functionality.c

File che raccoglie le implementazioni di alcune delle funzioni utilizzate all'interno del programma cammini.c (quelle non trattate si trovano nel file `xerrori.c`)

---

confrontaCodici

Implementazione:

```
// Funzione di confronto per la bsearch
int confrontaCodici(const void * codice1, const void *codice2){

    int *cod1 = (int *) codice1;
    attore *cod2 = (attore *) codice2;

    if(*cod1 < cod2->codice)
    {
        return -1;
    }

    if(*cod1 > cod2->codice)
    {
        return 1;
    }

    return 0;
}
```

La funzione `confrontaCodici` ha lo scopo di confrontare due codici di due attori.
Questa serve alla funzione `bserch(..)` per trovare l'attore con codice `codice2` che corrisponde al codice dell'attore, `codice1`, letto dal file `grafo.txt`.

---

inserimentoAttori

Prende come argomento il puntatore al file `nomi.txt` e la capacità iniziale dell'array che dovrà essere creato.
Restituisce il puntatore ad un array di `struct attore`, che rappresenterà il grafo che è stato letto.

Vengono fatti inizialmente dei controlli sul puntatore al file e sul puntatore alla capacità:

```
// Controllo se il punatore FILE è null
assert(nomeFile != NULL);

// Controllo sul valore della capacità dell'array
assert(*capacity == 0);
```

Successivamente viene impostata la dimensione `dimBuffArray`, cioè la dimensione di partenza dell'array che conterrà gli attori, e viene appunto chiamata la funzione `malloc(..)` per allocare dinamicamente l'array che rappresenta il grafo:

```
// Dimensione del buffer per allocare attori poco alla volta
int dimBufArray = 10000;

attore *arrGraph = malloc(dimBufArray * sizeof(attore));
```

si controlla che l'allocazione tramite la `malloc(..)` sia andata a buon fine ed inizializzo le variabili che mi serviranno per la creazione dell'array di attori:

```
// Inizializzo dichiarazione (ed inizializzazione per nomeAttore)
// delle variabili di appoggio per l'insirimento dei valori letti da file
int codiceAttore;
int annoAttore;
char *nomeAttore = NULL;
```

A questo punto dichiaro le variabili per l'utilizzo della funzione `getline(..)`, ovvero il suo buffer e la dimensione del buffer, ovviamente inizialmente settati a `NULL` e `0`:

```
// Variabili per utilizzo della getline()
char *buffer = NULL;
size_t n = 0;
```

Inizia ora il ciclo `while` in cui vengono lette le righe del file `nomi.txt`.
Come guardia del file è presente direttamente la `getline(..)` questo perché quando la lettura del file è terminata esco dal ciclo:

```
while(getline(&buffer, &n, nomeFile) > 0){
    /* ISTRUZIONI */
}
```

All'interno del corpo del `while` si trova un puntatore a carattere chiamato `puntatoreInterno` che serve per il funzionamento della `strtok_r(..)`.
Vengono estratti, grazie alla `strtok_r(..)`, dalla linea letta tramite la `getline(..)` i campi:

- `codiceAttore`
- `nomeAttore`
- `annoAttore`

```
char *puntatoreInterno;

// Leggo valori da una linea
codiceAttore = atoi(strtok_r(buffer,"\t",&puntatoreInterno));
nomeAttore = strtok_r(NULL,"\t",&puntatoreInterno);
annoAttore = atoi(strtok_r(NULL,"\t",&puntatoreInterno));
```

Prima di inserire all'interno del array di attori i valori trovati, si controlla se capacità è uguale alla dimensione del buffer, questo perché la capacità in questa funzione ha il compito di indice, cioè di scorrere l'array di attori (alla fine della funzione invece verrà aggiornata e quindi all'interno del file `cammini.c` verrà utilizzata proprio com capacità effettiva dell'array di attori).
Infatti serve per vedere se ho terminato lo spazio all'interno del buffer della `getline(..)`.
Se così fosse viene duplicato il valore del buffer e viene fatta una `realloc`:

```
// Controllo la dimensione dell'array di attori
if(*capacity == dimBufArray)
{
    // Aumento la dimensione dell'array di attori
    dimBufArray *= 2;

    // Alloco lo spazio necessario
    arrGraph = realloc(arrGraph, dimBufArray * sizeof(attore));
}
```

Ora posso inserire i valori letti dalla linea e salvati nella variabili precedentemente definite, all'interno dell'array di attori ed incremento il valore della capacità di `1` in modo da continuare a scorrere l'array di attori per l'inserimento:

```
// Inserisco i dati nell'attore
arrGraph[*capacity].codice = codiceAttore;
arrGraph[*capacity].nome = strdup(nomeAttore);
arrGraph[*capacity].anno = annoAttore;

// Incremento il valore dell'indice
(*capacity)++;
```

Con questo termina il corpo del ciclo `while`.

Finita la lettura del file `nomi.txt` passo al controllo tra capacità e dimensione dell'array di attori in modo tale da essere sicuro che non ci siano stati errori durante la creazione dell'array (infatti la `capacity` è necessariamente sempre più piccola della dimensione del buffer) .

Viene fatta una `realloc(..)` con la `capacity` in modo tale da avere allocato il quantitativo di memoria corretta.

In fine si libera la memoria occupata dal buffer della `getline(..)` e ritorno il puntatore all'array di attori che rappresenta il grafo letto:

```
// Controllo tra indice e dimBufArray
assert(*capacity <= dimBufArray);

// Realloco la memoria dell'array di attori in modo tale
// da non sprecarne
arrGraph = realloc(arrGraph, *capacity * sizeof(attore));

// Dealloco il buffer per getline
free(buffer);

return arrGraph;
```

---

completamentoInserimentoAttori

Prede per argomento il puntatore al file `grafo.txt` aperto in lettura, l'array di attori creato dalla funzione `inserimentoAttori`, il puntatore alla capacità dell'array di attori ed infine il numero di thread consumatori da creare.

Si inizia dichiarando l'indice per l'inserimento nel buffer del thread produttore, l'indice per l'estrazione dal buffer dei thread consumatori e la dichiarazione del buffer condiviso:

```
// Indice di inserimento nel buffer (da passare al producer)
int prodIndex = 0;

// Indice di estrazione dal buffer (da passare ai consumers)
int consIndex = 0;

// Dichiarazione del buffer
char* buffer[buffSize];
```

Si passa dichiarazione ed inizializzazione del `mutex` per i thread consumatori:

```
// Dichiarazione del mutex per i consumers
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Inizializzazione del mutex
xpthread_mutex_init(&mutex, NULL, QUI);
```

Successivamente si dichiarano le variabili di tipo `sem_t`, ovvero dei semafori per avvertire quando uno slot del buffer si riempie e si libera, Subito dopo si inizializzano tramite la funzione `xsem_init(..)`:

```
// Semaforo per il numero di slots vuoti nel buffer
sem_t sem_free_slots;

// Semaforo per il numero di posti occupati nel buffer
```

```
sem_t sem_data_items;

// Inizializzazione sempafori
xsem_init(&sem_free_slots,0,buffSize,QUI);
xsem_init(&sem_data_items,0,0,QUI)
```

Dopo dichiarazione ed inizializzazione dei semafori, si dichiara il thread Produce utilizzando la seguente `struct`:

```
// Struttura per la definizione del thread Producer
typedef struct {
    char **buffer;          // Buffer condiviso
    int *pIndex;
    sem_t *sem_free_slots;  // Semaforo per il numero di slots liberi
    sem_t *sem_data_items;  // Semaforo per il numero di slots occupati
    FILE *nomeFile;         // Puntatore al file in cui leggere
} datiProducer;
```

, l'array di thread Consumatori e l'array che contiene i dati di ogni thread Consumatore implementati tramite la seguente `struct`:

```
// Struttura per la definizione dei thread Consumer
typedef struct {
    char **buffer;          // Buffer condiviso
    int *cIndex;
    pthread_mutex_t *mutex; // Mutex
    sem_t *sem_free_slots;  // Semaforo per il numero di slots occupati
    sem_t *sem_data_items;  // emaforo per il numero di slots occupati
    attore *arrayGrafo;     // Array di attori a cui accedere
    int capacity;           // Capacità dell'array di attori
} datiConsumer;
```

(per ogni indice i, nell'array di thread si trova l'i-esimo thread consumatore mentre nell'array di dati si trovano i dati esattamente del thread consumatore i-esimo)

```
// Dichiarazione del thread Producer
datiProducer produttore;

// Definizione array di consumers
pthread_t arrThreadConsumer[numeroConsumer];

// Dichiarazione array dei dati da passare ai consumers
datiConsumer arrDatiConsumer[numeroConsumer];
```

Inizio ora la parte di inizializzazione dei thread consumatori, tramite un ciclo che mi scorre l'array di dati: ad ogni dato del thread assegna il valore che si aspetta e crea il thread i-esimo tramite la chiamata a `xpthread_create(..)`.
(questo ovviamente per tutta la lunghezza dell'array dati).

```
// Creazione dei thread, e li faccio partire
for(int i = 0; i < numeroConsumer; i++){
    arrDatiConsumer[i].buffer = buffer;
    arrDatiConsumer[i].cIndex = &consIndex;
    arrDatiConsumer[i].mutex = &mutex;
    arrDatiConsumer[i].sem_free_slots = &sem_free_slots;
    arrDatiConsumer[i].sem_data_items = &sem_data_items;
    arrDatiConsumer[i].arrayGrafo = arrayGrafo;
    arrDatiConsumer[i].capacity = *capacity;

    // Avvio il thread i-esimo

xpthread_create(&arrThreadConsumer[i],NULL,threadConsBody,&arrDatiConsumer[i],QUI)
;

}
```

Creati i thread consumatori inizializzo le variabili del thread produttore

```
// Inizializzazione delle variabili del Producer
produttore.buffer = buffer;
produttore.pIndex = &prodIndex;
produttore.sem_free_slots = &sem_free_slots;
produttore.sem_data_items = &sem_data_items;
produttore.nomeFile = nomeFile;
```

e avvio la lettura del file `grafo.txt` chiamando la funzione `threadProdBody`:

```
threadProdBody(&produttore);
```

Terminata la funzione precedente significa che devo terminare tutti i thread consumatori, perché ho finito il lavoro e lo faccio tramite un ciclo for: il produttore spetta uno slot libero nel buffer tramite il semaforo `sem_free_slots` e la funzione `xsem_wait`, quando si libera uno slot del buffer ci carica il valore `NULL` (questo perché i thread consumatori quando ottengono un `NULL` sono programmati in modo tale da terminare), successivamente tramite il semaforo `sem_data_items` e la funzione `wsem_post` il produttore notifica ai consumatori che ha riempito un nuovo slot del buffer condiviso e aggiorna l'indice di inserimento:

```
// Terminazione dei threads Consumers
for(int i = 0; i < numeroConsumer; i++){
    xsem_wait(&sem_free_slots,QUI);
    buffer[prodIndex] = NULL;
    xsem_post(&sem_data_items,QUI);
    prodIndex = (prodIndex + 1) % buffSize;
}
```

Successivamente viene effettuata la join dei thread consumatori

```
// Join dei threads
for(int i = 0; i < numeroConsumer; i++){
    xpthread_join(arrThreadConsumer[i],NULL,QUI);
}
```

e viene distrutta la mutex e i due semafori, tramite le rispettive funzione di destroy:

```
// Distruzione del mutex
xpthread_mutex_destroy(&mutex,QUI);

// Distruzione dei semafori
xsem_destroy(&sem_free_slots,QUI);
xsem_destroy(&sem_data_items,QUI);
```

---

threadProdBody

E' il corpo del thread produttore, il suo compito è quello di leggere dal file `grafo.txt` e caricare le linee lette nel buffer condiviso per essere processate dai thread consumatori.

Inizia con il controllo sulla validità degli argomenti passatogli, ovvero gli viene passata la struct che segue:

```
// Struttura per la definizione del thread Producer
typedef struct {
    char **buffer;          // Buffer condiviso
    int *pIndex;
    sem_t *sem_free_slots;  // Semaforo per il numero di slots liberi
    sem_t *sem_data_items;  // Semaforo per il numero di slots occupati
    FILE *nomeFile;         // Puntatore al file in cui leggere
} datiProducer;
```

Se l'argomento passato è `NULL` allora si è verificato un errore, altrimenti si procede a fare il cast esplicito:

```
if(args == NULL)
{
    xtermina("ERRORE: argomenti ... non validi (NULL)\n",QUI);
}

datiProducer *datiP = (datiProducer* )args;
```

Si dichiarano ed inizializzano le variabili utili all'utilizzo della `getline(..)` :

```
// Variabili per l'utilizzo di getline()
char *bufferLine = NULL;
size_t n = 0;
```

ed inizia il ciclo while per la lettura delle righe dal file `grafo.txt`.
Come anche per il file `nomi.txt`, nella guardia del while è presente direttamente la funzione `getline(..)` in modo tale da controllare il suo valore di ritorno per la terminazione del ciclo.

Il corpo del for inizia con la funzione `xsem_wait(..)`, questo perché il produttore aspetta che gli venga segnalato che si è liberato uno slot del buffer prima di inserire altri elementi.
Successivamente carica nel buffer alla posizione indicata dall'indice di inserimento `pIndex`, la linea letta dal file tramite la `getlin(..)`.
Viene aggiornato il l'indice di inserimento del produttore ed infine tramite la funzione `xsem_post(..)` viene segnalato che il produttore ha caricato in una cella una nuova linea del file.

Finito di eseguire le istruzioni del ciclo `while`, viene de-allocato, tramite la funzione `free(..)`, il buffer della `getline(..)` e termina.

---

threadConsBody

E' il corpo dei thread consumatori, il loro compito è quello di leggere dal buffer condiviso le linee del file `grafo.txt` caricate dal produttore, estrarre i codici degli attori con cui l'attore principale della linea ha lavorato, cercare l'attore corrispondente nell'array di attori ed infine inserire i valori degli interi degli attori con cui ha lavorato

---

Si inizia con il controllo della validità dell'argomento passato, altrimenti si passa al cast esplicito:

```
if(args == NULL)
{
    xtermina("ERRORE: argomanti ... non validi (NULL)\n",QUI);
}

datiConsumer *datiC = (datiConsumer* )args;
```

A questo punto ci si imbatte in un `while(true)` perché questo è quello che un thread deve eseguire fino anche non legge dal buffer il valore `NULL`.

All'interno del corpo del `while` troviamo subito la funzione `xsem_wait(..)`, questo perché il thread consumatore attende che il produttore abbia caricato nel buffer condiviso qualcosa.
Appena arriva la "notifica" allora il thread cerca di acquisire la mutex tramite la funzione `xpthread_mutex_lock` questo per evitare che due o più thread accedano alla stessa risorsa contemporaneamente (ZONA CRITICA !!).

```
// Aspetto che ci sia qualcosa nel buffer
xsem_wait(datiC->sem_data_items,QUI);
```

```
// Lock del mutex
xpthread_mutex_lock(datiC->mutex,QUI);
```

Non appena acquisita la mutex, la linea viene puntata da un puntatore a stringa e viene controllato se vale `NULL`, questo peché se la risposta fosse affermativa vuol dire che il thread deve terminare. Successivamente al controllo viene rilasciata la mutex tramite la funzione `xpthread_mutex_unlock` e viene inviato il semaforo `sem_free_slots`:

```
char *linea = datiC->buffer[*(datiC->cIndex)];
*(datiC->cIndex) = (*(datiC->cIndex) + 1) % buffSize;

if(linea == NULL)
    {
        xpthread_mutex_unlock(datiC->mutex,QUI);
        xsem_post(datiC->sem_free_slots,QUI);
        break;
    }

xpthread_mutex_unlock(datiC->mutex,QUI);
xsem_post(datiC->sem_free_slots,QUI);
```

A questo punto dichiaro il puntatore `punatoreInterno` per la gestione della `strtok_r` e vengono caricati su delle variabili di appoggio i valori letti dalla linea letta dal file `grafo.txt`:

```
char *puntatoreInterno;

int codAtt = atoi(strtok_r(linea,"\t",&puntatoreInterno));
int numeroCoprotag = atoi(strtok_r(NULL,"\t",&puntatoreInterno));
```

Viene dichiarato ed inizializzato un puntatore ad interi, `coprotagonisiti`, pari a `NULL`, e viene controllato se il valore che rappresenta il numero di coprotagonisti letto dal file sia strettamente maggiore di zero.
Questo perché se lo è allora alloco spazio in memoria per il puntatore `coprotagonisti`, faccio un controllo er vedere se la `malloc(..)` è andata a buon fine ed inizio a caricare all'interno dell'array appena allocato i valori degli interi che corrispondono ai codici identificativi degli attori coprotagonisti della linea letta dal buffer condiviso:

```
int *coprotagonisti = NULL;

if(numeroCoprotag > 0)
    {
        coprotagonisti = malloc(numeroCoprotag * sizeof(int));

        // Controllo se malloc di coprotagonisti è andato a buin fine
        if(coprotagonisti == NULL)
            {
                free(linea);
                xtermina("ERRORE: malloc ... NULL\n",QUI);
```

```
            }

            for(int i = 0; i < numeroCoprotag; i++){
                coprotagonisti[i] = atoi(strtok_r(NULL,"\t",&puntatoreInterno));
            }
    }
```

Se invece non entra dentro l'`if` allora viene de-allocata la linea letta tramite la `free(..)`, viene fatta una ricerca binaria, tramite `bserch(..)`, per trovare l'attore, all'interno dell'array di attori, che ha lo stesso codice di quello presente nella linea letta dal file `grafo.txt`.
Una volta trovato i valori, `numcop` e `cop`, della struttura di tale attore vengono aggiornati rispettivamente con il numero di coprotagonisti letti dalla linea del file e con il puntatore all'array precedentemente creato con tutti i codici numeri degli attori con cui ha lavorato:

```
attoreAppoggio->numcop = numeroCoprotag;
attoreAppoggio->cop = coprotagonisti;
```

Con questo termina il corpo del `while`, ed alla fine si trova la chiamata alla funzione `ptherad_exit(NULL)`.

---

threadSignalBody

Funzione che viene passata al thread gestore di segnali. Il suo compito è quello di gestire il segnale di `SIGINT`: se il programma NON è in fase di lettura della pipe allora lo comunica all'utente altrimenti, se invece è in fase di lettura della pipe allora si deve terminare (aspettare 20 secondi e de-allocare tutto)

Si inizia con il controllo della validità dell'argomento passato, successivamente si stampa il PID del thread gestore di segnale su `stdout` e si fa il cast esplicito:

```
if(args == NULL)
    {
        xtermina("ERRORE: argomanti ... non validi (NULL)\n",QUI);
    }

// Stampo il pid del thread gestore ddi segnali
fprintf(stdout,"Thread Gestore Signal: Il mio pid e': %d\n",getpid());

datiThreadSignal *datiTS = (datiThreadSignal* )args;
```

Si dichiara un set di segnali, lo si svuota (per evitare che gli sia stata assegnata della memoria sporca) e si aggiunge a tale set la tipologia di segnale da gestire:

```
sigset_t mask;
sigemptyset(&mask);
```

```
sigaddset(&mask,SIGINT);
```

Si dichiarano il valore di ritorno della funzione `sigwait(..)` e il numero di segnale che verrà catturato dallo `sigwait(..)`:

```
// Valore di ritorno della sigwait()
int err;

// Numero di segnale che verrà catturato dalla sigwait
int numS;
```

A questo punto troviamo un ciclo `while` che viene effettuato mentre il valore della variabile atomica `stato_terminazione` è uguale ad 0:

```
while(atomic_load(datiTS->stato_terminazione) == 0){
    /* CORPO DEL WHILE */
}
```

Il suo corpo inizia proprio con la funzione `sigwait(..)`, di cui salviamo il valore di ritorno e controlliamo se ci possono essere stati errori nel funzionamento della funzione.
Successivamente si controlla se il valore della variabile atomica `attiva_lettura_pipe` sia 1, questo perché quando tale variabile assume valore 1 significa che la pipe è in stato di lettura e quindi se arriva un segnale di `SIGINT` si può terminare, altrimenti si restituisce un messaggio su `stdout` e si aspetta perché vuol dire che la pipe non è ancora stata aperta:

```
err = sigwait(&mask,&numS);

// Se err == 0 ho ricevuto il segnale correttamente, se != da 0
// c'è stato un problema
if(err != 0)
    {
        xtermina("ERRORE: errore nelle sigwait (threadSignalBody)\n",QUI);
        break;
    }

// Se il valore di attiva_lettura_pipe è 0 vuol dire che non
// sono ancora in fase di lettura della pipe, mentre se è 1
// allora sono in fase di lettura della pipe e quindi il programma
// deve terminare come nelle specifiche
if(atomic_load(datiTS->attiva_lettura_pipe) == 1)
    {
        // Siccome la pipe è aperta allora posso dire al programma di terminare
        atomic_store(datiTS->stato_terminazione, 1);
    }
    else
    {
        fprintf(stdout,"Costruzione del grafo in corso\n");
    }
```

Con questo è finito il corpo del `while`, finita la sua esecuzione troviamo come ultima istruzione `return NULL`.

---

camminiMinimiThreadCreate

Legge dalla pipe 2 interi da 32 bit e per ogni coppia crea un thread che calcola il cammino minimo tra i due. Ovviamente deve gestire anche quando arriva un segnale di `SIGINT`.
Prende in ingresso il file descriptor della pipe, il puntatore alla variabile atomica `stato_terminazione`, il puntatore all'array di attori, il valore della capacità dell'array di attori e il puntatore al thread gestore di segnali.

Dichiaro un buffer per la pipe che accolga solamente due interi da 32 bit, dichiaro una variabile di tipo `ssize_t` che mi prenderà il numero di byte letti dalla pipe:

```
// Alloco staticamente un buffer per la lettura della pipe, di dimensione 2
int32_t pipeBuffer[2];

// Variabile che ospiterà il valore di ritorno della read, ovvero il
// numero di byte letti dalla pipe
ssize_t byteLetti;
```

A questo punto incontriamo un ciclo `while` che controlla se il valore della variabile atomica `stato_terminazione` è diversa da 1, questo perché possiamo uscire dal ciclo solo quando tale variabile è esattamente 1 proprio perché acquisisce il valore 1 o quando arriva un segnale oppure quando devo terminare il programma perché la pipe è stata chiusa:

```
// Ciclo while che termina solo quando stato_terminazione == 1, quando non lo è
// crea per ogni coppia di interi a 32 bit un thread per il calcolo del cammino
minimo
while (atomic_load(stato_terminazione) != 1) {
    /* CORPO WHILE */
}
```

Il corpo del while inizia con la lettura della pipe tramite la `read(..)` (chiamata di sistema) e scriviamo in `byteLetti` il numero di byte letti, inoltre facciamo dei controlli su questa variabile:

- se `byteLetti == 0` allora vuol dire che lo scrittore ha chiuso la pipe e quindi esco dal ciclo `while` grazie al `break`
- se `byteLetti < 0` significa che si è verificato un errore nella lettura della pipe
- se invece `byteLetti < 8` vuol dire che è avvenuta una lettura parziale della pipe

```
// lettura dalla pipe, ritorna -1 se ci sono stati problemi altrimenti
// ritorna il numero di byte letti
byteLetti = read(fd,pipeBuffer, 2 * sizeof(int32_t));
```

```
// Se byteLetti == 0 vuol dire che lo scrittore
// ha chiuso la sua estremità della pipe
if(byteLetti == 0)
    {
        // Esco dal while per la lettura della pipe
        break;
    }

// Se inevece è minore di 0 byte in qualunque caso c'è stato un errore
// nella lettura della pipe
if(byteLetti < 0)
    {
        xtermina("ERRORE: errore in lettura dalla pipe\n",QUI);
    }

// Se inevece byteLetti è < 8, allora c'è stata una lettura
// parziale dei dati
if(byteLetti < 8)
    {
        xtermina("ERRORE: errore di lettura dei dati dalla pipe\n",QUI);
    }
```

Successivamente prelevo dal buffer i due valori a 32 bit e li salvo in due variabili separate:

```
int32_t a = pipeBuffer[0];
int32_t b = pipeBuffer[1];
```

Alloco la memoria necessaria per la struttura `datiThreadMinPath`, che sarà come segue

```
typedef struct{
    int a;              // valore letto dalla pipe
    int b;              // valore letto dalla pipe
    attore *arrayGrafo; // Array di struct di attori (grafo sottoforma di array)
    int capacity;       // Dimnesione/capacità dell'array di attori
} datiThreadMinPath;
```

da passare a tutti i thread che calcola il cammino minimo tra due codici di attori, e controllo il successo della `malloc(..):

```
// Alloco lo spazio necessario per la struct dei dati da passare
// ai thread che calcolano il cammino minimo
datiThreadMinPath *datiTMP = malloc(sizeof(datiThreadMinPath));

// Controlli sulla corretta aloocazione della strict
if(datiTMP == NULL)
    {
        xtermina("ERRORE: errore ... cammino minimo", QUI);
    }
```

Se è andato tutto bene allora inizializzo i valori della `struct` , dichiaro il thread per il calcolo del cammino minimo e lo creo tramite la `xpthread_create(..)` :

```
// Inserisco i dati all'interno della struct
datiTMP->a = a;
datiTMP->b = b;
datiTMP->arrayGrafo = arrayGrafo;
datiTMP->capacity = capacity;

pthread_t threadMP;
xpthread_create(&threadMP, NULL, threadMinPathBody, datiTMP,QUI);
```

Come richiesto nelle specifiche, rendo il thread appena creato `detach` e controllo il suo valore di ritorno per vedere che sia andato tutto a buon fine:

```
// Rendo detach il thread
int detachControllo = pthread_detach(threadMP);
if(detachControllo != 0)
    {
        perror("ERRORE: errore in pthread_detach");
        fprintf(stderr,"== %d == Linea: %d, File: %s\n",getpid(),QUI);
        pthread_exit(&threadMP);
    }
```

Arrivati a questo punto il ciclo si ripete.

Finito il corpo del while si procede alla chiusura della pipe tramite la funzione `xclose(..)` , si procede all'eliminazione della pipe tramite la funzione `unlink(..)` e se ne controlla il valore di ritorno:

```
// Chiudo la pipe
xclose(fd,QUI);

// Elimino la pipe
int pipeControllo = unlink("./cammini.pipe");
if(pipeControllo != 0)
    {
        xtermina("ERRORE: errore nell'eliminazione della pipe\n",QUI);
    }
```

A questo punto viene controllato il valore della variabile atomica `stato_terminazione` , infatti quando esco dal `while` non si sa se siamo usciti a causa dell'arrivo di un segnale e dunque del conseguente settaggio di tale variabile ad 1 oppure perché lo scritto ha deciso di chiudere la pipe.
Quindi si controlla se `stato_terminazione == 1` :

- se lo è significa che il motivo è l'arrivo di un segnale e quindi si procede alla de-allocazione delle risorse tramite la chiama a `signalDeallocation` ed infine si termina il programma con il codice `EXIT_FAILURE`
- altrimenti non faccio nulla e quindi la terminazione avverrà in maniera naturale

```
// Controllo se stato_terminazione == 1 questo perchè non so se sono
// uscito dal while perchè la pipe è stata chiusa oppure perchè è
// arrivato un segnale SIGINT
if(atomic_load(stato_terminazione) == 1)
    {
        fprintf(stderr,"=sto ... camminiMinimiThreadCreate =\n");
        fprintf(stderr,"== TERMINAZIONE DOVUTA A SEGNALE SIGINT ==\n");

        // Dealloco le risorse del occupate dal grafo degli attori e
        // faccio la join del therad gestore di seganli
        signalDeallocation(arrayGrafo,capacity,signalThread);

        // Termino il programma tramite exit() con codice EXIT_FAILURE
        exit(EXIT_FAILURE);
    }
```

---

threadMinPathBody

Funzione che viene eseguita dai thread che hanno il compito di calcolare il cammino minimo tra due attori con codici `a` e `b`.

Inizia con il controllo della validità dell'argomento passato e poi si passa al cast esplicito:

```
if(args == NULL)
    {
        xtermina("ERRORE: argomenti ... non validi (NULL)\n",QUI);
    }

datiThreadMinPath *datiTMP = (datiThreadMinPath* )args;
```

Viene dichiarata ed inizializzata la variabile di tipo `clock_t` per il calcolo del tempo necessario a trovare il cammino minimo tra `a` e `b`:

```
// Inizio la misurazione del tempo di esecuzione
clock_t start = times(NULL);
```

Attuo due ricerche binarie tramite la funzione `bsearch(..)` per ottenere le `struct` dei due attori che hanno codici `a` e `b`:

```
// Ottenngo dall'array di attori gli attori con i codici cercati
attore *sorgente = bsearch(&datiTMP->a, datiTMP->arrayGrafo, datiTMP->capacity,
sizeof(attore), confrontaCodici);

attore *destinazione = bsearch(&datiTMP->b, datiTMP->arrayGrafo, datiTMP-
>capacity, sizeof(attore), confrontaCodici);
```

Finite le due ricerche si controlla la validità dei due codici, cioè verifico se sono state trovate le due `struct` per i due codici passati al thread.
Se almeno una delle due strutture ricavate (uno dei due puntatori alla struttura) è `NULL`, viene calcolato il tempo di "esecuzione", dichiarata la variabile `valoreStampa` che a seconda del valore che assume la funzione di stampa farà una stampa diversa,e si verifica quale dei due codici non era valido:

- se non era valida la sorgente `valoreStampa` viene settato a `-1`
- altrimenti viene settato a `0` (cioè non era valida la destinazione)
  Viene chiamata la funzione di stampa `stampaCamminoMinimo`, viene fatta la free della `struct` del thread che calcola il cammino minimo ed infine ritorna `NULL`:

```
// Controllo che i codici che mi sono stati dati siano validi
if(sorgente == NULL || destinazione == NULL)
    {
        // Uno dei due codici non è valido
        // Stoppo il timer e calcolo qunato tempo è passato
        //double tempo = calcolaTempo(start, time(NULL));
        double tempo = calcolaTempo(start, times(NULL));

        int valoreStampa;
        if(sorgente == NULL)
        {
            valoreStampa = -1;
        }
        else
        {
            valoreStampa = 0;
        }

        stampaCamminoMinimo(datiTMP->a, datiTMP->b, tempo, NULL, valoreStampa);

        // Dealloco gli argomenti passati (struct dei dati del thread)
        free(datiTMP);
        return NULL;
    }
```

Se supera quel controllo allora viene il primo nodo dell'ABR, che è una `struct` come segue:

```
typedef struct nodoABR{
    attore *actor;
    int codActShuffle;
    struct nodoABR* predecessore;
    struct nodoABR* figlioDx;
    struct nodoABR* figlioSx;
} nodoABR
```

passando come dati la struttura della sorgente, e `NULL` come predecessore; inoltre viene controllato che la creazione vada a buon fine:

```
nodoABR *visitati = creaABR(sorgente,NULL);
assert(visitati != NULL);
```

A questo punto viene controllato se la sorgente e la destinazione sono la stessa cosa, se così fosse viene stoppato il calcolo del tempo di "esecuzione", viene chiamata la funzione di stampa `stampaCamminoMinimo`, viene de-allocato l'ABR (cioè l'unico nodo creato), viene fatta la `free()` della struttura del thread che calcola il cammino minimo ed infine ritorna `NULL`:

```
// Controllo se la sorgente e la destinazione sono uguali
if(datiTMP->a == datiTMP->b)
    {
        // Stoppo il timer e calcolo qunato tempo è passato
        double tempo = calcolaTempo(start, times(NULL));

        stampaCamminoMinimo(datiTMP->a, datiTMP->b, tempo, visitati, 1);

        // Dealloco l'abr (unico nodo)
        deallocaABR(visitati);

        // Dealloco gli argomenti passati (struct dei dati del thread)
        free(datiTMP);
        return NULL;
    }
```

Se non fosse nemmeno il caso in cui la sorgente e la destinazione coincidono, allora si procede in un altro modo.

Viene dichiarata la coda FIFO, implementata con la seguente struttura:

```
typedef struct{
    attore **coda;
    int capacity;
    int codiceTesta;
    int codiceCoda;
} FIFO;
```

, inizializzate le sue variabili e controllata la buona creazione delle FIFO:

```
// Dichiaro ed inizializzo la FIFO
FIFO raggiunti;
raggiunti.capacity = 1024;
raggiunti.codiceTesta = 0;
raggiunti.codiceCoda = 0;
raggiunti.coda = malloc(raggiunti.capacity * sizeof(attore));

// Controllo per la malloc
if(raggiunti.coda == NULL)
    {
        perror("Errore malloc");
        fprintf(stderr, "== %d == Linea: %d, File: %s\n", getpid(),QUI);
```

```
        exit(1);
    }
```

Il primo passo e l'inserimento in coda alla FIFO della sorgente, questo viene fatto grazie alla funzione `inserimentoInCoda`:

```
// Inserisco la sotgente in FIFO
inserimentoInCoda(&raggiunti, sorgente);
```

Ora troviamo la guardia di `while` che presenta la seguente condizione:

```
while(raggiunti.codiceTesta < raggiunti.codiceCoda){
    /* CODICE BFS */
}
```

questo sta a significare che fino a che il valore di `codiceTesta` è strettamente minore di quello di `codiceCoda` allora devo continuare la BFS, cioè sto dicendo che fino a che FIFO non è vuota continuo il ciclo: infatti la FIFO sarà vuota solo quando `codiceTesta == codiceCoda`.

All'interno del ciclo `while` estraiamo subito l'elemento che si trova in testa, facciamo un rapido controllo con `assert(..)` e cerco all'interno dell'ABR il nodo che ha lo stesso codice di quello che ho estratto dalla FIFO e faccio un controllo con `assert(..)`:

```
// Estraggo la testa da FIFO
attore *attEstratto = estrazioneInTesta(&raggiunti);

assert(attEstratto != NULL);

nodoABR *attEstrattoABR = cercaInABR(visitati, shuffle(attEstratto->codice));
assert(attEstrattoABR != NULL);
```

A questo punto si avvia un ciclo for che cicla fino al numero massimo di coprotagonisti che l'attore estratto dalla FIFO possiede, sta andando ad inserire nell'ABR e nella FIFO i nodi adiacenti al nodo attualmente estratto (quelli raggiunti dal nodo attualmente estratto):

1. controlla se, tramite il codice dell'attore coprotagonista, l'attore i-esimo dell'array dei coprotagonisti è già stato inserito all'interno dell'ABR (cioè se è già stato raggiunto/processato) se non supera il controllo si salta quel nodo
2. cerca tramite una `bserch(..)` all'interno dell'array di attori la struct che possiede lo stesso codice dell'i-esimo attore dell'array di coprotagonisti (viene fatto anche un `assert(..)`)
3. viene creato un nodo dell'ABR che abbia codice, quello dell'i-esimo attore adiacente a quello appena estratto e come predecessore l'attore estratto stesso
4. a questo punto tale nodo viene inserito nell'ABR
5. infine vine controllato se il codice dell'attore raggiunto (i-esimo) è uguale al codice della destinazione:
    - se non lo è si inserisce in coda, tramite la funzione `inserimentoInCoda`, l'attore adiacente (i-esimo) e si passa all'intero dell'attore coprotagonista successivo

- altrimenti se invece abbiamo trovato la destinazione si calcola il tempo di "esecuzione" tramite la funzione `calcola_tempo(..)` , si chiama la funzione di stampa `stampaCamminoMinimo(..)` , viene de-allocato l'ABR, viene de-allocata la FIFO, viene fatta la free della `struct` passata al thread del calcolo del cammino minimo e viene ritornato `NULL`

```
// Inserisco i nodi adiacenti
for(int i = 0; i < attEstratto->numcop; i++){
    // Controllo se ho già visitato l'i-esimo nodo adiacente
    if(cercaInABR(visitati, shuffle(attEstratto->cop[i])) != NULL)
        {
            continue;
        }

    // In caso negativo, ottengo l'attore e aggiorno l'ABR
    attore *adiacente = bsearch(&attEstratto->cop[i], datiTMP->arrayGrafo,
datiTMP->capacity, sizeof(attore), confrontaCodici);

    assert(adiacente != NULL);

    nodoABR *adiacenteABR = creaABR(adiacente, attEstrattoABR);

    // Inserisco in visitti l'ABR di "adiacente" appena trovato
    inseriscoInABR(&visitati, adiacenteABR);

    // Controllo se il codice dell'attore adiacente sia uguale alla destinazione
    if(adiacente->codice == datiTMP->b)
        {
            // Ho trovato la destinazione
            // Stoppo il timer e calcolo quanto tempo è passato
            double tempo = calcolaTempo(start, time(NULL));

            stampaCamminoMinimo(datiTMP->a, datiTMP->b, tempo, adiacenteABR, 1);

            // Dealloco ABR creato
            deallocaABR(visitati);

            // Dealloco la FIFO
            deallocoFIFO(&raggiunti);

            // Dealloco gli argomenti passati (struct dei dati del thread)
            free(datiTMP);

            return NULL;
        }

    // Altrimenti aggiungo alla FIFO
    inserimentoInCoda(&raggiunti,adiacente);
}
```

Se invece si arriva fino a questo punto significa che non è stato trovato alcun cammino tra `a` e `b` . Quindi si procede a calcolare il tempo di "elaborazione", si chiama la funzione di stampa (ovviamente

con il proprio codice).
Inoltre viene de-allocato l'ABR, la FIFO, viene fatta la `free(..)` della `struct` del thread che calcola il cammino minimo e ritorna `NULL`

```
// A questo punto è stata eseguita la BFS senza però trovare un cammino da a ad b
// Stoppo il timer e calcolo qunato tempo è passato
double tempo = calcolaTempo(start, times(NULL));

stampaCamminoMinimo(datiTMP->a, datiTMP->b, tempo, NULL, 0);

// Dealloco l'ABR creato
deallocaABR(visitati);

// Dealloco la FIFO
deallocoFIFO(&raggiunti);

// Dealloco gli argomenti passati (struct dei dati del thread)
free(datiTMP);

return NULL;
```

---

creaABR

Funzione chiamata dal thread che calcola il cammino minimo tra `a` ed `b`. Crea un `nodoABR` inizializzando la variabile `actor` con la `struct attore` che gli viene passata ed imposta il valore del predecessore al `nodoABR` che gli viene passato per argomento (i figli sono di default `NULL`)

Viene allocato in memoria lo spazio che occorre per una `struct nodoABR`.
Viene controllato se la `malloc(..)` è andata a buon fine.
Vengono inizializzati i valori del nodo.
In particolare viene richiamata la funzione `shuffle(..)` sul codice dell'attore (questa serve per ottenere ABR bilanciati) ed infine ritorna il `nodoABR` appena creato

```
nodoABR *nodo = malloc(sizeof(nodoABR));
if(nodo == NULL)
    {
        perror("Errore malloc");
        fprintf(stderr, "== %d == Linea: %d, File: %s\n", getpid(),QUI);
        exit(1);
    }

nodo->actor = att;
nodo->codActShuffle = shuffle(att->codice);
nodo->predecessore = predec;
nodo->figlioDx = NULL;
nodo->figlioSx = NULL;
```

```
return nodo;
```

---

inseriscoInABR

Funzione chiamata dal tread che calcola il cammino minimo. Ha il compito di inserire all'interno di un ABR il nodo che gli viene passato per argomento.

Viene fatto un controllo sul nodo da inserire.
Si controlla se il puntatore alla radice è `NULL`, se lo è a tale valore viene assegnato il nodo passato per argomento e si termina.
Altrimenti si deve controllare che il valore del codice del nodo (ovviamente dopo l'applicazione di `shuffle`), è uguale a quello della radice.
Se la risposta è affermativa allora significa che il nodo è già presente nell'ABR e quindi, il nodo passato per argomento viene de-allocato tramite la `free(..)` e termina.
Se così non fosse si controlla se il codice shuffled del noto passato per argomento è strettamente maggiore del codice shuffled del puntatore alla radice:

- se è così allora si fa una chiamata ricorsiva alla funzione cui si passa il solito nodo da inserire e come radice si passa il riferimento al figlio sinistro del nodo puntato da radice
- se non è così si fa una chiamata ricorsiva alla funzione cui si passa il solito nodo da inserire e come radice si passa il riferimento al figlio destro del nodo puntato da radice

```
assert(nodo != NULL);
if(*radice == NULL)
    {
        *radice = nodo;
        return;
    }

if((*radice)->codActShuffle == nodo->codActShuffle)
    {
        // Nodo già presente allora si scarta
        free(nodo);
        return;
    }

// Controllo se inserire il nodo a nel ramo destro o sinistro
if((*radice)->codActShuffle > nodo->codActShuffle)
    {
        return inseriscoInABR(&(*radice)->figlioSx,nodo);
    }
else
    {
```

```
        return inseriscoInABR(&(*radice)->figlioDx, nodo);
    }
```

---

deallocaABR

Funzione chiamata dal thread che calcola il cammino minimo. Serve a de-allocare l'intero ABR.
Prende in input il puntatore al nodo radice dell'ABR

Se il puntatore a radice è diverso da `null` vengono fatte:

- due chiamate ricorsive cui viene passato prima il puntatore al figlio destro e poi il puntatore al figlio sinistro
- viene fatta la free della radice stessa

```
if(radice != NULL)
    {
        deallocaABR(radice->figlioDx);
        deallocaABR(radice->figlioSx);
        free(radice);
    }
```

---

inserimentoInCoda

Funzione chiamata dal thread che calcola il cammino minimo. Prende per argomento la coda FIFO e l'attore che dobbiamo inserire.

Inizialmente si fa un controllo sulla coda passata.

```
assert(fifo != NULL);
```

Poi si controlla se la coda è piena (cioè se l'indice di inserimento ha raggiunto la capacità).
Successivamente si controlla che se l'indice di inserimento è maggiore di 32.
Se lo è controllo quanti dati sono effettivamente in coda (grazie alla sottrazione tra il `codiceCoda` e `codiceTesta`), si effettua la `memmov(..)` cioè si sposta all'inizio della coda (area puntata da `fifo->coda`) tutti gli elementi validi (puntati da `fifo->coda + fifo->codice testa` = primo elemento valido nella FIFO) all'inizio della coda (puntata da `fifo->coda`) e si assegna nuovi valori a `codiceCoda` ed a `codiceTesta`.
Se non è valido invece si raddoppia la capacità, si effettua una `realloc` con il rispettivo controllo.
Alla fine si inserisce in coda l'attore passato per argomento e si incrementa l'indice di inserimento in coda:

```
// Controllo se la
if(fifo->codiceCoda >= fifo->capacity)
    {
        if(fifo->codiceCoda > 32)
        {
            int dim = fifo->codiceCoda - fifo->codiceTesta;

            memmove(fifo->coda, fifo->coda + fifo->codiceTesta, dim *
sizeof(attore*));

            fifo->codiceCoda = dim;
            fifo->codiceTesta = 0;
        }
        else
        {
            fifo->capacity *= 2;
            fifo->coda = realloc(fifo->coda, fifo->capacity * sizeof(attore*));

            // Controllo realloc
            if(fifo->coda == NULL)
            {
                perror("Errore remalloc");
                fprintf(stderr, "== %d == Linea: %d, File: %s\n", getpid(),QUI);
                exit(1);
            }
        }
    }

fifo->coda[fifo->codiceCoda] = att;
fifo->codiceCoda++;
```

---

estrazioneInTesta

Funzione chiamata dal thread che calcola il cammino minimo. Prende come argomento la FIFO e ritorna l'attore estratto dalla testa.

Viene fatto un controllo sulla FIFO:

```
assert(fifo != NULL);
```

Controllo che i due codici/indici di testa e di coda non coincidano (perché se coincidono vuol dire che la coda è vuota quindi non c'è nulla da togliere)

```
assert(fifo->codiceCoda != fifo->codiceTesta);
```

In fine ritorna l'elemento in testa:

```
return fifo->coda[(fifo->codiceTesta++)];
```

---

cercaInABR

Funzione chiamata dal thread che calcola il cammino minimo. Confronta il codice shuffled passato per argomento con quelli dei nodi che visita nell'ABR fino a che: o non trova una corrispondenza e ritorna `NULL` altrimenti ritorna il `nodoABR` che ha quel codice

---

Controllo se il puntatore alla radice è `NULL` allora ritona `NULL`:

```
if(radice == NULL)
    {
        return NULL;
    }
```

Altrimenti si controlla se il codice passato è uguale a quello della radice, se lo è si ritorna esattamente la radice:

```
if(radice->codActShuffle == codiceShuffle)
    {
        return radice;
    }
```

Se così non fosse allora controllo se il codice della radice è strettamente maggiore del codice passato per argomento:

- se si allora faccio una chiamata ricorsiva in cui il puntatore a radice è il figlio sinistro della radice e i codice in ingresso è sempre il solito
- altrimenti faccio una chiamata ricorsiva in cui il puntatore a radice è il figlio destro della radice e i codice in ingresso è sempre il solito

```
if(radice->codActShuffle > codiceShuffle)
    {
        return cercaInABR(radice->figlioSx, codiceShuffle);
    }
else
    {
        return cercaInABR(radice->figlioDx, codiceShuffle);
    }
```

---

signalDeallocation

De-allocazione dell'array di attori che rappresenta il grafo e del thread gestore di segnali (serve per garantire la terminazione richiesta dalle specifiche). Prende er argomento il puntatore all'array di attori, la capacità dell'array e il puntatore al thread gestore di segnali

---

Per prima cosa viene chiamata la `join` per il thread gestore di segnali:

```
// join del thread gestore di sehnali
xpthread_join(*signalThread, NULL, QUI);
```

Si attende 20 secondi:

```
// Attendo 20 secondi
sleep(20);
```

De-alloco le risorse all'interno dell'array di attori:

```
// Dealloco le risorse all'interno dell'array di attori
for(int i = 0; i < capacity; i++){
    free(arrayGrafo[i].nome);
    free(arrayGrafo[i].cop);
}
```

Infine de-alloco l'array di attori:

```
free(arrayGrafo);
```

---

deallocoFIFO

Funzione chiamata dal thread che calcola il cammino minimo. Funzione per la de-allocazione della coda FIFO. Prende in ingresso la coda da de-allocare.

Viene fatta la free di `coda` :

```
free(fifo->coda);
```

---

calcolaTempo

Funzione chiamata dal thread che calcola il cammino minimo. Funzione per il calcolo del tempo che impiega il thread che calcola il cammino minimo a trovare il cammino minimo.

Viene fatta la sottrazione tra le variabili di tipo `clock_t`, `inizio` e `fine`.
Vengono divise per il numero di ticks della CPU, converte in double e lo scrive nella variabile `tempoInSecondi`.

```
double tempoInSecondi = (double)(fine - inizio) / sysconf(_SC_CLK_TCK);
```

In fine ritorna tale valore:

```
return tempoInSecondi;
```

---

stampaCamminoMinimo

Funzione chiamata dal thread che calcola il cammino minimo. Prende come argomenti:

- codice sorgente e destinazione (`a` ed `b`)
- la variabile che contiene il tempo del calcolo del cammino minimo
- puntatore al `nodoABR` destinazione
- un intero che funge da valore di controllo
  A seconda del valore di controllo la funzione stampa risultati diversi

---

Viene inizialmente dichiarato un buffer di carattere per contenere il nome del file.

```
// Dichiaro un buffer per il nome del file
char bufferNomeF[32];
```

Successivamente si scrive all'interno del file la stringa con cui si dovrà chiamare il file da aprire, grazie alla chiamata della funzione `snprintf(..)`:

```
// Creo il nome del file
snprintf(bufferNomeF,sizeof(bufferNomeF),"%d.%d",a,b);
```

(Crea una stringa formattata nel modo in cui vogliamo e la va a scrivere nel buffer, ovviamente viene controllata dimensione del buffer per evitare buffer overflow)

Ora viene aperto il file `a.b` in scrittura:

```
// Apro il file dedicato alla stampa
FILE *fileStampa = xfopen(bufferNomeF, "w",QUI);
```

A questo punt entra in gioco il valore di controllo.
Se `controlValue == 1`, significa che ho trovato un cammino minimo tra la sorgente e la destinazione

e dunque eseguo il corpo dell' `if` :

```
if(controlValue == 1)
    {
        /* CORPO IF */
    }
```

A questo punto vengono dichiarate ed inizializzate due variabili che ci serviranno per l'allocazione del path all'interno di un array.
Viene creato l'array che rappresenterà il path e viene fatto il controllo sulla sua apertura (per essere sicuri che sia andata a buone fine)

```
// Andato tutto bene
int lunghezza = 0;
int capacity = 32;

attore *path = malloc(capacity * sizeof(attore));
if(path == NULL)
    {
        perror("Errore malloc");
        fprintf(stderr, "== %d == Linea: %d, File: %s\n", getpid(),QUI);
        exit(1);
    }
```

A questo punto si incontra un ciclo while la cui guardia e la seguente:

```
while(destinazione != NULL){
    /* CORPO WHILE */
}
```

questo perché lo scopo è quello di ripercorrere a ritroso, dalla destinazione alla sorgente, il percorso che collega i due nodi.

All'interno del `while` si controlla inizialmente se `lucghezza == capacity` , questo per verificare se l'array è pieno:

- se lo è si raddoppia la `capacity` e si fa una `realloc(..)`
- se non lo è si continua
  Successivamente si inserisce all'interno dell'array `path` l'attore presente all'interno del puntatore al nodo `destinazione` , si incrementa l'indice `linghezza` e si assegna a destinazione il puntatore al predecessore di `destinazione`

```
if(lunghezza == capacity)
    {
        capacity *= 2;
        path = realloc(path,capacity * sizeof(attore));
        if(path == NULL)
            {
                perror("Errore realloc");
```

```
                fprintf(stderr, "== %d == Linea: %d, File: %s\n", getpid(),QUI);
                exit(1);
            }
    }
path[lunghezza++] = *destinazione->actor;
destinazione = destinazione->predecessore;
```

In questo modo appena si esce dal ciclo `while` abbiamo l'intero percorso da `a` ad `b` all'interno dell'array: ovviamente in ordine inverso.
Dunque tramite un ciclo for si scorre in senso opposto l'array e si stampa all'interno del file i valori richiesti dalla specifica:

```
// Scorro all'indietro l'array di attori
for(int i = lunghezza - 1; i >= 0; i--){
    fprintf(fileStampa, "%d\t%s\t%d\n", path[i].codice, path[i].nome,
path[i].anno);
}
```

Fatto ciò viene de-allocato l'array `path` e il programma termina.

Se invece `controlValue` non è uguale ad 1, viene verificato se siamo nel caso in cui non ci siano cammini minimi da `a` ad `b`, cioè `controlValue == 0`, se lo è viene fatta una stampa dedicata.
Se invece `controlValue` a altri valori allora viene fatta un'altra stampa dedicata

```
// Se non ci sono cammini tra a e b
 if(controlValue == 0)
    {
        fprintf(fileStampa,"non esistono cammini da %d a %d\n",a,b);
    }
 else
    {
        // Nessun cammino perchè usorgente non è valida
        fprintf(fileStampa,"codice %d non valido\n",a);
}
```

In fine si chiude il file aperto in scrittura e relativo controllo:

```
if(fclose(fileStampa) == EOF)
{
    xtermina("ERRORE: errore chiusura file path",QUI);
}
```

---

shuffle

Funzione data dalle specifiche. Funzione di Hashing che previene che l'ABR si sbilanci.

```
int shuffle(int n){
  return ((((n & 0x3F) << 26) | ((n >> 6) & 0x3FFFFFF)) ^ 0x55555555);
}
```

---

unshuffle

Funzione data dalle specifiche. Mi permette di ottenere il valore inizialmente inserito in shuffle a partire dal risultato di shuffle

---

```
int unshuffle(int n){
  return ((((n >> 26) & 0x3F) | ((n & 0x3FFFFFF) << 6)) ^ 0x55555555);
}
```
