cammini.py

Questo programma python consiste nell'apertura della pipe in scrittura, per la comunicazione con `cammini.c`.
Questo passerà a `cammini.c` coppie di interi che rappresentano i codici degli attori di cui trovare il cammino minimo.

Tramite le due import, si importano le librerie fondamentali per questo programma:

```
import os, signal, argparse
import sys, struct, time
```

`os` e `signal` sono fondamentali per le chiamate di sistema (aprire file descriptor, inviare segnali).
`struct` è fondamentale perché permette di impacchettare in i dati che devo mandare alla parte C, in byte crudi.

Successivamente si trova il main.
Al suo interno si trova subito un `while true`, questo servirà ad aprire la pipe in scrittura.
Infatti nel suo corpo si trova il costrutto `try...cacth`: qua si prova ad aprire la pipe in scrittura tramite la funzione `os.open(..)` e se ci riesce, esce dal `while` tramite un `break`, altrimenti si fa il `cacth` dell'eccezione e si attende due secondi, grazie alla `time.sleep(2)` e si riprova ad aprire la pipe:

```
# apre la pipe in scrittura
 while True:
    try:
      pipe = os.open(args.p, os.O_WRONLY)
      break
    except FileNotFoundError:
      print(f"--- Pipe {args.p} non pronta, attendo...")
      time.sleep(2)
print("=== pipe aperta in scrittura")
```

questo si fa perché se si prova ad aprire la pipe, ma dall'altra parte non è ancora stata creata verrebbe sollevato un errore di `FileNotFoundError` e andrebbe in crash.
Nota tecnica: `os.open` qui è bloccante. Se la pipe esiste ma nessuno la sta leggendo dall'altra parte, lo script si fermerà su quella riga finché il server C non apre la pipe in lettura.

A questo punto, significa che la pipe è stata aperta, e che quindi si deve inviare le coppie di interi.
Abbiamo un ciclo `for` che scorre il numero di codici e per ognuno di essi chiama la funzione `struct.pack(..)` che prende due interi e li trasforma in una sequenza di 8 byte, li salva all'interno della variabile `buf` e scrive il contenuto di `buf` nella pipe:

```
# scrive le coppie di attori sulla pipe
for i in range(len(args.codici)-1):
    # scrive i due attori sulla pipe
```

```
    buf = struct.pack('ii', args.codici[i], args.codici[i+1])
    os.write(pipe, buf)
print("=== scrittura coppie completata")
```

A questo punto viene messo in pausa lo script:

```
# attende il numero di secondi specificato
print(f"=== attendo {args.s} secondi")
time.sleep(args.s)
```

A questo punto c'è una parte di codice che viene eseguita solo se era stato specificato l'opzione `-i`. Permette di passare un PID a cui inviare un segnale di `SIGINT`

```
# se opzione -i specificata, invia SIGINT al pid specificato
if args.i > 0:
    print(f"Invio SIGINT al processo {args.i}")
    try:
      os.kill(args.i, signal.SIGINT)
    except ProcessLookupError:
      print(f"--- Errore: processo {args.i} non trovato")
    except PermissionError:
      print(f"--- Errore: permesso negato per inviare SIGINT a {args.i}")
    except Exception as e:
      print(f"--- Errore: {e}")
```

A questo punto, viene chiusa la pipe e l'esecuzione termina:

```
# chiude la pipe
os.close(pipe)
print("=== pipe chiusa in scrittura, esecuzione terminata")
```

Alla fine del file troviamo il seguente codice:

```
if __name__ == '__main__':
  parser = argparse.ArgumentParser(description=Description,
formatter_class=argparse.RawTextHelpFormatter)
  parser.add_argument('codici', nargs='+', type=int, metavar='codice', help='lista
di codici di attori (almeno 2)')
  parser.add_argument('-p', help='nome pipe', type=str, default='cammini.pipe',
metavar='pipe')
  parser.add_argument('-i', help='pid a cui inviare SIGINT', type=int, default=0,
metavar='pid')
  parser.add_argument('-s', help='secondi attesa prima di inviare SIGINT (def. 0
sec)', type=float, default=0, metavar='secs')
  args = parser.parse_args()
  if len(args.codici) < 2:
    print("Devi passare almeno 2 codici di attori")
    sys.exit(1)
  main(args)
```

tutto questo serve per rendere utilizzabile da linea di comando lo script.