Bipartitní souvislý podgraf hranově ohodnoceného grafu s maximální vahou

Jorge Zuňiga
NI-PDP
ČVUT LS 2022/2023

Úvod

Cílem práce bylo navrhnout algoritmus nacházející bipartitní souvislý podgraf hranově ohodnoceného grafu s maximální vahou, tedy podmnožinu hran F takovou, že podgraf G(V,F) je bipartitní, souvislý a součet ohodnocení jeho hran je maximální mezi všemi bipartitními souvislými podgrafy grafu G(V,E).

Pro tento úkol byla navržena 4 řešení s různými přístupy a stupni paralelizace. Nejdříve byl vytvořen algoritmus čistě sekvenční. Následně algoritmy využívající OpenMP, funkční a datový paralelismus. Na závěr byl vytvořen algoritmus kombinující MPI a OpenMP.

Implementace

Algoritmus byl implementován v jazyce C++ s důrazem na OOP. V některých částech algoritmu byla pro snadné používání využita knihovna STL.

Části algoritmů, které přímo operovaly nad objekty s prvky z knihovny STL byly později nahrazeny statickými polemi. To výrazně pomohlo v rychlosti výpočtů natolik, že bylo potřeba využít větších testovacích vstupů.

Další vylepšení, které jemně zrychlilo výpočty bylo využití co nejmenších datových typů, tedy `uint8_t`, `uint16_t`,… tam kde se daly použít.

Vstup

Vstupem programu je textový soubor obsahující informace o grafu. Ukázka grafu graf_24_23.txt:

```
24
    0   81   87   80   86   96   89  100   82  119  108   82  107  117   81  106  112  118   92   86   86   92  106  107
   81    0  102  109   92  120  116  115  119  109  118   87  109   84  104  120  106  108   80   95  112  107   91  113
   87  102    0   94   83  112  106   89  120  118  118  106   99  106  120  100  103  115  101   91  112  108   81  118
   80  109   94    0   93   83  106   80   83   80  114  112   92  108   85   95   99  111  106   98  110  105   83   88
   86   92   83   93    0   92   85  110  115  120   90   87  113   80   89  111   93   92   98   96   97   98   89   88
   96  120  112   83   92    0  112  118   95   88   96   85  115  115  115  101  120   84  114   84  114  108   85   86
   89  116  106  106   85  112    0  116  119   86   86  111   99  100   88  115  117  108   86   86  100   83  101  108
  100  115   89   80  110  118  116    0  101  108  104   97  104   85   99  108  119  105  103  108  111  111  103  111
   82  119  120   83  115   95  119  101    0  117  111  101   98   90  111   94   86  100  100   94  120  103  117  110
  119  109  118   80  120   88   86  108  117    0   86  104   93  103   87  100   81  117  100  109  102   88   99   92
  108  118  118  114   90   96   86  104  111   86    0  113   91   91  104  112  109  114  104   82   82   83  103   98
   82   87  106  112   87   85  111   97  101  104  113    0   85   87   95  117   93  119   89  119   88  112   81   86
  107  109   99   92  113  115   99  104   98   93   91   85    0   91  110  108   99   90   82   92  101   93  118   94
  117   84  106  108   80  115  100   85   90  103   91   87   91    0   83  111  118   85  115   83  110   93   90  120
   81  104  120   85   89  115   88   99  111   87  104   95  110   83    0  108   86   94  108   97   92  116   88   94
  106  120  100   95  111  101  115  108   94  100  112  117  108  111  108    0   82  100   85  112   80   96  116   94
  112  106  103   99   93  120  117  119   86   81  109   93   99  118   86   82    0  117   90   91   91   93   84   90
  118  108  115  111   92   84  108  105  100  117  114  119   90   85   94  100  117    0  101  119   95   92   93  105
   92   80  101  106   98  114   86  103  100  100  104   89   82  115  108   85   90  101    0   91   80  113  106  111
   86   95   91   98   96   84   86  108   94  109   82  119   92   83   97  112   91  119   91    0   90  118  108   98
   86  112  112  110   97  114  100  111  120  102   82   88  101  110   92   80   91   95   80   90    0   93  110  120
   92  107  108  105   98  108   83  111  103   88   83  112   93   93  116   96   93   92  113  118   93    0   99  104
  106   91   81   83   89   85  101  103  117   99  103   81  118   90   88  116   84   93  106  108  110   99    0   80
  107  113  118   88   88   86  108  111  110   92   98   86   94  120   94   94   90  105  111   98  120  104   80    0
```

Formálně:

- n = přirozené číslo představující počet uzlů grafu G, 150 > n >= 10
- G(V,E) = jednoduchý neorientovaný hranově ohodnocený souvislý graf o n uzlech a průměrném stupni k, váhy hran jsou z intervalu <80,120>

Definice

Graf G(V,E) je bipartitní, jestliže můžeme rozdělit množinu uzlů V na disjunktní podmnožiny U a W tak, že každá hrana v G spojuje uzel z U s uzlem z W. Jinak řečeno, bipartitní graf je možné uzlově obarvit 2 barvami.

Výstup

Výstupem algoritmu jsou disjunktní množiny uzlů X a Y, seznam ohodnocených hran a součet vah ohodnocených hran.

```
Loaded file ../inputs/23_20
==============================================
Result of: ../inputs/23_20
----------------------------------------------
RED: {0, 1, 5, 8, 10, 12, 13, 16, 18, 21, 22}
BLUE: {2, 3, 4, 6, 7, 9, 11, 14, 15, 17, 19, 20}
----------------------------------------------
EDGES: {(7, 22), 120}, {(5, 20), 120}, {(4, 22), 120}, {(1
----------------------------------------------
Weights sum = 12902
----------------------------------------------
Recursive calls: 1,474,463,121
Took: 0h:0m:56.191s
==============================================
```

Ukázkový výstup ze sekvenčního algoritmu pro **graf_23_20.txt**:

Výstupy sekvenčního algoritmu obsahují i počet rekurzivních volání algoritmu. Ostatní implementace tuto metriku neobsahují, protože její výpočet je značně zpomaloval.

Sekvenční algoritmus

Implementace je postavena na přístupu OOP. Byly vytvořeny třídy `InputHandler`, `ProblemInstance`, `SolutionState` a `Edge`.

Třída **InputHandler** obsahuje metody pro načtení vstupů a zpracování parametrů. Program při použití flagu `-h`, `--help` vypisuje pomocnou hlášku a jinak přijímá `--file <seznam souborů>` a `--folder <seznam složek>` pro hromadné zpracování vstupů.

Třída Edge

Třída `Edge` reprezentuje hranu a využívá co nejméně prostoru pomocí datového typu `uint8_t`.

```
class Edge {
public:
    uint8_t u; // < 150
    uint8_t v; // < 150
    uint8_t weight; // 80 <= weight <= 120
};
```

Třída SolutionState

Třída **`SolutionState`** reprezentuje stav řešení. V poli colors si uchovává seznam barev hran grafu. Tento seznam je identifikátorem stavů. Další důležitou proměnnou je `edge_index`, který je indexem hrany, která je právě přidávána/odebírána.

```
class SolutionState {
public:
    color_t colors[MAX_VERTICES] = {NO_COLOR};
    uint8_t num_of_vertices = 0; // < 150
    uint16_t edge_index = 0; // < 11175
    uint16_t used_edges = 0; // < 11175
    uint32_t cost = 0;
    uint32_t sum_cost_all = 0;
    Edge * edges = nullptr;
    uint16_t edges_size = 0; // < 11175
    uint32_t edges_total_weight = 0;
};
```

Dále obsahuje informace potřebné pro prohledávání dalších stavů, výpočtu cen, prořezávání a řadu dalších funkcí jako například:

- `bool isLeaf()`
    - Vrací true pokud je stav listem ve stromu řešení.
- `bool isBetterThan(SolutionState state)`
    - Porovnává ceny stavů.
- `bool isBipartite()`
    - Vrací true pokud je graf ve stavu bipartitní.
- `bool isConnected()`
    - Vrací true pokud je graf ve stavu souvislý.
- `void skipEdge()`
    - Přeskakuje hranu.
- `void addEdge()`
    - Přidává hranu.

Třída ProblemInstance

Třída `ProblemInstance` repezentuje instanci problému, tedy uchovává informace o grafu, jeho velikosti, udržuje si výchozí a nejlepší stav. Dále drží informace o vstupu, jako třeba jméno souboru, ze kterého byl vstup načten, počet rekurzivních volání a čas kdy byl výpočet započat.

```
class ProblemInstance {
private:
    Edge edges[MAX_EDGES];
    uint16_t edges_size;
    SolutionState initial_state;
    SolutionState best_state;
    // Metrics
    string input_name;
    uint64_t recursive_calls;
    time_point start_time;
};
```

Třída je navržena tak aby poskytovala co nejjednodušší rozhraní, a tak v sekci public obsahuje pouze funkce:

- `string getInputName()`
  - Vrací jméno vstupního souboru.
- `uint32_t getBestStateCost()`
  - Vrací cenu nejlepšího řešení.
- `void findMaxConnectedBipartiteSubgraph()`
  - Nalezne řešení.

Poslední ze zmíněných je funkce spouštějící výpočet. Tato funkce nejdříve zkontroluje, zda vstupní graf již není validním řešením. V případě, že jím je, uloží výsledek do stavu `best_state` jinak spouští funkci `findBestStateDFS(initial_state)`. A nakonec vypíše výsledek na stdout.

V privátní sekci se nachází funkce:

- `void printResult()`
  - Vypíše výsledek na `stdout`.
- `bool noBetterSolutionPossible(SolutionState state)`
  - Vrací true pokud ze stavu již není možné získat lepší než současně nejlepší řešení. Slouží k prořezávání stavového prostoru. Stavy prořezává na základě dvou výpočtů.
  - Nejdříve kontroluje, zda při přidání všech zbylých hran je možné dostat vyšší cenu.
  - Následně kontroluje, zda při přidání všech zbylých hran je možné dostat souvislý podgraf.
- `void findBestStateDFS(SolutionState state)`
  - Ze zadaného stavu prohledává prostor stavů řešení algoritmem DFS.

Funkce `findBestStateDFS()` je nejdůležitější částí celého programu. Tato funkce prohledává prostor stavů algoritmem BB-DFS.

Funkce před samotným rekurzivním zanořováním kontroluje, zda bylo nalezeno lepší řešení. Pokud ano, uloží jej.

```
if (state.isLeaf()) {
    if (state.isConnected() and state.isBetterThan(best_state)) {
        best_state = state;
        return;
    } else {
        return;
    }
}
```

Následuje prořezávání stavového prostoru pro současný stav.

```
if (noBetterSolutionPossible(state))
    return;
```

Poté následuje nejdůležitější část programu, tedy samotné barvení vrcholů a rekurzivní zanořování. Algoritmus si nejdříve uloží indexy hran, nad kterými pracuje.

```
int u = edges[state.edge_index].u;
int v = edges[state.edge_index].v;
```

Dále vyzkouší veškeré kombinace barev tak, aby zachoval bipartitu řešení. Jednou z nich je například:

```
if ((state.colors[u] == RED and state.colors[v] == RED) or
    (state.colors[u] == BLUE and state.colors[v] == BLUE)){
    {
        SolutionState opt_skip = state;
        opt_skip.skipEdge();
        findBestStateDFS(opt_skip);
    }
} else if ...
```

Kde v případě, že oba vrcholy mají stejnou barvu se hrana přeskakuje.

Podobně se zpracovávají další případy. Po skončení této funkce je ve proměnné `best_state` uložen nejlepší stav, neboli stav s maximálním součtem vah hran.

Výkon sekvenčního řešení

Optimalizace popsané v úvodu sekce implementace, mají za následek to, že testovací vstupní data z courses jsou vyřešena prakticky okamžitě. Ukázkou za všechny je největší z grafů ze souboru graf_17_10.txt.

Vstupní soubor	Čas	Počet rekurzivních volání
graf_17_10.txt	0h:0m:0.97s	5,438,301

OpenMP - Funkční paralelismus

Funkční paralelismus s knihovnou OpenMP se od sekvenčního řešení příliš neliší. Program navíc přijíma parametr `-t <number of threads>` určující počet vláken.

Program již nepočítá počet rekurzivních zanoření.

Funkce `findMaxConnectedBipartiteSubgraph()` je doplněna o direktivu pro knihovnu OpenMP.

```
void findMaxConnectedBipartiteSubgraph() {
    start_time = chrono::high_resolution_clock::now();
    initial_state.edges = this->edges;
    if (initial_state.isBipartite() and
        initial_state.isConnected()) {
        best_state = initial_state;
    } else {
        initial_state.resetSolution();
        #pragma omp parallel num_threads(number_of_threads)
            #pragma omp single
                findBestStateDFS(initial_state);
    }
    printResult();
}
```

Ve funkci findBestStateDFS() je kolem kontroly nejlepšího výsledku vytvořena kritická sekce protože proměnná `best_state` je sdílená.

```
if (state.isLeaf()) {
    if (state.isConnected() and state.isBetterThan(best_state)) {
        #pragma omp critical
        {
            if(state.isBetterThan(best_state))
                best_state = state;
        }
        return;
    } else {
        return;
    }
}
```

Každé rekurzivní zanoření je označeno direktivou #pragma omp task.

```
if ((state.colors[u] == RED and state.colors[v] == RED) or
    (state.colors[u] == BLUE and state.colors[v] == BLUE)){
    {
        SolutionState opt_skip = state;
        opt_skip.skipEdge();
        #pragma omp task
        {
            findBestStateDFS(opt_skip);
        }
    }
} else if ...
```

OpenMP - Datový paralelismus

Implementace datového paralelismu se od sekvenční verze liší o něco více. Program opět přijímá parametr pro počet vláken a nepočítá počet rekurzivních volání.

Zásadní rozdíl je ve funkci `findMaxConnectedBipartiteSubgraph()`, která je doplněna o funkci `generateStatesQueue()` a direktivu `#pragma omp parallel for`.

```
void findMaxConnectedBipartiteSubgraph() {
    start_time = chrono::high_resolution_clock::now();
    initial_state.edges = this->edges;
    if (initial_state.isBipartite() and
        initial_state.isConnected()) {
            best_state = initial_state;
    } else {
        initial_state.resetSolution();
        generateStatesQueue();
        #pragma omp parallel for num_threads(number_of_threads)
        for (auto & solution_state : solution_states_queue) {
            findBestStateDFS(solution_state);
        }
    }
    printResult();
}
```

Funkce `generateStatesQueue()` spouští funkci `findBestStateBFS()`, který podobně jako `findBestStateDFS()` prohledává stavový prostor. BFS verze prohledává prostor až do určité velikosti fronty.

```
void generateStatesQueue() {
    this->solution_states_queue.push_back(initial_state);
    while (solution_states_queue.size() < solutionQueueLimit()) {
       findBestStateBFS(solution_states_queue.front());
       solution_states_queue.erase(solution_states_queue.begin());
    }
}
```

Po dosažení požadované velikosti fronty se paralelně spouští sekvenční řešení ve funkci `findMaxConnectedBipartiteSubgraph()`.

MPI (+ funkční paralelismus)

Implementace s využitím MPI funguje na principu Main-Worker. Main proces, podobně jako v datovém paralelismu, nejdříve naplní frontu stavy pomocí algoritmu BFS a poté Worker procesům posílá tyto stavy k dokončení. V případě, že už není co posílat, čeká na doběhnutí všech Worker procesů a poté vypíše nejlepší nalezený výsledek.

V implementaci přibyla statická třída `MyMpi`, sloužící jako obal pro funkce MPI a přibyla třída `Graph`, reprezentující graf seznamem hran.

Dále přibyly serializační prvky pro třídu `SolutionState` a `Graph`, konkrétně funkce `toString()` a `fromString()`.

Novou důležitou třídou je třída `Worker`, reprezentující Worker proces. Worker procesy přijímají od Main procesu nejdříve graf a následně stavy, které dále pomocí funkčního paralelismu prohledávají.

```
void workerMain(){
    string graph_str;
    MyMpi::recvString(MPI_MAIN, TAG_GRAPH, graph_str);
    graph.fromString(graph_str);
    while (true) {
        string initial_state_str;
        MyMpi::recvString(MPI_MAIN, TAG_STATE, initial_state_str);
        initial_state.fromString(initial_state_str, &graph);
        if (all_of(initial_state.colors, initial_state.colors +
MAX_VERTICES, [](color_t c){return c == NO_COLOR;})) {
            break;
        } else if (initial_state.isLeaf() and
                   initial_state.isBetterThan(best_state)) {
            best_state = initial_state;
        } else {
            #pragma omp parallel num_threads(number_of_threads)
                #pragma omp single
                    findBestStateDFS(initial_state);
            MyMpi::sendString(MPI_MAIN, TAG_BEST,
best_state.toString());
        }
    }
}
```

Měření, analýza a hodnocení paralelizace

Měření proběhlo na klastru star.fit.cvut.cy. Následující tabulka ukazuje výsledky pro 3 vybrané grafy a časy daných běhů. Byl měřen sekvenční čas. Poté funkční a datový paralelismus pro 2, 4, 8, 16, 20 vláken. A následně byl měřen výkon MPI s 16 vlákny. Tabulka obsahuje časy v počtu sekund.

Měření	**c**	**t**	**graf_23_20.txt**	**graf_24_23.txt**	**graf_26_25.txt**
Sequential	1	1	56	138	473
Parallel task	1	1	177	451	přesčas
Parallel task	1	2	98	250	přesčas
Parallel task	1	4	48	124	přesčas
Parallel task	1	8	24	64	314
Parallel task	1	16	14	36	178
Parallel task	1	20	16	46	210
Parallel data	1	1	55	157	přesčas
Parallel data	1	2	40	109	497
Parallel data	1	4	35	88	256
Parallel data	1	8	20	49	137
Parallel data	1	16	12	31	74
Parallel data	1	20	9	22	78
MPI + PT (1M + 1W)	2	16	115	TCP network fail	527
MPI + PT (1M + 2W)	3	16	72	174	501
MPI + PT (1M + 3W)	4	16	49	149	489

Paralelní zrychlení: $S(n,\ p)\ =\ SU(n)\ \div\ T(n,\ p)$, kde $SU(n)$ je sekvenční čas a $T(n,\ p)$ je paralelní čas při p procesech.

Zrychlení	**c**	**t**	**graf_23_20.txt**	**graf_24_23.txt**	**graf_26_25.txt**
Parallel task	1	1	0.3163841808	0.3059866962	přesčas
Parallel task	1	2	0.5714285714	0.552	přesčas
Parallel task	1	4	1.166666667	1.112903226	přesčas
Parallel task	1	8	2.333333333	2.15625	1.506369427
Parallel task	1	16	4	3.833333333	2.657303371
Parallel task	1	20	3.5	3	2.252380952
Parallel data	1	1	1.018181818	0.8789808917	přesčas
Parallel data	1	2	1.4	1.266055046	0.9517102616
Parallel data	1	4	1.6	0.5568181818	1.84765625
Parallel data	1	8	2.8	2.816326531	3.452554745
Parallel data	1	16	4.666666667	4.451612903	6.391891892
Parallel data	1	20	6.222222222	6.272727273	6.064102564
MPI + PT (1M + 1W)	2	16	0.4869565217	TCP network fail	0.8975332068
MPI + PT (1M + 2W)	3	16	0.7777777778	0.7931034483	0.9441117764
MPI + PT (1M + 3W)	4	16	1.142857143	0.9261744966	0.9672801636

Paralelní efektivnost: $E(n,\ p)\ =\ SU(n)\ \div\ C(n,\ p)$, kde $SU(n)$ je sekvenční čas a $C(n,\ p)$ je paralelní cena, pro kterou platí $C(n,\ p)\ =\ p\ \cdot\ T(n,\ p)$.

Efektivnost	**c**	**t**	**graf_23_20.txt**	**graf_24_23.txt**	**graf_26_25.txt**
Parallel task	1	1	0.3163841808	0.3059866962	přesčas
Parallel task	1	2	0.2857142857	0.276	přesčas
Parallel task	1	4	0.2916666667	0.2782258065	přesčas
Parallel task	1	8	0.2916666667	0.26953125	0.1882961783
Parallel task	1	16	0.25	0.2395833333	0.1660814607
Parallel task	1	20	0.175	0.15	0.1126190476
Parallel data	1	1	1.018181818	0.8789808917	přesčas
Parallel data	1	2	0.7	0.6330275229	0.4758551308
Parallel data	1	4	0.4	0.3920454545	0.4619140625
Parallel data	1	8	0.35	0.3520408163	0.4315693431
Parallel data	1	16	0.2916666667	0.2782258065	0.3994932432
Parallel data	1	20	0.3111111111	0.3136363636	0.3032051282
MPI + PT (1M + 1W)	2	16	0.0152173913	TCP network fail	0.02804791271
MPI + PT (1M + 2W)	3	16	0.0162037037	0.01652298851	0.01966899534
MPI + PT (1M + 3W)	4	16	0.01785714286	0.01447147651	0.01511375256

Z výsledků měření je vidět, že datový paralelismus překonal funkční. Předpokládám, že je to způsobeno převážně nižšími nároky na režii datového paralelismu oproti funkčnímu paralelismu s #pragma omp task.

Závěr

Cílem práce bylo vytvořit programy pro řešení úlohy maximálního souvislého bipartitního grafu a následně provést vyhodnocení metrik těchto programů.

Tyto cíle jsem splnil a naučil se novým věcem. Úloha mi ze začátku přišla těžká a nepříjemná, ale po prvním odevzdání sekvenčního řešení, jsem si k ní našel cestu a dal si navíc práci s optimalizací kódu, ze které mám radost.

Měření na clusteru star bylo příjemným zpestřením, ale pocit z rychlého běhu při spouštění na lokálním PC se nedá překonat.