Università di Pisa

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Implementazione dell'interfaccia tra i solutori MCF di LEMON e SMS++

Relatore:
Prof. Antonio Frangioni

Candidato:
Daniele Caliandro

Anno Accademico 2023/2024

Contents

Chapter 1

Stato dell'arte

In questo capitolo analizzeremo lo stato dell'arte del problema del flusso di costo minimo, descrivendone le proprietà e le applicazioni, sino ad arrivare agli algoritmi più diffusi per risoluzione dei problemi MCF.

1.1 Il problema del flusso di costo minimo

Consideriamo un grafo $G = (N, A)$ e supponiamo che sugli archi siano definiti un costo c_{ij} per unità di flusso ed una capacità superiore del flusso u_{ij}. Sui nodi siano assegnati dei bilanci b_i tali che la loro somma sia nulla. Il problema del flusso di costo minimo consiste nel determinare un flusso che rispetti i bilanci dei nodi e le capacità degli archi e che abbia costo totale minimo. Tale problema si può formulare nel modo seguente:

$$\begin{cases} \min c^T x \\ Ex = b \\ 0 \leq x \leq u \end{cases}$$

dove x è il vettore dei flussi sugli archi ed E è la matrice di incidenza.

Il problema del flusso di costo minimo può essere specializzato in problemi come il problema dei cammini minimi oppure il problema del flusso massimo, che affrontano le diferse componenti del problema generale del flusso di costo minimo. Per esempio, il problema dei cammini minimi considera il costo del flusso dell'arco ma non la capacità del flusso; il problema del flusso massimo considera le capacità ma solo la struttura dei costi più semplice. Dato che il problema del flusso di costo minimo combina questi ingredienti del problema, si possono combinare i metodi risolutivi per i cammini minimi ed il flusso massimo per sviluppare le condizioni di ottimalità e gli algoritmi.

1.2 Applicazioni

Il problema del flusso di costo minimo ha numerose applicazioni in diversi settori, grazie alla sua capacità di modellare situazioni in cui è necessario ottimizzare il trasporto o la distribuzione di risorse minimizzando i costi, sorge in quasi tutte le industrie, incluse agricoltura, comunicazioni, difesa, fabbricazione, medicina, trasporti, etc...

Esempi di applicazioni del problema del flusso di costo minimo sono i seguenti:

- Problema di distribuzione
- Pianificazione con costi di differimento
- Problema di trasporto
- Problema del flusso in reti di comunicazione
- Gestione delle risorse idriche
- Problema del flusso di lavoro

Ogni problema ha in comune con gli altri il fatto che è modellizzabile attraverso un grafo con costi sugli archi, quindi applicare algoritmi per la soluzione del problema del flusso di costo minimo garantisce di trovare la soluzione al problema posto, e.g per il problema del flusso in reti di comunicazione, la funzione obiettivo mira a minimizzare i costi di trasmissione in una rete di nodi (router o server) e archi (connesioni tra i router), quindi la soluzione ottimale ricevuta da un algoritmo che risolve il problema del flusso di costo minimo garantisce che la trasmissione avvenga con il minor costo di trasmissione nella rete.

1.3 Proprietà

Per determinare le soluzioni del problema si sfruttano, le proprietá delle matrici di incidenza e la definizione di grafo connesso. Una matrice di incidenza M è una matrice binaria NxA, con la quale viene rappresentato il grafo. Il generico elemento M_{ij} è definito come segue:

$$\begin{cases} M_{ij} = 1 \ sse \ \exists a_{ij} \\ M_{ij} = 0 \end{cases}$$

dove a_{ij} rappresenta l'arco che connette il nodo i al nodo j.

Sia $G = (N, A)$ un grafo, dove N è l'insieme dei vertici ed $A \subseteq NxN$ è l'insieme degli archi. Un grafo G si dice **connesso** se per ogni coppia di nodi u e v appartenenti a N, esiste un cammino che li connette. Formalmente:

$$\begin{aligned} &\forall u, v \in N, u \neq v \implies \exists \{w_0, w_1, ..., w_k\} \subseteq N \ tale \ che \ w_0 = u, w_k = v, \\ &\forall i \in \{0, 1, ..., k-1\}, (w_i, w_{i+1}) \in A \end{aligned} \tag{1.1}$$

Dalla definizione di grafo connesso si rappresentano gli **Alberi di copertura**. Nel grafo (N, A) Un **Albero di copertura** è un sottoinsieme di archi $T \subseteq A$ tale che il sottografo (N, T) sia connesso e non contenga cicli. **Una proposizione** che deriva da queste definizioni è:

Un grafo è **connesso** se e solo se contiene almeno un **albero di copertura**.

La matrice di incidenza quindi ha rango $n-1$ e il sistema $Ex = b$ è sovradeterminato, il sistema ammette soluzione specifica, che esiste sicuramente. Più precisamente tutte le soluzioni sono un sottospazio, {w + av} dove w è una soluzione specifica e v è un vettore del kernel della matrice d'incidenza.

Il duale del problema di flusso di costo minimo è definito come segue:

$$\begin{cases} max \ \pi^T b \\ \pi^T E \leq c^T \end{cases} \tag{1.2}$$

dove E è la matrice d'incidenza. Il duale è chiamato problema dei potenziali ai nodi della rete. Il potenziale di base è $\pi^T = c_T^T E_T^{-1}$ che è ammissibile se e solo se $\pi^T E_L \leq c_L$.

Riassumiamo nella tabella i possibili casi per i flussi ed i potenziali di base:

	flusso di base $x_T = E_T^{-1} b$	potenziale di base $\pi^T = c_T^T$
ammissibile	$\forall (i,j) \in T$ si ha $x_{ij} \geq 0$	$\forall (i,j) \in L$ si ha $c_{ij}^{\pi} \geq 0$
non ammissibile	$\exists (i,j) \in T$ tale che $x_{ij} < 0$	$\exists (i,j) \in L$ tale che $c_{ij}^{\pi} < 0$

Per quanto riguarda le condizioni di ottimalità dei flussi e dei potenziali, ci avvaliamo del teorema degli **Scarti complementari**:

- Un **flusso ammissibile** x è ottimo se e solo se esiste un potenziale π tale che :

$$\begin{cases} c_{ij}^{\pi} \geq 0 \text{ se } x_{ij} = 0 \\ c_{ij}^{\pi} = 0 \text{ se } 0 < x_{ij} < u_{ij} \\ c_{ij}^{\pi} \leq 0 \text{ se } x_{ij} = u_{ij} \end{cases} \tag{1.3}$$

- Un **potenziale** π è ottimo se e solo se esiste un flusso x tale che :

$$\begin{cases} Ex = b \\ x_{ij} = u_{ij} \text{ se } c_{ij}^{\pi} < 0 \\ 0 \leq x_{ij} \leq u_{ij} \text{ se } c_{ij}^{\pi} = 0 \\ x_{ij} = 0 se c_{ij}^{\pi} > 0 \end{cases} \tag{1.4}$$

1.4 Algoritmi

1.4.1 Cycle Canceling

L'algoritmo della cancellazione dei cicli mantiene una soluzione fattibile e ad ogni iterazione prova a migliorare il valore della sua funzaione obiettivo. L'algoritmo inizialmente stabilisce un flusso fattibile x nella rete risolvendo un problema di flusso massimo, dopo, itera cercando cicli di costo negativo nella rete residua ed aumenta il flusso su questi cicli. L'algoritmo termina quando la rete residua non contiene nessun ciclo di costo negativo. In pseudo-codice:

```
algorithm cycle-canceling;
begin
    estabilish a feasible flow x in the network
    while G(x) contains a negative cycle do
    begin
        use some algorithm to identify a negative
        cycle W;
        delta : = min(r_{ij} : (i,j) in W;
        augment delta units of flow in the cycle W
        and update G(x);
    end;
end;
```

La complessità dell'algoritmo Cycle Cancelling è $O(E^2VCP)$ dove E e V sono il numero di archi ed il numero di vertici rispettivamente; C - capacità massima degli archi; P - costo massimo di attraversamento degli archi

1.4.2 Capacity Scaling

L'algoritmo Capacity Scaling fa parte degli algoritmi di stile Ford-Fulkerson, è una variante dei cammini minimi successivi, è correlato all'algoritmo del cammino minimo successivo, proprio come l'algoritmo Capacity scaling per il problema del flusso massimo. Capacity scaling richiama l'algoritmo del labeling che performa $O(nU)$ incrementi; mandando il flusso attraverso cammini con sufficientemente larghe capacità residue, l'algoritmo capacity scaling riduce il numero di incrementi a $O(mlogU)$. In modo simile, l'algoritmo Capacity Scaling per il problema di flusso di costo minimo assicura che ogni incremento nel cammino minimo porta con se un sufficiente ammontare di flusso. Questo algoritmo non migliora solamente le performance dell'algoritmo dei cammini minimi successivi, ma mostra come piccoli cambiamenti in un algoritmo possono produrre significanti miglioramenti algoritmici. L'algoritmo Capacity Scaling è applicato al generico problema capacitato del flusso di costo

minimo. L'algoritmo mantiene uno pseudoflusso x che soddisfa le condizioni di ottimalità di costo ridotto e gradualmente converte lo pseudoflusso in un flusso identificando i cammini minimi dai nodi con eccessi a nodi con deficit e aumenta il flusso su questi cammini. Performa un numero di fasi di ridimensionamento con uno specifico valore Δ come il Δ-scaling phase. Inizialmente, $\Delta = 2^{logU}$. L'algoritmo assicura che nel Δ-scaling phase ogni incremento porta con se esattamente Δ unità di flusso.

In pseudo-codice:

```
algorithm capacity scaling;
begin
    x : = 0 and pi : = 0
    Delta = 2^{log U};
    while Delta >= 1
    begin
        for every arc (i,j) in the residual network G(x) do
            if(r_{ij} >= Delta and C_{ij}^{pi} < 0 then
            send r_{ij} units of flow along arc (i,j),
            update x and the imbalances Theta;

        S(Delta) : = i in N t.c Theta(i) >= Delta;
        T(Delta) : = i in N t.c Theta(i) <= -Delta;

        while S(Delta) != empty and T(Delta) != empty do
        begin
        select a node k in S(Delta) and a node l in T(Delta)
        determine shortest path distances d() from node k
        to all other nodes in the Delta-residual network
        G(x, Delta) with respect to
        the reduced costs c_{ij}^{pi};

        let P denote shortest path from node k to node l
        in G(x, Delta);
        update pi : = pi - d;
        augment Delta units of flow along the path P;
         update x, S(Delta), T(Delta) and G(x, Delta);
         end;
         Delta : = Delta/2;
     end;
end;
```

Il tempo computazionale per il CapacityScaling è $O(E^2(I + log_2 C_{max}))$ dove E è l'insieme degli archi, I è il numero di volte che viene eseguito il loop con controllo di $\Delta \geq 1$

1.4.3 Cost Scaling

L'algoritmo di Cost scaling per il flusso di costo minimo può essere visto come una generalizzazione dell'algoritmo preflow-push per il problema di flusso massimo, infatti, l'algoritmo ha un interessante relazione tra i problemi del flusso massimo e del flusso di costo minimo.

Un flusso x o uno pseudoflusso x è detto essere ϵ-otpimal for some $\epsilon > 0$ se per alcuni potenziali dei nodi π, la coppia (x, π) soddisfa le seguenti condizioni di ϵ-ottimalità

- Se $c_{ij}^{\pi} > \epsilon$ allora $x_{ij} = 0$.
- Se $-\epsilon \leq c_{ij}^{\pi} \leq \epsilon$ allora $0 \leq x_{ij} \leq u_{ij}$
- Se $c_{ij}^{\pi} < -\epsilon$ allora $x_{ij} = u_{ij}$

L'algoritmo cost scaling tratta ϵ come un parametro ed iterativamente ottiene un flusso ϵ-ottimale per successivi valori più piccoli di ϵ. Inizialmente, $\epsilon = C$ ed un flusso fattibile è ϵ-ottimale. L'algoritmo quindi performa fasi di scalatura del costo ripetendo un applicazione di miglioramento dell'approssimazione, procedura che trasforma un flusso ϵ-ottimale in un flusso $1/2\epsilon$-ottimale. Dopo $1 + log(nC)$ fasi di scalamento del costo, $\epsilon < 1/n$ e l'algoritmo termina con un flusso ottimale.

In pseudocodice:

```
algorithm cost scaling;
begin
    pi = 0 and epsilon = C;
    let x be any feasible flow
    while epsilon >= 1/n do
    begin
        improve-approximation(epsilon, x, pi);
        epsilon = epsilon/2;
    end;
    x is an optimal flow for the MCF problem9
end
```

1.4.4 Network Simplex

Prima di definire l'algoritmo del simplesso di rete, bisogna definire gli strumenti che utilizza per restituire una soluzione ottimale, ovvero gli spanning tree. Uno spanning tree è definitio come un sottoinsieme di grafi indiretti connessi che hanno tutti i vertici coperti con il minor numero di archi possibili.

L'algoritmo del simplesso di rete mantiene un fattibile spanning tree ad ogni iterazione e successivamente lo trasforma in uno spanning tree migliorato finchè non diventa ottimale.

Pseudo codice:

```
algorithm network simplex
begin
    determine an initial feasible tree structure(T, L, U);
    let x be the flow and pi be the node
    potentials associated with this tree structure;
    while some nontree arc violates the optimality conditions
    do
    begin
        select an entering arc (k, l)
        violating its optimality condition;
        add arc (k, l) to the tree and
        determine the leaving arc (p, q);
        perform a tree update and
        update the solution x and pi;
    end;
end;
```

La complessità temporale dell'algoritmo del simplesso di rete è $O(V^2 Elog(VC))$ dove C è il costo massimo di tutti gli archi, V è il numero di vertici ed E il numero di archi.

Chapter 2

Il framework SMS++

In questo capitolo presenteremo il framework utilizzato nel progetto della tesi, SMS++, e ne descriveremo le sue principali funzionalità e la sua struttura attuale.

2.1 Presentazione Framework

SMS++ è un insieme di classi C++ che forniscono un sistema per modellizzare modelli matematici complessi e strutturati a blocchi, in particolare ma non esclusivamente, problemi di ottimizzazione con una singola funzione obiettivo reale, e risolverli attraverso sofisticati algoritmi che ne sfruttano la struttura come ad esempio approcci di decomposizione. L'obiettivo di SMS++ è quello di fornire un sistema che provvede per i concetti che:

- il modello e gli approcci risolutivi possono essere strettamente integrati
- Nell'ambito di approcci risolutivi complessi, possono essere necessari diversi solutori con riferimento allo stesso (parte del) modello
- Nell'ambito di approcci risolutivi complessi, diverse parti del modello possono cambiare in modo arbitrario ed i solutori devono essere in grado di sfruttare le informazioni sulla soluzione precedente e l'elenco dei dati modificati per (cercare di) riottimizzare in modo efficiente (nella misura in cui è possibile)
- Diversi solutori possono avere bisogno di rappresentazioni diverse dello stesso modello (o parte del modello) sottostante, che quindi devono essere mantenute sincronizzate quando cambiano
- Gli approcci risolutivi complessi possono procedere in parallelo

2.1.1 Principali funzionalità di SMS++

SMS++ porta una serie di caratteristiche più in là cercando di avvicinare il modello e gli algoritmi di soluzione. Si tratta di un sistema di modellazione basato su un insieme di classi C++, dove un modello è rappresentato come un blocco: una classe astratta di base destinata a rappresentare il concetto generale di "una parte di un modello matematico con un'identità ben conosciuta". In altre parole, le classi derivate rappresenteranno in linea di principio un modello con una struttura specifica (ad esempio, un knapsack problem, un Traveling Salesman Problem, un SemiDefinite problem, ...) che in linea di principio consentirà di sviluppare algoritmi di soluzione specializzati per la loro soluzione. Questi dovrebbero essere implementati come classi derivate da Solver, una classe base astratta che definisce l'interfaccia generale tra un blocco che rappresenta un modello matematico e qualsiasi algoritmo progettato per risolverlo, ovvero produrre soluzioni. La classe CDASolver, leggermente specializzata, rappresenta gli algoritmi di soluzione che sfruttano le strutture convesse (e, in particolare, la dualità) del modello.

Dopo aver descritto SMS++ in un impostazione piuttosto classica, descriviamo più in dettaglio le funzionalità offerte dal framework.

- Un blocco può avere qualsiasi numero di blocchi interni, ricorsivamente, che rappresentano separate parti del modello. Una conseguenza di ciò, è che un Solver può essere collegato all'individuo sotto-blocco(ricorsivamente) di un blocco più grande, facilità l'attuazione risoluzioni basate in techiche avanzate, come la precedentemente citata **Decomposizione**.

- Solver specializzati saranno in gradi di sfruttare la struttura semanticamente definita del blocco, i.e, la **Rappresentazione fisica** dell'istanza (un insieme di oggetti con pesi, grafi pesati, liste di matrici quadrate, ...) per sviluppare i loro approcci risolutivi. Comunque, ogni blocco specializzato presuppone che possa fornire una **Rappresentazione astratta** di se stesso, in termini delle sue **Variable**, **Costraints** e **Objective**, in modo da essere in grado sia di implementare metodi di risoluzione che mischiano approcci a scopo generale e Solver ad-hoc per una più specifica.

- **Variable**, **Costraints and Objective** sono classi astratte che provano a stabilire l'interfaccia minima possibile di un sistema di modellizzazione, facendo il minor numero di piccole assunzioni sulla forma attuale dei corrispondenti oggetti matematici. Questo dovrebbe permettere di estendere il sistema ad applicazioni e strutture matematiche differenti. Possono essere statici e dinamici, in modo da permettere strategie dove possono essere dinamicamente generate mentre forniscono il Solver con informazioni riguardanti quali non aandranno mai a cambiare. Variable/-Costraint sia statici e dinamici possono essere singole o inserite in un std::vector oppure boost::multiarray; le dinamici sono sempre all'interno di std::list dato che il "nome" della Variable/Costraint è il suo puntatore, e quindi Variable/Costraint non possono essere mai spostate nella memora.

- Per consentire di affrontare le applicazioni reali, un insieme "minimale" di implementazioni concrete sono fornite dal sistema "core", come **ColVariable** (una singola Variable reale, possibilmente con restrizioni di integrità), **RowCostraint** (a **Costraint** della forma:

  $$"LHS \Leftarrow (singlerealexpression) \Leftarrow RHS$$

  con un valore duale reale corrispondente), e **RealObjective** (una funzione obiettivo a valori reali). Anche una classe generale **Function** è fornita per esprimere un concetto generale di funzioni a valori reali, insieme alla sua derivata **C05Function** (che fornisce linearizzazione, i.e, informazioni al primo ordine, ma non necessariamente continue) e **C15Function** (che fornisce modelli quadratici, i.e, informazioni al secondo ordine, ma non necessariamente continue). Con questo, **FRow-Constraint** and **FRealObjective** sono definite come **RowCostraint**

e **RealObjective** prendendo una generica **Function** per implementarle. Qualche "semplice" **Function** è fornita, come **LinearFunction** e **DQuadFunction**, le quali permettono di rappresentare l'importante classe di **Mixed-integer Linear Problems**. Anche la classe **OneVarConstraint** e qualche altra specializzazione è separatamente derivata da **RowConstraint** per implementare il caso dove la "singola espressione reale" è una singola **ColVariable**, i.e, "vincoli di box".

- Per mezzo delle classi sopra SMS++ può essere anche usato in un modo affine a un linguaggio algebrico standard di modellizzazione; uno specifico **AbstractBlock** è fornito, che, diversamente dai **Block** regolari, permette di avere solo la **"Rappresentazione astratta"** ed essere fornito dall'utente, senza la classe che fa assunzioni per il momento, sul tipo di **Variable/Constraint and Objective**.

- Cambiamenti nei dati del **Block**, sia nella "rappresentazione fisica" che nella "rappresentazione astratta"(**Variable**, **Constraint**, **Objective**. **Function**, ...) sono trattate in un modo totalmente lazy, i.e, sono registrate in oggetti di(derivate dalla classe di base) classe **Modification**, che sono fornite a tutti gli oggetti **Solver** registrati al **Block** e tutti i suoi antenati. **Modification** sono mandate attraverso **Smart pointers** in modo che la gestione della memoria sia banale. Il meccanismo della **Modification** fornisce anche supporto cruciale per il meccanismo da dove la "rappresentazione astratta" di un **Block** è cambiato, quindi la "fisica rappresentazione" deve essere modificata conseguentemente.

- **Block** supporta computazioni asincrone fornendo i metodi lock() e readlock(); questi performano automaticamente la stessa operazione in tutti i **Block** interni. Le operazioni di Lock stabiliscono un proprietario, che permette di assicurare che il **Block** non è manipulato da differenti esecuzioni nello stesso thread (std::mutex è usata per gestire thread differenti, ma non funzionerebbe in questo contesto). **Solver** supporta anche computazione asincrona avendo la lista delle **Modification** in sospeso modificate solamente sotto un'attiva guardia(la std::atomic) e un meccanismo che può prendere in prestito l'identità del proprietario esistente per il caso in cui un **Solver** è collegato a un **Block** ha bisogno di delegare dei tasks al **Solver** collegato al **Block** interno (ricorsivamente).

- **Block** supporta (anche se solo nell'interfaccia, non in pratiche implementazioni) il concetto che un **Block** può produrre versioni differenti di se stesso che sono entrambe equivalenti (una riformulazione), rilassamento o restrizione. L'interfaccia fornissce il fatto che entrambe le informazioni delle soluzioni e le **Modification** possono essere mappate all'indietro e

in avanti da un **Block** a qualunque del suo "R3 Block", anche se questo è tutto basato sul fatto che **Block** stesso lo implementi.

- Siccome **Block** ha un arbitrariamente largo oggetto strutturato ad albero, configurarlo ed attaccarlo al Solver di bisogno e a tutti i **Block** interni necessari è un task complesso. Per questo la classe **Configuration** è fornita ed intende di rappresentare oggetti per collezionare alcuni (strutturati) parameteri algoritmici. La classe è la base per **BlockConfig** (che colleziona parametri per tasks differenti che un **Block** ha da compiere), **ComputeConfig** (colleziona parametri algoritmici per **Solver** e **Function** complessa), e **BlockSolverConfig** ha classi derivate che permettono di fornire per le arbitrarie strutture ad albero.

- Lo status della soluzione di un **Block**(il valore della sua **Variable**) può essere salvata in oggetti appropriati derivati dalla classe base **Solution**, e dopo rileggerli da questi nel **Block**. Gli oggetti **Solution** possono possibilmente salvare qualsiasi sottoinsieme dell'informazione(come dettato dall'oggetto **Configuration** e la soluzione duale se disponibile. Generali **ColVariableSolution**, **RowConstraintSolution** and **ColRowSolution** sono forniti per fare questo usando solo la "rappresentazione astratta", per la primale, duale, o entrambe le informazioni rispettivamente.

- **Block**, **Solver** e **Configuration** tutte forniscono una factory per generare dinamicamente oggetti di qualsiasi classe derivata, solamente passando la stringa del nome della classe (la classe fornita è registrata alla factory, il quale supporto è fornito con alcune macros).

- **Block** e **Configuration** sono supposti di essere capaci di serializzare e deserializzare se stessi, in particolare usando lo standard industriale , i file netCDF; questo può essere utile anche per supporti programmati in futuro a computazioni asinccrone e sistemi a memoriia non condivisa.

2.2 Struttura

I progetti correnti presenti nel framework SMS++ sono:

- **SMS++ core library**, in cui sono definite le funzionalità generali del framework SMS++.
- **BinaryKnapsackBlock**, un **Block** che implementa il problema dello zaino binario e il corrispondente **Solver** basato su una diretta implementazione con l'approccio della programmazione dinamica (per pesi interi).
- **CapacitatedFacilityLocationBlock**, un implementazione di un concetto di **Block** per una versione base del problema Capacitated Facility Location (CFL), inteso primariamente come una implementazione "didattica" per mostrare alcune funzionalità di SMS++
- **BundleSolver**, a **Solver** per problemi di ottimizzazione che coinvolgono funzioni obiettivo non differenziabili, basato sul "bundle method". Usa alcuni moduli da **NDOSolver / FiOracle project**, anche se la dipendenza sarà sperabilmente rimossa in tempo
- **InvestmentBlock**, un **Block** designato per modellare l'investimento in differenti beni definiti in **UCBlock**. Il corrispondente problema può essere anche deterministico o stocastico, nel qual caso il modulo usa anche **SDDPBlock** e quindi **StochasticBlock**.
- **LagrangianDualSolver**, un generico **Solver** Lagrangian-based per **Block** con una struttura appropriata.
- **LukFiBlock**, un semplice **Block** definendo molte funzioni di test dalla letteratura per **Solvers** di ottimizzazioni non differenzizabile (proprio come **BundleSolver**
- **MCFBlock / MCFSolver**, definisce la classe **MCFBlock** per (continui e lineari) problemi MCF e il suo associato **MCFSolver**, che è un wrapper per solver dal **MCFClass project**. Parleremo più nello specifico di MCFBlock nei prossimi capitoli.
- **MILPSolver**, definisce il generale **MILPSolver**, un **Solver**, che mira a risolvere qualsiasi **Block** il quale astrae la codifica della rappresentazione per problemi lineari Mixed-integer(**ColVariable**, **FRowConstraint** and **FRealObjective** con **LinearFunction** all'interno, **OneVarConstraint**, insieme con le classi derivate che attualmente interfacciano con esistenti MILP solver. Correntemente, classi derivate disponibili sono:

  - **CPXMILPSolver**, che fornisce l'interfaccia con **Cplex**.
  - **SCIPMILPSolver**, fornisce l'interfaccia con l'open-source **SCIP**.
  - **GRBMILPSolver**, che fornisce l'interfaccia con **GUROBI Optimizer**, usato nei test del progetto per testare la correttezza delle soluzioni trovate dal **Solver** di LEMON.
  - **HiGHSMIKPSolver**, che fornisce l'interfaccia con l'open-source **HiGHS**

- **MMCFBlock**, che definisce **MMCFBlock** per rappresentare Multi-commodity MCF problems.

- **SDDPBlock**, definisce **SDDPBlock** per ottimizzazioni stocastiche multi-stage risolvibili dall'approccio della programmazione dinamica duale, e **SDDPSolver** che interfaccia con SDDP sovler nel progetto **StOpt**

- **StochasticBlock** "meta-Block" che prende qualsiasi **Block** deterministico e lo "rende stocastico" accettando cambiamenti su alcuni dei suoi dati in un modo molto generale e astratto (usando metodi factory di SMS++)

- **UCBlock**, definisce differenti **Block** per problemi di Unit Commitment: la classe root **UCBlock**, molti **Block** per specifiche unita di generazione **UnitBlock** e interconnettere reti **NetworkBlock**, con qualche **Solver** specializzato.

- **tests**, definisce testers complessi per diverse componenti del progetto che richiedono elementi **Block** e/o **Solver**.

- **tools**, definisce qualche strumento che può essere utile per gli utenti(come "main files" che prendono istanze dei problemi e li rsiolvono) e che richiede elementi **Block** e/o **Solver**.

Chapter 3

Il progetto LEMON

In questo capitolo descriveremo LEMON, discuteremo sulle scelte dei template e daremo un accenno alla sua struttura.

3.1 Presentazione Libreria

LEMON sta per Library for Efficient Modeling and Optimization in Networks, è una libreria C++ template che fornisce implementazioni efficienti di comuni strutture dati ed algoritmi con l'obiettivo sull'ottimizzazione combinatoria dei task connessi principalmente con grafi e reti.

LEMON è un progetto open source, con licenza Boost 1.0, che è una licenza molto permissiva. La libreria aiuta a scrivere programmi che risolvono vari problemi di ottimizzazione, i quali spesso vengono fuori mentre si progettano e testano certe reti, per esempio nelle telecomunicazioni, reti di computer, logistica, scheduling e altre aree. Un modo naturale di modellare queste redi è attraverso i grafi.

3.1.1 Funzionalità principali

Le funzionalità principali di LEMON sono strutture dati, algoritmi e strumenti ausiliari che rendono possibile di rappresentare grafi e lavorarci in maniera semplice ed efficiente. Vi sono due tipi di grafi principali utilizzati nel progetto di tesi, e sono due classi derivate da Digraph, SmartDigraph e ListDigraph. SmartDigraph è molto efficiente in memoria ma al prezzo di non supportare cancellazione di nodi e di archi. ListDigraph è basata su linked lists, ed è possibile cancellare nodi e archi, quindi questa sarà la classe del grafo utilizzata nelle modification che sono implementate nel progetto. I parametri del problema, come le capacità, costi, supply, sono rappresentate all'interno degli algoritmi di LEMON tramite delle mappe, UpperMap, CostMap e SupplyMap. Nel progetto passiamo agli algoritmi queste mappe in questo modo.

```
// defining names for types for readability
using MCFArcMapV = typename GR::template ArcMap< V >;
using MCFNodeMapV = typename GR::template NodeMap< V >;

// now we are going to fill up ArcMap and NodeMap
// this is the case of upperMap
if( ! MCFB->get_U().empty() ) {
 MCFArcMapV um( * dgp );  // create the upper map
 auto & u = MCFB->get_U();
 for( MCFBlock::Index i = 0 ; i < m ; ++i )
  um.set( dgp->arcFromId( i ) , u[ i ] );

 f_algo->upperMap( um );  // pass it to the Algo
 }

// this is the case of CostMap
if( ! MCFB->get_C().empty() ) {
 MCFArcMapV cm( * dgp );  // create the cost map
 auto & c = MCFB->get_C();
 for( MCFBlock::Index i = 0 ; i < m ; ++i )
  cm.set( dgp->arcFromId( i ) , c[ i ] );

 f_algo->costMap( cm );  // pass it to the Algo
 }

// this is the case of supplyMap
if( ! MCFB->get_B().empty() ) {
 MCFNodeMapV bm( * dgp );  // create the supply map
 auto & b = MCFB->get_B();
 for( MCFBlock::Index i = 0 ; i < n ; i++ )
  bm.set( dgp->nodeFromId( i ) , -b[ i ] );

 f_algo->supplyMap( bm );  // pass it to the algo
 }
```

LEMON fornisce implementazioni dei quattro algoritmi citati nella sezione Algoritmi precedente:

- CycleCancelling
- CapacityScaling
- CostScaling
- NetworkSimplex

A partire da queste il progetto di tesi definisce le quattro classi derivate MCFLemonSolverCycleCanceling, MCFLemonSolverCapacityScaling, MCFLemonSolverCostScaling e MCFLemonSolverNetworkSimplex, di cui parleremo meglio nei prossimi capitoli.

3.1.2 Templates

I Templates sono la base per la programmazione generica in C++. Essendo un linguaggio fortemente tipato, C++ richiede che tutte le variabili abbiano un tipo specifico, sia dichiarate dal programmatore che dedotte dal compilatore. Comunque, molte strutture dati ed algoritmi sembrano gli stessi con qualsiasi tipo loro operino. I Templates permettono di definire le operazioni di classi o funzioni, e lasciare all'utente la specifica delle operazioni sulle quali i tipi concreti dovrebbero lavorare. La libreria LEMON ha molte classi che sono definite template, cioè tutti gli algoritmi e le strutture dati. La maggior parte dei parametri template riguardano i tipi di grafo usati, i tipi dei valori e dei costi, ed i Traits.

3.2 Struttura

3.2.1 Strutture Dati

In questa sezione ritroviamo strutture dati per grafi, mappe, Cammini, Heap e strutture dati ausiliaree, come matrici e strutture dati geometriche.

Le implementazioni di algoritmi combinatori si affidano pesantemente a implementazioni efficienti di grafi. Le strutture dati offerte da LEMON sono studiate per essere facili da usare in fasi sperimentali e studi implementativi.

I grafi implementati possono essere diretti o indiretti, in questa lista ritroviamo la classe base dei Digraph e la classe base dei Graph.

3.2.2 Algoritmi

Nella lista di LEMON degli algoritmi per il problema MCF troviamo

- NetworkSimplex, algoritmo primale del simplesso di rete con varie pivot-strategies
- CostScaling, algoritmo basato sul push/augmentig e operazioni di relabeling
- CapacityScaling, algoritmo basato sul metodo dei cammini minimi successivi
- CycleCanceling, due implementazioni per l'algoritmo CyCle-Canceling, fortemente polinomiali.

In generale, **NetworkSimplex** e **CostScaling** sono le implementazioni più efficienti. NetworkSimplex è di solito la più veloce su grafi relativamente piccoli (diverse migliaia di nodi) e su grafi densi, mentre **CostScaling** è tipicamente più efficiente su grandi grafi(centinaia di migliaia di nodi), specialmente se sono sparsi. Comunque, altri algoritmi possono essere più veloci in casi speciali. Per esempio **CapacityScaling**, se i supply values e/o le capacità sono piuttosto piccole, è di solito l'algoritmo più veloce.

Queste classiu sono create per essere usate con tipo di valori di input inter (capacità, supply values e costi) eccetto per **CapacityScaling**, che è capace di usare costi di archi a valori reali(gli altri valori devono essere interi).

Nel progetto usiamo le classi CapacityScaling, CostScaling, CycleCanceling, NetworkSimplex, tutte template su almeno 3 parametri, le uniche due con 4 parametri sono CapacityScaling e CostScaling che accettano i Traits, nel progetto vengono passati come DefaultTraits definiti da LEMON.

3.2.3 Strumenti e utilità

Forniscono dei solver specifici per ogni algoritmo e dei tool base per operare su grafi, che includono copia di grafi di ogni tipo, iteratori su nodi e archi

3.2.4 Input-Output

Il tipo di file scelto per il progetto è il dimacs, da LEMON sono fornite varie funzionalità per leggere e scrivere da file dimacs. Una funzione utilizzata nel progetto è la **writeDimacsMat** che permette di scrivere su file dimacs un grafo.

Chapter 4

MCFBlock

MCFBlock è parte del framework SMS++ implementa il concetto di Block per problemi MCF. Il dato del problema consiste in un grafo diretto $G = (N, A)$ con $n = |N|$ nodi e $m = |A|$ archi diretti. Ogni nodo i ha un deficit $b[i]$, i.e, l'ammontare del flusso che è consumata dal nodo: i nodi sorgente (che producono il flusso) hanno deficit negativi e nodi pozzo (che consumano il flusso) che hanno deficit positivi. Ogni arco (i, j) ha una capacità superiore $U[i, j]$ e un coefficiente di costo lineare $C[i, j]$. Variabili di flusso $X[i, j]$ che rappresentano l'ammontare del flusso che deve essere mandato su un arco (i, j). Archi paralleli, i.e, copie multiple dello stesso arco (i, j) sono permesse in generale; ci si aspetta che loro abbiano costi differenti(altrimenti possono essere fuse in un unico arco), ma questo non è strettamente forzato.

Il grafo G è dinamico in parte. L'insieme dei nodi e degli archi che sono input all'inizio è assunto che non cambino (eccetto per cambiare costi, capacità, deficit e per archi da aprire o chiudere). Dopo, nuovi archi e nodi, ad un massimo previsto, possono essere dinamicamente aggiunti o cancellati. Questo significa che il grafo può essere completamente statico (se il numero massimo degli archi dinamici e nodi è 0) così come totalmente dinamico (se il grafo iniziale è vuoto). Nota che cancellare gli ultimi archi decresce il numero degli archi, cancellando in mezzo può solo lasciare un buco: l'arco non è li e qualunque arco nuovo può prendere il suo nome, ma il numero totale degli archi riportato non cambia.

Nota che cambiare costi, capacità e derficit è permesso anche attraverso la rappresentazione astratta. Similmente, aprire e chiudere archi può essere performato dalla disattivazione e attivazione della variabile di flusso corrispondente. Comunque, tutte le altre operazioni richiedono lavoro complesso sulla rappresentazione astratta e quindi non possono essere eseguiti attraverso questo. Si può in principio permettere di gruppare le Modification in GroupModification permettendo di controllare che tutte le operazioni necessarie a creare e cancellare archi sono fatte correttamente nella rappresentazione as-

tratta, ma questo non è ancora implementato. Quindi, aggiungere/rimuovere Variable da un vincolo di conservazione del flusso, o anche cambiare i suoi coefficienti, non è permesso attraverso la rappresentazione astratta, così come non aggiungere/rimuovere Constraint dinamici(che siano conservazioni del fluisso o limitiazioni). Similmente, archi cancellati al centro corrispondono a variabili del flusso fissate a 0, che non possono essere cambiate dalla rappresentazione astratta. In tutti questi casi, verranno lanciate eccezioni.

Nel progetto vengono utilizzate le funzioni:

```
/// get the vector of starting nodes

[[nodiscard]] c_Subset & get_SN( void ) const { return( SN ); }


/// get the starting node of arc i (0 <= i < get_NArcs())

[[nodiscard]] Index get_SN( Index i ) const { return( SN[ i ] ); }
```

per riempire il grafo da passare a LEMON.

4.1 Struttura

La struttura del file MCFBlock.h è definita come segue:

- Getter/Setter per le soluzioni primali e duali
- Funzioni per gestire le Modification
- Funzioni per la serializzazione

4.1.1 Getter/Setter per le soluzioni primali e duali

```
/// gets the flow potential for an arbitrary subset of nodes
void get_pi( Vec_CNumber_it PSol , c_Subset & nms ) const;

/// gets the flow solution for an arbitrary subset of arcs
void get_x( Vec_FNumber_it FSol , c_Subset & nms ) const;

/// sets a generic subset of the flow solution
void set_x( c_Vec_FNumber_it fstrt , c_Subset sbst )

/// sets a generic subset of the potential solution
void set_pi( c_Vec_FNumber_it pstrt , c_Subset sbst );
```

4.1.2 Funzioni per gestire le modification

La versione di MCFBlock deve intercettare le modifiche astratte che modificano la rappresentazione astratta del MCFBlock, e traducono le modification in cambiamenti sulle strutture dati e la corrispondenti modification fisiche. Queste modification sono quelle per cui Modification::concernsBlock() è vero. Nota, comunque, che prima di mandare una modification al solver e al blocco padre, la funzione concernsBlock() è settata a false. Questo perchè una volta che si è passati da questo metodo, la Modification astratta ha già svolto il proprio dovere di fornire informazioni all'MCFBlock, e questo non deve essere ripetuto. In particolare, Questo sarebbe un problema se la Modification potrebbe essere mappata in avanti o indietro, perchè dentro questo metodo una fisica Modification che fa lo stesso lavoro è sicuramente compromessa. Questa Modification potrebbe anche essere mappata in avanti o all'indietro, insieme con l'originale Modification astratta che passa di nuovo attraverso questo metodo, che vuol dire che la Modification fisica sarà usata due volte.

```
/// adding a new Modification to the MCFBlock
void add_Modification( sp_Mod mod , ChnlName chnl = 0 )
override;
```

Ci sono anche funzioni che permettono di gestire il cambiamento dei costi, capacità, deficits, chiudere archi ed aprirli, aggiungere archi e rimuoverli. Tutto questo va implementato nel progetto della tesi e sarà oggetto di discussione della sezione Modification del capitolo MCFLemonSolver.

4.1.3 Funzioni per la serializzazione

Ci sono chiaramente le funzioni per serializzare/deserializzare MCFSolution in file netCDF. Questa è la sintassi:

```
/// extends Block::deserialize( netCDF::NcGroup )
void deserialize( const netCDF::NcGroup & group ) override;

/// serialize a MCFSolution into a netCDF::NcGroup
void serialize( netCDF::NcGroup & group ) const override final;
```

Esse sono definite in overriding dei metodi di base della Block class, per quanto riguarda la deserialize group contiene informazioni sul grafo, quantità di nodi e archi, vari array contenenti costi, capacità e deficit degli archi e dei nodi, gli insiemi SN ed EN ed altre informazioni per manipolare archi e nodi.

Invece la serialize contiene informazioni come numero di archi e nodi, la variabile FLowSolution e la variabile Potentials che contengono i flussi e i potenziali.

Chapter 5

MCFLemonSolver

In questo capitolo descriveremo l'implementazione di MCFLemonSolver, le scelte implementative, le strutture utilizzate e ...

5.1 Classe Base

Gli algoritmi nel progetto LEMON sono template su almeno tre tipi

- GR, il tipo di grafo. Le possibilità sono SmartDigraph e ListDigraph.
- V, che è il tipo dei flussi/deficit; tipicamente , double può essere usato per la massima compatibilità, ma int è piu performante.
- C, che è il tipo del costo degli archi; tipicamente double può essere usato per la massima compatibilità, ma int è piu performante.

Inoltre, gli algoritmo di tipo scaling possono avere differenti modi a seconda di quali V e C usati, e ci sono differenti traits per questo che sono un altro parametro template. Comunque, preferiamo che MCFLemonSolver sia sempre template sopra tre parametri solamente, questo è il motivo per cui definiamo SMSppCapacityScaling e SMSppCostScaling come template sopra $< GR, V, C >$ e usando trait di default.

Quindi, il concept LEMONGraph è definito in modo che accetti solo i tipi di grafo LEMON, cioè :

- SmartDigraph non supporta cancellazione di archi e nodi
- ListDigraph supporta cancellazione di archi e nodi.

```
 /// concept for "one of the LEMON graphs"
template< typename Type >
concept LEMONGraph =
 std::is_same< Type , SmartDigraph >::value   ||
 std::is_same< Type , ListDigraph >::value;
```

5.1.1 Classe MCFLemonSolver

MCFLemonSolver è un CDASolver per MCFBlock basato sul progetto LEMON. MCFLemonSolver implementa l'interfaccia Solver per MCFBlock che rappresenta problemi MCF lineari, usando algoritmi della libreria di LEMON. Dato che MCF è un programma lineare che ha un esatto duale, MCFLemonSolver implementa l'interfaccia CDASolver per anche dare informazioni duali.

MCFLemonSolver è template sopra quattro tipi differenti:

- GR, tipo del grafo
- V, tipo del flusso/deficit

- C, tipo del costo
- Algo, che è lo specifico algoritmo(è un template template parametro, che ha altri 3 parametri template che sono GR, V, C. Le possibilità sono:
  - NetworkSimplex implementa il Simplesso di rete primale
  - CycleCanceling implementa tre differenti algoritmi
  - CostScaling implementa un algoritmo cost scaling
  - CapacityScaling implementa la versione di capacity scaling dell'algoritmo dei cammini minimi successivi per trovare un MCF.

```
    template< template< typename , typename , typename > class Algo ,
          LEMONGraph GR , typename V , typename C >
class MCFLemonSolver : public CDASolver
```

Ho scelto di implementare una classe base template e sottoclassi derivate per gestire separatamente i parametri algoritmici delle sottoclassi derivate

5.2 Classi Derivate

Le classi derivate da MCFLemonSolver sono quattro, ognuna rappresenta uno dei possibili algoritmi implementati da LEMON per il problema MCF. Abbiamo:

- MCFLemonSolverNetworkSimplex
- MCFLemonSolverCycleCanceling
- MCFLemonSolverCapacityScaling
- MCFLemonSolverCostScaling

ognuna di loro definisce una gutsofcompute differente, in modo da utilizzare i corretti parametri algoritmici.

5.2.1 MCFLemonSolverNetworkSimplex

$MCFLemonSolver < NetworkSimplex, GR, V, C >$ specializzato.

La classe è specializzata su NetworkSimplex come primo parametro template, lasciando liberi gli altri tre. Il costruttore chiama gutsOfConstructor della classe base e assegna alla variabile che contiene il parametro algoritmico PivotRule il suo valore di default

```
MCFLemonSolverNetworkSimplex( void ) {
  BaseClass::guts_of_constructor();
  f_pivot_rule = NSPivotRule::BLOCK_SEARCH;
}
```

La classe contiene enums per indicizzare i parametri algoritmici e le corrispondenti set/get*par per gestirli.

```
enum LEMON_NS_int_par_type{
  kPivot = intLastParCDAS,
  intLastParLEMON_NS
};

void set_par( idx_type par , int value ) override {
      if( par == kPivot ) {
       if( ( value < 0 ) || ( value > 4 ) )
        throw( std::invalid_argument( "Error: invalid kPivot " +
                      std::to_string( value ) ) );

       if( value == f_pivot_rule )
        return;  // nothing is changed

       f_pivot_rule = NSPivotRule( value );
       return;
       }

      CDASolver::set_par( par , value );
      return;
}
```

L'enum LEMONNSintpartype definisce l'indice per il parametro algoritmico PivotRule. La funzione setPar di cambiare parametro algoritmico

La gutsofcompute di MCFLemonSolverNetworkSimplex è implementata in questo modo:

```
void guts_of_compute(void) override {
status = f_algo->run(NSPivotRule(f_pivot_rule));
}
```

NSPivotRule indica con quale regola viene scelto il pivot nell'algoritmo.

5.2.2 MCFLemonSolverCycleCanceling

La classe è specializzata su CycleCanceling come primo parametro template, lasciando liberi gli altri tre. Il costruttore chiama gutsOfConstructor della classe base e assegna alla variabile che contiene il parametro algoritmico Method il suo valore di default

```
MCFLemonSolverCycleCanceling( void ){
    BaseClass::guts_of_constructor();
    f_method = CycleCanceling<GR, V, C>::Method::
CANCEL_AND_TIGHTEN;
}
```

La classe contiene enums per indicizzare i parametri algoritmici e le corrispondenti set/get*par per gestirli.

```
enum LEMON_CC_int_par_type{
        kMethod = intLastParCDAS,
        intLastParLEMON_CC
};

 void set_par(idx_type par, int value) override {

    if( par == kMethod ) {
    if( ( value < 0 ) || ( value > 4 ) )
      throw( std::invalid_argument( "Error: invalid kPivot " +
            std::to_string( value ) ) );

    if( value == f_method )
      return;  // nothing is changed

    f_method = CCMethod( value );
    return;
    }


    CDASolver::set_par( par , value );
    return;
}
```

L'enum LEMONCCintpartype definisce l'indice per il parametro algoritmico Method. La funzione setPar permette di cambiare valore del parametro algoritmico.

La gutsofcompute di MCFLemonSovlerCycleCanceling è implementata in questo modo:

```
void guts_of_compute() override {
status = f_algo->run(CCMethod(f_method));
}
```

5.2.3 MCFLemonSolverCapacityScaling

La classe è specializzata su SMSppCapacityScaling come primo parametro template, lasciando liberi gli altri tre. Il costruttore chiama gutsOfConstructor della classe base e assegna alla variabile che contiene il parametro algoritmico Method il suo valore di default

```
/// constructor: does nothing
 MCFLemonSolverCapacityScaling( void ) : BaseClass() {
 BaseClass::guts_of_constructor();
}
```

Inizializza solamente la classe base, dato che non ha parametri algoritmici. La gutsofcompute è implementata in questo modo:

```
void guts_of_compute() override {
    status = f_algo->run();
}
```

Non avendo parametri algoritmici, la run ha solo un opzione con cui essere chiamata.

5.2.4 MCFLemonSolverCostScaling

La classe è specializzata su SMSppCostScaling come primo parametro template, lasciando liberi gli altri tre. Il costruttore chiama gutsOfConstructor della classe base e assegna alla variabile che contiene il parametro algoritmico Method il suo valore di default La classe contiene enums per indicizzare i

```
MCFLemonSolverCostScaling( void ) {
    BaseClass::guts_of_constructor();
    f_method = SMSppCostScaling<GR, V, C>::Method::
PARTIAL_AUGMENT;
}
```

parametri algoritmici e le corrispondenti set/get*par per gestirli.

```
enum LEMON_CS_int_par_type{
    kMethod = intLastParCDAS,
    intLastParLEMON_CS
};

void set_par(idx_type par, int value) override {

    if( par == kMethod ) {
    if( ( value < 0 ) || ( value > 4 ) )
      throw( std::invalid_argument( "Error: invalid kPivot " +
            std::to_string( value ) ) );

    if( value == f_method )
      return;  // nothing is changed

    f_method = CSMethod( value );
    return;
    }


    CDASolver::set_par( par , value );
    return;
}
```

L'enum LEMONCSintpartype definisce l'indice per il parametro algoritmico Method. La funzione setPar permette di cambiare valore del parametro algoritmico.

La gutsofcompute è implementata in questo modo:

```
void guts_of_compute() override {
    status = f_algo->run(CSMethod(f_method));
}
```

CSMethod indica con quale metodo è stato avviato l'algoritmo di CostScaling.

5.3 Problemi di interfacciamento

La classe MCFLemonSolver e le 4 classi derivate permettono di interfacciare gli algoritmi di lemon e le sue strutture dati ad un MCFBlock, infatti i dati vengono innanzitutto deserializzati da un file netCDF in un MCFBlock, ogni informazione ricavata viene passata alle strutture di LEMON grazie ai metodi forniti da LEMON. Un problema riscontrato sul quale ci ho speso diverso tempo è stato capire come specializzare un MCFLemonSolver::compute in una compute $MCFLemonSolver < NetworkSimplex, GR, int, int >:: compute$ senza successo, quindi ho dovuto usare l'ereditarietà definendo classi derivate dalla classe base specializzata sugli algoritmi. La principale differenza fra LEMON e SMS++ è che LEMON è basato sui template mentre SMS++ è basato sulle classi. SMS++ gestisce in modo chiaro le tolleranze come parametri algoritmici mentre LEMON non da questa possibilità, dà una tolleranza base usata in tutti gli algoritmi che non è modificabile. Nei casi UNFEASIBLE SMS++ riporta un taglio, mentre i developer di LEMON hanno deciso di non implementare questa funzionalità. Nel caso UNBOUNDED SMS++ riporta un ciclo di costo negativo, mentre LEMON si limita a riconoscere che il problema non è limitato, senza ritornare nessun tipo di informazione all'utente.

Per un debug più accurato si può utilizzare la funzione writeDimacsMat che scrive su file il grafo, ma questa funzionalità non è supportata dai grafi di tipo ListDigraph, quindi abbiamo dovuto gestire il caso tirando un'opportuna eccezzione.

È stato fondamentale riconoscere la differenza tra la specializzazione template e l'ereditarietà, che sono due concetti ortogonali tra loro, in particolar modo le specializzazioni template non ereditano i membri del template primario, quindi è stato difficoltoso comprendere come separare logicamente ogni funzionalità di LEMON da quelle di SMS++. Da questo fatto derivano le classi specializzate, che derivano da una classe template primaria specializzata (nell'algoritmo richiesto), queste classi sono servite principalmente per distinguere i parametri algoritmici logici di LEMON, le funzioni ereditate dalla classe base definite virtual (ovvero la gutsofCompute).

Un altro problema sorto a causa di LEMON e dei mancati aggiornamenti relativi ad esso, era la compatibilità con la versione di c++20, quindi ho dovuto eseguire una patch alla loro libreria su un certo file **arraymap.h**, il quale usava le funzioni di uno std::allocator **construct** e **destroy**, che sono state rimosse da c++20 in poi, quindi per risolverle ho utilizzato **std::allocatortraits** che permettono alla libreria di essere compatibile con la versioni di c++ 20 e superiori.

Un problema descritto nella classe base MCFLemonSolver è quello dei parametri template degli algoritmi, che sono al minimo tre, ma ci sono due algoritmi: CostScaling e CapacityScaling che portano con se un quarto parametro

template, ovvero i Traits. Abbiamo gestito questo problema creando delle classi template su tre parametri che ereditano dalle classi principali degli algoritmi in questione, quelle con quattro parametri template, usando dei trait di default (quindi specializzate su questi ultimi), in codice ecco quello che abbiamo fatto:

```
/// CapacityScaling algorithm using the default trait
template< LEMONGraph GR , typename V , typename C >
class SMSppCapacityScaling : public
 CapacityScaling< GR , V , C , CapacityScalingDefaultTraits< GR , V
   , C > >
{
 public:
 SMSppCapacityScaling( const GR & dgp ) :
  CapacityScaling< GR , V , C ,
                   CapacityScalingDefaultTraits< GR , V, C > >( dgp
   ) {}

 ~SMSppCapacityScaling() = default;
 };

/// CostScaling algorithm using the default trait
template < LEMONGraph GR , typename V , typename C >
class SMSppCostScaling : public
 CostScaling< GR , V , C , CostScalingDefaultTraits< GR , V , C > >
{
 public:
 SMSppCostScaling(const GR & dgp ) :
 CostScaling< GR , V , C ,
              CostScalingDefaultTraits< GR , V , C > > ( dgp ) {}

 ~SMSppCostScaling() = default;
 };
```

Questo permette alle classi SMSppCostScaling ed SMSppCapacityScaling di essere registrate nella factory dei solver del file .cpp con soli tre parametri template, come tutte le altre del resto.

5.4 Modification

5.4.1 Premessa

Per la gestione delle modification si è dovuto implementare una sottoclasse di ListDigraph, MCFListDigraph che consente una gestione del grafo personalizzata per i nostri scopi. SmartDigraph non supporta le modifiche sugli archi, mentre ListDigraph supporta tutto l'insieme di Modification fornito. Le Modification supportano modifiche sul grafo utilizzato dall'algoritmo istanziato e possono essere di quattro tipi:

- Ranged, sono modifiche che avvengono in un range di archi/nodi, ovvero il file di test crea un range su cui verranno attuate le modifiche e queste ultime sono processate per ogni indice del range. Sono supportate le operazioni di modifica:
  - ChgCost, ossia cambio di costo di uno o più archi.
  - ChgCap, ossia cambio di capacità di uno o più archi.
  - ChgDfc, ossia cambio di deficit/supply di uno o più nodi.
  - AddArc, ossia l'aggiunta di un nuovo arco.
  - RmvArc, ossia la rimozione di un arco già esistente.
  - OpenArc, ossia la riapertura di un arco chiuso in precedenza.
  - CloseArc, ossia la chiusura di un arco già esistente.
- Subset, sono modifiche che avvengono su un sottoinsieme di archi/nodi, non necessariamente in un range, ossia il tester passa ai solver un sottoinsieme qualsiasi di archi/nodi che vengono processati. Sono supportate le seguenti operazioni di modifica:
  - ChgCost
  - ChgCap
  - ChgDfc
  - OpenArc
  - CloseArc
- NB Modification, ossia l'opzione che cambia totalmente l'MCFBlock su cui l'algoritmo lavora, viene processata all'interno della addModification, ossia se la modifica arrivata è di questo tipo, viene richiamata una gutsofsetBlock per riaggiornare totalmente le strutture dati da passare all'algoritmo.
- Group Modification, ossia un gruppo di modifiche che vengono spacchettate all'interno della gutsofModification, questo gruppo può contenere sia modifiche Ranged che Subset.

5.4.2 Modifica costi/capacità/domanda

Queste modifiche sono le più semplici in quanto basta cambiare il valore del costo/capacità/domanda presenti nelle mappe di LEMON in quelli presenti nell'MCFBlock di riferimento. Sono progettate per eseguire una modifica "all-in-one", ovvero ogni modifica va a settare un flag booleano che viene controllato prima di lanciare l'algoritmo, se questo flag è settato allora si aggiornano le strutture di costi/capacità/domanda in modo da soddisfare la modifica. Esempi di codice:

```
case (MCFBlockMod::eChgCost): {
    for (auto i : tmod->nms())
        if (dgp->valid(dgp->arcFromId(i))) {
            auto arc = dgp->arcFromId(i);
            auto cost = MCFB->get_C(i);
            cm->set(arc, std::isnan(cost) ? 0 : cost);
            cost_changed = true;
        }
    return;
}
case (MCFBlockMod::eChgCaps): {
    auto &CC = MCFB->get_C();
    auto &U = MCFB->get_U();
    for (auto i : tmod->nms())
        if (!std::isnan(CC[i]) &&
        dgp->valid(dgp->arcFromId(i))) {

            um->set(dgp->arcFromId(i), U[i]);
            cap_changed = true;
        }
    return;
}
case (MCFBlockMod::eChgDfct): {
    MCFBlock::Vec_FNumber NDfct(tmod->nms().size());
    MCFBlock::Subset nmsI(tmod->nms().size() + 1);
    auto B = MCFB->get_B();
    for (MCFBlock::Index i = 0; i < NDfct.size(); i++) {
        bm->set(dgp->nodeFromId(i), -B[nmsI[i]]);
        supply_changed = true;
    }
    return;
}
```

Queste modifiche sono esempi di modifiche Subset, infatti ognuna di loro itera su un vettore di indici nms() in cui sono contenuti gli indici degli archi/nodi da modificare

5.4.3 Aggiunta/rimozione/apertura/chiusura archi

Queste sono le modifiche più complicate in quanto vanno a cambiare la struttura del grafo di riferimento, per quanto riguarda l'aggiunta di archi semplicemente il nuovo grafo conterrà un nuovo arco con costi e capacità intere, mentre per la rimozione l'arco viene rimosso se e solo se esiste nel grafo ed è valido, indipendentemente se è chiuso o meno. Esempio di codice per aggiunta archi:

```
    case (MCFBlockMod::eAddArc): {
            // only if the graph is ListDigraph
            if constexpr (std::is_same<GR, MCFListDigraph>::value) {

                auto ca = MCFB->get_C(rng.first);
                auto startNode = MCFB->get_SN(rng.first) - 1;
                auto endNode = MCFB->get_EN(rng.first) - 1;
                auto capacity = MCFB->get_U(rng.first);

                dgp->addArc(startNode, endNode);
                auto arc = dgp->arcFromId(rng.first);

                n_arcs_added++;
                cm->set(arc, std::isnan(ca) ? 0 : ca);
                um->set(arc, capacity);

                cost_changed = true;
                cap_changed = true;

                return;
            } else {
                throw(std::logic_error(
                    "SmartDigraph doesn't support operations on arc")
);
            }

            return;
        }
```

La parte cruciale di questa modifica è la chiamata alla funzione addArc, una funzione della sottoclasse MCFListDigraph, una sotttoclasse di ListDigraph aggiunta nel progetto. Questa sottoclasse fornisce un metodo addArc che, a differenza dell'addArc di ListDigraph, aggiunge l'arco nel buco con indice più piccolo, mentre ListDigraph aggiunge l'arco nell'ultimo buco creato con le rimozioni dell'arco.

Esempio di codice di rimozione archi:

```
case (MCFBlockMod::eRmvArc): {
        // only if ListDigraph is used
        if constexpr (std::is_same<GR, MCFListDigraph>::value) {
            // if the arc was previous deleted, return
            if (!dgp->valid(dgp->arcFromId(rng.second - 1))) {
                return;
            }
            auto arc = dgp->arcFromId(rng.second - 1);

            // if the arc isClosed, only 'mark' it as eliminated,
            // otherwise normally erase it
            if (dgp->isClosed(rng.second - 1)) {
                dgp->eraseClosed(rng.second - 1);
            } else {
                dgp->erase(arc);
            }

            n_arcs_deleted++;
        } else {
            throw(std::logic_error(
                "SmartDigraph doesn't support operations on arc")
);
        }

        return;
    }
```

La parte cruciale della rimozione degli archi è la chiamata all'erase corretta, che può essere eraseClosed(), sempre implementata nella sottoclasse MCFListDigraph, oppure l'erase() implementata da ListDigraph. Le due funzioni differiscono sulla gestione della stella entrante e uscente dei nodi sorgenti e destinazione dell'arco, in quanto se l'arco viene chiuso in precedenza vengono settate in modo da non far "vedere" l'arco all'algoritmo, quindi bisogna solamente 'marcare' l'arco come cancellato. Mentre se l'arco non è stato chiuso in precedenza, bisogna cancellarlo totalmente, gestendo stella entrante e uscente ed il flag che indica l'arco come cancellato.

Per quanto riguarda chiusura ed apertura archi, invece, si è fatto in modo di sfruttare le strutture fornite da LEMON in modo da rendere "inutilizzabili" gli archi chiusi, semplicemente modificando la stella entrante ed uscente dei nodi sorgente dell'arco in questione e l'ID univoco del nodo sorgente, che viene cambiato di segno, in caso di chiusura ed in riapertura viene ripristinato al suo valore originale. Esempio di codice per apertura archi:

```
case (MCFBlockMod::eOpenArc): {
        // only if ListDigraph is used
        if constexpr (std::is_same<GR, MCFListDigraph>::value) {
            for (auto arc : tmod->nms())
                // checking if the arc is closed
                if ((!MCFB->is_deleted(arc)) &&
                    dgp->isClosed(arc) &&
                    dgp->valid(dgp->arcFromId(arc))) {
                    dgp->openArc(arc);
                    n_arcs_added++;
                }
        } else {
            throw(std::logic_error(
                "SmartDigraph doesn't support operations on arc")
        }
        return;
```

Durante la riapertura di un arco viene controllato se è stato già chiuso in precedenza, guardando semplicemente se l'ID del nodo sorgente è negativo e, se vero, viene riaperto cambiandogli di segno l'ID e ripristinando la stella entrante ed uscente. Viene incrementato un counter di archi aggiunti in modo tale che, prima che l'algoritmo esegua una run, vengano ripristinate le strutture dati contenenti gli archi, necessario per far rendere conto all'algoritmo che un arco è stato riaperto.

Esempio di codice per chiusura archi: Quando un arco viene chiuso invece,

```
        case (MCFBlockMod::eCloseArc): {
        // only if ListDigraph is used
        if constexpr (std::is_same<GR, MCFListDigraph>::value) {

            for (auto arc : tmod->nms())
                // checking if the arc is not closed
                if ((!MCFB->is_deleted(arc)) &&
                    dgp->valid(dgp->arcFromId(arc)) &&
                    !dgp->isClosed(arc)) {
                    dgp->closeArc(arc);
                    n_arcs_deleted++;
                }

            return;
        } else {
            throw(std::logic_error(
                "SmartDigraph doesn't support operations on arc")
        }
    };
```

viene prima di tutto controllato se non è già stato chiuso in precedenza, altrimenti andremmo a negare l'ID dell'arco che ritornerebbe positivo falsando tutti i controlli futuri. Viene chiamata la funzione closeArc che va a gestire l'ID del nodo sorgente, negandolo e modificando la stella entrante ed uscente dei nodi dell'arco, in modo da renderlo "invisibile" all'algoritmo, ma rimane nella struttura degli archi in quanto il suo flag di cancellazione non è stato settato.

Chapter 6

Tests

Questa sezione descrive il file di test utilizzato per comparare le soluzioni trovate dagli algoritmi e come sono stati organizzati ed eseguiti.

6.1 Premessa

Bisogna innanzitutto fare una grossa premessa, gli algoritmi di LEMON lavorano per lo più su valori e costi interi, ma è possibile farli funzionare anche con valori di tipo double. I test per le versioni che usano costi interi però, riscontrano un problema ad un certo punto dei test, ovvero, appena arriva un grafo con costi troppo grandi, si hanno degli integer overflow, in quanto l'aritmetica di macchina per quel tipo non supporta operazioni con valori così grandi, quindi abbiamo ovviato il problema usando costi e valori di tipo *long*, mentre per i parametri di tipo double non ci sono problemi, i test sono superati da tutte le istanze disponibili.

6.2 Struttura del file di test

Il file di test.cpp è strutturato in due parti:

- Controllo e gestione parametri di input
- Fase di confronto fra soluzioni

6.2.1 Controllo e gestione parametri di input

Quando il file di test viene avviato, tipicamente tramite il batch, che descriveremo dopo, gli viene dato in pasto un certo numero di parametri che sono:

- file, rappresenta il grafo .netCDF
- seed, seme del generatore pseudo-random
- wchg, what to change, codificato bit a bit
- rounds, numero di round di cambiamento
- chng , numero medio di elementi da cambiare
- chng, probabilità del singolo cambiamento

Inizialmente il parametro file, viene deserializzato in un puntatore ad MCFBlock se il file è un .nc4, altrimenti viene semplicemente caricato nel block.

In seguito, troviamo le configurazioni del BlockSolverConfig, in cui si deserializza un file molto importante *BSPar.txt* che rappresenta quali solver verranno eseguiti e come sono gestiti i parametri algoritmici.

Poi vengono fatti calcoli per il numero di cambiamenti e poi arriva la parte più interessante del file di test, ovvero:

```
bool AllPassed = SolveBoth();
```

6.2.2 Fase di confronto fra soluzioni

SolveBoth() confronta il flusso di output di entrambi i Solver e il valore di ritorno della compute(). Ecco il codice della SolveBoth():

```
static bool SolveBoth( void )
{
 try {
  // solve with the 1st Solver
  auto Slvr1 = MCFB->get_registered_solvers().front();
  #if DETACH_1ST
   MCFB->unregister_Solver( Slvr1 );
   MCFB->register_Solver( Slvr1 , true );  // push it to the front
  #endif

  int rtrn1st = Slvr1->compute( false );
  bool hs1st = ( ( ( rtrn1st >= Solver::kOK ) && ( rtrn1st < Solver::
   kError )
                  && ( rtrn1st != Solver::kUnbounded )
                  && ( rtrn1st != Solver::kInfeasible ) )
                || ( rtrn1st == Solver::kLowPrecision ) );
  double fo1st = hs1st ? Slvr1->get_var_value() : -CInf;

  // solve with the 2nd Solver
  auto Slvr2 = MCFB->get_registered_solvers().back();
  #if DETACH_2ND
   MCFB->unregister_Solver( Slvr2 );
   MCFB->register_Solver( Slvr2 );  // push it to the back
  #endif

  int rtrn2nd = Slvr2->compute( false );

  bool hs2nd = ( ( ( rtrn2nd >= Solver::kOK ) && ( rtrn2nd < Solver::
   kError )
                  && ( rtrn2nd != Solver::kUnbounded )
                  && ( rtrn2nd != Solver::kInfeasible ) )
                || ( rtrn2nd == Solver::kLowPrecision ) );
  double fo2nd = hs2nd ? Slvr2->get_var_value() : -CInf;

  if( hs1st && hs2nd && ( std::abs( fo1st - fo2nd ) <= 5e-7 *
              std::max( double( 1 ) ,
                    std::max( std::abs( fo1st ) ,
                          std::abs( fo2nd ) ) ) ) ) {
   LOG1( "OK(f)" << std::endl );
   return( true );
   }

  if( ( rtrn1st == Solver::kInfeasible ) &&
      ( rtrn2nd == Solver::kInfeasible ) ) {
   LOG1( "OK(e)" << std::endl );
   return( true );
   }
```

In seguito, troviamo molte guardie che controllano quali tipi di Modification sono richieste dal parametro di input wchg , un intero che viene messo in and-bitwise in modo da rendere gestibile solo dal parametro di input del programma quale Modification verrà chiamata.

- $wchg \wedge 1$, ossia abilitazione del cambio di costo
- $wchg \wedge 2$, ossia abilitazione del cambio di capacità
- $wchg \wedge 4$, ossia abilitazione del cambio di deficit
- $wchg \wedge 8$, ossia abilitazione dell'apertura degli archi
- $wchg \wedge 16$, ossia abilitazione della chiusura degli archi
- $wchg \wedge 32$, ossia abilitazione della cancellazione di archi
- $wchg \wedge 64$, ossia abilitazione della aggiunta di archi

Le istanze degli algoritmi MCFListDigraph supportano tutte le modifiche, ovvero $whcg \wedge 255$, mentre le istanze degli algoritmi SmartDigraph supportano solamente le prime tre modifiche, ovvero $whcg \wedge 7$.

6.2.3 Tempi di esecuzione

Di seguito riportati i tempi di esecuzione per le istanze testate, riportando per ogni tipologia di grafo il tempo totale di esecuzione dell'algoritmo su tutta la tipologia di grafo , e.g la colonna net8_8 conterrà i tempi di esecuzione di ogni algoritmo eseguendo ogni grafo nell'insieme net8_8, quindi da net8_8_1 a net8_8_5. La prima colonna contiene le istanze degli algoritmi, di seguito riportata la codifica

- NS = NetworkSimplex
- CS = CostScaling
- CC = CycleCanceling
- CaS = CapacityScaling
- LD = MCFListDigraph
- SD = SmartDigraph
- L = long
- D = double
- GRB = GurobiMilpSolver

	net8_8	net8_32	net10_8	net10_64	net12_8	net14_8
GRB	1.42	2.03	2.96	13.77	11.20	297.38
NS<LD,L,L>	9.34	38.31	43.78	291.25	203.62	764.52
CS<LD,L,L>	19.16	41.67	53.50	392.25	234.04	1501.90
CC<LD,L,L>	12.48	45.33	59.60	491.67	393.34	2113.32
CaS<LD,L,L>	20.78	153.59	77.74	4564.71	454.68	3380.02
NS<LD,D,D>	0.94	2.41	2.86	30.59	36.66	481.74
CS<LD,D,D>	0.68	1.89	2.84	40.69	54.60	822.10
CaS<LD,D,D>	1.58	74.21	26.88	3438.81	381.08	2881.72
CC<LD,D,D>	1.30	3.89	8.46	123.29	123.24	923.32

Table 6.1: La tabella contiene i tempi in secondi delle esecuzioni di tutte le istanze degli algoritmi con grafo LD su tutti i grafi net.

	net8_8	net8_32	net10_8	net10_64	net12_8	net14_8
GRB	0.95	1.51	1.89	8.16	12.23	96.05
NS<SD,L,L>	16.06	48.07	48.38	352.23	222.16	921.04
CS<SD,L,L>	12.20	48.61	51.44	356.03	226.30	949.62
CC<SD,L,L>	14.12	45.89	52.68	367.85	378.64	2646.02
CaS<SD,L,L>	15.74	138,61	73.80	4484.49	389.08	3777.24
NS<SD,D,D>	0.93	1.71	3.27	19.44	26.31	222.41
CS<SD,D,D>	1.01	0.81	1.91	15.68	52.35	1220.01
CC<SD,D,D>	1.55	1.65	3.21	28.48	132.11	1593.57
CaS<SD,D,D>	2.53	76.75	22.45	3442.76	315.23	3396.69

Table 6.2: La tabella contiene i tempi in secondi delle esecuzioni di tutte le istanze degli algoritmi con grafo SD su tutti i grafi net.

	goto8_8	goto8_32	goto10_8	goto10_64	goto12_8	goto14_8
GRB	1.38	3.05	7.57	30.58	26.87	1109.45
NS<LD,L,L>	13.26	66.37	69.95	767.42	463.39	3720.77
CS<LD,L,L>	9.96	47.09	68.01	365.20	224.93	3207.13
CC<LD,L,L>	21.62	88.51	184.29	764.28	1201.63	35717.35
CaS<LD,L,L>	10.52	79.93	68.09	1570.78	1910.97	9016.85
NS<LD,D,D>	4.04	19.03	37.51	490.98	386.71	6579.09
CS<LD,D,D>	6.38	11.83	27.25	203.24	202.01	374.12
CaS<LD,D,D>	6.92	51.20	49.03	1201.30	1558.39	4070.89
CC<LD,D,D>	16.14	29.23	186.11	360.62	1194.43	36348.55

Table 6.3: La tabella contiene i tempi in secondi delle esecuzioni di tutte le istanze degli algoritmi con grafo LD su tutti i grafi goto.

	goto8_8	goto8_32	goto10_8	goto10_64	goto12_8	goto14_8
GRB	1.16	2.60	8.87	46.88	55.40	284.95
NS<SD,L,L>	12.74	44.77	51.69	698.70	654.27	10466.81
CS<SD,L,L>	15.08	69.79	75.95	780.82	517.95	9295.29
CC<SD,L,L>	31.30	106.35	323.57	1371.48	4171.89	30688.73
CaS<SD,L,L>	15.40	80.51	116.43	3414.52	2795.11	2645.93
NS<SD,D,D>	5.38	12.06	12.49	324.48	461.66	11046.51
CS<SD,D,D>	7.4	18.92	52.13	356.28	370.16	10162.03
CC<SD,D,D>	11.14	43.8	330.79	1051.72	4583.12	34715.65
CaS<SD,D,D>	9.42	62.92	68.21	2735.7	2177.5	2870.29

Table 6.4: La tabella contiene i tempi in secondi delle esecuzioni di tutte le istanze degli algoritmi con grafo SD su tutti i grafi goto.

Chapter 7

Conclusioni

Nel corso della realizzazione di questa tesi, abbiamo esplorato il problema di interfacciare il framework SMS++ e la libreria LEMON, per permettere di farle cooperare fra di loro, con l'obiettivo di usare gli algoritmi forniti da LEMON su istanze di MCFBlock definiti da SMS++. I risultati raggiunti in questo lavoro hanno dimostrato che gli algoritmi di LEMON sono in grado di essere eseguiti partendo da una struttura di blocco definita da SMS++, in particolare superano tutti i test le istanze con tipi di costi e capacità long. L'esperienza di lavorare su questo progetto è stata profondamente arricchente. Ho acquisito competenze nel linguaggio C++, e ho imparato a gestire le difficoltà dovute alla scarsa documentazione di LEMON, spulciando dove potevo il loro codice cercando di comprenderlo al meglio.

7.0.1 Repository

Il repository creato durante questo progetto, disponibile sul gitlab di SMS++, contiene i file in cui ci sono le implementazioni di quanto descritto in questa tesi. Il progetto è distribuito secondo una licenza FOSS (Free and OpenSource software). Questo permette a futuri sviluppatori di contribuire apertamente a questo progetto.

7.0.2 Ringraziamenti

Ringrazio il mio relatore per avermi seguito molto accuratamente durante questo percorso e tutti coloro che mi hanno sostenuto.