Informatica - Area scientifica
Dipartimento di Scienze matematiche, informatiche e multimediali
Università di Udine

Progetto di Architetture Parallele

Rasera Giovanni (143395)

Anno accademico 2024/2025

Indice

1 Descrizione del problema affrontato

Scrivere un programma che, dato un nodo x in V (diverso da s), determini un sottoinsieme D di nodi tale che:

- 1) la sorgente s non appartiene a D e il nodo x non appartiene a D
- 2) ogni cammino diretto da s a x passa per almeno un nodo in D
- 3) D è minimale rispetto alla cardinalità

Un algoritmo che risolve un problema analogo è l'algoritmo MinCutMaxFlow con la differenza che permette di trovare gli archi che rappresentano il massimo flusso che può attraversare un grafo.

1.1 MinCutMaxFlow in un contesto reale

Scenario bellico: Gestione strategica delle vie di comunicazione
In un contesto bellico, un comandante deve impedire al nemico di trasportare rifornimenti dalla loro base principale (sorgente s) a un obiettivo strategico (t) attraverso una rete stradale.
La rete è rappresentata come un grafo, dove:

- I nodi rappresentano le intersezioni stradali.
- Gli archi rappresentano le strade che collegano queste intersezioni, con capacità che indicano la quantità massima di rifornimenti che possono attraversare ciascuna strada.

Obiettivo:
Individuare il set minimo di strade (arco taglio) che, se bloccate o distrutte, interromperebbero efficacemente tutti i rifornimenti dal punto s al punto t.

2 Metodologia adottata per la soluzione

2.1 Sfruttare l'algoritmo MinCut per risolvere il problema richiesto

L'algoritmo MinCut risolve il problema di trovare gli archi con il costo minimo. Il problema presentato cerca di tagliare i nodi, una soluzione è quella di procedere come suggerito da: Professor Andrea Formisano
Un nodo v in G=(V, E) diventa v' in G'=(V', E'), composto da (v'even -> v'odd).

- Tutti gli archi originali in(v) che arrivavano a v sono ora collegati a v'even con costo |V'| * 2.
- Tutti gli archi originali out(v) che partivano da v partono ora da v'odd con costo infinito.
- Il costo da v'odd a v'even è pari a 1.

Esempio:

- G : 0 -a-> 1 -b-> 2 -c-> 3
- G' : (0 -1-> 1) -16-> (2 -1-> 3) -16-> (4 -1-> 5) -16-> (6 -1-> 7)

Il risultato del MinCut nel grafo G' darà come risultato il taglio di archi di costo 1 e indicheranno i nodi da eliminare con un semplice calcolo nodo_da_eliminare = (v'even / 2).

2.1.1 Condizione di taglio del nodo

Nell'algoritmo che verrà presentato, per verificare quali archi tagliare in G' (di conseguenza i nodi in G) bisogna controllare che per ogni arco (u -> v) siano verificate contemporaneamente le condizioni:

- excess_flow del nodo u abbia un eccesso di (V'*2)-1
- forward_flow di (u -> v) 0

- backward_flow di (u -> v) 1

Tali nodi tagliati comporranno l'insieme D del problema proposto.
IMPORTATE: Le strutture dati excess_flow, forward_flow e backward_flow derivano da come è stato risolto il problema, e le condizioni di taglio dipendono dal tipo di algoritmo e di struttura dati che si decide di utilizzare per risolvere il problema.

2.1.2 Esempio 1

L'insieme D è il nodo 2 e l'output dell'algoritmo di taglio dovrebbe essere:
Nodes D are:
(2) : 4 -/-> 5
|D|: 1

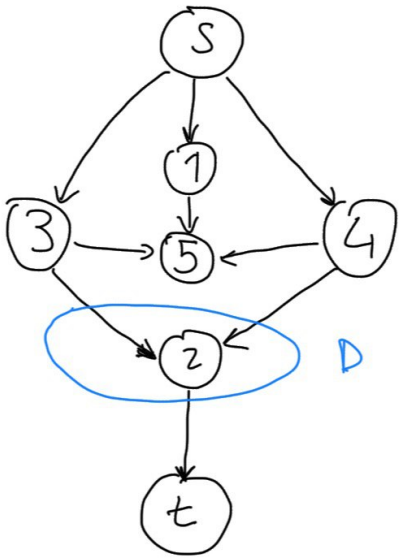

Figura 2.1: /graphs/esempio1.txt

2.1.3 Esempio 2

L'insieme D sono i nodi 1 2 3 4 5 l'output dell'algoritmo di taglio dovrebbe essere:
Nodes D are:
(1) : 2 -/-> 3
(2) : 4 -/-> 5
(3) : 6 -/-> 7
(4) : 8 -/-> 9
(5) : 10 -/-> 11
|D|: 5

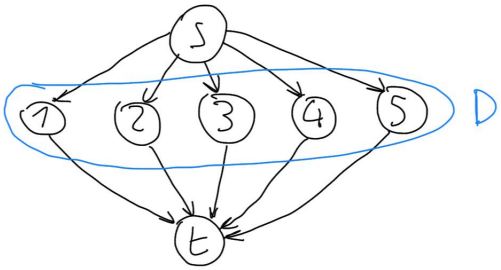

Figura 2.2: /graphs/esempio2.txt

2.1.4 Esempio 3

L'insieme D sono i nodi 1 2 l'output dell'algoritmo di taglio dovrebbe essere:
Nodes D are:
(1) : 2 -/-> 3
(2) : 4 -/-> 5
|D|: 2

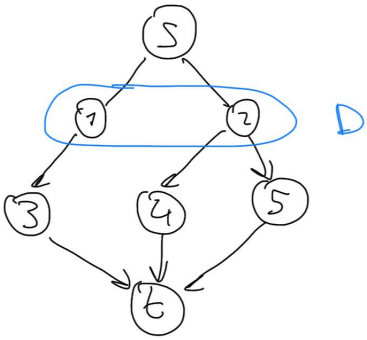

Figura 2.3: /graphs/esempio3.txt

2.1.5 Esempio Reale

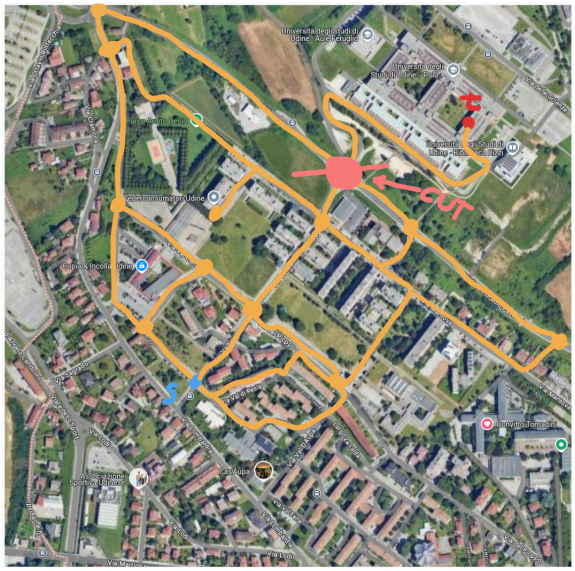

Figura 2.4: Esempio di difesa dell'aula A036

In questo caso il minimo taglio di un nodo che si può fare per separare (s) e (t) è il taglio del nodo rosso in figura.

2.2 MinCutMaxFlow

Esistono diversi algoritmi di MinCutMaxFlow, i due che vengono presi in considerazione sono:

1. Ford-Fulkerson
2. Goldberg-Tarjan
   (a) Goldberg-Tarjan Parallelo

2.2.1 Ford-Fulkerson

Di seguito viene presentato lo peseudo codice.

Ford-Fulkerson

```
auto minCutMaxFlow(graph,rGraph, source, to){
    // init structs
    //...

    while(bfs(rGraph, parent, source, to)){
        path_flow = INFINITY;
        // a path from to -> source
        for (v = to; v != source; v = parent[v]){
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        // update the flow in the residual graph
        for (v = to; v != source; v = parent[v]){
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }
    }

    // run dfs on the residual graph
    dfs(rGraph, visited, source);

    // here you can calculate nodes to cut
}
```

Utilizza BFS e DFS per determinare se l'ultimo nodo può essere raggiunto. Tale algoritmo è difficile da parallelizzare perché BFS e DFS sono due algoritmi notoriamente complicati da parallelizzare.

2.2.2 Goldberg-Tarjan

Di seguito viene presentato lo peseudo codice.

Golberg-Tarjan

```
auto minCutMaxFlow(source, sink){
    preflow(source);

    while(any_active()) {
        push(active_node);
        relabel(active_node);
    }
}
```

L'idea che sta alla base è quella di cercare un nodo con delle caratteristiche particolari da cui è possibile far passare del flusso, e procedere in tal modo fino a quando non è più possibile inserire del flusso.
Per capire l'algoritmo ovviamente c'è bisogno della funzione di push e relabel.

Push e Relabel

```
auto push(x){
    if(active(x)){
        for (y=neighbor(x)) {
            if (height(y) == height(x)-1) {
                flow = min( capacity(x,y), excess_flow(x));

                // update the flow
                excess_flow(x) -= flow;
                excess_flow(y) += flow;
                capacity(x,y) -= flow;
                capacity(y,x) += flow;
            }
        }
   }
}

auto relabel(x){
    if (active(x)) {
        my_height = V;

        // init to max height
        for (y=neighbor(x)){
            if capacity(x,y) > 0 {
                my_height = min(my_height, height(y)+1);
            }
        }
        height(x) = my_height;
```

```
        }
    }
```

Il nodo x è attivo: se capacity(x) $>$ 0 e height(x) $<$ HEIGHT_MAX:
Nodo attivo x:

- può spingere verso il vicino y: se capacity(x,y) $>$ 0, height(y) = height(x) – 1
- viene relabel: se per tutti capacity(x, *) $>$ 0, height(*) $<=$ height(x)

Tale algoritmo è un'ottimo candidato per la realizzazione di un algoritmo parallelo. Ogni nodo può essere visualizzato da un thread ed accede in modo atomico alle strutture che modificherà.

2.3 Goldberg-Tarjan Parallelo

L'implementazione dell'algoritmo deriva dal paper [CK24]. L'algoritmo analizzato nel documento risolve il problema di trovare IL VALORE del taglio minimo di archi. Per risolvere il problema del taglio dei nodi sono state apportate delle modifiche. Viene applicata la trasformazione discussa nel capitolo 2.1.
Per capire quali nodi sono di interesse vine fatto il controllo presentato nel capitolo 2.1.
Di seguito l'algoritmo di pseudo codice:

Algorithm 1: The lock-free push-relabel algorithm

Input: $G(V, E)$: the directed graph, $G_f(V, E_f)$: the residual graph, $c_f(v, u)$: the residual flow on (u, v), $e(v)$: the excess flow of the vertex v, $h(v)$: the height of the vertex v, $Excess\_total$: the sum of excess flow
Output: $e(t)$: the maximum flow value
Data: Initialize $c_f(v, u), e, h$, $Excess\_total \leftarrow 0$

```
   /* Step 0: Preflow                                              */
 1 foreach (s, v) ∈ E do
 2     c_f(s, v) ← 0
 3     c_f(v, s) ← c(s, v)
 4     e(v) ← c(s, v)
 5     Excess_total ← Excess_total + c(s, v)
 6 while e(s) + e(t) < Excess_total do
       /* Step 1: Push-relabel kernel (GPU)                        */
 7     cycle = |V|
 8     while cycle > 0 do
 9         foreach u ∈ V and e(u) > 0 and h(u) < |V| do
10             h' ← ∞ , v' = -1
11             foreach (u, v) ∈ E_f do
12                 if h(v) < h' then
13                     h' ← h(v) , v' = v        // the value
                                                 of the min
                                                 node
14             if h(u) > h' then
15                 d ← MIN(e(u), c_f(u, v'))
16                 AtomicSub(c_f(u, v'), d)
17                 AtomicSub(e(u), d)
18                 AtomicAdd(c_f(v', u), d)
19                 AtomicAdd(e(v'), d)
20             else
21                 h(u) ← h' + 1
22         cycle ← cycle − 1
       /* Step 2: Heuristic Optimization (CPU)                     */
23     GlobalRelabel()
```

Figura 2.5: Golberg-Tarjan, parallelo

Nella Figura 2.3 sono presenti delle correzioni rispetto all'implementazione originale, questo perché durante lo studio del paper sono stati scoperti degli errori che sono stati riportati agli autori; un riferimenti agli errori riportati: GitHub Issue 4.
L'algoritmo presentato assegna un thread ad ogni vertice **u** e controlla se tale vertice è attivo. Se il vertice **u** è attivo, allora il thread trova un vertice direttamente collegato chiamato vertice **v** e che ha altezza minima tra tutti i vertici vicini. Quando **h(u)** > **h(v)** allora il thread spinge del flusso da **u** a **v**, negli altri casi l'altezza del nodo **u** viene aumentata di **1** rispetto all'altezza del nodo **v**.

Quando tutti i thread hanno finito deve essere eseguita una procedura chiamata **GlobalRelabel** che permette di ridurre al minimo le altezze del nodi e di diminuire il valore di Excess_total in modo da far terminare l'algoritmo.
L'accesso ai nodi viene suddiviso in **BLS** numero di blocchi di thread con **THS** thread. Il parametro può essere impostato in fare di compilazione tramite il comando:
make custom_params BLS=1 THS=1024
Il kernel viene lanciato utilizzando la chiamata a **cudaLaunchCooperativeKernel**.

Gpu Call

```
// Configure the GPU
int device = -1;
cudaGetDevice(&device);
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, device);
dim3 num_blocks(deviceProp.multiProcessorCount * numBlocksPerSM);
dim3 block_size(numThreadsPerBlock);
size_t sharedMemSize = 3 * block_size.x * sizeof(int);

....
// Algo
    // Update GPU values
    ( cudaMemcpy ... , cudaMemcpyHostToDevice) ) ;

    // Gpu Call
    // calling the kernel
    cudaLaunchCooperativeKernel(push_relabel_kernel, ...args);
    cudaDeviceSynchronize();

    // Get results
    ( cudaMemcpy ... , cudaMemcpyDeviceToHost ) ) ;
```

2.4 Grafo CSR(Compressed Sparse Row)

Per la realizzazione dell'algoritmo parallelo è necessario introdurre la rappresentazione CSR di un grafo.
Data un grafo G(V, E) la sua rappresentazione CSR (Compressed Sparse Row) è rappresentata dai seguenti parametri:

- Un vettore offsets grande |V| che per ogni nodo rappesenta offset dove trovare i nodi a cui punta nel vettore destinations,
- un vettore destinations grande |E| che rappresenta i nodi destinazione,
- un vettore capacities grande |E| che rappresenta la capacità di ogni arco.

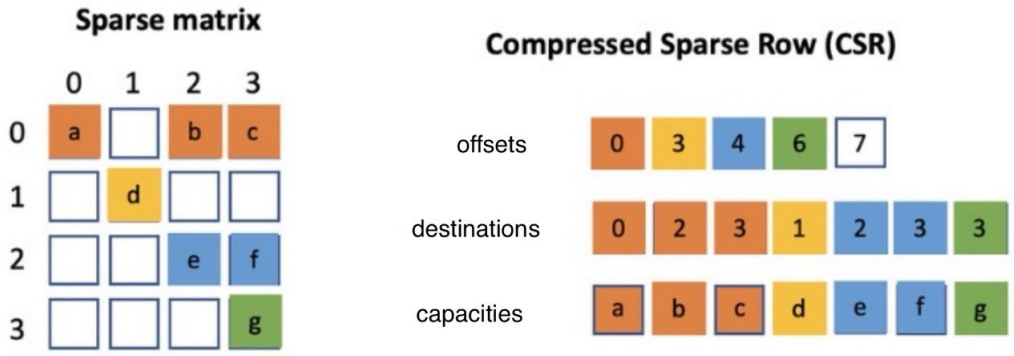

Figura 2.6: https://www.researchgate.net

2.5 Strutture di supporto all'algoritmo

L'algoritmo a bisogno delle strutture dati:

- excessTotal: Un numero che rappresenta la quantità totale di flusso in eccesso; viene inizializzato con la somma di tutte le capacità degli archi in uscita dalla sorgente (s).
- excesses: Un vettore grande |V| che rappresenta la quantità di flusso in eccesso che arriva ad un nodo. Viene aggiornato a coppia di due nodi collegati ad un arco che è stato selezionato come arco di costo minimo. L'accesso a tale struttura viene eseguito il modo atomico.
- flow_index: Un vettore grande |E| che rappresenta quale è l'arco forward che è interessato da un arco backward. Visualizzando la sezione di codice di inizializzazione di tale struttura si può capire capire meglio cosa si intende: parallel/loader.hppL258-L280
- heights: Un vettore grande |V| che rappresenta l'altezza di un nodo, i nodi più alti possono spingere del flusso verso quelli più bassi. La struttura viene aggiornata alzando id 1 i nodi da cui non è possibile spingere del flusso.
- forward_flows, backward_flows: Dei vettori grandi |E| che rappresentano la quantità di flusso che viene spinta tramite un arco.

3 Risultati sperimentali

I risultati sui tempi di esecuzione sono stati presi generando grafi di grandezze diverse con numero di nodi diverso. In particolare sono stati utilizzati i seguenti grafi:

1. graphs/basic1.txt –nodes 70 –edges 6000
2. graphs/basic2.txt –nodes 130 –edges 10000
3. graphs/basic3.txt –nodes 260 –edges 20000
4. graphs/basic4.txt –nodes 520 –edges 50000
5. graphs/basic5.txt –nodes 1030 –edges 100000
6. graphs/nanoone.txt –nodes 2050 –edges 200000
7. graphs/microone.txt –nodes 4100 –edges 400000
8. graphs/minione.txt –nodes 8200 –edges 800000
9. graphs/midone.txt –nodes 16400 –edges 1600000
10. graphs/bigone.txt –nodes 32800 –edges 3200000

3.1 Comparazione tra algoritmi diversi

Nella tabella vengono riportati i tempi di esecuzione in microsecondi delle diverse implementazioni.

Tempi

Grafo	Nodi	Archi	GPU GodbergTarjan	CPU GoldbergTarjan	FordFulkerson	CPUgt/GPUgt	FoFu/GPUgt
basic1	70	6000	1386	931	9581	0,6	6,9
basic2	130	10000	2873	8617	37298	2,9	12,9
basic3	260	20000	6025	21724	183634	3,6	30,4
basic4	520	50000	6353	637	758920	0,1	119,4
basic5	1030	100000	31103	833646	2790481	26,8	89,7
nanoone	2050	200000	17285	5954	11523114	0,3	666,6
microone	4100	400000	135952	39740650	41343893	292,3	304,1
minione	8200	800000	57861	81863	179611458	1,4	3104,1
midone	16400	1600000	122384	304478	699148515	2,4	5712,7
bigone	32800	3200000	241725	1065764		4,4	

Figura 3.1: Tabella dei tempi di microsecondi

Dalla tabella possiamo notare che l'implementazione dell'algoritmo di GPU ha portato ad un miglioramento della velocità di esecuzione. Di seguito vengono

presentati i grafici che presentano meglio il guadagno in performance.

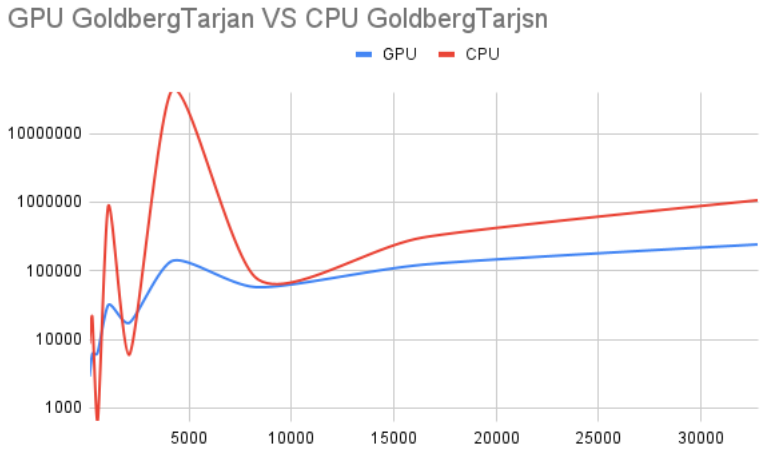

Figura 3.2: X: numero di nodi, Y: log(tempo di esecuzione)

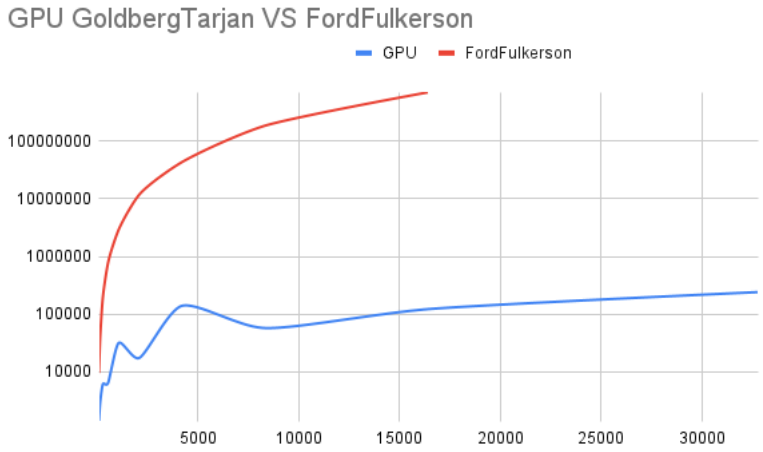

Figura 3.3: X: numero di nodi, Y: log(tempo di esecuzione)

3.2 Comparazione kernel con parametri BLS e THS

Grafo	Nodi	BLS=1, THS=1024	BLS=2, THS=512	BLS=4, THS=256	BLS=8, THS=128	BLS=16, THS=64
basic1	70	1520	1386	1667	2719	4669
basic2	130	3007	2873	3177	5837	10624
basic3	260	5959	6025	6328	11654	22137
basic4	520	6506	6353	6867	12272	22404
basic5	1030	31600	31103	32828	63981	126934
nanoone	2050	16860	17285	17910	29478	38247
microone	4100	145528	135952	146370	240393	350766
minione	8200	53598	57861	57614	81312	139594
midone	16400	123635	122384	145042	219204	241164
bigone	32800	266265	241725	240535	391576	583796

Figura 3.4: Tabella dei tempi kernel con parametri diversi

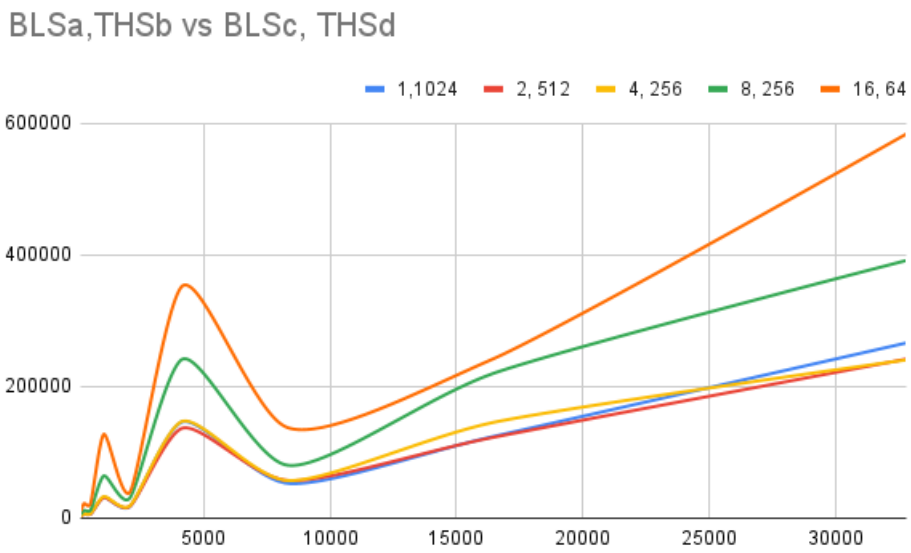

Figura 3.5: Grafico dei tempi kernel con parametri diversi

Dal grafico si evince che è meglio tenere BLS=1 e THS=1024.

4 Valutazioni ed osservazioni

I risultati sono buoni, potrebbero comunque essere migliorati utilizzando delle tecniche di suddivisione dei dati e di WORK BALANCING più complesse. Nel paper [CK24] sono presentate alcune delle implementazioni.

5 Build del codice

I tre algoritmi presi in considerazione sono nelle cartelle:

- sequential: Versione FordFulkerson
- paperSerialCode: Versione GoldbergTarjan CPU
- parallel: Versione GoldbergTarjan GPU

Scrivendo sul terminale:
make test
varranno eseguiti i test su diversi grafi.
Nella versione CUDA è possibile settare dei parametri BLS e THS
make custom_params BLS=1, THS=1024
L'output degli eseguibili sarà il tempo di esecuzione dell'algoritmo.

Bibliografia

[CK24] Po-Chieh Lin Chou-Ying Hsieh e Sy-Yen Kuo. «Engineering A Workload-balanced Push-Relabel Algorithm for Massive Graphs on GPUs». In: *arXiv* (2024).

[] URL: https://web.stanford.edu/class/archive/cs/cs161/cs161.1172/CS161Lecture16.pdf.

[] URL: https://www.tutorialspoint.com/data_structures_algorithms/dsa_kargers_minimum_cut_algorithm.htm.

[] URL: https://www.baeldung.com/cs/minimum-cut-graphs.

[] URL: https://it.wikipedia.org/wiki/Algoritmo_di_Ford-Fulkerson.

[] URL: https://www.nvidia.com/content/GTC/documents/1060_GTC09.pdf.

[] URL: https://en.wikipedia.org/wiki/Push%E2%80%93relabel_maximum_flow_algorithm.

[] URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4563095.

[] URL: https://arxiv.org/pdf/2404.00270.

[] URL: https://github.com/NTUDDSNLab/WBPR/tree/master/maxflow-cuda.

[] URL: https://www.adrian-haarbach.de/idp-graph-algorithms/implementation/maxflow-push-relabel/index_en.html.

[] URL: https://www.geeksforgeeks.org/push-relabel-algorithm-set-2-implementation/.