Documentazione del Progetto di Elementi di Intelligenza Artificiale

Adriano Oliviero N46006115

2 Luglio 2024

Contents

1 Obiettivo del progetto

Il progetto si propone di applicare algoritmi di ricerca ad alcuni dataset disponibili su `https://snap.stanford.edu/data/` per valutare l'efficacia e l'efficienza di tali algoritmi.

2 Descrizione delle metodologie e tecniche adoperate

2.1 Il linguaggio

Per la realizzazione del progetto, ho utilizzato il linguaggio di programmazione Rust.

Questo linguaggio è stato scelto per diversi motivi:

- **Performance**: È un linguaggio di programmazione ad alte prestazioni con gestione automatica della memoria.
- **Semplicità**: È moderno con una sintassi pulita e concisa, compatibile con alti livelli di astrazione, più difficile da scrivere rispetto a Python, ma più facile di C/C++.
- **Esperienza**: Ho già esperienza con Rust e ritengo che sia un linguaggio adatto per progetti di questo tipo.

2.2 Software di profilazione

Al fine della valutazione dell'efficienza degli algoritmi, ho scelto di misurare il tempo di esecuzione degli stessi utilizzando la libreria `std::time` di Rust.

In aggiunta a questa metrica, ho scelto di misurare anche la quantità di memoria utilizzata dagli algoritmi, utilizzando un software esterno al progetto: `valgrind` con il tool `massif`. Tuttavia, tali tool, generano un enorme overhead, infatti l'esecuzione dei cinque algoritmi sui sette dataset (35 esecuzioni), ha richiesto precisamente 8 ore, 54 minuti e 53 secondi.

Ho quindi deciso di avviare una seconda volta il programma, disattivando `valgrind`, per generare degli output che restituiscano misurazioni più precise riguardo al tempo di esecuzione. Questa seconda esecuzione ha richiesto 37 minuti e 21 secondi.

Per la generazione dei grafici che si basano sui dati ottenuti dalle misurazioni con `valgrind`, ho utilizzato la libreria `matplotlib` di Python. I file LaTeX `grafici.tex` e `risultati.tex` sono stati generati automaticamente dallo script Python fornito nel progetto.

2.3 Organizzazione del progetto

Il progetto è organizzato come segue:

- `src/` - Directory contenente il codice sorgente in Rust, nel quale sono effettivamente implementati gli algoritmi.
- `run.py` - Script Python per eseguire il progetto, e generare benchmark e grafici.
- file aggiuntivi

2.4 Compilazione ed esecuzione

Per compilare ed eseguire il progetto, è necessario avere Rust –e il suo package manager, Cargo– installati sul proprio sistema. Inoltre è necessario scaricare almeno un dataset, come descritto nella sezione dedicata.

Se le dipendenze sono soddisfatte, è possibile procedere con la compilazione ed esecuzione del progetto:

- Al fine di compilare il progetto, è possibile eseguire il seguente comando:

  ```
  $ cargo build --release
  ```

- Al fine di eseguire il programma compilato, è possibile eseguire il seguente comando:

  ```
  $ ./target/release/eia <opzioni>
  ```

Se si desidera consultare una lista delle opzioni disponibili, è possibile eseguire il programma con il flag `-h` o `--help`:

```
$ ./target/release/eia --help
```

Inoltre, è possibile eseguire lo script Python fornito per avviare automaticamente alcuni o tutti gli algoritmi alcuni o tutti i dataset, e generare grafici e benchmark:

```
$ python3 run.py <lista dei dataset> <lista degli algoritmi>
```

2.5 Strutture dati

Per rendere utilizzabili i dataset, ho implementato le seguenti strutture dati:

- `State: u32` - Un semplice alias per rendere più leggibile il codice.
- `Action` - Struttura dati per rappresentare un'azione:
  - `risultato: State` - Stato risultante dall'azione.
  - `costo: i32` - Costo dell'azione.
- `Node` - Struttura dati per rappresentare un nodo del grafo:
  - `stato: State` - Stato corrispondente al nodo.
  - `azioni: Vec<Action>` - Azioni possibili dal nodo.
  - `genitore: Node` - Nodo genitore, utile per risalire il percorso.
  - `costo_cammino: i32` - Costo del cammino partendo dal nodo iniziale per raggiungere il nodo.
  - `profondita: usize` - Profondità del nodo rispetto al nodo iniziale.
- `Graph` - Struttura dati contenente una astrazione del grafo:
  - `gtype: String` - Tipo del grafo (direzionato, non direzionato o con pesi).
  - `nodi: Vec<Node>` - I nodi del grafo.
  - `edge_count: u32` - Il numero di archi del grafo, utile per essere sicuri che il caricamento del dataset sia avvenuto correttamente.
  - `load_dataset(dataset_path)` - Legge il file del dataset e costruisce il grafo.
- `Problem` - Struttura dati contenente il grafo e i dati e le funzioni necessarie per la ricerca:
  - `stato_inziale: State` - Nodo dal quale iniziare la ricerca.
  - `stato_finale: State` - Nodo da raggiungere.
  - `grafo: Graph` - Struttura dati contenente il grafo.
  - `limite: usize` - Limite di profondità per gli algoritmi di ricerca limitata.
  - `goal_test(&self, stato) -> bool` - Funzione per verificare se il nodo obiettivo è stato raggiunto.
  - le funzioni di ricerca, delle quali parlerò più avanti.

2.6 Algoritmi di ricerca

Gli algoritmi di ricerca che ho scelto di implementare sono:

- **Tree Search (tree-search)** - Ricerca semplice, per default disattivata a causa della sua eccessiva inefficienza.
- **Breadth-First Search (breadth-first)** - Ricera in ampiezza.
- **Uniform-Cost Search (uniform-cost)** - Ricerca a costo uniforme. Differisce dal Breadth-First Search esclusivamente nel caso di grafi pesati.
- **Depth-Limited Search (depth-limited)** - Ricerca in profondità limitata.
- **Iterative Deepening Depth-First Search (iterative-deepening)** - Ricerca in profondità iterativa.
- **Bidirectional Search (bi-directional)** - Ricerca bidirezionale.

Tutti gli algoritmi sono compatibili sia con grafi direzionati che non direzionati, e con grafi pesati.

3 Dataset

I dataset utilizzati sono stati scaricati dal sito dell'università di Stanford (`https://snap.stanford.edu/data/`). Per scaricare i dataset, è possibile procedere manualmente recandosi alle reciproche pagine sul sito, oppure utilizzare lo script shell fornito nel progetto:

```
$ chmod +x ./download-datasets.sh
$ ./download-datasets.sh
```

Lo script utilizza `wget` ed è scritto per sistemi UNIX & UNIX-like.

3.1 Dataset Utilizzati

Nome	Nodi	Archi	Tipologia	Dimensione
soc-sign-bitcoin-alpha	3783	24186	Con pesi	152KB
email-Enron	36692	183831	Non direzionato	1.1MB
com-Youtube	1134890	2987624	Non direzionato	11MB
roadNet-CA	1965206	2766607	Direzionato	18MB
as-Skitter	1696415	11095298	Non direzionato	33MB
cit-Patents	3774768	16518948	Direzionato	85MB
com-LiveJournal	3997962	34681189	Non direzionato	124MB

Table 1: Dataset Utilizzati

I dataset contengono alcune informazioni nelle prime righe. Sono in formato `txt` con compressione `.gz` e le proprie righe sono formate da due numeri (Nodo Sinistro e Nodo Destro), ad eccezione del dataset soc-sign-bitcoin-alpha, che è in formato `csv` con le colonne:

- **SOURCE** (id del nodo Sinistro),
- **TARGET** (id del nodo Destro),
- **RATING** (il costo delle azioni),
- **TIME** (non rilevante).

4 Codice Sviluppato

Il codice sviluppato è stato consegnato insieme alla documentazione del progetto e può essere consultato nei file allegati.

Alternativamente è possibile trovare il codice sorgente su GitHub al seguente indirizzo:
ad-oliviero/progetto_eia

5 Risultati sperimentali

I risultati ottenuti sono stati valutati in termini di efficacia ed efficienza, come descritto di seguito. Essi sono organizzati in ordine di completamento.

Inoltre, sono stati generati grafici automaticamente dallo script Python fornito nel progetto. È possibile visualizzarli nel documento grafici.pdf

5.1 soc-sign-bitcoinalpha.csv.gz

Tipo di Grafo: Labeled
Durata caricamento: 0.024s
Nodi cercati: 1496 e 160

Algoritmo	Risultato	Profondità	Costo	Tempo
breadth-first	Trovato	2	15	0.498004s
uniform-cost	Trovato	2	15	0.447142s
depth-limited	Trovato	10	95	1.540672s
iterative-deepening	Trovato	2	15	0.491949s
bi-directional	Trovato	3	10	0.14705s

Table 2: soc-sign-bitcoinalpha.csv.gz

5.2 email-Enron.txt.gz

Tipo di Grafo: Directed
Durata caricamento: 0.208s
Nodi cercati: 1 e 44

Algoritmo	Risultato	Profondità	Costo	Tempo
breadth-first	Trovato	1	0	0.348722s
uniform-cost	Trovato	1	0	0.445537s
depth-limited	Trovato	9	0	2.581543s
iterative-deepening	Trovato	1	0	0.402705s
bi-directional	Trovato	3	0	0.434386s

Table 3: email-Enron.txt.gz

5.3 com-youtube.ungraph.txt.gz

Tipo di Grafo: Undirected
Durata caricamento: 1.528s
Nodi cercati: 4 e 2

Algoritmo	Risultato	Profondità	Costo	Tempo
breadth-first	Trovato	1	0	498.187974s
uniform-cost	Trovato	1	0	447.720300s
depth-limited	Trovato	2	0	363.215622s
iterative-deepening	Trovato	1	0	359.271743s
bi-directional	Trovato	3	0	461.343379s

Table 4: com-youtube.ungraph.txt.gz

5.4 roadNet-CA.txt.gz

Tipo di Grafo: Directed
Durata caricamento: 1.441s
Nodi cercati: 419 e 420

Algoritmo	Risultato	Profondità	Costo	Tempo
breadth-first	Trovato	1	0	0.859953s
uniform-cost	Trovato	1	0	0.680200s
depth-limited	Trovato	10	0	16.347660s
iterative-deepening	Trovato	1	0	1.43092s
bi-directional	Trovato	3	0	1.267525s

Table 5: roadNet-CA.txt.gz

5.5 as-skitter.txt.gz

Tipo di Grafo: Undirected
Durata caricamento: 8.312s
Nodi cercati: 0 e 1

Algoritmo	Risultato	Profondità	Costo	Tempo
breadth-first	Trovato	1	0	126.375315s
uniform-cost	Trovato	1	0	117.507084s
depth-limited	Trovato	1	0	103.863850s
iterative-deepening	Trovato	1	0	103.447892s
bi-directional	Trovato	0	0	184.993955s

Table 6: as-skitter.txt.gz

5.6 cit-Patents.txt.gz

Tipo di Grafo: Directed
Durata caricamento: 8.695s
Nodi cercati: 3858244 e 3858246

Algoritmo	Risultato	Profondità	Costo	Tempo
breadth-first	Fallito	0	0	3.781682s
uniform-cost	Fallito	0	0	3.869857s
depth-limited	Fallito	0	0	3.24814s
iterative-deepening	Fallito	0	0	6.349138s
bi-directional	Fallito	0	0	5.25531s

Table 7: cit-Patents.txt.gz

5.7 com-lj.ungraph.txt.gz

Tipo di Grafo: Undirected
Durata caricamento: 15.680s
Nodi cercati: 3 e 1

Algoritmo	Risultato	Profondità	Costo	Tempo
breadth-first	Trovato	1	0	483.560797s
uniform-cost	Trovato	1	0	432.646101s
depth-limited	Trovato	1	0	374.565855s
iterative-deepening	Trovato	1	0	372.558098s
bi-directional	Trovato	0	0	468.17906s

Table 8: com-lj.ungraph.txt.gz

6 Conclusioni

6.1 Completezza

La completezza di un algoritmo indica se l'algoritmo è in grado di trovare una soluzione se essa esiste. Gli algoritmi breadth-first, uniform-cost, depth-limited, iterative-deepening e bi-directional sono completi, come dimostrato dai risultati sui vari dataset, ad eccezione del dataset `cit-Patents.txt.gz` dove, a causa di un errore nella selezione degli stati, i test non sono stati validi.

6.2 Complessità spaziale

La complessità spaziale misura la quantità di memoria richiesta da un algoritmo durante la sua esecuzione. Nel nostro studio, abbiamo osservato che:

- Gli algoritmi breadth-first e iterative-deepening presentano una complessità spaziale elevata nei grafi di grandi dimensioni come `com-youtube.ungraph.txt.gz` e `com-lj.ungraph.txt.gz`, richiedendo significative risorse di memoria.
- L'algoritmo bi-directional ha dimostrato di essere più efficiente in termini di memoria nei test sui grafi meno densi, come `soc-sign-bitcoinalpha.csv.gz` e `email-Enron.txt.gz`.

6.3 Complessità temporale

La complessità temporale rappresenta il tempo necessario per l'esecuzione di un algoritmo. Dai risultati ottenuti:

- Gli algoritmi depth-limited e bi-directional hanno mostrato una complessità temporale inferiore rispetto agli altri, specialmente su grafi di dimensioni ridotte come `soc-sign-bitcoinalpha.csv.gz` e `email-Enron.txt.gz`.
- Per grafi molto grandi, come `com-youtube.ungraph.txt.gz` e `com-lj.ungraph.txt.gz`, tutti gli algoritmi hanno evidenziato tempi di esecuzione prolungati, con bi-directional che si è dimostrato inefficace in alcuni casi.

6.4 Ottimalità

L'ottimalità si riferisce alla capacità di un algoritmo di trovare la soluzione migliore (più economica). Gli algoritmi uniform-cost, breadth-first e iterative-deepening hanno garantito soluzioni ottimali nei test condotti, confermando la loro efficienza nel trovare percorsi di costo minimo.

6.5 Nota sui risultati del dataset `cit-Patents.txt.gz`

A causa di un errore nella selezione degli stati per il dataset `cit-Patents.txt.gz`, i test eseguiti su questo dataset sono stati invalidati. Pertanto, i risultati ottenuti non possono essere utilizzati per trarre conclusioni affidabili riguardo le performance degli algoritmi su tale dataset.

References

[1] Jure J. Leskovec et al. “Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters”. In: *Internet Mathematics*. Vol. 6. 1. 2009, pp. 29–123.

[2] Benjamin Klimmt and Yiming Yang. “Introducing the Enron corpus”. In: *CEAS conference*. 2004.

[3] Srijan Kumar et al. “Edge weight prediction in weighted signed networks”. In: *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE. 2016, pp. 221–230.

[4] Srijan Kumar et al. “Rev2: Fraudulent user prediction in rating platforms”. In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM. 2018, pp. 333–341.

[5] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. “Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations”. In: *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2005.

[6] Jaewon Yang and Jure Leskovec. “Defining and Evaluating Network Communities based on Ground-truth”. In: *ICDM*. 2012.