Report progetto Ingegneria degli Algoritmi

Alessio Susco mat. 266383

10 settembre 2021

1 Introduzione

Al giorno d'oggi, moltissime sono le applicazioni che fanno uso intensivo di algoritmi per la risoluzione del problema del calcolo dei cammini minimi da singola sorgente. In particolare, il Single Source Shortest Path problem (SSSP) consiste nel trovare i cammini minimi tra un dato vertice sorgente s e tutti gli altri vertici nel grafo. L'algoritmo di Dijkstra è un algoritmo esatto che risolve tale problema. Per algoritmo esatto si intende quell'algoritmo che restituisce soluzioni ottime per il problema dato. In alcune applicazioni moderne, uno dei principali problemi è che i grafi cambiano continuamente la loro struttura ed in modo non predicibile a priori. In particolare gli eventi che vanno a cambiare la struttura del grafo possono essere: aggiunta o rimozione di nodi e/o archi e incrementi/decrementi dei pesi degli archi. Al verificarsi di ogni evento, i cammini minimi calcolati precedentemente potrebbero non essere più minimi, rendendo così necessaria un ulteriore risoluzione del problema. L'obiettivo di questo studio è quello di andare a valutare le performance di due algoritmi, per il calcolo dei cammini minimi di un grafo da singola sorgente, a fronte di eventi di aggiunta di archi ed incremento dei pesi degli archi. I due algoritmi presi in esame sono quelli messi a disposizione dal toolkit Networkit nella libreria distance, in particolare: distance.Dijkstra e distance.DynDijkstra.

1.1 Notazione

Sia $G = (V, E)$ un grafo e siano

- $V = v_1, \ldots, v_n$ insieme finito di vertici
- $E = (v_i, v_j) : v_i, v_j \in V$ insieme finito di archi
- $|V| = n$
- $|E| = m$

Definiamo:

- $w : E \rightarrow R$ funzione peso di $G = (V, E)$
- peso di un cammino $\prod =< v_1, \ldots, v_k >$ come somma dei pesi degli archi che lo costituiscono, cioè: $w(\prod) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

- peso di un cammino minimo da un vertice u a un vertice v come: $d(u, v) = \begin{cases} min\{w(\prod) : u \to v\} & \text{Se esiste un cammino da } u \text{ a } v \\ \infty & \text{Altrimenti} \end{cases}$
- cammino minimo da un vertice u a un vertice v come un qualunque cammino $\prod$ il cui peso è pari a $d(u, v)$
- albero dei cammini minimi con radice in s il sottografo $G' = (V', E')$, con $V' \subseteq V$ ed $E' \subseteq E$, tale che
  - V' è l'insieme dei vertici raggiungibili da s
  - G' è un albero con radice in s
  - Per ogni $v \in V'$. l'unico cammino da s a v in G' è un cammino minimo da s a v in G
  - con w si fa riferimento a quei nodi la cui distanza da s è cambiata come conseguenza degli eventi nel batch β
  - $p(w) :=$ priorità del nodo w
  - $d :=$ vettore delle distanze da s
  - σ vettore dei cammini minimi da s
  - $d' = max\{d(s, u) | u \in V, d(u, v) \neq +\infty\}$
  - $d'' = max\{d(s, v) | v \in V, v \neq u, d(s, v) \neq +\infty\}$

1.2 Algoritmo DynDijkstra

L'algoritmo DynDijkstra, dato un grafo G e un batch di eventi β, permette di mantenere aggiornato l'albero dei cammini minimi, radicato in s, andando a ricalcolare solo i cammini che sono stati influenzati dagli eventi in β. Tale aggiornamento viene effettuato attraverso l'algoritmo *updateSSSP* illustrato nel seguito. Per completezza si sottolinea che l'algoritmo DynDijkstra fa uso dell'algoritmo di Dijkstra per il calcolo dell'albero dei cammini minimi nella sua prima iterazione.

1.2.1 Pseudocodice

Algorithm 1 $updateSSSP(grafo\,G, vettore\,d, vettore\,\sigma, batch\,\beta) \rightarrow \{updated\,d(v), updated\,\sigma(v)\ \forall v \in V\}$

$CodaConPriorita\ S$
for each $arco\ e \in \beta, d(u) < d(v)$ **do**
    $S.InsertOrDecreaseKey(v, p(v) = min\{d(u) + w(u, v), d(w)\}$
$\overline{w} \leftarrow min\{\overline{w}, w(e) \in \beta\}$
while $(not\ S.IsEmpty())$ **do**
    $\{w, p(w)\} \leftarrow S.ExtractMin()$
    $con(w) \leftarrow min_{z:(z,w)\in E} d(z) + w(z, w)$
    if $(con(w) = p(w))$ **then**
        $update\ d'\ and\ d''$
        $d(w) \leftarrow p(w)$
        $\sigma(w) \leftarrow 0$
        for each *incident edge* (z, w) **do**
            if $d(w) = d(z) + w(z, w)$ **then**
                $\sigma(w) \leftarrow \sigma(w) + \sigma(z)$
            if $d(z) \geq d(w) + w(z, w)$ **then**
                $S.InsertOrDecreaseKey(z, p(z) = d(w) + w(z, w))$
    else
        if $(d(w) \neq +\infty)$ **then**
            if $(con(w) \neq +\infty$ **then**
                $S.InsertOrDecreaseKey(w, p(w) = con(w))$
                for each *incident edge* (z, w) **do**
                    if $d(z) = d(w) + w(w, z)$ **then**
                        $S.InsertOrDecreaseKey(z, p(z) = d(w) + w(z, w))$
                $d(w) \leftarrow +\infty$

1.3 Algoritmo di Dijkstra

L'algoritmo di Dijkstra consente di risolvere il problema del Single-Source Shortest Path. É applicabile in grafi diretti e non, con o senza cicli, pesati o non pesati purché abbiano archi con pesi non negativi. L'algoritmo Dijkstra, dato in input un grafo $G = (V, E, w)$ orientato o non orientato e con pesi reali non negativi sugli archi ed un vertice sorgente s, permette di calcolare l'albero dei cammini minimi G' con radice in s e per ogni vertice v il valore $d(s, v)$. Al contrario del precedente appena descritto è intuibile che, a fronte di tali eventi, è necessario rieseguirlo per avere i cammini minimi corretti ed aggiornati. Pertanto, tale algoritmo è stato utilizzato per quantificare quanto fosse più veloce l'algoritmo dinamico in particolari set di grafi.

1.3.1 Pseudocodice

Algorithm 2 *Dijkstra* $(grafo\ G, vertice\ s) \rightarrow albero\ G'$

for each *vertice* $u \in G$ **do**
    $d_{su} \leftarrow +\infty$
CodaConPriorità S
$G' \leftarrow$ *albero formato dal solo vertice s*
$d_{ss} \leftarrow 0$
S.Insert$(s, 0)$
while (*not S.IsEmpty*()) **do**
    $u \leftarrow$ *S.DeleteMin*()
    for each (*arco* $(u, v) \in G$) **do**
        if $(d_{sv} = +\infty)$ **then**
            S.Insert$(v, d_{su} + w(u, v))$
            $d_{sv} \leftarrow d_{su} + w(u, v)$
            Rendi u padre di v in G'
        else if $(d_{su} + w(u, v) < d_{sv})$ **then**
            S.DecreaseKey$(v, d_{sv} - (d_{su} + w(u, v))$
            $d_{sv} \leftarrow d_{su} + w(u, v)$
            Rendi u nuovo padre di v in G'
return G'

2 Progettazione degli esperimenti

Con questo studio si vuole determinare come varia lo speedup tra l'algoritmo statico e quello dinamico al variare delle taglie dei grafi in input, pertanto per la scelta dei fattori e dei design points si è fatto riferimento alle linee guida relative alle categorie di domande di tipo Assessment. Di seguito verranno brevemente riportate le metriche ed i parametri scelti:

- Performance Metrics: Speedup e Time
- Performance Indicator: CPU Time speso dai due algoritmi, scelta obbligata per ridurre al minimo l'influenza degli altri processi sui risultati durante l'elaborazione
- Algorithm Parameters: istanze di grafi e nodo sorgente (Dijkstra(graph, source)). Inoltre, anche gli eventi di modifica delle istanze dei grafi sono stati considerati come parametri algoritmici
- Instance Parameters:
  - Input Source: I grafi sono stati generati randomicamente da due classi di generatori, cioè ErdosRenyi e BarabasiAlbert
  - Input Size: n: numero dei nodi, p: probabilità dell'esistenza di un arco per i grafi ER e k:numero di archi uscenti da un nodo per i grafi BA
- Factors: I parametri che vengono esplicitamente manipolati nei seguenti esperimenti sono il numero dei nodi nei grafi. In particolare, i livelli sono stati scelti seguendo la logica del Doubling Experiment partendo da un valore iniziale abbastanza grande che permettesse di evitare risultati spuri dovuti al floor effect. Il numero di raddoppi è stato scelto come compromesso tra generalità dei risultati, somiglianza a taglie di grafi dinamici reali e tempi di sperimentazione accettabili.
- Fixed Parameters: Il numero di eventi di modifica del grafo durante gli esperimenti è fissato a priori. Inoltre, anche p e k (parametri di istanza) sono stati fissati. In particolare $p = 0.02$ per i grafi Erdos-Renyi è stato scelto in modo da soddisfare le ipotesi di un teorema che ci garantisce che la probabilità del grafo generato $P(\{G(n,p) \quad isconnected\}) \to 1$
- Noise Parameters: Il tipo di evento dinamico di modifica del grafo è scelto casualmente ma in modo semicontrollato dato che i possibili eventi sono due, ovvero l'aggiunta di un arco al grafo e l'incremento del peso di un arco già presente nel grafo. Anche i pesi degli archi vengono scelti randomicamente tra due valori fissati a priori. Per diminuire il bias statistico dovuto al nodo sorgente, è stato scelto di cambiare, ad ogni evento, il nodo sorgente per il calcolo dei cammini minimi..
- Design Points: Date 6 taglie di input per ogni grafo

  $$n = (500, 1000, 2000, 4000, 8000, 16000)$$

  e due tipologie di grafi

  $$GraphType = (BarabasiAlbert, ErdosRenyi)$$

  abbiamo ottentuto 12 design points

3 Test Environment

In questo paragrafo verranno riportate tutte le implementazioni e gli script necessari all'esecuzione degli esperimenti precedentemente descritti con l'obiettivo di riassumere il loro funzionamento. L'intero ambiente di test è rappresentato dal seguente workflow:

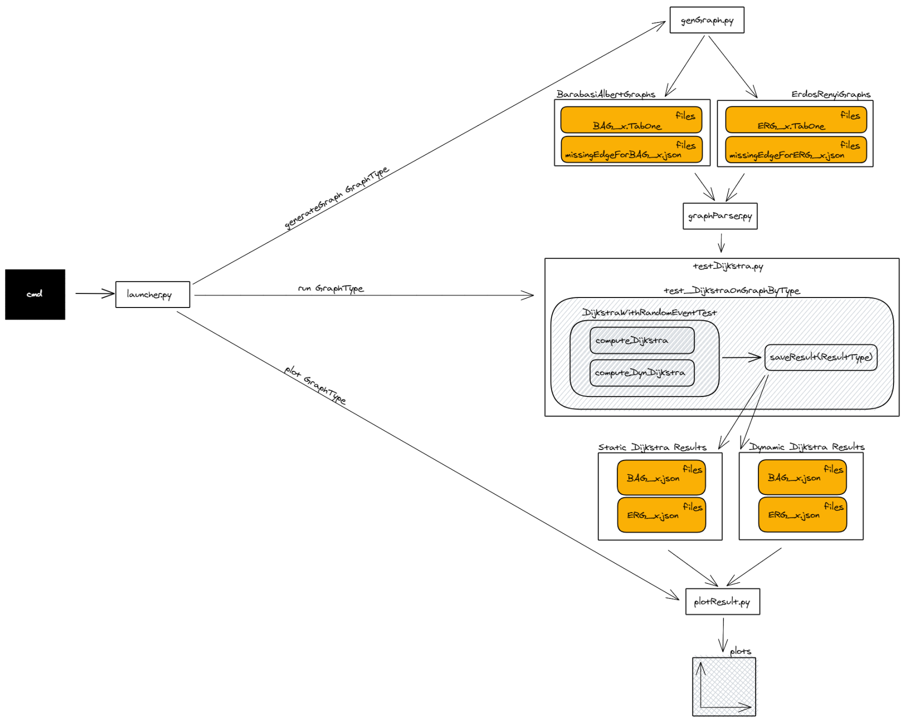

Figura 1: Test Enviroment Workflow

3.1 Implementazioni

E' possibile eseguire i test attraverso lo script python *launcher.py* con i seguenti comandi:

- -h per visualizzare i comandi
- -g GraphType per generare i grafi del tipo GraphType
- -r GraphType per eseguire i test dei due algoritmi sui grafi di tipo GraphType precedentemente generati
- -p GraphType per plottare i risultati ottenuti dall'esecuzione dei test

E' presente un ulteriore script chiamato *utility.py* nel quale sono presenti funzioni comuni tra i vari script e parametri di settings dell'intero ambiente di test e verranno descritti successivamente. E' necessario anticipare la presenza in *utility.py* di:

- path relativi delle folder nelle quali vengono salvati i grafi e i risultati dell'esecuzione dei test
- enum *GraphTypes* per identificare i tipi di grafi
- enum *DijkstraAlgoTypes* per identificare l'algoritmo statico o dinamico

3.1.1 Phase 1 Graph Generation

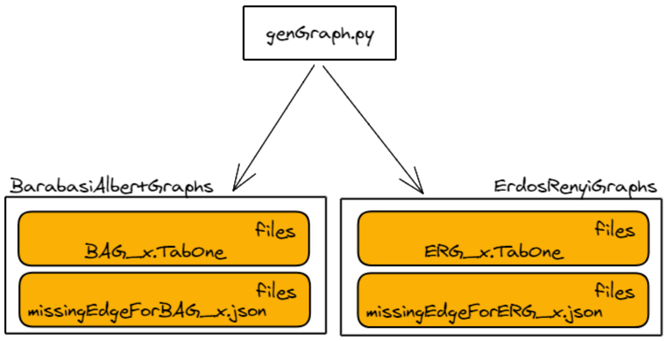

Figura 2: Phase 1: cmd to graphFiles

Lo script *genGraph.py* si occupa di generare i grafi e salvarli su file. Per la generazione dei grafi vengono utilizzati i seguenti parametri di input presenti nello script *utility.py*:

- MIN_NODES : numero minimo di nodi
- MIN_K : numero minimo di collegamenti per ogni nodo (parametro di input per la generazione dei grafi di Barabasi-Albert
- MIN_PROB : probabilità minima dell'esistenza di un arco (parametro di input per la generazione dei grafi di Erdos-Renyi
- N_GRAPH : numero di grafi da generare attraverso il doubling, ovvero il numero di volte che viene raddoppiato il numero di nodi del grafo da generare

Per ogni grafo così generato verranno calcolati gli archi mancanti attraverso un semplice algoritmo random e ad ognuno di essi verrà assegnato un peso randomico compreso tra due valori definiti dalle seguenti variabili presenti nello script *utility.py*, cioè:

- MIN_EDGE_WEIGHT
- MAX_EDGE_WEIGHT

I grafi generati verranno salvati su file, attraverso metodi dedicati offerti dalla libreria networkit, in formato $.TabOne$. Invece gli archi mancanti, con i relativi pesi, verranno salvati in formato $.json$. La folder nella quale verranno salvati è configurabile nello script *utility.py* con le variabili:

- $BAGs\_FOLDER$: folder nella quale verranno salvati i grafi di tipo Barabasi-Albert in formato $.TabOne$
- $ERGs\_FOLDER$: folder nella quale verranno salvati i grafi di tipo Erdos-Renyi in formato $.TabOne$

E' stato necessario adottare tale strategia di generazione e salvataggio in quanto il tempo di generazione dei grafi era di poco inferiore al tempo di esecuzione dell'esperimento e ciò ha permesso un notevole risparmio di tempo nella sperimentazione degli algoritmi oggetto di studio. Proprio grazie al risparmio di tempo recuperato dalla fase di generazione dei grafi è stato possibile aumentare notevolmente la taglia degli stessi mantenendo un tempo di sperimentazione accettabile.

3.1.2 Phase 2 Graph Deserialize

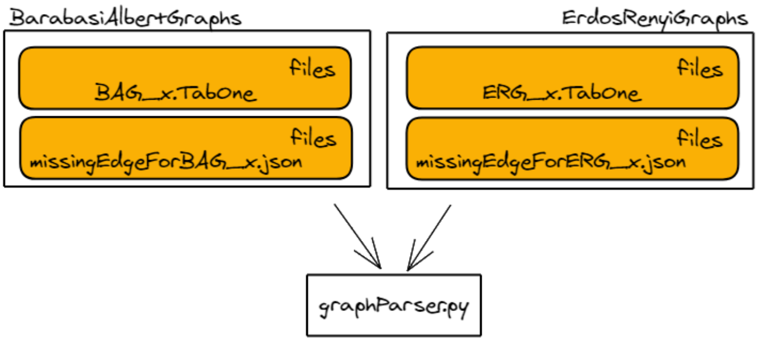

Figura 3: Phase 2: graphFiles to graphParser

Affinché i grafi e gli archi mancanti, salvati nella fase precedente, fossero utilizzabili è stato necessario implementare dei metodi di lettura da file. Per fare ciò è stata creata una classe di appoggio chiamata *GraphParser* e utilizzabile importando lo script *graphParser.py*. Tale oggetto si occupa di leggere da file, ricostruire e restituire al chiamante una lista così costituita: $[indice, grafo, lista\_archi\_mancanti]$ relativa al *GraphType* passato in input.

3.1.3 Phase 3 Run Test Dijkstra Algorithms

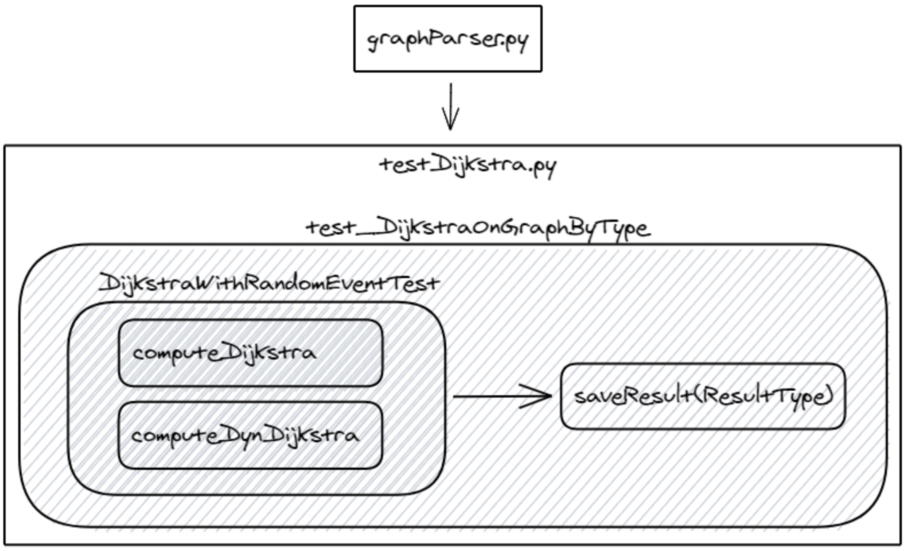

Figura 4: Phase 3: graphParser to testDijkstra

Lo script *testDijkstra.py* implementa il core di questo studio sperimentale. In particolare si occupa di eseguire l'algoritmo di Dijkstra e l'algoritmo Dyn-Dijkstra sui grafi precedentemente generati. Il flusso della sperimentazione è il seguente:

```python
def test_DijkstraOnGraphByType(graphType):
    if(isinstance(graphType, GraphTypes) == False):
        return

    parser = graphParser.file_Parser()

    while(True):
        response = parser.getNextByType(graphType)

        if(response[0] == "no_more_graphs" or response == "not_exist"):
            if logger.isEnabledFor(logging.DEBUG):
                logger.debug(f"break: {response}")
            break
        elif(isinstance(response[0], int) and response[0] <= utility.GRAPH_TO_CHECK - 1):
            index, graph, randomMissingEdgeList = response
            # Static Dijkstra test - worst-case time O(m+n logn) con Fib. Heap
            g0_nodes = graph.numberOfNodes()
            g0_edges = graph.numberOfEdges()

            static_result_list, dynamic_result_list = DijkstraWithRandomEventTest(graph, utility.EVENT_NUMBER_IN_EXP, randomMissingEdgeList)
```

Figura 5: test_DijkstraOnGraphByType function

- Viene creato l'oggetto graphParser precedentemente illustrato
- Viene eseguita la function DijkstraWithRandomEventTest finchè il graphParser trova grafi generati in precedenza e tali grafi vengono passati alla funzione. Un ulteriore input è il numero di esperimenti. Tale numero rappresenta la quantità di eventi di aggiunta di archi o decremento del peso di archi esistenti del grafo ed è un parametro fisso configurabile nello script *utility.py*

```python
def DijkstraWithRandomEventTest(graph, event_number, missing_edge_to_add):
    # Dijkstra’s SSSP algorithm
    static_computing_time_list = []
    # new local graph instance foreach experiment
    localGraph = copy.copy(graph)

    # DynDijkstra’s SSSP algorithm
    dynamic_computing_time_list = []
    # new local graph instance foreach experiment
    dyn_localGraph = copy.copy(graph)

    # first dijkstra computation without adding missing edge
    sssp = networkit.distance.Dijkstra(localGraph, 0)

    # first DynDijkstra computation without adding missing edge
    dynSssp = networkit.distance.DynDijkstra(dyn_localGraph, 0)

    if(utility.COMPUTE_FIRST_ALGO_RUN):
        static_computing_time_list.append(("FIRST_RUN", computeDijkstra(sssp)))
        dynamic_computing_time_list.append(("FIRST_RUN", computeDynDijkstra(dynSssp, "FIRST_RUN")))
    else:
        sssp.run()
        dynSssp.run()

    if logger.isEnabledFor(logging.DEBUG):
        assert dynSssp.getDistances() == sssp.getDistances()
```

Figura 6: DijkstraWithRandomEventTest function

La prima parte della funzione DijkstraWithRandomEventTest si occupa di

- allocare le strutture dati contenenti i grafi
- eseguire gli algoritmi di Dijkstra su tali grafi

Questa esecuzione non viene presa in considerazione in quanto, essendo la prima, sia l’algoritmo dinamico che quello statico impiegano lo stesso tempo e dunque non aggiungono informazioni utili al test.

```python
for i in range(event_number):

    if logger.isEnabledFor(logging.DEBUG):
        logger.debug(f"edge_addition_counter: {edge_addition_counter}")

    # scelgo randomicamente uno dei 2 eventi tra EDGE_ADDITION, EDGE_WEIGHT_UPDATE
    event = getRandomGraphEdgeEvent()

    # scelgo randomicamente un nodo sorgente da cui calcolare sssp per ammortizzare il bias
    random_source_node = networkit.graphtools.randomNode(localGraph)

    sssp.setSource(random_source_node)
    dynSssp.setSource(random_source_node)

    if(event == networkit.dynamic.GraphEvent.EDGE_ADDITION):
        edge_addition_counter = handleEdgeAdditionEvent(event, localGraph, dyn_localGraph, sssp, dynSssp,
        static_computing_time_list, dynamic_computing_time_list,
        edge_addition_counter, missing_edge_to_add)
    elif(event == networkit.dynamic.GraphEvent.EDGE_WEIGHT_UPDATE):
        handleEdgeWeightUpdateEvent(event, localGraph, dyn_localGraph, sssp, dynSssp,
        static_computing_time_list, dynamic_computing_time_list)
    elif(event == networkit.dynamic.GraphEvent.EDGE_REMOVAL):
        handleEdgeRemovalEvent(event, localGraph, dyn_localGraph, sssp, dynSssp,
        static_computing_time_list, dynamic_computing_time_list)
    elif(event == networkit.dynamic.GraphEvent.EDGE_WEIGHT_INCREMENT):
        handleEdgeWeightIncrementEvent(event, localGraph, dyn_localGraph, sssp, dynSssp,
        static_computing_time_list, dynamic_computing_time_list)
```

Figura 7: DijkstraWithRandomEventTest function

La seconda parte della funzione DijkstraWithRandomEventTest si occupa invece di

- scegliere randomicamente tra due possibili eventi di modifica del grafo
- scegliere randomicamentre il nodo sorgente, dal quale verrà effettuato Single Source Shortest Path, tra tutti i possibili nodi del grafo. Tale scelta è stata fatta per evitare il bias statistico causato dal nodo sorgente, ovvero per evitare che la casualità degli eventi di modifica del grafo rendessero meno oneroso il calcolo da parte dell'algoritmo DynDijkstra da nodo sorgente fisso
- gestire l'evento scelto randomicamente attraveso i rispettivi metodi handle

```python
def handleEdgeAdditionEvent(event, localGraph, dyn_localGraph, sssp, dynSssp,
        static_computing_time_list, dynamic_computing_time_list, edge_addition_counter, missing_edge_to_add):
    if logger.isEnabledFor(logging.DEBUG):
        assert localGraph.numberOfNodes() == dyn_localGraph.numberOfNodes()
    if logger.isEnabledFor(logging.DEBUG):
        assert localGraph.numberOfEdges() == dyn_localGraph.numberOfEdges()
    # controllo che gli archi aggiunti siano minori della taglia dei missingEdges calcolati e salvati nel json
    if logger.isEnabledFor(logging.DEBUG):
        assert edge_addition_counter < missing_edge_to_add.index.size

    from_node = missing_edge_to_add['from_node'][edge_addition_counter]
    to_node = missing_edge_to_add['to_node'][edge_addition_counter]
    weight = missing_edge_to_add['weight'][edge_addition_counter]

    # controllo che i nodi dell'arco da aggiungere siano diversi
    if logger.isEnabledFor(logging.DEBUG):
        assert from_node != to_node
    # controllo che l'arco non sia presente in entrambi i grafi
    if logger.isEnabledFor(logging.DEBUG):
        assert localGraph.hasEdge(from_node,to_node) == dyn_localGraph.hasEdge(from_node, to_node)

    # EDGE ADDITION EVENT
    localGraph.addEdge(from_node, to_node, weight)
    dyn_localGraph.addEdge(from_node, to_node, weight)

    edge_addition_counter+=1
    static_computing_time_list.append(("EDGE_ADDITION", computeDijkstra(sssp)))

    dyn_event = networkit.dynamic.GraphEvent(event, from_node, to_node, weight)
    dynamic_computing_time_list.append(("EDGE_ADDITION", computeDynDijkstra(dynSssp, dyn_event)))

    # controllo che l'arco sia stato aggiunto in entrambi i grafi
    if logger.isEnabledFor(logging.DEBUG):
        assert localGraph.hasEdge(from_node,to_node) == dyn_localGraph.hasEdge(from_node, to_node)
    # controllo che il peso dell'arco aggiunto sia uguale in entrambi i grafi
    if logger.isEnabledFor(logging.DEBUG):
        assert localGraph.weight(from_node, to_node) == dyn_localGraph.weight(from_node, to_node)
    # controllo che le distanze dei cammini minimi siano uguali
    if logger.isEnabledFor(logging.DEBUG):
        assert dynSssp.distance(to_node) == sssp.distance(to_node)
    # controllo che il peso totale dei due grafi sia uguale
    if logger.isEnabledFor(logging.DEBUG):
        assert localGraph.totalEdgeWeight() == dyn_localGraph.totalEdgeWeight()

    return edge_addition_counter
```

Figura 8: handleEdgeAdditionEvent function

Le funzioni handle si occupano di gestire i vari eventi di modifica del grafo e nel seguito verrà presentato il funzionamento della function che gestisce l'evento di aggiunta di un arco nel grafo. In particolare, si occupa di

- prendere un arco dalla lista di archi mancanti letta in precedenza da file
- aggiungere tale arco al grafo
- eseguire le funzioni *computeDijkstra* e *computeDynDijkstra* le quali si occupano di eseguire i rispettivi algoritmi e restituire il tempo di esecuzione impiegato per tale computazione. Per evitare che la misura fosse influenzata da interrupt di sistema e altri processi presenti nella macchina sulla quale sono stati eseguiti i test è stato scelto di utilizzare come metro di misura il CPU time.

Ulteriori precisazioni sono necessarie riguardo la correttezza e l'efficienza dei test. Per quanto riguarda la correttezza è sono state utilizzate in modo intensivo le assertion le quali hanno permesso di verificare la correttezza dei passaggi per le modifiche sul grafo. Inoltre è stato possibile validare i risultati dei due algoritmi asserendo che le distanze da loro calcolate dovessero essere necessariamente uguali. Proprio attraverso la validazione effettuata è stato possibile scoprire che l'algoritmo DynDijkstra oggetto di test non gestisce correttamente l'evento di incremento del peso e di rimozione di un arco restituendo distanze dei cammini minimi non corrette. Per tale motivo è stato necessario limitare il test di tale algoritmo agli eventi di aggiunta di un arco e decremento del peso di un arco. Per quanto riguarda l'efficienza dei test, l'uso intensivo delle assertion e delle print su console ha avuto un impatto molto negativo sul tempo di esecuzione. Per tale motivo è stato utilizzata utilizzata una libreria python per implementare il sistema di logging. Quando il logger è settato in modalità DEBUG è possibile osservare i print su console per valutare il comportamento del test a runtime e assicurarci tramite le assertion che l'ambiente di test funzioni correttamente. Invece quando il logger è configurato in modalità INFO vengono saltati i passaggi precedentemente descritti e questo ha portato ad un risparmio notevole soprattutto sui running time dei due algoritmi. Di seguito i tempi di esecuzione dell'intero test (generazione, esecuzione, plot) nelle configurazioni rispettivamente di DEBUG e INFO eseguiti su 6 grafi Erdos-Renyi con numero di nodi (500, 1000, 2000, 4000, 8000, 16000) e 35000 eventi eseguiti per ogni grafo:

```
test ended in 1065.175002163 seconds
```

Figura 9: running time with logger in DEBUG mode

```
test ended in 628.77793786 seconds
```

Figura 10: running time with logger in INFO mode

3.1.4 Phase 4 Serialize Test Data

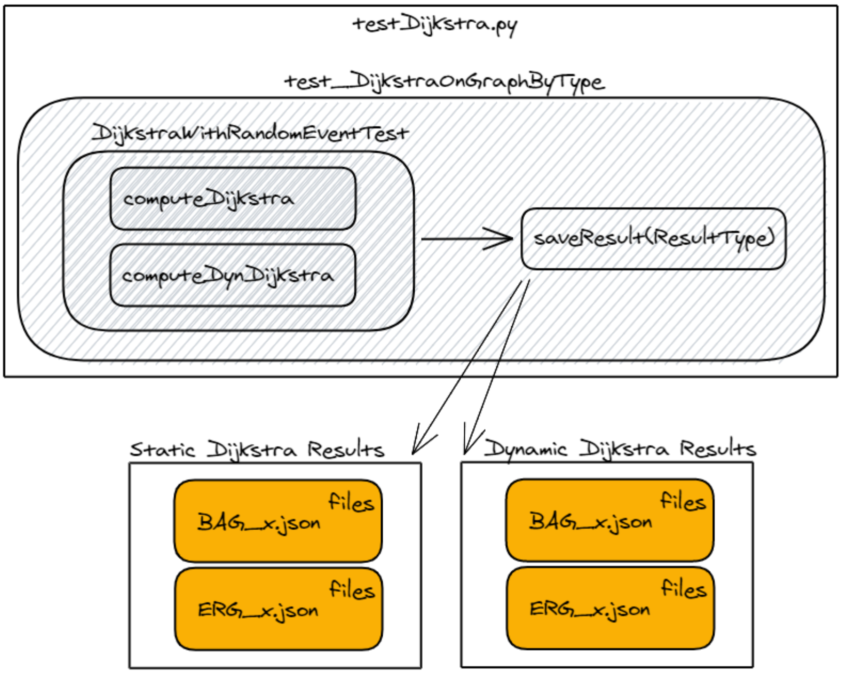

Figura 11: Phase 4: testDijkstra to resultFiles

Alla fine dell'esecuzione della funzione *DijkstraWithRandomEventTest* verranno restituite due liste, contenenti rispettivamente i running time impiegati dall'algoritmo di Dijkstra e DynDijkstra, le quali verranno salvate su file insieme alle informazioni relative alla sperimentazione effettuata. In particolare, verranno generati due file in formato json, uno per ogni tipo di algoritmo eseguito, contenenti le seguenti informazioni:

- graph_type: GraphType
- graph_number: index
- nodes: numero di nodi del grafo
- edges: numero di archi del grafo
- total_weight: peso totale del grafo
- result_list: lista contenente tutti i running time dell'algoritmo per ogni evento di aggiunta/decremento del peso

3.1.5 Phase 5 Plot Results

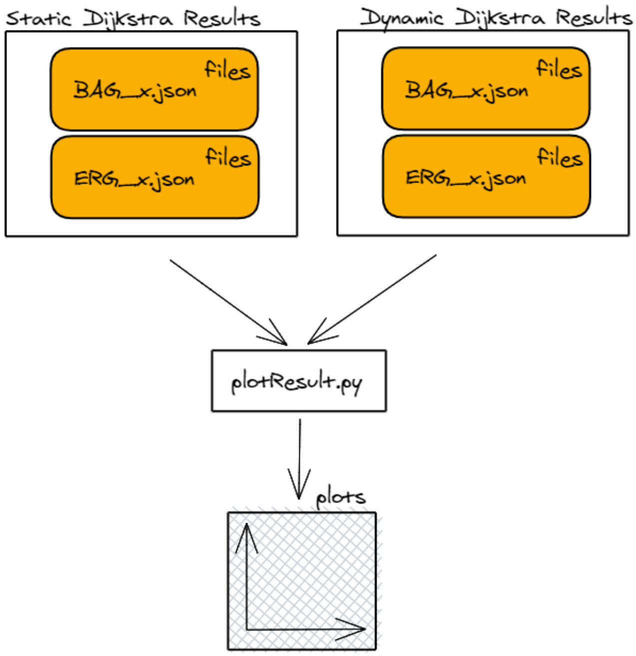

Figura 12: Phase 5: resultFiles to plots

Lo script *plotResult.py* si occupa di leggere i file json generati dai test, manipolare tali dati e plottare i grafici opportuni che verranno discussi nel paragrafo successivo. La separazione in fasi del test enviromment ha consentito lo sviluppo e la validazione degli script senza la necessità di attendere il tempo necessario all'elaborazione degli algoritmi.

4 Risultati

I risultati sono stati ottenuti eseguendo gli esperimenti su un notebook con le seguenti caratteristiche hardware:

- CPU Intel Core i9-9980HK (8 Core, 16 Thread, 5Ghz)
- GPU NVIDIA GeForce GTX 1650 (4 GB GDDR5)
- RAM 16 GB DDR4 2666MHz
- Disco a stato solido M.2 2230/2280 (PCIe Gen3.0x4 NVMe, 32 Gb/s)

Al fine di quantificare l'efficienza dell'algoritmo di Dijkstra dinamico rispetto a quello statico, verranno di seguito riportati i grafici dei risultati ottenuti dai test precedentemente illustrati. In particolare verranno distinti i risultati ottenuti eseguendo i due algoritmi su grafi generati secondo Barabasi-Albert da Erdos-Renyi.

Per prima cosa, per validare i due algoritmi, è stato confrontato il running time dei due algoritmi con la complessità nota in letteratura (in termini asintotici). Sono stati riportati due grafici all'interno della stessa figura. Il primo per evidenziare la differenza dei running time tra i due algoritmi. Il secondo per vedere quale andamento ha il running time dell'algoritmo dinamico, che a causa dei diversi ordini di grandezza con quello statico, non è ben comprensibile dalla prima figura.

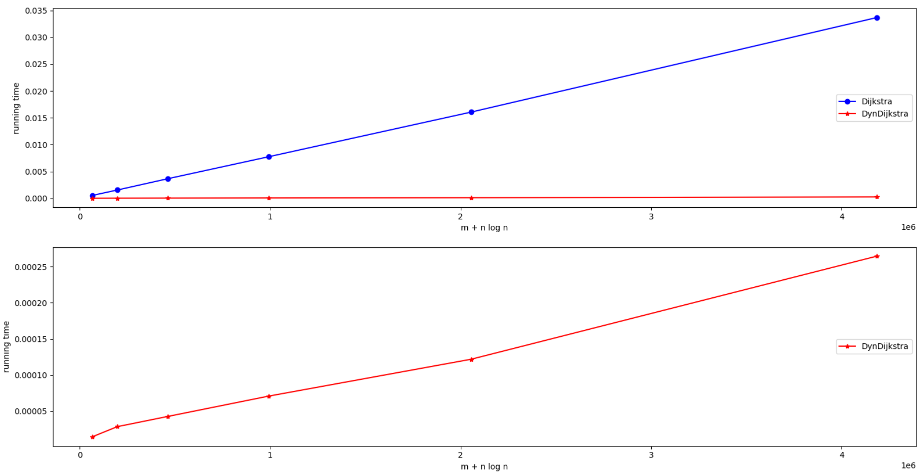

Figura 13: Barabasi-Albert Graph: Asymptotic Comparison

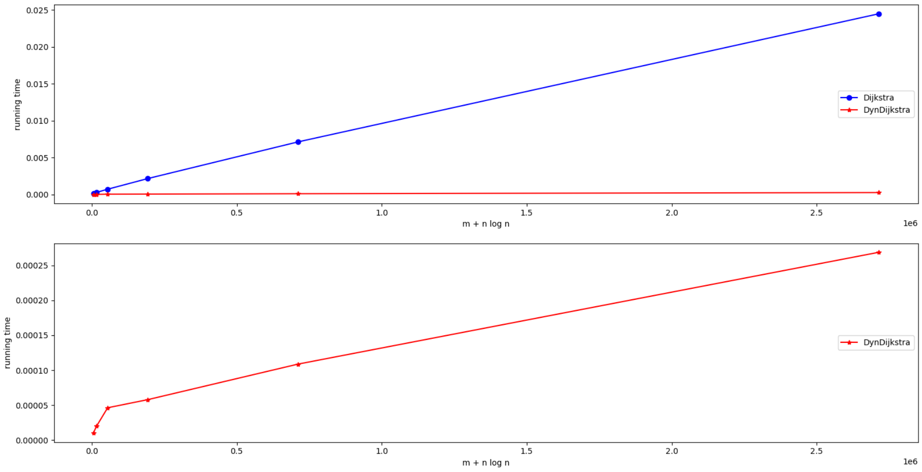

Figura 14: Erdos-Renyi Graph: Asymptotic Comparison

Successivamente è stato ritenuto opportuno osservare come lo speedup varia al variare del numero dei nodi del grafo. In particolare il numero dei nodi del grafo è stato scelto attraverso la tecnica del doubling experiment.

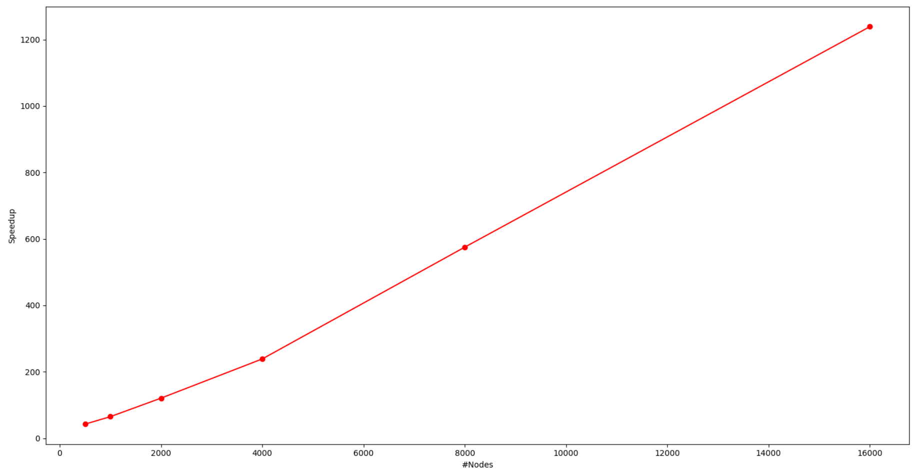

Figura 15: Barabasi-Albert Graph: #Nodes vs Speedup

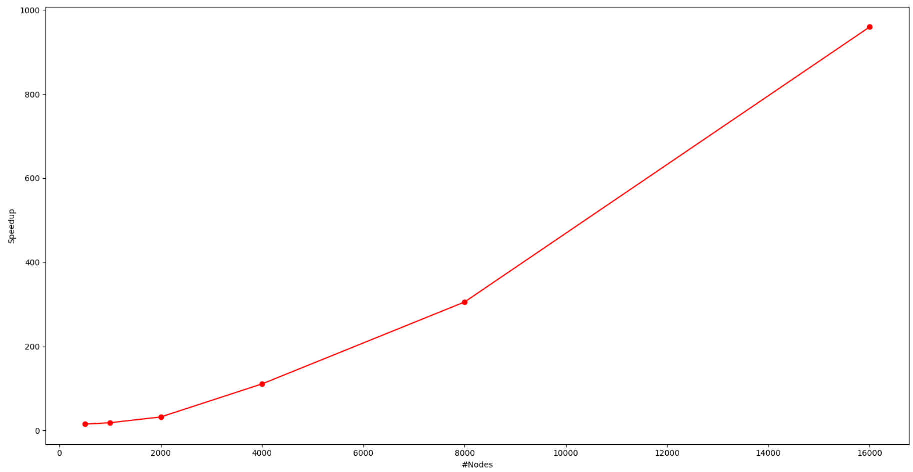

Figura 16: Erdos-Renyi Graph: #Nodes vs Speedup

Infine è stato utilizzato un modello di regressione polinomiale per stabilire l'andamento dello speedup tra i due algoritmi. Ogni punto arancione in figura rappresenta lo speedup tra i due algoritmi ad ogni evento di modifica del grafo. In particolare, per ogni taglia di grafo, sono stati eseguiti esattamente 35000 eventi riportati nell'ordine in cui si sono succeduti.

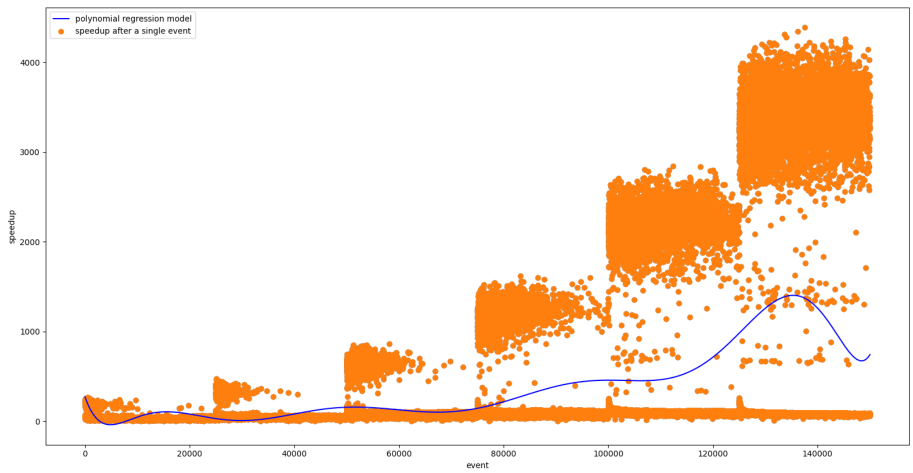

Figura 17: Barabasi-Albert Graph: event vs Speedup

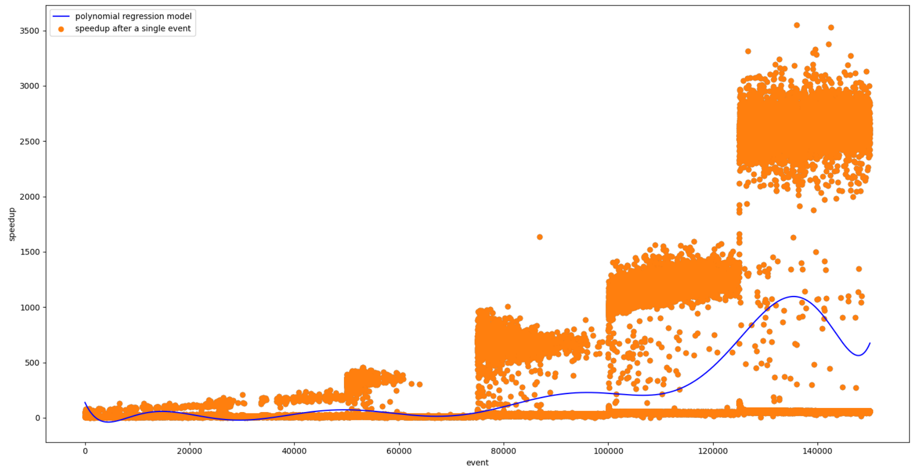

Figura 18: Erdos-Renyi Graph: event vs Speedup

5 Conclusioni e sviluppi futuri

Dall'analisi dei risultati ottenuti è possibile concludere, limitatamente agli esperimenti condotti, che:

- Lo speedup tra i due algoritmi varia dal trascurabile a tre ordini di grandezza
- L'algoritmo dinamico, seppur molto più veloce in determinate situazioni, presenta comunque un andamento tipico di Dijkstra che varia da sublineare a quadratico in base alla densità del grafo
- dalla figura 15 e 16 non vi è una differenza sufficiente per concludere che i grafi Barabasi-Albert e Erdos-Renyi influenzano l'andamento dell'algoritmo dinamico rispetto a quello statico. Nonostante ciò, si può comunque concludere che lo speedup aumenta in modo lineare all'aumentare dei nodi di un grafo purché non aumenti in modo rilevante la densità.
- Il numero di eventi di modifica rispetto alla grandezza del nodi del grafo ne influenza in modo significativo l'andamento. In particolare: - Aggiungere molti archi in un grafo piccolo fa decrescere lo speedup. A conferma del fatto che in un grafo completo lo speedup dei due algoritmi è circa uguale se non negativo (a causa del fatto che l'algoritmo dinamico avrà un'overhead dovuto all'identificazione delle variazioni). - Al crescere delle dimensioni dei grafi lo speedup aumenta (mediamente) come già visto in precedenza ma , dalle figure 17 e 18, emergono dati importanti circa la distribuzione degli eventi molto, poco e mediamente onerosi in termini di tempo. In particolare, ci sono eventi a causa dei quali l'algoritmo dinamico deve ricalcolare tutto esattamente come quello statico. Infatti, ci sono eventi al di sotto dello 0 nell'asse delle ascisse proprio perchè, a fronte di quell'evento, l'algoritmo dinamico è stato addirittura più lento di quello statico. - E' intuibile ma non dimostrabile con certezza attraverso i dati ottenuti che lo speedup crescerà fino al suo massimo teorico nel caso di

un grafo completo a fronte di un singolo evento ottimo ma mediamente tenderà a scendere man mano che si ci si avvicina alle condizioni di completezza.

Questi risultati offrono quel minimo di dati che consentono di fare delle scelte guidate, quantificandone a priori i risultati che sarà possibile ottenere, li dove vi è l'esigenza di scegliere tra l'enorme complessità di implementazione di un algoritmo dinamico e l'accontentarsi di determinati running time a vantaggio di una complessità di implementazione minore.

Eventuali sviluppi futuri potrebbero essere:

- ripetere i test con particolari e contestualizzati set di grafi
- parallelizzare i test su più core e/o dispositivi li dove possibile
- approfondire la ricerca di caratteristiche determinanti e/o correlazioni tra parametri attraverso studi workhorse mirati