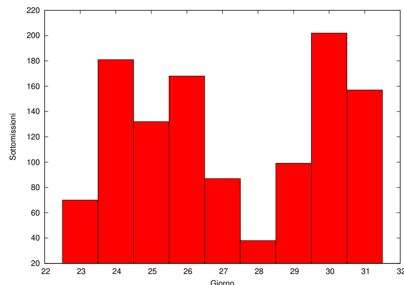
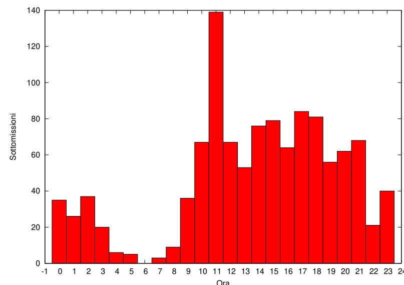
RIDERS PER IL CONCERTONE

Numero sottoposizioni: `1175`





- 51 gruppi hanno fatto almeno una sottoposizione, di cui 46 hanno raggiunto la sufficienza;
- 170 studenti iscritti, di cui 110 appartenenti a gruppi che hanno fatto almeno una sottoposizione;

Risultati

Punteggi

- $P < 30$ $\longrightarrow$ progetto non passato
- $30 \leq P < 63.8$ $\longrightarrow$ 1 punto bonus (9 gruppi)
- $63.8 \leq P < 75$ $\longrightarrow$ 2 punti bonus (17 gruppi)
- $75 \leq P < 77$ $\longrightarrow$ 3 punti bonus (13 gruppi)
- $P >= 77$ $\longrightarrow$ 3.5 punti bonus (7 gruppi)

Classifiche e sorgenti sul sito (controllate i numeri di matricola):
`https://judge.science.unitn.it/slides/asd21/classifica_prog2.pdf`

Il problema I

Riassumiamo brevemente il problema proposto in questo progetto.

- Abbiamo un grafo completo, pesato e indiretto, che rappresenta la nostra mappa di punti di ristoro con le rispettive strade;
- Ogni punto di ristoro ha un certo numero di clienti in attesa al tempo 0. Ogni punto di ristoro può essere visitato una sola volta. Ad ogni unità di tempo, un cliente se ne va;
- Abbiamo R rider che devono consegnare le pizze ai punti di ristoro;

Obiettivo

L'obiettivo è **massimizzare** il numero di clienti soddisfatti.

Il problema II

Multiple Traveling Repairman Problem with Profits

Il progetto è ispirato ad un problema conosciuto come Multiple Traveling Repairman Problem with Profits (MTRPP). MTRPP è un problema NP-HARD, quindi trovare la soluzione ottima per questo problema diventa rapidamente intrattabile.

- MTRPP è un problema simile al problema del Commesso Viaggiatore (TSP), con alcune difficoltà in più, come l'avere più di un commesso e il fatto di avere profitti che diminuiscono nel tempo;
- Le applicazioni di questo problema nel mondo reale sono svariate, spesso nel campo della logistica.

Verso la soluzione

Prima di passare alla nostra proposta di soluzione, vediamo qualche idea su come approcciare in generale il problema. Una cosa di cui tenere conto è che **per arrivare a una soluzione è necessario usare più di un rider.**

È sempre cosa buona e giusta partire implementando la soluzione più semplice che vi possa venire in mente: **una baseline.** Ci servirà per capire l'impatto di strategie o tecniche potenzialmente più furbe che andremo ad implementare in futuro, oltre che ad essere sicuri di ottenere i primi punti in classifica.

Finché ci sono nodi da visitare, eseguiamo il seguente algoritmo:

- Scegliamo un rider;
- Scegliamo il prossimo nodo che deve visitare.

SCELTE GREEDY (I)

Iniziamo con un algoritmo semplice:

- Prendiamo un rider alla volta in sequenza (round-robin);
- Mandiamo il rider al nodo che ci da il massimo profitto. Possiamo mantenere una coda di priorità per farci dare il prossimo nodo, bisogna solo tenere conto di quali nodi sono già stati assegnati agli altri rider.

Punteggio: **31.43**

Scelte greedy (II)

«Si può fare di meglio.» (cit.)

Come scegliamo **il prossimo rider**?

- Prendiamo un rider alla volta;
- Per fare una scelta (probabilmente) migliore, prendiamo il rider che ha percorso meno strada[1].

Come scegliamo **il prossimo nodo**? Scegliendo una di queste euristiche greedy:

- Il nodo più vicino a quello dove si trova il rider;
- Il nodo che massimizza l'aumento di punteggio;
- Il nodo con migliore rapporto $\frac{\textit{Profitto}}{\textit{Distanza}}$

Come baseline va bene una qualsiasi di queste euristiche.

Punteggio: **68.00**

[1] Tramite una priority queue

Miglioriamo la baseline

Abbiamo a disposizione 5 secondi per testcase. Perché limitarci ad utilizzare una sola euristica?

- **Diamo un secondo di tempo ad ogni euristica**;
- Essendo un algoritmo greedy sarà molto veloce, nell'ordine dei millisecondi. Finché non esauriamo il tempo a disposizione dell'euristica, **continuiamo a generare soluzioni seguendo questa euristica**, **randomizzando** la scelta del prossimo nodo;
- **Più passa il tempo, più randomizziamo la scelta del prossimo nodo**, in modo da generare soluzioni diverse tra loro per provare a trovarne di migliori;
- Ci teniamo da parte la soluzione migliore.

Miglioriamo la baseline

Miglioriamo un po' la scelta greedy del prossimo nodo:

- Invece di scegliere il nodo che massimizza l'euristica, **consideriamo più possibilità**;
- Consideriamo **i migliori percorsi di profondità** D visitabili dal nodo corrente;
- Scegliamo come nodo da visitare il **primo nodo appartenente al percorso migliore** tra quelli analizzati;
- Questo potrebbe **evitarci scelte** greedy **che avrebbero un effetto disastroso negli hop successivi**.

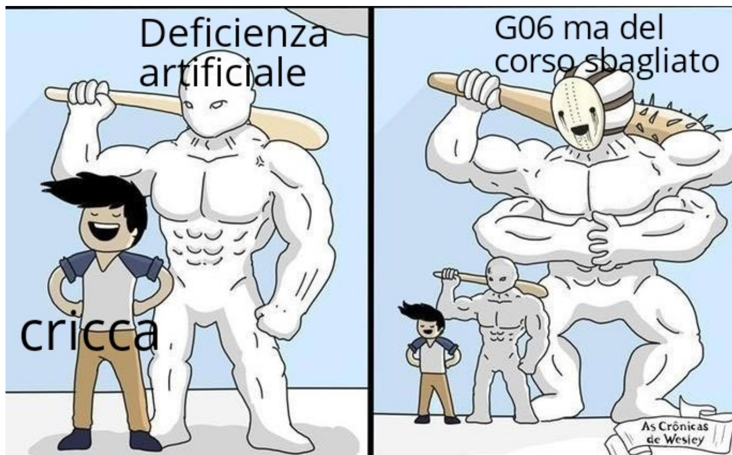
Ricerca locale

Ora che abbiamo in mano la migliore soluzione che abbiamo trovato generata da tre euristiche diverse randomizzate, proviamo a migliorarla con un po' di **ricerca locale**.

- Scelgo due nodi a caso **all'interno del percorso di un rider**. Cosa succede se li scambio?
- Scelgo due nodi a caso **appartenenti a due rider diversi**. Cosa succede se li scambio?
- Se il punteggio migliora, rendo lo scambio definitivo, se peggiora revoco lo scambio.

Punteggio: **77.15**

Mentre gli esercitatori si godono la Grill Valley, *Deficienza Artificiale* e *G06 ma del corso sbagliato* superano `cricca`.



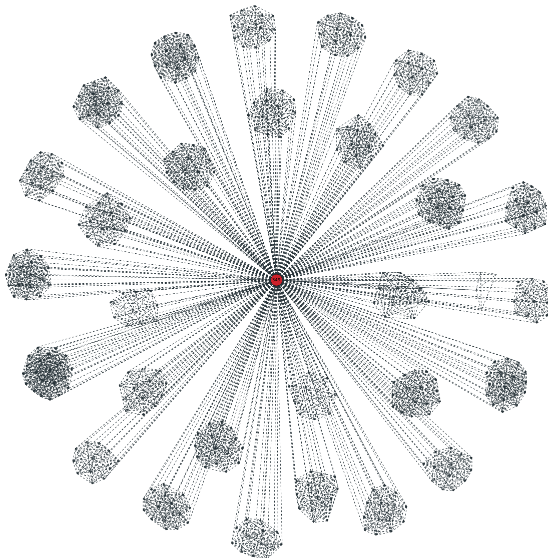
Non sempre le scelte greedy sono le migliori

Solitamente nei problemi NP-HARD le euristiche greedy funzionano molto bene nella maggior parte dei casi, ma ci sono dei **corner-case** che le rendono poco efficaci.

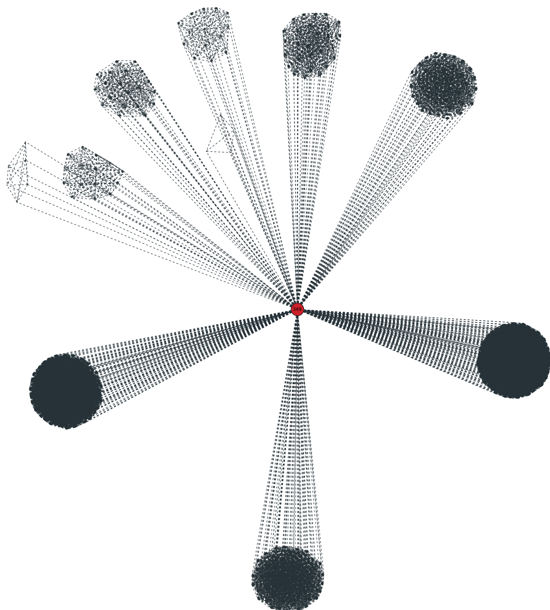
Avendo generato i testcase, noi sapevamo cosa dovevamo ottimizzare se volevamo avere dei risultati migliori. Voi invece avevate solamente il dataset di esempio. Quindi mettendoci nei vostri panni li abbiamo **visualizzati** per avere un'idea di come fossero fatti gli input sui quali il nostro algoritmo faceva più fatica.

- Essendo grafi completi, è difficile visualizzarli. Per questo abbiamo tolto tutti gli archi che con alta probabilità non sarebbero stati scelti dalle nostre euristiche (quelli troppo pesanti).

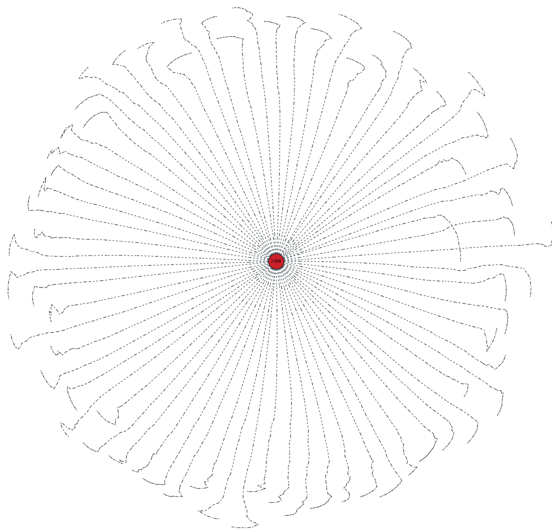
Input 11 - Comunità bilanciate



Input 12 - Comunità sbilanciate



Input 19 - Narrow paths



Clustering

Ci siamo resi conto che l'**assegnamento iniziale** è molto importante. Se due rider finiscono nella stessa comunità si spartiranno la ricompensa e dopo dovranno fare dei salti molto costosi verso altre comunità.

Quindi dividiamo il grafo in cluster:

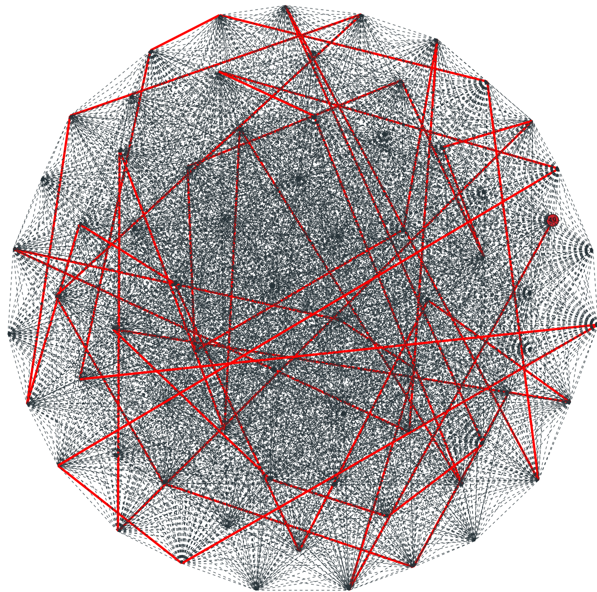
- Eliminiamo gli archi troppo pesanti[2];
- Facciamo partire un algoritmo per trovare le componenti connesse che rimangono[3] - i nostri cluster;
- Assegniamo ad ogni cluster un numero di rider proporzionato al numero di nodi in quel cluster[2].

Punteggio: **77.61**

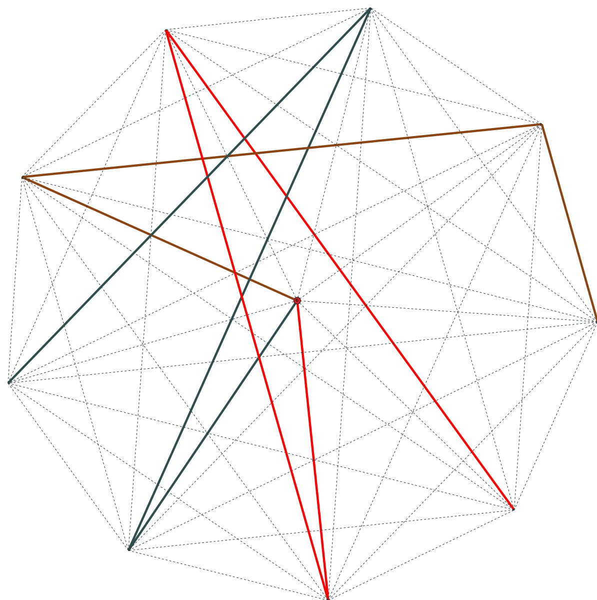
[2] È un po' più complicato di così

[3] Utilizziamo un merge-find set per essere più efficienti

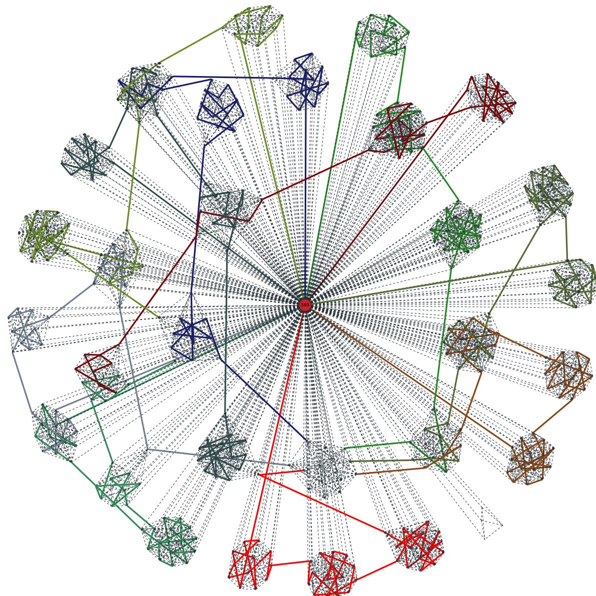
Testcase - Grafi random, singolo rider



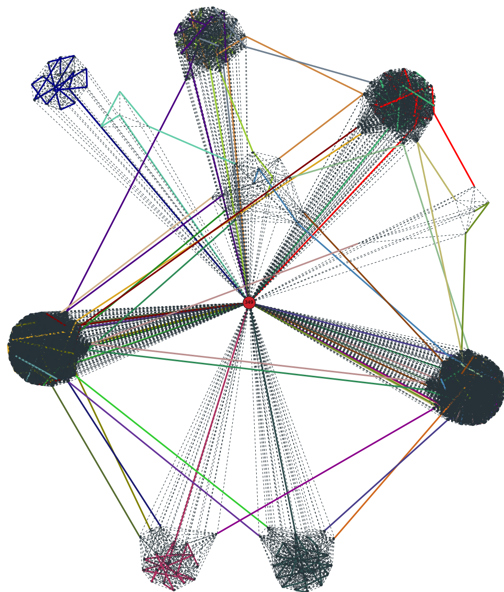
Testcase - Grafi random, rider multipli



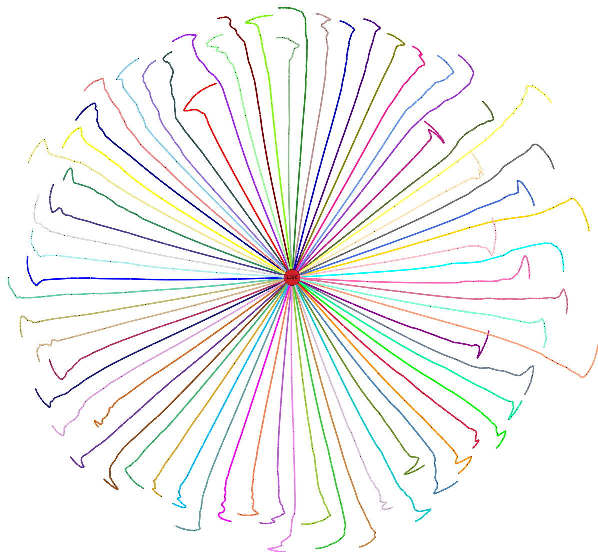
Testcase - Grafi di comunità, bilanciati



Testcase - Grafi di comunità, sbilanciati



Testcase - Narrow paths



Per battere i Turchi serve allenamento

- Si potevano modellizzare tutti i percorsi come una permutazione con R-1 nodi S e ogni altro nodo una volta sola.
  - Es: 1, 3, 2, 4, 5
- Quindi bisogna ottimizzare quella permutazione, con un algoritmo che non richieda di provarle tutte (max 4000! combinazioni)
- Un noto algoritmo per approssimare una buona soluzione su questo tipo di problemi è il Simulated Annealing

```
T = beginTemp()
curPos = new pos()
while T > 0 do
    nextPos = modify(curPos)
    delta = evaluate(nextPos) − evaluate(curPos)
    if delta > 0 or random(0.0, 1.0) < exp(delta/T) then
        curPos = nextPos
    end if
    T.update()
end while
```

Deficienza Artificiale - Ottimizzazioni

- Da solo il precedente algoritmo non era molto efficiente (63.08)
- Ma eseguito dopo il Greedy (e con un bel po' di tuning dei parametri) funziona abbastanza bene (77.21)