Partizione di grafo e Large Neighborhood Search

Progetto per Programmazione su Architetture Parallele

Definizioni: grafo pesato
Consideriamo un grafo diretto pesato $G(V, E)$ in cui V è un insieme di elementi e E è l'insieme degli archi.

Supponiamo anche che non vi siano elementi "isolati" in V, cioè elementi che non appartengono a nessun arco e neanche archi "cappio" (cioè archi del tipo (v, v), che partono e arrivano allo stesso nodo)

Il grafo è pesato, ovvero sono definite due funzioni:

- una funzione per i nodi: $wv(v) : V \to \mathbb{N}$ a valori interi non negativi
- una funzione per gli archi: $we(e) : E \to \mathbb{N}$ a valori interi non negativi

Supponiamo di partizionare i nodi di V in $S_1, \ldots, S_k$ insiemi (quindi $V = S_1 \cup \cdots \cup S_k$ e gli S_i sono a due a due disgiunti).

Per ogni S_i, S_j nella partizione, con $i \neq j$, definiamo:

- μ_i = la somma dei pesi $we(e)$ degli archi $e = (v, u)$ con sia v che u in S_i (somma dei pesi degli archi del sottografo composto dai soli nodi in S_i (gli archi "interni" a S_i)).
- $\epsilon_{i,j}$ = la somma dei pesi $we(e)$ degli archi $e = (v, u)$ con $v \in S_i$ e $u \in S_j$ (somma dei pesi degli archi che partono da un nodo di S_i e arrivano in un nodo di S_j

In base a queste definizioni possiamo dare un peso ad ogni parte della partizione, in base al peso degli archi interni e degli archi esterni. Per ogni parte S_i definiamo:

$$F(S_i) = \begin{cases} 0 & \text{se } \mu_i = 0 \\ \frac{2\mu_i}{2\mu_i + \Sigma_{j=1, j\neq i}^{k} (\epsilon_{i,j} + \epsilon_{j,i})} & \text{altrimenti} \end{cases}$$

Il peso di una partizione $S_1, \ldots, S_k$ è la somma dei valori $\Sigma_i F(S_i)$.

Definizioni: Large Neighborhood Search
La euristica Large Neighborhood Search (LNS) si può applicare per risolvere problemi di ottimizzazione. LNS procede come nell'algoritmo illustrato nella figura.

Ovvero: si parte da una soluzione x iniziale (linea 1); Si inizializza x^b alla migliore soluzione finora ottenuta (linea 2); Si ripete il loop linee 3-11 fino a che un criterio di terminazione dice che ci si può fermare (linea 11); In ogni ciclo si "distrugge" una parte della soluzione attuale (funzione `destroy`) e si ricostruisce la parte distrutta in modo diverso (funzione `repair`). In pratica si modifica x cambiandone alcune parti e si ottiene un'altra candidata soluzione x^t. Alla linea 5 si verifica se si possa accettare x^t come soluzione (per esempio si verifica che sia veramente una soluzione accettabile). Alla linea 8 si valuta se x^t sia migliore di x^b (il test può usare il confronto $>$ oppure $<$ a seconda del tipo di ottimizzazione che si vuole eseguire), e se lo è si aggiorna x^b (linea 9).

Algorithm 1 Large Neighborhood Search Algorithm.

```
1: x ← a feasible solution
2: x^b ← x
3: repeat
4:     x^t ← repair(destroy(x))
5:     if accept(x^t, x) then
6:         x ← x^t
7:     end if
8:     if cost(x^t) > cost(x^b) then
9:         x^b = x^t
10:    end if
11: until stop condition is met
12: return x^b
```

Progetto

Dato un grafo $G(V, E)$, un numero intero B, ed una partizione $S_1, \ldots, S_k$ dei nodi V tale che ogni S_i si abbia $\quad massa(S_i) = \Sigma_{v \in S_i} wv(v) \leq B$.

Implementare un programma CUDA che utilizza Large Neighborhood Search (LNS) per migliorare la partizione, cioè per trovare un'altra partizione $R_1, \ldots, R_h$ (è possibile che h e k siano diversi) con peso totale inferiore (cioè $\Sigma_j F(R_j) < \Sigma_i F(S_i)$) e tale che valga comunque che per ogni R_j si abbia che $massa(R_j) = \Sigma_{v \in R_j} wv(v) \leq B$.

Definizioni: Large Neighborhood Search per il partizionamento di grafo

Volendo usare lo schema dell'Algorithm 1 per migliorare una partizione di un grafo, dobbiamo scegliere alcune componenti dell'algoritmo: la funzione `destroy`; la funzione `repair`; la funzione `accept`; e la condizione di terminazione del loop.

Possiamo fare queste scelte in questo modo[1] (ove figura un parametro del programma m):

- la funzione `destroy`: se vi sono $|V|$ nodi nel grafo, scegliere casualmente $m < |V|$ nodi e toglierli dalla parte a cui appartengono (ovviamente il risultato non sarà più una partizione perchè questi m nodi non apparterranno a nessuna parte (la funzione `repair` correggerà la situazione).

  (Si osservi che questo passo potrebbe rendere vuoto uno o più degli S_i)

- la funzione `repair`: si sceglie casualmente uno dei m nodi scelti da `destroy` e si calcola di quanto incrementerebbe il valore $\Sigma_i F(S_i)$ se si aggiungesse tale nodo a S_i. Si compie questo calcolo per tutti gli S_i. Poi si aggiunge realmente il nodo alla parte S_i per la quale si ottiene il miglioramento maggiore.

  Si ripete per tutti gli m nodi (quindi alla fine del passo tutti gli m sono stati re-inseriti in parti della partizione).

  (Si osservi che se `destroy` ha reso vuoto uno degli S_i, la funzione `repair` potrebbe inserirci degli elementi oppure lasciarlo vuoto. In questo ultimo caso diminuisce il numero di parti della partizione (cioè diminuisce k))

- la funzione `accept`: accetteremo solo nuove soluzioni che migliorano il valore $\Sigma_i F(S_i)$ sempre imponendo che per ogni parte S_i valga che $massa(S_i) = \Sigma_{v \in S_i} wv(v) \leq B$.

- la condizione di terminazione del loop: ci sono due alternative che possiamo considerare:
  - si fissa un numero MAX e ci si ferma dopo MAX iterazioni del loop
  - ci si ferma dopo che ci sono state MAX iterazioni ciascuna delle quali ha portato un miglioramento della soluzione inferiore a un valore soglia t.

[1] non sono le uniche scelte possibili

Il parametro m è un dato di input dell'algoritmo e indica la percentuale di soluzione che deve essere distrutta ad ogni iterazione. Per esempio si potrebbe sceglire di distruggere il 10% (cioè $m = 0.10|V|$), o il 15%, o il 25% della soluzione, ecc.

OSSERVAZIONE: si noti che per come è definita $F(S_i)$ i calcoli da compiere per implementare `destroy` e `repair` possono essere fatti in modo incrementale, senza ricalcolare ad ogni passo tutti termini della espressione $\frac{2\mu_i}{2\mu_i+\Sigma_{j=1,j\neq i}^{k}(\epsilon_{i,j}+\epsilon_{j,i})}$.