Relazione Progetto Ragionamento Automatico

Definizione del problema:

Il problema presentato riguarda la costruzione di ponti in una laguna che contiene k piccole isolette di varie dimensioni e forme. Le dimensioni e la forma delle isolette non sono rilevanti. È nota la distanza minima in linea d’acqua tra ogni isoletta e le sue più vicine. L'obiettivo è costruire ponti tra le isolette in modo tale da garantire due condizioni:

1. Assicurare la completa connessione via ponte tra tutte le isolette.
2. Minimizzare i costi complessivi di costruzione.

I costi sono divisi in due categorie: costi fissi che dipendono da ogni singola isola, relativi alle infrastrutture necessarie, e costi variabili di 100k€ ogni 10 metri lineari di ponte. Inoltre, alcuni ponti potrebbero essere già esistenti tra alcune isolette, il che implica che non vi siano costi aggiuntivi per quei collegamenti.

Il problema richiede, quindi, la determinazione del modo più efficiente in termini di costi per connettere tutte le isolette della laguna rispettando queste condizioni. L’obiettivo è quello di confrontare gli approcci risolutivi implementati in Minizinc e in Clingo.

Modello Minizinc:

Il primo approccio risolutivo consiste in questa implementazione:

include "globals.mzn";

int : n; % number of nodes

array[1..n] of int: fixed\_costs;

array[1..n, 1..n] of var 0..1 : edges;

array[1..n, 1..n] of int: existing\_bridges;

array[1..n, 1..n] of int: distances;

predicate reach(int: n1, int: n2, set of int : visited) =

edges[n1, n2] == 1

\/

existing\_bridges[n1, n2] == 1

\/

(not (n1 in visited) /\

exists(j in 1..n)((edges[n1, j] = 1 \/

existing\_bridges[n1, j] == 1)

/\ reach(j, n2, visited union{n1})));

% Symmetry of bridges

constraint forall(i,j in 1..n where i < j)(

edges[i,j] = edges[j,i]

);

% All islands must be able to reach each other

constraint forall(i, j in 1..n where i < j)(

reach(i, j, {})

);

var int: cost = sum(i, j in 1..n where i < j /\ existing\_bridges[i, j] = 0) ((

distances[i,j]\*10

+ fixed\_costs[i]

+ fixed\_costs[j])\*edges[i,j]);

solve minimize cost;

output [

"Value of cost: ", show(cost), "\n",

"Value of edges: ", show(edges), "\n"

];

Descrizione del Codice

* Dichiarazione dei parametri:
  + n: Intero che rappresenta il numero di nodi (isole) presenti nella laguna.
  + fixed\_costs: Un array di interi di dimensione n che rappresenta i costi fissi associati a ogni isola per la costruzione delle infrastrutture dei ponti.
  + edges: Una matrice n×n di variabili di decisione (0 o 1) che rappresentano la presenza o assenza di un ponte tra due isolette. Se edges[i,j] = 1, significa che è presente un ponte tra l'isola i e l'isola j. Questa variabile è utilizzata per la ricerca delle soluzioni.
  + existing\_bridges: Una matrice n×n di interi che indica la presenza di ponti già esistenti tra le isole. Se existing\_bridges[i,j] = 1, c'è già un ponte tra i e j.
  + distances: Una matrice n×n di interi che rappresenta le distanze tra ogni coppia di isolette in metri.
* Predicato reach:

Definisce la nozione di raggiungibilità tra due nodi n1 e n2, verificando se esiste un percorso diretto o indiretto tra di essi. Questo predicato utilizza ricorsione e insiemi per tracciare i nodi visitati e verificare se una connessione è possibile attraverso ponti esistenti o ponti che si stanno pianificando.

* Vincoli del problema:
  + Simmetria dei ponti: Il vincolo forall(i, j in 1..n where i < j)( edges[i,j] = edges[j,i] ) garantisce che la matrice degli edges sia simmetrica, poiché un ponte tra due isolette è bidirezionale.
  + Connettività: Il vincolo forall(i, j in 1..n where i < j)( reach(i, j, {}) ) assicura che ogni coppia di isole sia raggiungibile, ovvero il grafo rappresentato dagli edges e existing\_bridges deve essere connesso.
* Funzione di costo:

cost: La variabile cost rappresenta il costo totale della costruzione dei ponti in migliaia di euro. Viene calcolata come la somma di:

* + - Distanza (in metri) × 10

+ costo fisso dell’isola di partenza

+ costo fisso dell’isola di arrivo

* + - Se esiste già un ponte tra due isole (existing\_bridges[from[i], to[i]] == 1), il costo è 0 perché il ponte non deve essere costruito.
* Obiettivo:

solve minimize cost: Il modello cerca di minimizzare la variabile cost, ovvero il costo totale per la costruzione dei ponti necessari a connettere tutte le isolette.

* Output:

Viene mostrato il valore finale del costo e la configurazione della matrice edges, che rappresenta quali ponti sono stati selezionati.

Viene riportato qui di seguito un secondo approccio alla risoluzione, che è risultato esponenzialmente più efficiente del primo, sfruttando vincoli predefiniti del linguaggio:

include "globals.mzn";

par int : n;

array[1..n] of int: fixed\_costs;

array[1..n, 1..n] of int: distances;

array[1..n, 1..n] of int: existing\_bridges;

array[1..n\*(n-1) div 2] of int: from = [i | i in 1..n, j in 1..n where i < j];

array[1..n\*(n-1) div 2] of int: to = [j | i in 1..n, j in 1..n where i < j];

array[1..n] of var bool : ns;

array[1..n\*(n-1) div 2] of var bool : es;

% Ensure all nodes are reachable from the root

constraint forall(r in 1..n)(

reachable(n, n\*(n-1) div 2, from, to, r, ns, es)

);

% Ensure all nodes are included

constraint forall(i in 1..n)(ns[i] = true);

var int : cost = sum(i in 1..n\*(n-1) div 2)(

if(existing\_bridges[from[i], to[i]] == 0) then

es[i]\*(distances[from[i], to[i]]\*10

+ fixed\_costs[from[i]] + fixed\_costs[to[i]])

else 0

endif);

solve minimize cost;

output [

"Value of es: ", show(es), "\n",

"Value of cost: ", show(cost), "\n",

];

Descrizione del codice:

* Dichiarazione dei parametri:

Oltre agli stessi parametri del precedente approccio sono stati definiti anche:

* + from: Array di interi che elenca il nodo di partenza di ogni possibile ponte in un ipotetico grafo completamente connesso (evitando duplicazioni simmetriche, es. il nodo 1 comparirà n-1 volte il nodo 2 comparirà n-2 volte ecc.).
  + to: Array di interi che elenca il nodo di destinazione corrispondente di ogni possibile ponte (rispetto a ogni posizione dell’array from).

Queste due strutture rappresentano tutte le coppie di nodi di un grafo completamente connesso, in cui vengono cercate le soluzioni.

* Definizione delle variabili di decisione:
  + ns: Array di variabili booleane che rappresenta se un'isola specifica è inclusa nel grafo risultante. In questo contesto, ns[i] è true se l'isola i è inclusa.
  + es: Array di variabili booleane che indica se un ponte tra la coppia from[i] e to[i] è stato selezionato (es[i] = true significa che il ponte è presente).
* Vincolo di raggiungibilità:
  + constraint forall(r in 1..n)( reachable(n, n×(n-1) div 2, from, to, r, ns, es) ): Questo vincolo assicura che ogni isola r possa essere raggiunta dalle altre attraverso i ponti selezionati. Si utilizza il vincolo predefinito “reachable” per verificare la connettività del grafo risultante, che utilizza tutti gli array elencati sopra e due parametri numerici che indicano il numero di nodi del grafo dove cercare le soluzioni(n) e il numero degli archi di questo(n×(n-1)), r è il nodo root di partenza ed è variabile.
* Vincolo di inclusione delle isole:
  + constraint forall(i in 1..n)(ns[i] = true): Questo vincolo garantisce che tutte le isolette debbano essere incluse nella soluzione finale. Quindi, ns[i] è sempre true.
* Funzione di costo:
  + cost: La variabile cost rappresenta il costo totale della costruzione dei ponti in migliaia di euro. Il costo è calcolato come la somma dei costi per ogni ponte selezionato (es[i] = 1) e viene calcolato in base alla formula:
    - Distanza (in metri) × 10

+ costo fisso dell’isola di partenza

+ costo fisso dell’isola di arrivo

* + - Se esiste già un ponte tra due isole (existing\_bridges[from[i], to[i]] == 1), il costo è 0 perché il ponte non deve essere costruito.
* Obiettivo:
  + solve minimize cost: Il modello cerca di minimizzare il costo totale necessario per connettere tutte le isole
* Output:
  + es: la configurazione dei ponti selezionati
  + cost: il costo totale della costruzione

Questo approccio è risultato empiricamente più efficiente del precedente, ciò è imputabile all’utilizzo di un vincolo predefinito di Minizinc (reachability).

Inoltre, la definizione di questa relazione è molto simile a diversi altri vincoli predefiniti del linguaggio utili alla risoluzione del problema e dunque si presta naturalmente ad un confronto con questi. Per questa comparazione sono stati usati i vincoli “steiner” (steiner(n, e, from, to, w, ns, es, k)) e “weighted\_spanning\_tree” (weighted\_spanning\_tree(n, e, from, w, es, k)), in cui non c’è un esplicito nodo root r da cui parte la ricerca, w è un array di interi che rappresenta i pesi per ogni arco e k è il peso complessivo dell’albero da minimizzare.

Modello Clingo:

% Ensure that fixed edges are always selected

selected(X, Y) :- fixed\_edge(X, Y).

% Choose edges to include in the spanning tree

{selected(X, Y) : edge(X, Y, C)}.

selected(Y, X) :- selected(X, Y).

% Define reachability

reachable(X, Y) :- selected(X, Y).

reachable(X, Z) :- reachable(X, Y), reachable(Y, Z).

% All nodes must be reachable from any other node (connected graph)

:- node(X), node(Y), X < Y, not reachable(X, Y).

% Minimize the sum of the weights of the selected edges

#minimize { C \* 10 + C1 + C2, X, Y : selected(X, Y), fixed\_cost(X, C1), fixed\_cost(Y, C2), edge(X, Y, C), not fixed\_edge(X, Y)}.

#show selected/2.

Descrizione del codice:

* La prima regola assicura che, se un arco (X, Y) è predefinito come "fixed\_edge", esso venga automaticamente incluso nell'insieme degli archi selezionati.
* La seconda regola utilizza la sintassi della regola di scelta di Clingo, che permette al solver di scegliere non deterministicamente quali archi includere nell'insieme selected tra gli archi disponibili edge(X, Y, C). Il predicato edge(X, Y, C) rappresenta tutti gli archi possibili (X, Y) insieme ai loro rispettivi pesi C.
* La terza regola garantisce che la selezione degli archi sia simmetrica. Se l'arco (X, Y) è selezionato, allora anche l'arco (Y, X) è considerato selezionato. Ciò è necessario perché il grafo è non orientato, il che significa che la connessione tra nodi è bidirezionale.
* La quarta regola, che definisce se due nodi sono “reachable” tra loro definisce in modo ricorsivo la relazione di raggiungibilità.
* Il quinto vincolo assicura che il grafo sia completamente connesso. In particolare, verifica che per ogni coppia di nodi X e Y (dove X < Y per evitare ridondanze), ci deve essere un percorso raggiungibile da X a Y.
* L'istruzione #minimize, infine, indica a Clingo di trovare una soluzione che minimizzi l'espressione specificata. L'espressione che viene minimizzata è C \* 10 + C1 + C2, dove:
  + C è il peso dell'arco (X, Y), cioè la distanza tra le isole in metri.
  + C1 è il costo fisso associato al nodo X.
  + C2 è il costo fisso associato al nodo Y.

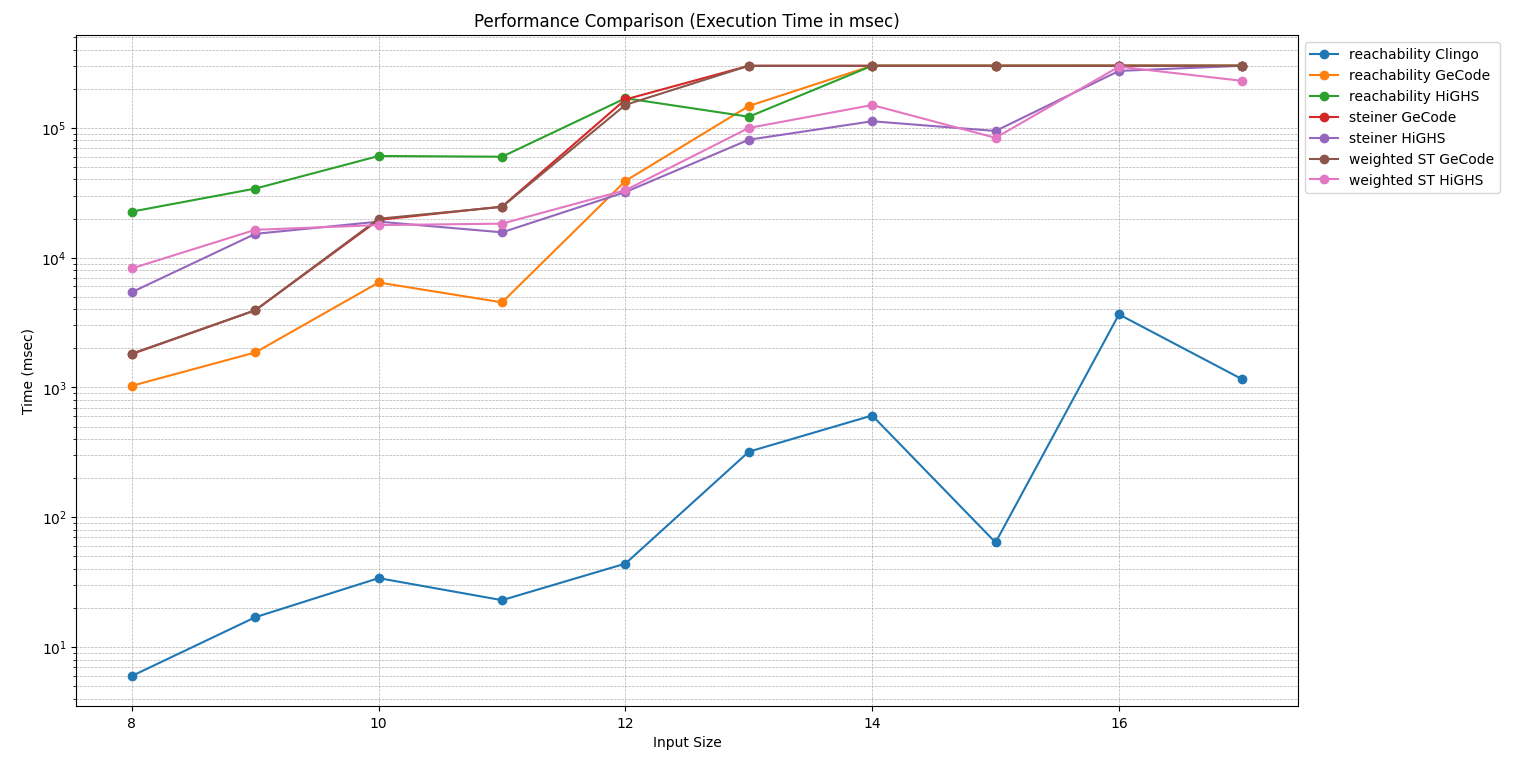
Il processo di minimizzazione considera solo gli archi che sono in selected(X, Y) e non in fixed\_edge(X, Y) per evitare connessioni obbligatorie. Il costo complessivo calcolato è in migliaia di euro.

Risultati:

L’obiettivo del benchmark è valutare le prestazioni di Minzinc (con diversi solver) e Clingo nell’affrontare problemi di ottimizzazione soddisfacendo determinati vincoli. I solver considerati per MiniZinc sono GeCode e HiGHS. Ogni configurazione è stata testata su problemi di dimensioni crescenti (da 8 a 17 nodi) con diversi vincoli per Minizinc (rechable, steiner e weighted\_spanning\_tree) e un’unica definizione del problema per Clingo. Per la generazione dei casi di test sono stati utilizzati gli script python instance\_generator.py e times\_output\_prettify.py e per il test dei risultati computati è stato utilizzato il file checker.py. Tutti i file menzionati sono presenti all’interno del progetto, nelle relative sottocartelle. Vengono di seguito riportati i risultati per le varie istanze, in millisecondi:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Dimensione | reachability GeCode | steiner GeCode | weighted ST GeCode | reachability HiGHS | steiner HiGHS | weighted ST HiGHS | reachability Clingo |
| 8 | 1030 | 1810 | 1829 | 22610 | 5420 | 8260 | 6 |
| 9 | 1860 | 3930 | 3930 | 34000 | 15240 | 16370 | 17 |
| 10 | 6430 | 19539 | 19910 | 60600 | 18920 | 17800 | 34 |
| 11 | 4520 | 24690 | 24590 | 59830 | 15650 | 18250 | 23 |
| 12 | 38960 | 165140 | 150090 | 168720 | 31960 | 33050 | 44 |
| 13 | 147070 | TIMEOUT | TIMEOUT | 121520 | 80870 | 99630 | 320 |
| 14 | TIMEOUT | TIMEOUT | TIMEOUT | TIMEOUT | 112190 | 149590 | 608 |
| 15 | TIMEOUT | TIMEOUT | TIMEOUT | TIMEOUT | 94440 | 83620 | 64 |
| 16 | TIMEOUT | TIMEOUT | TIMEOUT | TIMEOUT | 274020 | 295310 | 3651 |
| 17 | TIMEOUT | TIMEOUT | TIMEOUT | TIMEOUT | TIMEOUT | 229630 | 1155 |

Qui di seguito viene riportata una visualizzazione dei dati menzionati sopra:



Analisi dei Risultati:

MiniZinc con GeCode:

* Prestazioni: Le prestazioni iniziali sono abbastanza buone per dimensioni del problema fino a 12 nodi, con tempi di esecuzione inferiori a 17 secondi.
* Punto critico: Dopo 13 nodi, si nota un incremento esponenziale del tempo di esecuzione, raggiungendo il limite di 5 minuti (300000 ms) nei problemi di dimensione maggiore (da 14 in poi). Questo suggerisce che GeCode soffre di un aumento drammatico della complessità quando la dimensione del problema cresce.
* Comportamento simile tra i tre problemi: Per tutte le istanze ("reachability", "steiner", "weighted\_ST"), il comportamento è quasi identico, con solo leggere variazioni nei tempi. Questo indica che il solver gestisce queste varianti di problemi in modo analogo.

MiniZinc con HiGHS:

* Prestazioni: HiGHS ha tempi più lunghi rispetto a GeCode fin dalle prime dimensioni del problema. Ad esempio, per il problema di dimensione 8, HiGHS impiega già più di 20 secondi per "reachability".
* Punto critico: HiGHS raggiunge anch'esso il limite di 5 minuti (300000 ms) per problemi più grandi, ma ha un andamento meno stabile, con improvvisi incrementi e decrementi nei tempi di esecuzione, rivelandosi in diversi casi più efficiente di GeCode.
* Variabilità nei problemi: C’è maggiore differenza tra "reachability", "steiner", e "weighted\_ST". Ad esempio, "weighted\_ST" sembra gestito meglio da HiGHS rispetto a "reachability" nei problemi più grandi, suggerendo un comportamento meno uniforme rispetto a GeCode.

Clingo:

Prestazioni: Clingo mostra prestazioni nettamente superiori rispetto a MiniZinc sia con GeCode che con HiGHS. Anche per i problemi di dimensioni maggiori, Clingo gestisce il problema "reachability" in meno di 4 secondi.

Conclusioni:

GeCode è generalmente più efficiente di HiGHS per problemi di dimensioni più piccole, ma entrambi i solver tendono a soffrire di una crescita esponenziale del tempo di esecuzione con l'aumento della dimensione del problema.

Clingo invece mostra un livello di efficienza e scalabilità notevolmente superiore, gestendo le istanze di grandi dimensioni con tempi di esecuzione significativamente inferiori. MiniZinc (sia con GeCode che con HiGHS) non è scalabile oltre una certa soglia, con tempi di esecuzione che aumentano drasticamente. Clingo, invece, mostra un comportamento più lineare e prevedibile.

In sintesi, se il problema non supera determinate dimensioni, MiniZinc è una buona scelta, specialmente con GeCode, mostrando anche un andamento prevedibile rispetto alla crescita dell’input. Tuttavia, per problemi di raggiungibilità e ottimizzazione di grafo, Clingo è nettamente la soluzione più efficiente e scalabile.