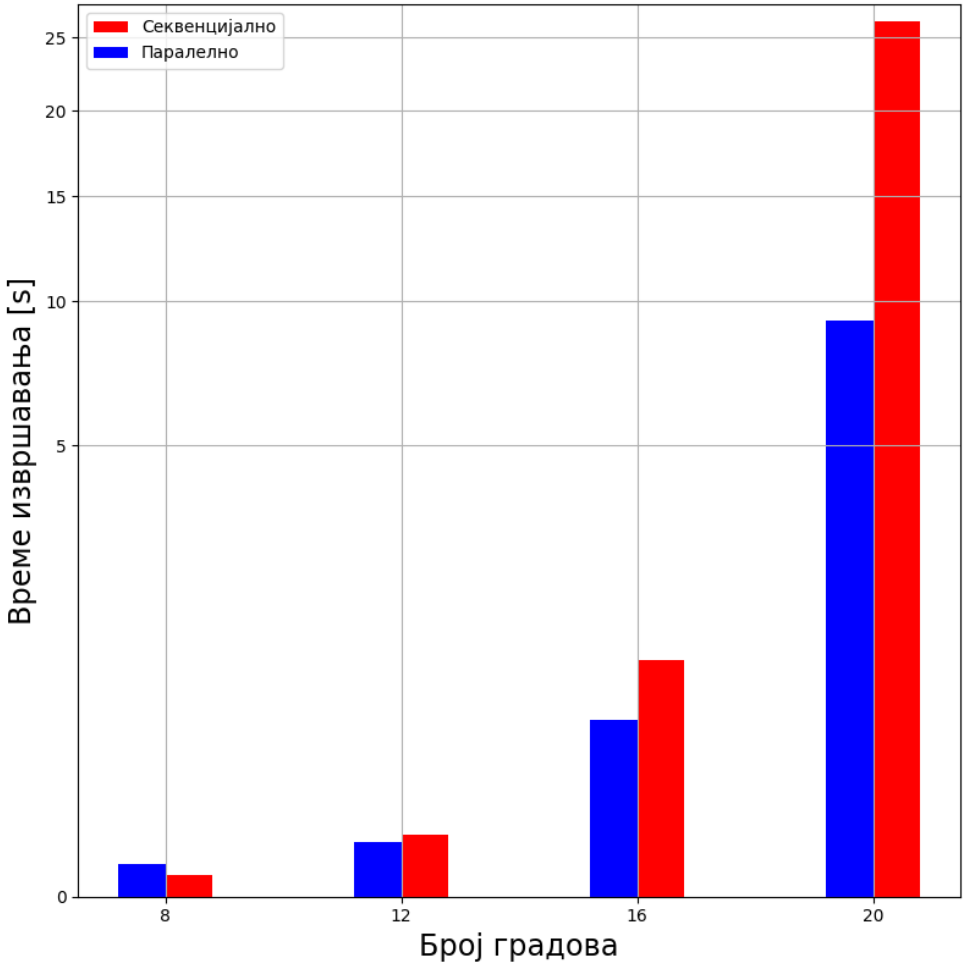
Слика 5: Најкраћа путања кроз 20 градова

4.2 Перформансе

Табела 1 приказује времена извршавања за сваки од тестних скупова градова, како за секвенцијалну тако и за паралелну имплементацију алгоритма. Извршавање је обављено на процесору са 6 физичких и 12 логичких језгара. Ради веће прегледности и лакше визуелне анализе, резултати су додатно представљени графички на слици 6.

	Секвенцијална	Паралелна
8 градова	2.6ms	6.8ms
12 градова	35.5ms	25.8ms
16 градова	1s	481.6ms
20 градова	26.2s	9.2s

Табела 1: Времена извршавања секвенцијалног и паралелног алгоритма



Слика 6: Поређење времена извршавања секвенцијалне и паралелне имплементације

На основу времена извршавања може се приметити да за мали број градова (≤ 8 паралелна имплементација постиже за нијансу спорије перформансе од секвенцијалне. Ово је последица додатног трошка повезаног са покретањем и управљањем нитима. Међутим, како се број градова повећава, време извршавања секвенцијалног алгоритма расте нагло, док паралелна верзија показује све израженије убрзање. Ово указује на то да се корист од паралелизације значајно испољава тек при већим и рачунски интензивнијим инстанцама проблема.

5 Закључак

На основу спроведене анализе и експеримената може се закључити да *Held–Karp* алгоритам, иако теоријски захтеван, представља поуздан метод за егзактно решавање проблема трговачког путника за мање и средње велике скупове градова. Имплементација у програмском језику Rust омогућила је детаљно испитивање изазова конкурентног програмирања без ослањања на спољне библиотеке, што је допринело бољем разумевању стратегија синхронизације и управљања нитима. Резултати показују да паралелна имплементација не доноси значајно убрзање код једноставних задатака, али пружа значајне добитке у перформансама како број градова расте. Ово указује на потенцијал паралелног извршавања у решавању комбинаторно сложених проблема, посебно када се примењује уз пажљиву контролу ресурса.

Библиографија

[1] Wikipedia. Held–Karp algorithm --- Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Held%E2%80%93Karp%20algorithm&oldid=1266011856, 2025. [Online; accessed 02-July-2025].

[2] Wikipedia. Round-robin scheduling --- Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Round-robin%20scheduling&oldid=1290674072, 2025. [Online; accessed 03-July-2025].

[3] Wikipedia. Travelling salesman problem --- Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Travelling%20salesman%20problem&oldid=1297255372, 2025. [Online; accessed 02--July-2025].