
Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

# Крушкалов алгоритам у рачунарству високих перформанси

*Аутор:*
Вељко Николић

*Индекс:*
E2 37/2024

2. фебруар 2026.

**Сажетак**

У овом раду анализирају се секвенцијална и паралелна реализација Крушкаловог алгоритма за проналажење минималног разапињућег стабла (*Minimum Spanning Tree, MST*). Крушкалов алгоритам спада у похлепне алгоритме и заснива се на избору ивица са најмањом тежином, при чему се води рачуна да њихово укључивање не доведе до појаве циклуса у графу.

Рад обухвата развој и анализу секвенцијалне и паралелне имплементације алгоритма. Паралелна имплементација ослања се на *OpenMP* библиотеку и намењена је извршавању на вишејезгарним процесорима.

Циљ истраживања био је да се утврди у којој мери се Крушкалов алгоритам може ефикасно паралелизовати и колики добитак у перформансама се може остварити у односу на класичну секвенцијалну верзију.

Резултати показују да Крушкалов алгоритам није могуће паралелизовати у довољној мери да се постигну значајна побољшања у перформансама. Приликом тестирања постигнути су 5-15% боље перформансе код паралелне имплементације у поређењу са секвенцијалном имплементацијом.

Изворни код пројекта доступан је на *GitHub*-у, на линку.

# Садржај

# Списак изворних кодова

# 1 Увод

Проблем проналажења минималног разапињућег стабла (енгл. *Minimum Spanning Tree, MST*) представља један од кључних проблема у теорији графова и алгоритмима, са значајним бројем практичних примена. Минимално разапињуће стабло омогућава повезивање свих чворова графа уз минималан укупни трошак, што је од посебне важности у областима као што су пројектовање рачунарских и комуникационих мрежа, оптимизација инфраструктурних система, анализа великих скупова података и решавање проблема у оперативним истраживањима. Са развојем савремених апликација, које све чешће подразумевају рад са великим и густим графовима, ефикасност алгоритама за решавање овог проблема постаје све значајнија.

Крушкалов алгоритам [1] је један од најчешће коришћених алгоритама за одређивање минималног разапињућег стабла. Заснива се на похлепној стратегији, при чему се ивице графа сортирају по тежини и затим поступно додају у решење, под условом да њихово укључивање не доведе до формирања циклуса. Уопштено, алгоритам се састоји од следећих корака:

1. У почетној фази алгоритма формира се скуп дисјунктних стабала, при чему сваки чвор графа представља засебну компоненту и сам себи је корен.

2. Све ивице графа се затим уређују у неопадајућем редоследу на основу својих тежина.

3. Након сортирања, алгоритам редом разматра ивице из добијене листе. За сваку ивицу проверава се да ли њени крајњи чворови припадају различитим компонентама. Уколико је тај услов испуњен, ивица се укључује у минимално разапињуће стабло, а одговарајуће компоненте се обједињују. У супротном, ивица се прескаче, јер би њено додавање довело до формирања циклуса.

4. Поступак се завршава када се сви чворови налазе у резултујућем стаблу. Резултујуће стабло увек садржи све чворове ($V$ чворова) и $V$-$1$ ивица.

Овај рад бави се развојем и анализом паралелних имплементација Крушкаловог алгоритма. Методологија решавања подразумева идентификацију делова алгоритма погодних за паралелно извршавање, као и њихову реализацију коришћењем *OpenMP* библиотеке за вишејезгарне процесоре. Детаљан опис предложених приступа и имплементационих решења дат је у посебном поглављу рада.

# 2 Имплементација

У овом поглављу приказана је имплементација посматраног алгоритма у програмском језику *C++*. Прво је приказан начин припреме улазних података, а затим

и имплементација самог алгоритма, прво у секвенцијалној изведби, а затим и у паралелној изведби (*OpenMP*).

## 2.1 Помоћне структуре и методе

Прво су приказане неке помоћне структуре и методе које помажу при имплементацији самог Крушкаловог алгоритма, али нису од кључног значаја у разумевању алгоритма.

### 2.1.1 Припрема улазних података

За припрему података написана је функција *generate_graph* приказана у примеру 1, која за улазне аргументе прима број чворова *num_of_vertices* и број ивица *num_of_edges* и враћа граф. Због сажетости, приказан је само потпис функције, јер сама имплементација функције није од кључног значаја за овај рад. Изворни код ове функције је могуће наћи на репозиторијуму пројекта, на линку.

```
1 Graph generate_graph(unsigned num_of_vertices, unsigned
  ↪  num_of_edges);
```

Изворни код 1: Функција за генерисање улазног графа

### 2.1.2 Дисјунктни скупови

Такође, имплементирана је помоћна класа која нам омогућава да рукујемо дисјунктним скупом ивица, који нам је потребан у имплементацији самог Крушкаловог алгоритма. Имплементација је приказана на примеру 2.

```cpp
class DisjointSet {
private:
    map<string, string> parent;
    map<string, int> rank_;

public:
    DisjointSet(const vector<string>& vertices) {
        for (const string& v : vertices) {
            parent[v] = v;
            rank_[v] = 0;
        }
    }

    string find(const string& vertex) {
        if (parent[vertex] != vertex) {
            parent[vertex] = find(parent[vertex]);
        }
        return parent[vertex];
    }

    void union_(const string& root1, const string& root2) {
        if (rank_[root1] > rank_[root2]) {
            parent[root2] = root1;
        } else if (rank_[root1] < rank_[root2]) {
            parent[root1] = root2;
        } else {
            parent[root2] = root1;
            rank_[root1]++;
        }
    }
};
```

Изворни код 2: Класа која дефинише дисјунктни скуп

### 2.1.3 Остале помоћне методе

На примеру 3 приказане су методе за екстраховање чворова и ивица из датог графа.

```
vector<Edge> extract_edges(Graph graph) {
    vector<Edge> edges;
    set<tuple<string, string, int>> seen;
    for (const auto& [vertex, neighbors] : graph) {
        for (const auto& [neighbor, weight] : neighbors) {
            if (seen.find({neighbor, vertex, weight}) ==
             ↪  seen.end()) {
                edges.push_back({vertex, neighbor, weight});
                seen.insert({vertex, neighbor, weight});
            }
        }
    }
    return edges;
}

vector<string> extract_vertices(Graph graph) {
    vector<string> vertices;
    for (const auto& [vertex, _] : graph) {
        vertices.push_back(vertex);
    }
    return vertices;
}
```

Изворни код 3: Функције за екстраховање чворова и ивица из датог графа

На примеру 4 приказана је имплементација алгоритма проналажења *MST-a*.

```cpp
vector<Edge> find_mst(vector<string> vertices, vector<Edge>
 ↪  edges) {
    DisjointSet disjointSet(vertices);

    // find edges
    vector<Edge> mst;
    for (const Edge& edge : edges) {
        string root_u = disjointSet.find(edge.u);
        string root_v = disjointSet.find(edge.v);

        if (root_u != root_v) {
            mst.push_back(edge);
            disjointSet.union_(root_u, root_v);
        }
    }

    return mst;
}
```

Изворни код 4: Функција за проналажење *MST-a*

## 2.2 Крушкалов алгоритам

Крушкалов алгоритам је имплементиран у 2 изведбе: секвенцијална и паралелна уз коришћење *OpenMP* библиотеке. У наставку су прво приказани заједнички делови имплементације, а затим и разлике у секвенцијалној и паралелној имплементацији.

Оно што је заједничко за обе имплементације јесте алгоритам за сортирање ивица, као што је приказано на примеру 5.

```cpp
void quicksort_edges(vector<Edge>& arr, int left, int right, bool parallel, size_t threshold) {
    if (left >= right) return;

    auto compare = [](const Edge& a, const Edge& b) {
        if (a.weight != b.weight) return a.weight < b.weight;
        if (a.u != b.u) return a.u < b.u;
        return a.v < b.v;
    };

    Edge pivot = arr[left + (right - left) / 2];
    int i = left, j = right;

    while (i <= j) {
        while (compare(arr[i], pivot)) i++;
        while (compare(pivot, arr[j])) j--;

        if (i <= j) {
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }

    bool should_parallelize = parallel && (right - left > threshold);

    if (should_parallelize) {
        #pragma omp task shared(arr)
        quicksort_edges(arr, left, j, parallel, threshold);

        #pragma omp task shared(arr)
        quicksort_edges(arr, i, right, parallel, threshold);

        #pragma omp taskwait
    } else {
        quicksort_edges(arr, left, j, parallel, threshold);
        quicksort_edges(arr, i, right, parallel, threshold);
    }
}
```

Изворни код 5: *QuickSort* алгоритам за сортирање ивица

### 2.2.1 Секвенцијална имплементација

Секвенцијална имплементација Крушкаловог алгоритма је у основи једноставна и састоји се од главне функције и неколико помоћних функција. На примеру 6 је приказана имплементација главне функције Крушкаловог алгоритма.

```
vector<Edge> kruskal_sequential(const Graph& graph) {
    // extract edges and deduplicate
    vector<Edge> edges = extract_edges(graph);

    // sort edges
    sequential_sort_edges(edges);

    // extract vertices
    vector<string> vertices = extract_vertices(graph);

    // find MST
    vector<Edge> mst = find_mst(vertices, edges);

    return mst;
}
```

Изворни код 6: Секвенцијална имплементација Крушкаловог алгоритма

У наставку је приказана имплементација функције *sequential_sort_edges* на примеру 7.

```
void sequential_sort_edges(vector<Edge> &edges) {
    if (edges.empty()) return;
    quicksort_edges(edges, 0, edges.size() - 1, false, 0);
}
```

Изворни код 7: Секвенцијална употреба *QuickSort* алгоритма

Као што се може видети на примеру 7, секвенцијална употреба *QuickSort* алгоритма се своди само на позивање основне функције *quicksort_edges*.

### 2.2.2 Паралелна *OpenMP* имплементација

Као и секвенцијална имплементација, паралелна имплементација користи помоћне функције и у основи је иста као и секвенцијална верзија. На примеру је при-

казана паралелна имплементација главне функције Крушкаловог алгоритма.

```
vector<Edge> kruskal_omp(const Graph& graph) {
    // extract edges and deduplicate
    vector<Edge> edges = extract_edges(graph);

    // sort edges
    parallel_sort_edges(edges);

    // extract vertices
    vector<string> vertices = extract_vertices(graph);

    // find MST
    vector<Edge> mst = find_mst(vertices, edges);

    return mst;
}
```

Изворни код 8: Паралелна имплементација Крушкаловог алгоритма

У наставку је приказана имплементација функције *parallel_sort_edges* на примеру 10.

```
void parallel_sort_edges(vector<Edge> &edges) {
    if (edges.empty()) return;

    const size_t SEQUENTIAL_THRESHOLD = 100;

    #pragma omp parallel
    {
        #pragma omp single
        quicksort_edges(edges, 0, edges.size() - 1, true,
        ↪  SEQUENTIAL_THRESHOLD);
    }
}

```

Изворни код 9: Паралелна употреба *QuickSort* алгоритма

Као што се може видети на примеру 10, паралелна употреба *QuickSort* алгоритма се своди само на позивање основне функције *quicksort_edges* уз одговарајуће *OMP*

директиве. Такође, дефинисана је и граница величине низа од које се низ сортира секвенцијално (*SEQUENTIAL_THRESHOLD*).

## 2.3 Тестирање

У наставку на примеру је приказан *main.cpp* у ком се дефинише начин тестирања имплементираних алгоритама.

```
int main() {
    cout << "Generating graph..." << flush;
    Graph graph = generate_graph(10000, 2500000);
    cout << " finished." << endl;

    cout << "Sequential..." << flush;
    auto start = chrono::high_resolution_clock::now();
    vector<Edge> mst_sequential = kruskal_sequential(graph);
    auto end = chrono::high_resolution_clock::now();
    auto seq_time =
     ↪  chrono::duration_cast<chrono::milliseconds>(end -
     ↪  start).count();
    cout << " finished, time: " << seq_time << " ms"<< endl;

    cout << "Parallel..." << flush;
    start = chrono::high_resolution_clock::now();
    vector<Edge> mst_omp = kruskal_omp(graph);
    end = chrono::high_resolution_clock::now();
    auto omp_time =
     ↪  chrono::duration_cast<chrono::milliseconds>(end -
     ↪  start).count();
    cout << " finished, time: " << omp_time << " ms"<< endl;

    printf("Speedup: %.2fx\n", (double)seq_time / omp_time);

    return 0;
}
```

Изворни код 10: Паралелна употреба *QuickSort* алгоритма

У *main* функцији прво се генерише улазни граф. Затим се позива секвенцијална имплементација Крушкаловог алгоритма, а после ње и паралелна имплементација. Мери се време извршавања за обе функције и на крају се исписују резултати.

Главни програм се покреће командом *g++ -o main -fopenmp ./src/main.cpp && ./main* или покретањем скрипте *run.sh*. Очекивани излаз изгледа као што је приказано на примеру 11.

```
Generating graph... finished.
Sequential... finished, time: 12788 ms
Parallel... finished, time: 10645 ms
Speedup: 1.20x
```

Изворни код 11: Пример

# 3 Резултати тестирања и закључак

Резултати тестирања у просеку показују да је паралелна имплементација 5-15% бржа од секвенцијалне имплементације. Међутим, резултати нису конзистентни и нису довољно значајни. Разлог је што Крушкалов алгоритам у основи веома секвенцијалан и готово је немогуће паралелизовати га. Једина паралелизабилна целина јесте сортирање ивица. Међутим, та целина сама по себи не заузима довољно велик део процесорског времена у алгоритму, те у већини случајева паралелизација сортирања не доноси значајна побољшања у перформансама.

# Библиографија

[1] Stanford University. Cs 106b, lecture 24 dijkstra's and kruskal's.