



UNIVERSITY  
OF FERRARA  
- EX LABORE FRUCTUS -

DE Department of  
Engineering  
Ferrara

# Ricerca Operativa

**Carpooling**


Università degli Studi di Ferrara, Ingegneria Informatica e dell'Automazione

Andrea Bazerla - 151792

# Progetto

## Briefing

“32. Car Pooling (2 persone) Sono noti i luoghi di origine delle persone che si recano quotidianamente al Polo Scientifico Tecnologico UniFe a inizio giornata (stessa ora di arrivo). Sono noti quelli disponibili ad operare come autisti in un servizio di Car Pooling. Tutti gli altri sono interessati a viaggiare come passeggeri e prendono la propria macchina solo se nessuno li serve: in tal caso non vogliono nessun altro a bordo. Ogni auto porta fino a 5 persone (conducente incluso). Si determini la soluzione che minimizza il numero di km percorsi dalle auto.”



# Progetto

## Sintesi

- Progettare un sistema di **Carpooling per studenti** con destinazione finale l'**università**.
- Gli studenti ogni mattina possono decidere di fare da **autisti**, quindi caricare altri studenti che faranno da **passengeri**.
- Ogni auto degli autisti ha una **capacità** massima di 5 studenti (autista compreso).
- Gli studenti che non sono stati caricati dovranno arrangiarsi e andare in università da soli con la propria auto.
- Tutte le auto devono arrivare in università alla stessa ora.

# Progetto

- **Obiettivo:** minimizzare il numero totale di chilometri percorsi da tutte le auto degli studenti (**NP-Hard**).
- **Soluzione iniziale:** soluzione ammissibile iniziale (caso peggiore): grafo diretto pesato a stella centrato nel nodo università, cioè ogni studente andrà in università con la propria auto.
- **Soluzione finale?** Per ogni autista creare delle routes per andare a prendere altri studenti in modo tale da ridurre i chilometri percorsi totali il più possibile.

# Indice

1. Descrizione del progetto
2. Modello matematico
3. Panoramica del codice
4. Scelte progettuali
5. Euristiche costruttive greedy
  - 5.1. Random Greedy
  - 5.2. Nearest Neighbor
  - 5.3. Clarke & Wright
6. Euristiche neighbor based
  - 6.1. Tabu Search
  - 6.2. Simulated Annealing
7. Euristiche population based
  - 7.1. Genetic Algorithm
  - 7.2. Ant Colony Optimization

# Modello matematico (1/2)

$$\min \sum_{i \in N} \sum_{j \in N} \sum_{v \in V} d_{ij} x_{ij}^v \quad \forall i \in N, j \in N, v \in V$$

$$D = \{\dots, n\} \quad 1 \leq n \leq m$$

$$P = \{n + 1, \dots, m\} \quad 1 \leq n \leq m, \quad m \in \mathbb{N}^+$$

$$V = D$$

$$N = D \cup P$$

$$x_{ij}^v \in \{0, 1\} \quad \forall i, j \in N, \quad v \in V, \quad i \neq j$$

$$y_i^v \in \{0, 1\} \quad \forall i \in P, \quad v \in V$$

**Funzione obiettivo:** minimizzare  
distanza totale percorsa da ogni veicolo  
 $d_{ij}$  = distanza tra il nodo  $i$  e il nodo  $j$

[1]

Insieme degli autisti

Insieme dei passeggeri

Insieme dei veicoli

Insieme di tutti gli studenti (autisti + passeggeri)

$x_{ij}^v = 1$  se il veicolo  $v$  va dal nodo  $i$  al nodo  $j$ , 0 altrimenti

$y_i^v = 1$  se il veicolo  $v$  passa per il nodo passeggero  $i$ , 0 altrimenti

## Modello matematico (2/2)

$$\sum_{v \in V} y_i^v \leq 1 \quad \forall i \in P, v \in V$$

Ogni passeggero deve essere visitato al massimo da un veicolo

$$\sum_{i \in N} x_{ij}^v = y_j^v \quad \forall j \in P, v \in V$$

Ogni passeggero deve essere visitato e lasciato dallo stesso veicolo

$$\sum_{j \in P \cup 0} x_{ij}^v = y_i^v \quad \forall i \in P, v \in V$$

$$\sum_{i \in N} x_{ij}^v = 0 \quad \forall j \in D, v \in V$$

Ogni autista non deve essere visitato da nessuno

$$\sum_{i \in N} x_{i0}^v \leq 1 \quad \forall v \in V$$

Ogni veicolo una volta giunto in università termina il suo viaggio

$$\sum_{i \in N} y_i^v \leq 5 \quad \forall v \in V$$

Ogni veicolo ha una capacità massima di 5 studenti (autista compreso)

# Codice

- **main.py**: menu interattivo per selezionare azioni, metodi, euristiche, ecc. da eseguire.
- **generator.py**: generatore e visualizzatore di istanze.
- Più altre classi di supporto come Enums, convertitori, documentazione, licenza, ecc.





# Librerie

- [Matplotlib](#): per la visualizzazione di grafici, come istogrammi, scatterplot, ...
- [Networkx](#): per la creazione, manipolazione e visualizzazione di grafi
- [haversine](#): package di Python per il calcolo delle distanze tra due coordinate geografiche (Latitudine, Longitudine) in chilometri.



# Generatore di istanze

- **Input:** coordinate geografiche università, numero studenti totali, percentuale studenti-autisti; media, deviazione standard, limite inferiore e limite superiore di una distribuzione normale troncata (“mezza campana”).
- **Output:** soluzione ammissibile (caso peggiore), ovvero un grafo a stella con tutti i nodi (studenti autisti e passeggeri) connessi direttamente all’università.
- **Scelta progettuale:** studenti distribuiti normalmente con media “vicina” all’origine, mentre radialmente distribuiti uniformemente (Gli studenti preferiscono vivere vicino all’università).
- **Features:** ~~import/export di istanze~~, visualizzazione istogrammi distribuzioni, calcolo lunghezza totale, ecc.

# Scelte progettuali (1/2)

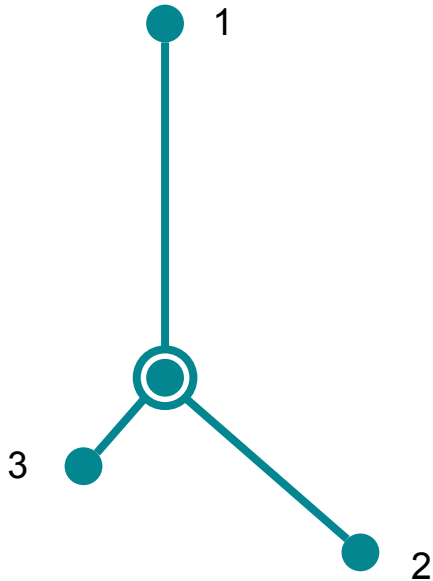
- **Stessa ora di arrivo degli studenti in università:** più uno studente sarà distante dall'università, prima partirà per iniziare il suo viaggio.  
Esempio  $T_f = 8:00$  AM, quindi se due viaggi durano  $D_0 = 25'$  e  $D_1 = 10'$ , allora l'autista 0 partirà alle ore 7:35 AM, prima dell'autista 1 che partirà alle ore 7:50 AM.
- **Pianificazione prossima destinazione?** La decisione delle routes è lasciata agli autisti: i passeggeri verranno assegnati dinamicamente in modo iterativo...

## Iterazioni:

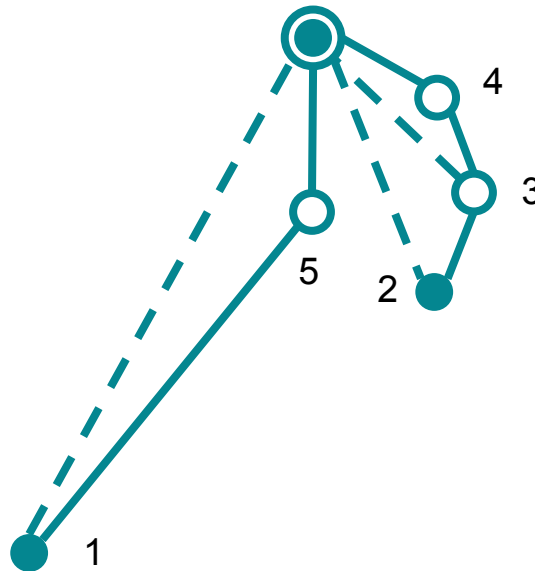
1. Ordino gli autisti in base alla distanza dall'università in modo decrescente: il più distante deciderà per primo la sua prossima destinazione (passeggero o università).
2. Ordino gli autisti in base alla distanza percorsa in modo crescente: l'autista che avrà percorso meno strada per arrivare al suo prossimo passeggero, deciderà per primo la sua prossima destinazione.
- n. ... (Uguale alla 2)

## Scelte progettuali (2/2)

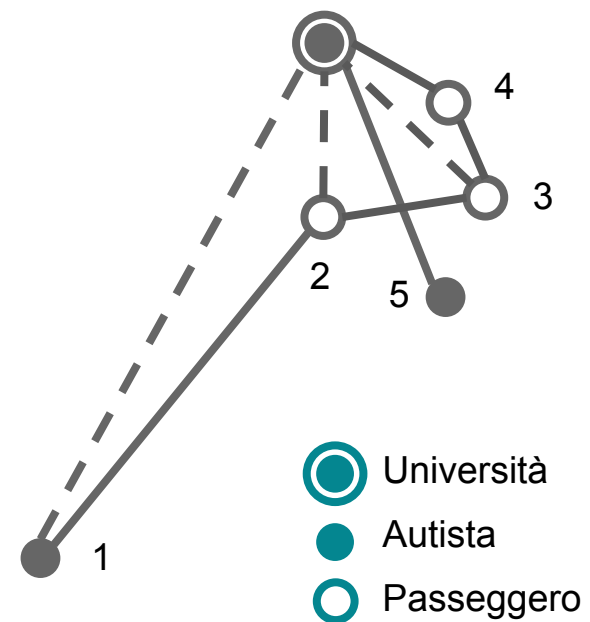
Priorità degli autisti basata sulle distanze dall'università ordinate in modo decrescente



Autisti che concorrono dinamicamente nel prenotarsi un passeggero



Autisti che sequenzialmente pianificano le loro route prima di partire




# Parallel Random Greedy (1/2)

1. Per ogni autista controllo se ha ancora posto in auto per caricare un nuovo passeggero.
  2. Se l'autista corrente ha ancora posto in auto, decido stocasticamente se caricargli o meno un nuovo passeggero.
  3. Se decido di caricargli un nuovo passeggero, allora ne scelgo uno casualmente tra quelli ancora disponibili.
- Ovviamente, questa greedy produrrà soluzioni ammissibili anche peggiori della soluzione ammissibile di base: la utilizzerò come soluzione iniziale per le euristiche di miglioramento.

# Parallel Random Greedy (2/2)

## Pseudocode

```
1.  if len(passengers) > 0:
2.      while capacity <= 4:
3.          for driver in drivers:
4.              if len(passengers) > 0:
5.                  new_passenger = random(False, True)
6.                  if new_passenger:
7.                      next_passenger = random(passengers)
8.                      routes[driver].append(next_passenger)
9.                      passengers.remove(next_passenger)
10.             capacity++
11. for route in routes:
12.     route.append(id_università)
```



# Parallel Nearest Neighbor (1/3)

- Anziché partire da uno studente qualsiasi, creo una route dall'autista più distante dall'università: più distante sarà, prima dovrà partire per arrivare alla stessa ora degli altri studenti più vicini.
- Anziché iterare l'algoritmo sui passeggeri successivi fino a riempire l'auto (oppure all'arrivo in università), lo applico all'autista che ha percorso meno distanza fino all'iterazione corrente: questo ordinamento avverrà dalla seconda iterazione in poi sugli autisti, quindi dopo che tutti gli autisti si siano prenotati almeno un passeggero.
- L'algoritmo termina quando tutti gli autisti hanno raggiunto la loro capienza massima, oppure se sono arrivati in università.
- Gli studenti restanti andranno da soli per conto loro.
- Ogni autista sceglierà come suo prossimo passeggero quello più vicino tra quelli ancora disponibili.

# Parallel Nearest Neighbor (2/3)

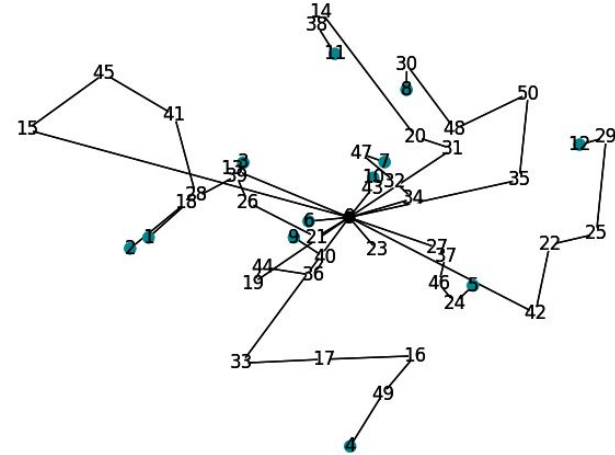
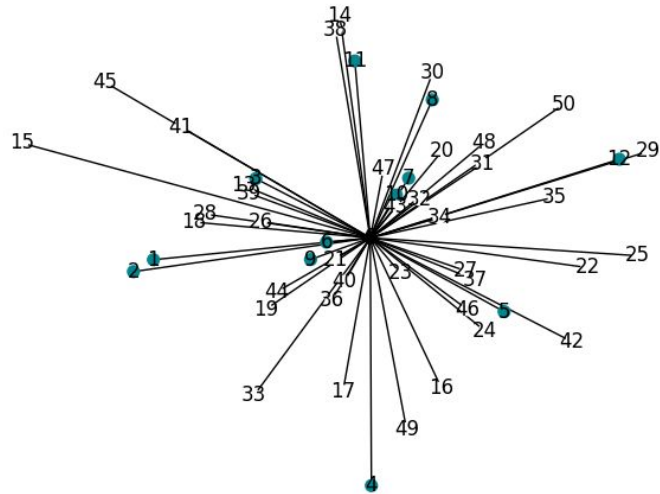
## Pseudocodice

1. **order** autisti in modo decrescente in base alla distanza dall'università
2. **while** tutti gli autisti non hanno raggiunto capacità massima o l'università:
3.     Seleziono il primo autista della lista (Alla prima iterazione quello più distante, dalla seconda in poi invece quello che ha percorso meno distanza)
4.     Per ogni passeggero ancora libero e università calcolo rispettivamente la distanza autista-passeggero e autista-università
5.     Ordino i passeggeri in ordine crescente per distanza dall'autista corrente
6.     Se la distanza autista-università è inferiore a quella di autista-passeggero, allora la prossima destinazione per l'autista corrente sarà l'università e terminerà il suo viaggio.
7.     Altrimenti, la sua prossima destinazione sarà il primo passeggero della lista, cioè quello a lui più vicino.
8.     Ordino gli autisti in ordine crescente in base alla distanza percorsa.



# Parallel Nearest Neighbor (3/3)

## Esempio



# Clarke & Wright (1/2)

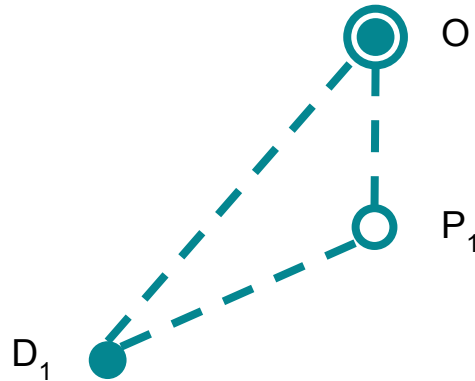
1. **Petali:** composti solo da archi diretti dagli studenti all'università. (Soluzione ammissibile per costruzione).
2. **Merging:** fusione di route solo se capacità inferiore a quella massima e se viene calcolato un risparmio (saving).

**Attenzione:** per proprietà geometriche basate sulle disuguaglianze triangolari il grafo finale è uguale a quello ottenuto dall'algoritmo **Nearest Neighbor**, quindi il passeggero più vicino per un autista è **sempre** anche quello che porterà ad un saving più alto.

## Clarke & Wright (2/2)

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

$$s_{ij} = d_{i0} - d_{ij}$$



**Distanza:** formula per il calcolo della distanza tra due punti su di un piano cartesiano.

**Saving:** formula per il calcolo del risparmio (saving).

Se la somma delle distanze  $D_1P_1$  e  $P_1O$  è inferiore alla somma  $D_1O$  e  $P_1O$  (soluzione ammissibile base), allora l'autista  $D_1$  caricherà il passeggero  $P_1$  se la sua capacità sarà inferiore a quella massima.

# Sweep Algorithm

- **Non implementato!** Perché sarebbe andato contro il vincolo che tutti gli studenti sarebbero arrivati in università alla stessa ora. (Non parallelizzabile)



# Tabu Search (1/4)

- Ad ogni iterazione seleziono la migliore soluzione dell'intorno, diversa dalla soluzione corrente, anche peggiore.
- Ricerca dell'ottimo basata su una memoria che mantiene l'ottimo candidato
- Nella **lista tabù** ad ogni iterazione si salva l'inversa della mossa appena effettuata: resterà proibita per le prossime - dimensione lista tabù - iterazioni.
- Il **criterio di aspirazione** mi permette di visitare soluzioni con valore di funzione obiettivo migliore di quella dell'ottimo candidato.



# Tabu Search (2/4)

- **Inizializzazione:** soluzione ammissibile generata da *Random Greedy*.
- **Intorno (mossa):** *String Exchange (Swap)*.
- **Lista Tabù:** memorizzo come mosse di swap i passeggeri delle routes (selezionati/e casualmente).
- **Criterio di stop:** raggiunto numero massimo di iterazioni, numero massimo di non miglioramenti (stallo) oppure nessuna mossa disponibile (ottimo locale).
- **Attenzione:** le mosse non mi producono soluzioni inammissibili, quindi nessuna penalità applicata.

# Tabu Search (3/4)

- **Input:**

$S$  = soluzione ammissibile iniziale

$S_{\text{current}}$  = soluzione ciclo corrente =  $S$

$F_{\text{current}}$  = costo soluzione corrente =  $c(S)$

$S_{\text{best}}$  = migliore soluzione trovata =  $S$

$F_{\text{best}}$  = costo migliore soluzione trovata =  $F_{\text{current}}$

$\text{moves}_{\text{max}}$  = numero massimo tentativi di mosse = 100

$\text{iter}_{\text{max}}$  = numero massimo di iterazioni = 1000

$\text{stall}_{\text{max}}$  = numero massimo iterazioni senza miglioramenti = 50

$\text{tabu\_list}_{\text{max}}$  = dimensione lista tabù (FIFO) = 5

- **Output:**

$S_{\text{best}}$  = migliore soluzione trovata

# Tabu Search (4/4)

```
1. Inizializzazione dei parametri
2. while iter < itermax AND stall < stallmax:
3.     while moves < movesmax:
4.         Seleziono casualmente 2 routes e 2 passeggeri
5.         Applico move String Exchange (Swap)
6.         if mossa migliorativa:
7.             Salvo mossa nella lista delle mosse provate
8.             moves++
9.         if len(moves_list) == 0:
10.            break // Non esistono mosse disponibili o migliorative
11.        Estraggo migliore mossa trovata move
12.        if c(move(S)) < c(move(Scurrent)) OR move not in tabu_list:
13.            if c(move(S)) < c(move(Sbest)):
14.                Sbest = move(S); stall = 0
15.            else: stall++
16.            Aggiungo move alla tabu_list
17.        Fcurrent = c(move(S))
18.        iter++
19.    return Sbest
```



# Simulated Annealing (1/3)

[2]

- Ad ogni iterazione una soluzione è scelta a caso nell'intorno della soluzione corrente: l'intorno non viene esplorato, bensì esplorato.
- Se la mossa applicata alla soluzione porta ad un miglioramento, viene sempre accettata.
- Altrimenti, la decisione se spostarsi su di una soluzione peggiore dipende da una funzione casuale  $e^{-\Delta/T}$  dove
  - $\Delta = c(S_{\text{next}}) - c(S_{\text{current}})$
  - $T$  = parametro temperatura che diminuisce durante l'euristica
- Ispirazione: annealing è il processo di raffreddamento secondo cui un solido raggiunge uno stato di energia minima che gli conferisce stabilità.
- Il metodo converge asintoticamente all'ottimo globale con opportuni parametri

# Simulated Annealing (2/3)

- **Input:**

**S** = soluzione ammissibile iniziale

**S<sub>iter</sub>** = soluzione iterazione corrente

**T<sub>0</sub>** = temperatura iniziale = 900

**T<sub>f</sub>** = temperatura finale = 0.1

**alpha** = tasso di raffreddamento = 0.92

**iter<sub>max</sub>** = numero massimo di iterazioni = 400

- **Output:**

**S<sub>best</sub>** = migliore soluzione trovata



# Simulated Annealing (3/3)

```
1. Inizializzazione dei parametri
2. while  $T > T_f$ :
3.     while  $\text{iter} < \text{iter}_{\max}$  OR  $\text{stall} < \text{stall}_{\max}$ :
4.          $S_{\text{iter}} = \text{move}(S)$ 
5.          $\Delta = c(S_{\text{iter}}) - c(S)$ 
6.         if  $\Delta < 0$ :
7.              $S = S_{\text{iter}}$ 
8.             if  $c(S_{\text{iter}}) < c(S_{\text{best}})$ :
9.                  $S_{\text{best}} = S_{\text{iter}}$ ;
10.             $\text{stall} = 0$ 
11.        else:
12.             $R = \text{random}(0, 1)$ 
13.            if  $R < e^{-\Delta/T}$ :
14.                 $S = S_{\text{iter}}$ 
15.                 $\text{stall} = 0$ 
16.            else:  $\text{stall}++$ 
17.         $\text{iter}++$ 
18.     $T = T \cdot \alpha$ 
19. return  $S_{\text{best}}$ 
```

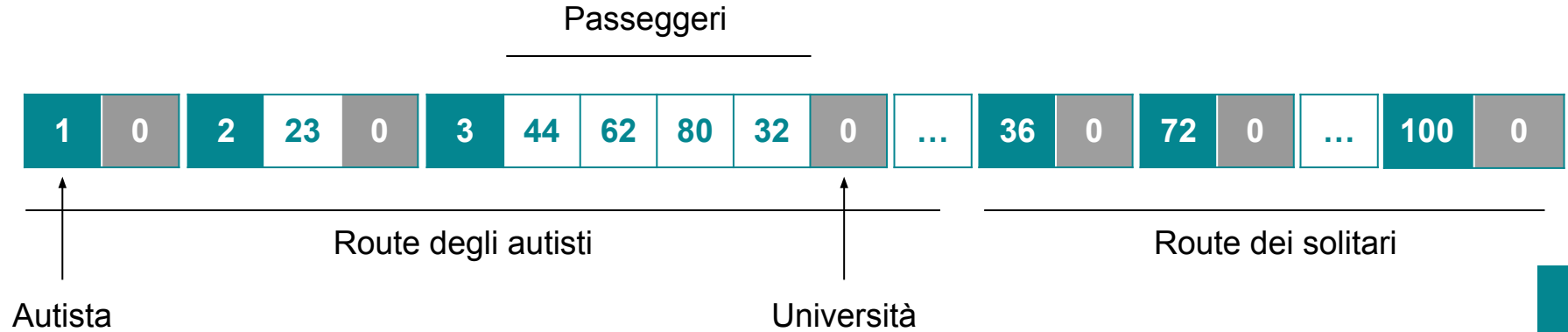
# Genetic Algorithm (1/9)

[3]

- Algoritmi basati sulla teoria dell'evoluzione di Darwin.
- **Popolazione:** generata mediante una *Random Greedy*.
- **Fitness:** distanza totale in chilometri percorsi.
- **Codifica:** una lista di liste contenenti gli ID dei nodi.
- Genitori scelti dalla popolazione tramite *Roulette Wheel*.
- **Crossover e Mutazione:** operazioni pesate ed eseguite stocasticamente.
- **Elitismo:** il migliore individuo passa alla generazione successiva sovrascrivendo il peggiore.
- **Condizione di stop:** numero iterazioni massima e numero iterazioni massime senza miglioramento (stallo).

# Genetic Algorithm (2/9)

## Codifica



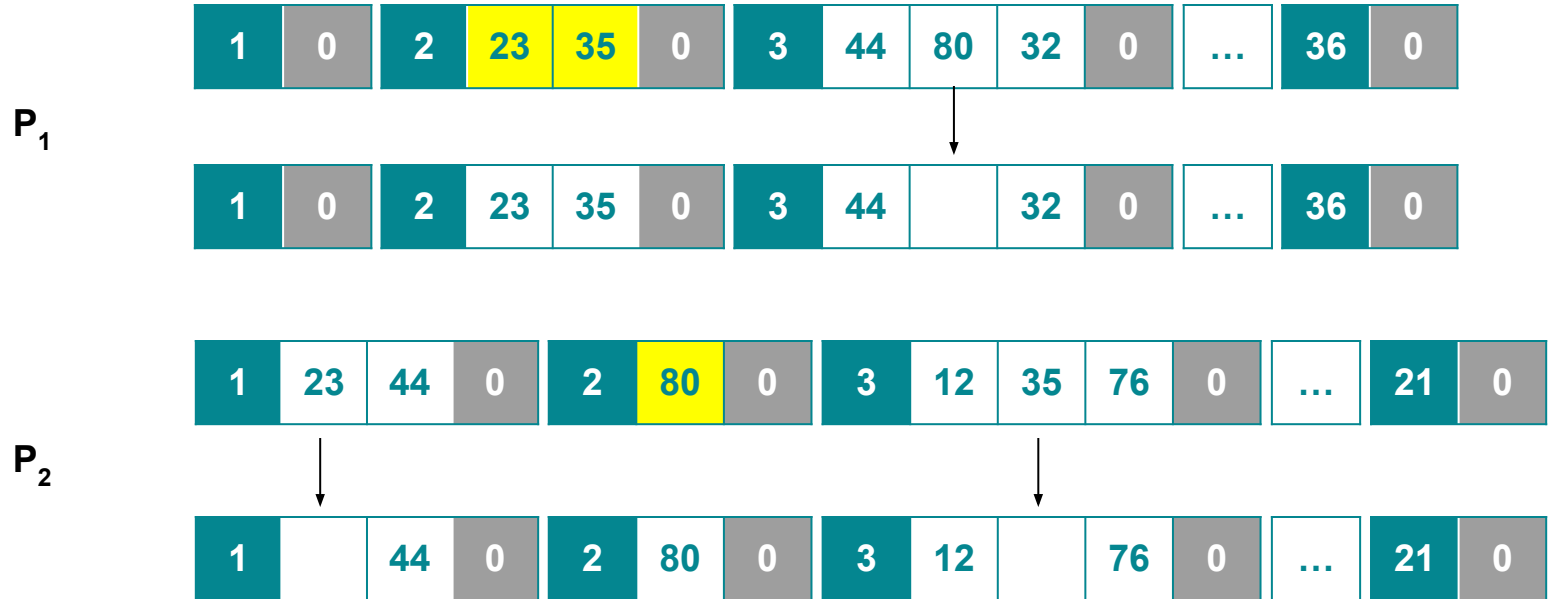
# Genetic Algorithm (3/9)

## Crossover

- **Crossover custom**: genera sempre soluzioni legali e ammissibili (**Repair**) e aspira a generare il figlio migliore in modo esaustivo (**Best improvement**). (**Attenzione**: più autisti saranno presenti nell'istanza e più l'algoritmo sarà lento!)
- Proprietà di **legalità**, **ammissibilità**, **unicità** e **lamarkianicità**: rispettate.
- Composto da **2 fasi**: Removal stage + Insertion stage
  1. **Removal stage**: seleziono mediante roulette wheel 2 genitori e casualmente una route da entrambi: se composta da un autista e almeno un passeggero oppure solo un passeggero, la tengo, altrimenti ne cerco un'altra. Una volta trovata, ci prelevo gli id dei passeggeri e li rimuovo dalla route dell'altro genitore, e viceversa. (Rimozione incrociata)
  2. **Insertion stage**: una volta rimossi, li ri-inserisco all'interno di una route se c'è posto, oppure ne creo una ulteriore: cerco la migliore posizione dove inserirli diminuendo il più possibile la fitness (Best improvement).

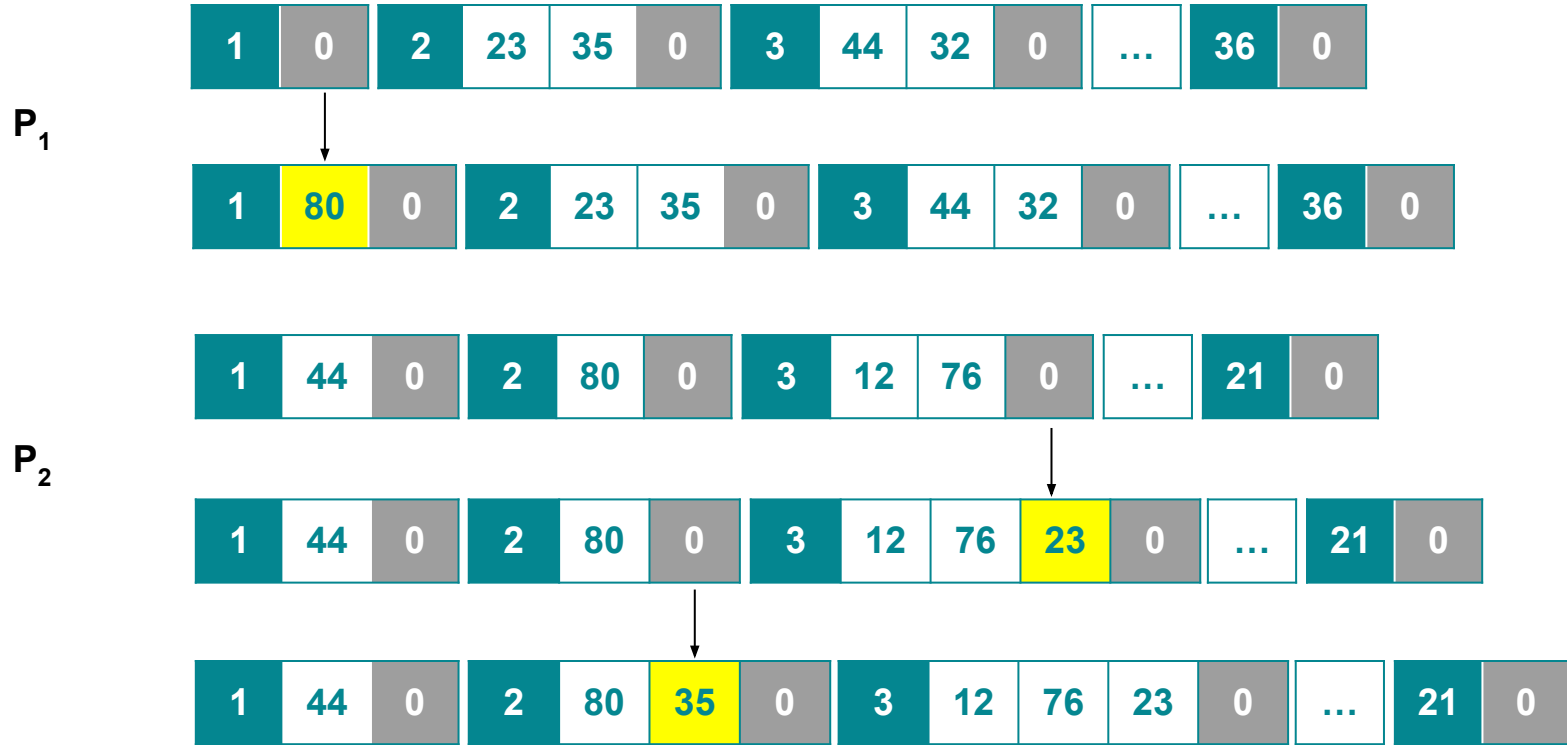
# Genetic Algorithm (4/9)

## Crossover



# Genetic Algorithm (5/9)

## Crossover





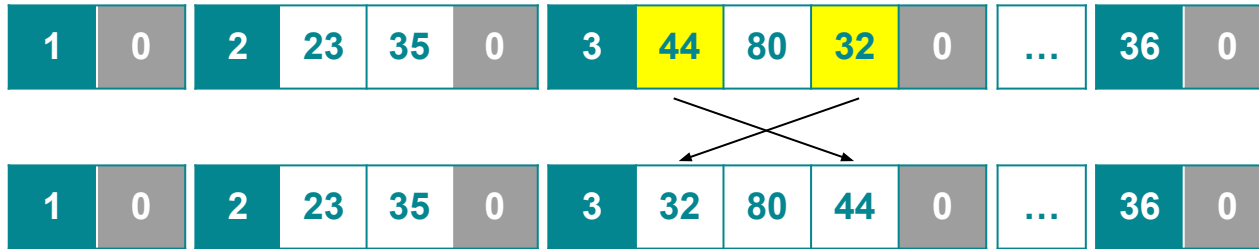
# Genetic Algorithm (6/9)

## Mutation

- Anziché lavorare su 2 individui come il **Crossover**, la **Mutazione** viene eseguita su di un solo individuo.
- Anche questa operazione non genera soluzioni non ammissibili
- Per ogni individuo della popolazione, decido stocasticamente se mutarlo o meno: se lo muterò, allora seleziono casualmente da questo individuo una route avente almeno 2 passeggeri. Selezionata la route, eseguirò la mutazione che consiste nello swap di 2 passeggeri scelti casualmente.

# Genetic Algorithm (7/9)

## Mutation



# Genetic Algorithm (8/9)

- **Input:**

$P_c$  = probabilità di crossover = 0.9

$P_m$  = probabilità di mutazione = 0.5

$S$  = dimensione popolazione = 100

$I_{\max}$  = numero massimo iterazioni = 100

$L_{\max}$  = iterazioni di stallo = 10

- **Output:**

$B$  = migliore individuo, cioè grafo ammissibile con minore distanza totale

# Genetic Algorithm (9/9)

1. Inizializzazione dei **parametri di input** e  $I = 0$ ,  $L = 0$
2. Genero **popolazione iniziale**  $S_0$
3. Calcolo **fitness** per ogni individuo
4. **while** finchè  $I < I_{\max}$  **AND**  $L < L_{\max}$ :
  5. Salvo il migliore individuo della popolazione corrente (**Elitismo**)
  6. Seleziono due genitori casualmente (**Roulette Wheel**)
  7. Genero numero random  $R \in [0,1]$
  8. **if**  $R < P_c$ : genero 2 figli (**Offspring**) dai genitori selezionati (**Crossover**)
  9. **else**: tengo i 2 genitori selezionati per la generazione successiva
10. **repeat** per ogni figlio generato
  11. Genero numero random  $R \in [0,1]$
  12. **if**  $R < P_m$ : applico mutazione al figlio corrente
  13. **else**: non applicare mutazione
14.  $I++$
15. **if** trovata soluzione migliore nella popolazione corrente:  $L = 0$
16. **else**:  $L++$
17. **return** migliore individuo ultima popolazione

# Parallel Ant Colony Optimization (1/6)

- Template proposto da **Marco Dorigo** negli anni '90 ispirato ai sistemi naturali: le colonie di formiche si organizzano per la ricerca del cibo attraverso una forma di comunicazione indiretta basata sulle modifiche ambientali: ***stigmergy***.
- Le formiche hanno un comportamento collaborativo orientato alla conservazione della **colonia** anziché dell'**individuo**.
- La ricerca del cibo avviene mediante una **esplorazione casuale** nei dintorni del formicaio.
- Le formiche durante gli spostamenti rilasciano una traccia chimica a base di **feromoni**, memoria collettiva e distribuita delle scelte locali di instradamento in direzione del cibo: nel tragitto di ritorno la quantità di feromoni è proporzionale alla **quantità** e **qualità** del cibo individuato.
- Nella scelta del percorso da seguire le formiche tendono in probabilità a seguire percorsi con alte concentrazioni di feromoni. Tuttavia, il feromone *svanisce* col tempo.
- Il percorso più breve raccoglie così più feromone per unità di tempo e in breve diventa il preferito.

## Parallel Ant Colony Optimization (2/6)

- Nelle istanze del mio problema, i **formicai** saranno identificati con gli **autisti** mentre il **cibo** con l'**università**; i possibili **passengeri** invece con i **nodi** per cambiare instradamento verso la destinazione successiva.
- Anziché eseguire l'algoritmo su una soluzione ammissibile di base a stella (caso peggiore, tutti gli studenti si arrangiano per andare in università), parto da un **grafo semi-completo**: archi presenti tra **autista-passeggero**, **autista-università**, **passeggero-passeggero**, **passeggero-università** e quindi privo di archi del tipo **autista-autista**.
- Anche questo algoritmo ho deciso di implementarlo in modo **parallelo**, per far sì che ogni autista sia concorrente agli altri fino al raggiungimento della capienza massima o dell'arrivo in università: del percorso migliore trovato autista-università, terrò conto solo della prima destinazione da fargli raggiungere.

# Parallel Ant Colony Optimization (3/6)

[4]

- **Input:**

$f_0$  = valore iniziale feromone = 1

$n_a$  = numero formiche = 100

$i_{\max}$  = numero iterazioni = 10

$p$  = velocità di evaporazione

**alpha** = peso del feromone = 2

**beta** = peso dell'inverso della distanza = 12

**gamma** = peso sul saving = 4

- **Output:** grafo ammissibile con distanza minima totale

# Parallel Ant Colony Optimization (4/6)

1. Inizializzazione parametri algoritmo
2. Creazione grafo semi-completo
3. Ordinamento autisti decrescente in base alla distanza dall'università
4. **while** tutti gli autisti non sono arrivati in università:
5.     **while**  $i < i_{\max}$ : // Numero di volte che mando in spedizione  $n_a$  formiche
6.         **while** tutte le formiche non sono arrivate in università:
7.             **for** ogni formica:
8.                 **if** la formica corrente è già arrivata in università: continua
9.                 **if** la formica ha capacità inferiore a quella massima:
10.                     Calcola nodo successivo su cui si sposterà
11.                 **else**:
12.                     Termina viaggio formica in università
13.             Aggiornamento feromoni
14.     Trovo l'arco con feromone massimo su cui il driver si muoverà
15. **return** soluzione



# Parallel Ant Colony Optimization (5/6)

$$p_{ij}^k = \begin{cases} \frac{\tau_{ij}^\alpha \eta_{ij}^\beta s_{ij}^\gamma}{\sum_{l \in N_i^k} \tau_{il}^\alpha \eta_{il}^\beta s_{il}^\gamma} & , \text{ if } j \in N_i^k \\ 0 & , \text{ otherwise} \end{cases}$$

Probabilità che una formica  $k$  dal nodo  $i$  scelga di raggiungere il nodo  $j$ . Se  $j$  è appartenente all'intorno del nodo  $i$  (tutti i passeggeri non ancora prenotati), allora la probabilità sarà diversa da 0 e proporzionale alla **quantità di feromone** presente su quell'arco  $(i, j)$ , ad una **euristica custom** e al **saving** (risparmio) che porterà sulla distanza totale.

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Formula per calcolare la distanza tra due nodi  $i$  e  $j$  proiettati su di un piano cartesiano

$$s_{ij} = d_{i0} - d_{ij}$$

Formula per calcolare il saving (risparmio) nel decidere di passare a prendere un passeggero nel nodo  $j$  partendo dal nodo  $i$

# Parallel Ant Colony Optimization (6/6)

$$\eta_{ij} = \frac{s_{ij}}{d_{ij}}$$

Formula per calcolare l'**euristica custom** riferita nel passare dal nodo i al nodo j: direttamente proporzionale al **saving** (risparmio) e inversamente alla **distanza** tra i due nodi.

$$\Delta \tau_{ij}^k = \begin{cases} \eta_{ij}^k & , \text{ if ant } k \text{ go from } i \text{ to } j \\ 0 & , \text{ otherwise} \end{cases}$$

Incremento/decremento di feromone dipeso dalla formica k lungo l'arco (i, j) basato sull'euristica custom.

$$\tau_{ij} = (1 - p) \tau_{ij} + \sum_{k \in K} \Delta \tau_{ij}^k$$

Aggiornamento di feromone sull'arco (i, j) dipeso dalla sommatoria degli incrementi/decrementi delle K formiche che lo hanno percorso e dall'evaporazione dipendente dal coefficiente p compreso tra 0 e 1: se p = 1 non c'è evaporazione.

# Risultati

**# Studenti = 100**

**% Autisti = 25%**

**km iniziali = 467km**

**km iniziali RG = 547km**

	km soluzione	% soluzione	Tempo
<b>Nearest Neighbor</b>	231km	-49%	11''
<b>Tabu Search</b>	355km	-36%	33''
<b>Simulated Annealing</b>	387km	-30%	5''
<b>Genetic Algorithm</b>	199km	-64%	44'
<b>Ant Colony Optimization</b>	???	???	???

# Conclusioni

- Nonostante la complessità degli **algoritmi genetici** di un ordine di grandezza temporale superiore (dovuta alla best improvement), producono per piccole istanze risultati migliori più del doppio rispetto agli altri algoritmi delle euristiche.
- La % di autisti sul totale degli studenti per individuare l'istanza migliore per l'ottimizzazione la troviamo tra il 25-35%: circa  $\frac{1}{4}$  degli studenti è un autista.



# Bibliografia

- [1] Pichpibul, Tantikorn & Kawtummachai, Ruengsak. (2012). An improved Clarke and Wright savings algorithm for the capacitated vehicle routing problem. ScienceAsia. 38. 307. 10.2306/scienceasia1513-1874.2012.38.307.
- [2] Normasari, Nur & Yu, Vincent & Bachtiyar, Candra & Sukoyo, Sukoyo. (2019). A Simulated Annealing Heuristic for the Capacitated Green Vehicle Routing Problem. Mathematical Problems in Engineering. 2019. 1-18. 10.1155/2019/2358258.
- [3] N. Lin, Y. Shi, T. Zhang and X. Wang, "An Effective Order-Aware Hybrid Genetic Algorithm for Capacitated Vehicle Routing Problems in Internet of Things," in IEEE Access, vol. 7, pp. 86102-86114, 2019, doi: 10.1109/ACCESS.2019.2925831.
- [4] Tan, W.F. & Lee, Lai Soon & Abdul Majid, Zanariah & Seow, Hsin-Vonn. (2012). Ant Colony Optimization for Capacitated Vehicle Routing Problem. Journal of Computer Science. 8. 846-852.