Algoritmi e Strutture Dati

Appello del 27/1/2025 - Prova di programmazione (12 punti)

Commento introduttivo. Si propongono soluzioni adattate al contesto dell'esame. Si evitano per semplicitá controlli di errori su apertura file e/o allocazione, né si presentano main, input e/o stampe, necessari per verificare la correttezza dei programmi. I programmi completi sono forniti a parte.
Attenzione: la soluzione non e' stata testata estensivamente. Sono possibili piccoli errori, che tuttavia non ne dovrebbero modificare il valore qualitativo e lo scopo di punto di riferimento e confronto.

1. (**2 punti**)

Sono dati due vettori di interi a, b (non ordinati e contenenti eventualmente dati ripetuti), di dimensione rispettivamente na e nb. Si scriva una funzione che verifichi se il primo vettore sia una sotto-sequenza del secondo. La funzione deve poter essere chiamata come

ris = subSeq(a,na,b,nb);

Commento alla soluzione

Si tratta di un algoritmo lineare di iterazione in parallelo sui due vettori, ispirato all'intersezione (con vettori ordinati).

```
int subSeq(int a[], int na, int b[], int nb) {
  int i, j;
  for (i=0, j=0; i<na&&j<nb; j++) {
    if (a[i] == b[j]) {
      i++;
    }
  }
  return i==na;
}
```

2. (**4 punti**)

È dato un BST avente come valori e chiavi delle stringhe. Si scriva una funzione che determini il cammino radice-foglia contenente il massimo numero di nodi aventi 2 figli (non nulli/vuoti). La funzione stampi il cammino e ne ritorni la lunghezza. Nel caso di più cammini di lunghezza massima la scelta è arbitraria. La funzione abbia prototipo:

int BSTprintMax2(BST b);

Si richiede, oltre alla funzione, la definizione del tipo BST e del tipo usato per il nodo.

Commento alla soluzione

Il problema comprende due sottoproblemi

- calcolare la lunghezza (si usa per semplicità questo termine, per indicare il peso/valore del cammino ottimo richiesto) massima: è una variante del calcolo dell'altezza di un albero binario, a cui occorre aggiungere un filtro sui nodi da conteggiare (solo quelli con 2 figli non nulli)
- stampare il cammino: per fare questo sono possibili due strategie
    A. impostare un problema di ottimizzazione, con un vettore in grado di salvare la soluzione corrente e la soluzione migliore (un vettore di nodi): questa strategia ha il difetto di dover allocare (eventualmente sovra-allocare) un vettore
    B. procedere in due fasi: una prima funzione cerca il cammino ottimo, ritornandone sia la lunghezza che il valore (o la chiave, è indifferente) del nodo terminale. Nota la chiave del nodo terminale, il cammino puó facilmente essere stampato con una successiva variante della BSTsearch

La soluzione proposta fa riferimento all'ADT BST e adotta la strategia B per la stampa. Si utilizza un Item quasi ADT con un solo campo stringa. La funzione ricorsiva di calcolo della lunghezza massima torna come risultato la lunghezza e gestisce l'ltem ritornato "by pointer".

```
static int findMaxlR(link h, link z, Item *pathLeafP) {
  int l_right, l_left, l_this=0;
  Item pathLeafR, pathLeafL;

  if (h->r==z && h->l==z) { // nodo foglia
    *pathLeafP = h->item;
    return 0;
  }

  if (h->r!=z && h->l!=z) // nodo con due figli non vuoti
    l_this=1;

  l_left = findMaxlR(h->l,z,&pathLeafL);
  l_right = findMaxlR(h->r,z,&pathLeafR);

  if (l_right>l_left) {
    *pathLeafP =pathLeafR;
    return l_right+l_this;
  }
  else {
    *pathLeafP =pathLeafL;
    return l_left+l_this;
  }
}

static void printPathToLeafR(link h, Key pathLeafKey) {
  int cmp = KEYcmp(pathLeafKey,KEYget(&h->item));
  ITEMstore(h->item);
  if (cmp<0)
    printPathToLeafR(h->l, pathLeafKey);
  else if (cmp>0)
    printPathToLeafR(h->r, pathLeafKey);
}


int BSTprintMax2(BST b) {
  Item pathLeaf;
  int maxl = findMaxlR(b->root,b->z,&pathLeaf);
  printf("maxl: %d\n", maxl);
  printf("path nodes\n");
  printPathToLeafR(b->root,KEYget(&pathLeaf));
}
```

3. (**6 punti**)

Due stringhe s1 e s2 possono essere concatenate se il prefisso (proprio) di ordine I di s2 coincide col suffisso (proprio) di ordine I di s1. Si ricorda che il prefisso (suffisso) di ordine I di una stringa è una sottostringa iniziale (terminale) di I elementi, che si dice proprio se non coincide con la stringa stessa (I è minore della lunghezza della stringa). Si scriva una funzione checkConcat che, date due stringhe, verifichi se possono essere concatenate. Si scriva poi una funzione che, dato un insieme S di stringhe, trovi la più lunga sequenza di stringhe concatenabili, appartenenti all'insieme. L'insieme S è rappresentato come vettore. Non sono ammesse ripetizioni. come lunghezza (da massimizzare) si intende il conteggio delle stringhe nella sequenza.

La funzione abbia prototipo:

int maxConcatSeq(char **S, int n);

Commento alla soluzione

La funzione checkConcat consiste in un algoritmo di verifica che, utilizzando il principio dei quantificatori, cerca un valore di I tale che il suffisso di ordine I della prima stringa coincida col prefisso di ordine I dell'altra. La funzione maxConcatSeq realizza un algoritmo di ottimizzazione, facendo da wrapper a una funzione ricorsiva di generazione delle permutazioni delle parole, con pruning mediante checkConcat.

```
int checkConcat(char *s1, char *s2) {
  int I, k, l1=strlen(s1), l2=strlen(s2);
  int minL = l1<l2?l1:l2;
  for (I=1; I<minL; I++) {
    int concat = 1;
    for (k=1; k<=I; k++)
      if (s1[l1-k]!=s2[I-k])
        concat = 0;
    if (concat)
      return 1;
  }
  return 0;
}

void perm_sempl(char **S, int n, int pos, int *mark, char *last, int *bestP) {
  int i;
  if (pos > *bestP) {
    *bestP = pos;
  }

  for (i=0; i<n; i++)
    if (mark[i] == 0 && (last==NULL || checkConcat(last,S[i]))) {
      mark[i] = 1;
      perm_sempl(S, n, pos+1, mark, S[i], bestP);
      mark[i] = 0;
    }
}

int maxConcatSeq(char **S, int n) {
  int best=0, *mark=calloc(n,sizeof(int));
  perm_sempl(S,n,0,mark,NULL,&best);
  free(mark);
  return best;
}
```

Algoritmi e Strutture Dati

Appello del 27/1/2025 - Prova di programmazione (18 punti)

Descrizione del problema

Un campus universitario comprende più edifici, dotati ognuno di una rete locale (LAN). Si vogliono realizzare nuove connessioni in fibra ottica, che permettano di espandere le LAN in modo che comprendano più edifici. Le connessioni in fibra ottica possono essere fatte solo tra coppie selezionate di edifici (quindi non tutte) e hanno ognuna un costo.
Dato il grafo (non orientato e pesato, coi costi) delle possibili connessioni, si vuole fare in modo che ogni componente connessa del grafo (non si garantisce infatti che il grafo sia connesso) diventi un'unica LAN, realizzando un sotto-insieme delle possibili connessioni, in modo tale che siano soddisfatti i seguenti criteri:

- per ogni componente connessa, vanno scelti solo archi appartenenti a un albero ricoprente (spanning tree)
- gli alberi ricoprenti vanno scelti in modo che, detti C_{min}, C_{max} e C_{avg} i costi minimo, massimo e medio (media aritmetica) delle connessioni selezionate, si minimizzi il valore dell'espressione $0.6*C_{avg} + 0.4*(C_{max}-C_{min})$.

Richieste del problema

A seguire una sintesi delle richieste del problema. Per ogni richiesta si troverà una domanda dedicata nelle sezioni a seguire con una descrizione più dettagliata per le richieste.

Strutture dati e letture

Si utilizzi come struttura dati un grafo standard, realizzato con liste di adiacenza. Non è richiesta una funzione di input, in quanto si può supporre di utilizzare una GRAPHload standard. Si richiede unicamente la definizione della struct wrapper utilizzata per il grafo.
Si richiede poi di definire una opportuna struttura dati (o una modifica della struttura dati esistente per il grafo) che permetta, nella soluzione del problema di ottimizzazione, di disabilitare (o rimuovere temporaneamente) un arco oppure abilitarlo (dopo averlo precedentemente disabilitato): la struttura dati va aggiunta all'ADT Graph e vanno scritte le due funzioni GRAPHedgeDisable(Graph g, Edge e), e GRAPHedgeEnable(Graph g, Edge e).
Si definisca, come ADT di prima classe, una struttura dati CC, in grado di rappresentare le componenti connesse del grafo (per ogni componente l'elenco dei vertici appartenenti alla componente) , e si realizzi la funzione CC CCgen(GRAPH g), che, dato il grafo, calcoli le componenti connesse.

Commento alla soluzione

Il problema va risolto con versioni del grafo modificate dinamicamente, a seconda degli archi che si vogliano abilitare o meno. Per far questo si potrebbero generare piú grafi oppure modificarne una unica versione, abilitando o disabilitando di volta in volta uno o piú archi. Per abilitare o disabilitare un arco si possono usare varie strategie: cancellazione e reinserimento nella lista di adiacenza, aggiunta di un flag (strategia usata) all' arco in lista di adiacenza, rappresentazione (ridondante) con matrice di adiacenza in aggiunta alle liste di adiacenza (costa di piú in termini di memoria, specie con grafi sparsi, ma consente operazioni di enable/disable O(1)).
L' ADT CC fa riferimento a una struct wrapper che contiene numero di vertici e di componenti connesse e rappresentazione delle componenti come vettore parallelo ai vertici, in cui per ogni vertice viene indicata la relativa componente. Sono possibili altre informazioni, eventualmente anche informazioni necessarie per gli algoritmi.
successivi. L'algoritmo alla base delle CCgen è una variante del calcolo delle componenti connesse in cui va tenuto contro (dipende dall'implementazione) del flag di abilitazione.

```
// al nodo in lista di adiacenza si aggiunge il campo en (enable)
struct node { int v; int wt; int en; link next; } ;

typedef struct components *CC; // da Grapg.h
struct components {
  int V;
  int nCC;
  int *vertexCC;
};
void GRAPHedgeDisable(Graph g, Edge e) {
  link t;
  for (t=g->ladj[e.v]; t != g->z; t = t->next) {
    if (t->v == e.w) {
      t->en = 0;
      break;
    }
```

```
  }
}
void GRAPHedgeEnable(Graph g, Edge e) {
  link t;
  for (t=g->ladj[e.v]; t != g->z; t = t->next) {
    if (t->v == e.w) {
      t->en = 1;
      break;
    }
  }
}

// CCinit e CCfree omesse per semplicita'
static void dfsRcc(Graph G, int v, int id, int *cc) {
  link t;
  cc[v] = id;
  for (t = G->ladj[v]; t != G->z; t = t->next)
    if (cc[t->v] == -1 && t->en)
      dfsRcc(G, t->v, id, cc);
}

CC CCgen(Graph G) {
  int v, id = 0;
  CC cc = CCinit(G->V);
  if (cc == NULL)
    return NULL;
  for (v = 0; v < G->V; v++)
    cc->vertexCC[v] = -1;
  for (v = 0; v < G->V; v++)
    if (cc->vertexCC[v] == -1)
      dfsRcc(G, v, id++, cc->vertexCC);
  cc->nCC = id;
  return cc;
}
```

Problema di verifica
Si scriva una funzione in grado di verificare se un elenco di archi, ricevuto come parametro, contenga tutti e soli gli archi appartenenti a un insieme di alberi ricoprenti (uno per ogni componente connessa). La funzione abbia prototipo GRAPHcheckTreeEdges(Graph g, CC comp, Edge *ev, int en);

Commento alla soluzione

Servono due tipi di verifica:

- verificare che non manchino archi, cioè che il sotto-grafo limitato agli archi in ev copra lo stesso numero di componenti connessi del grafo originale (invece di generarne di piú)
- verificare che non ci siano troppi archi, che farebbero degenerare gli alberi in grafi con cicli. Questa verifica è banale: è sufficiente notare che in un albero il numero di archi è dato dal numero di vertici/nodi -1. Quindi complessivamente il numero di archi in ev (parametro en) deve essere pari a <numero vertici> - <numero componenti connesse>.

Il sotto-grafo limitato agli archi in ev puó essere generato come un nuovo grafo temporaneo, oppure si modifica quello esistente (soluzione proposta, in cui si disabilitano ptima tutti gli archi e poi si abilitano solo quelli necessari).

```
int GRAPHcheckTreeEdges(Graph g, CC comp, Edge *ev, int en) {
  /* si confronta il conteggio delle componenti connesse
     originali con quelle ottenute abilitando solo gli archi in ev
  */
  int i, ok;
  printf("checking %d edges on a grash with %d nodes and %d ccs\n",
```

```
                                        en, g->V, comp->nCC);

  if (en!=g->V-comp->nCC) {
    /* wrong number of edges */
    return 0;
  }
  Edge *all = malloc(g->E * sizeof(Edge));
  GRAPHedges(g, all);
  for (i=0; i<g->E; i++)
    GRAPHedgeDisable(g, all[i]);
  for (i=0; i<en; i++)
    GRAPHedgeEnable(g, ev[i]);
  CC ccNew  = CCgen(g);
  ok = ccNew->nCC == comp->nCC;
  for (i=0; i<g->E; i++)
    GRAPHedgeEnable(g, all[i]);
  free(all);
  return ok;
}
```

Problema di ricerca e ottimizzazione
Si scriva una funzione che, dato il grafo e le relative componenti connesse, permetta di trovare gli alberi ricoprenti, secondo il criterio di ottimizzazione richiesto. Si fa notare che, non trattandosi solo di minimizzare la somma dei pesi degli archi, le funzioni standard per calcolo di MST (alberi ricoprenti minimi) non possono essere utilizzate. È pertanto necessario realizzare un algoritmo ricorsivo in grado di trovare l'insieme ottimo di archi. La funzione deve corrispondere al prototipo seguente
Edge *GRAPHgenOptTrees(Graph g, CC comp);
Si chiede inoltre di specificare il modello di calcolo combinatorio usato e lo spazio in cui si cercano le soluzioni. Si dica inoltre se la funzione di ottimizzazione viene applicata ad ogni componente connessa individualmente (unendo successivamente le soluzioni ottime trovate) oppure su tutto il grafo (le risposta va motivata).

ATTENZIONE
Si noti che la differenza (C_{max}-C_{min}) non fa riferimento a ogni componente connessa ma a tutto il grafo (minimo e massimo potrebbero infatti appartenere a due diverse componenti connesse), il che ha conseguenze sulla scelta dell'algoritmo di ottimizzazione.
Si noti poi che, benché la funzione GRAPHcheckTreeEdges possa essere sufficiente per verificare l'eventuale ammissibilità di un sottoinsieme di archi, l'eventuale strategia adottata per il pruning sarà oggetto di valutazione.

Commento alla soluzione

Il problema prevede di trovare un albero ricoprente per ogni componente connessa, quindi il numero di archi da selezionare è noto (<numero vertici> - <numero componenti connesse>). Visto come problema di ottimizzazione, lo spazio in cui si cerca è l'insieme degli archi, in cui vanno prese in considerazioni tutti i sottoinsiemi di k archi (con k = <numero vertici> - <numero componenti connesse>). A seconda del criterio di ottimizzazione, l'algoritmo potrebbe essere iterato su ogni componente connessa, in cui cercare un albero ricoprente ottimo. Purtroppo questo non è possibile in quanto il criterio di ottimizzazione non puó essere limitato a una singola componente (C_{max} e C_{min} possono appartenere a due componenti diverse. Il criterio di ottimizzazione non incrementale pregiudica anchge lútilizzo di algoritmi greedy come Prim o Kruskal.
Quindi, in linea di principio, lo schema da adottare è: generare le combinazioni di k degli n archi del grafo, per ognuna applicare la funzione di verifica `GRAPHcheckTreeEdges.`
Questa strategia risulta peró estremamente costosa, in quanto priva di pruning. Si propone pertanto una soluzione alternativa che genera unicamente soluzioni accettabili, grazie al concetto di arco "sicuro" (concetto alla base degli algoritmi di Prim e Kruskal): per ogni arco selezionato si verifica che sia sicuro rispetto alla foresta di archi giá selezionati e vertici coperti. Per far questo si sfrutta Union-Find, con una piccola variante: occorre che la UFunion permetta backtrack: si 'realizzata una funzione UFbacktrack, qui non illustrata ma visibile nell'implementazione completa (è sufficiente uno stack che permetta di "ricordare" gli stati dei vettori id e sz da ripristinare in backtrack.
Sono possibili altre forme di pruning parziale, che evitino la Union-Find ma richiederanno una verifica completa nei casi terminali.

```
void checkBestSol(Edge *sol, Edge *bestSol, float *bestCostP, int k) {
  int i, min, max;
  float avg=0.0, cost;
  for (i=0; i<k; i++) {
    if (i==0 || sol[i].wt < min)
      min = sol[i].wt;
    if (i==0 || sol[i].wt > max)
      max = sol[i].wt;
    avg += sol[i].wt;
  }
  avg /= k;
  cost = 0.6*avg + 0.4*(max-min);
  if (*bestCostP<0.0 || cost<*bestCostP) {
    *bestCostP = cost;
    for (i=0; i<k; i++) {
      bestSol[i]=sol[i];
    }
  }
}
int checkSecure (Edge e) {
  return (!UFfind(e.v, e.w));
}
void comb_sempl(int pos, Edge *val, Edge *sol,
           Edge *bestSol, float *bestCostP, int n, int k, int start) {
  int i;
  if (pos >= k) {
    checkBestSol(sol,bestSol,bestCostP,k);
    return;
  }
  for (i=start; i<n; i++) {
    if (checkSecure(val[i])) {
      sol[pos] = val[i];
      UFunion(val[i].v, val[i].w);
      comb_sempl(pos+1, val, sol, bestSol, bestCostP,
                                          n, k, i+1);
      UFbacktrack();
    }
  }
}

Edge *GRAPHgenOptTrees(Graph g, CC comp) {
  Edge *edges, *sol, *bestSol;
  int k = g->V - comp->nCC;
  float bestCost = -1.0;
  edges = malloc(g->E * sizeof(Edge));
  sol = calloc(k,sizeof(Edge));
  bestSol = calloc(k,sizeof(Edge));
  UFinit(g->V);

  GRAPHedges(g, edges);
  comb_sempl(0, edges, sol, bestSol, &bestCost, g->V, k, 0);

  free(sol);
  free(edges);
  UFfree();
  return bestSol;
}
```