Progetto di Advanced Scheduling Systems

Fornasiere Raffaello

13 giugno 2020

Indice

Capitolo 1

Problema

1.1 Descrizione del problema

Il progetto consiste nella creazione di un programma che ottimizzi il ritardo di propagazione di un circuito digitale combinatorio formato da *celle* interconnesse tra loro. Ognuna di queste introduce un determinato ritardo al segnale che la attraversa il quale dipende dalle celle a cui è collegata. Il tempo di propagazione del circuito, quindi, è dato dalla somma di tutti questi ritardi. Poiché il circuito ha diversi input e output, esistono più percorsi di segnale e per caratterizzare la temporizzazione di un circuito è necessario calcolare il ritardo accumulato attraverso ogni percorso.

Il programma deve essere in grado di analizzare il circuito e selezionare le celle che lo comporranno, con lo scopo di sceglierne un insieme tra quelle fornite al fine di ottimizzare il ritardo peggiore, ovvero il percorso per cui il segnale impiega più tempo per attraversare il circuito. Un'ulteriore richiesta potrebbe essere quella di minimizzare l'area occupata dall'intero circuito. La complessità di implementazione aggiunta da questa richiesta è minima quindi il progetto è stato sviluppato ponendo più attenzione sul primo problema. Tutti i dati relativi a celle e circuito vengono forniti attraverso dei file con una struttura ben definita.

1.1.1 Calcolo del ritardo

Il ritardo di ogni cella dipende da 3 parametri: durata della transizione in ingresso, area e capacità che questa "vede" in uscita. In realtà, l'area non rientra esplicitamente nel calcolo poiché è strettamente correlata con la capacità dei *pin*[1] di uscita della cella stessa. Una tabella fornisce il tempo di propagazione in funzione di diverse coppie di questi due parametri. Nel caso in cui per una data coppia di valori in ingresso non corrisponde nessuna entrata nella tabella, viene eseguita un'interpolazione in funzione di altre entrate della tabella.

Il calcolo del ritardo del circuito consiste nel ricavare progressivamente il tempo di propagazione di ogni cella a partire dalle quelle che costituiscono gli input del circuito. É doveroso procedere in questo modo poiché il ritardo di ogni cella dipende dalla transizione in uscita di

[1] Contatto elettrico di una cella

quella precedente, che deve essere stata precedentemente calcolata. Questo sistema funziona per circuiti combinatori, che non presentano quindi percorsi di segnale ciclici.

1.1.2 Ipotesi e semplificazioni

Al fine di rendere la struttura del programma più semplice, sono state introdotte alcune ipotesi semplificative:

- Ogni ingresso condiziona ogni uscita di una specifica cella. L'ipotesi è giustificata dal fatto che nelle istanze assegnate la maggior parte delle celle sono porte logiche o circuiti combinatori base come *Half Adder* o *Full Adder*. Nel caso in cui fosse necessario introdurre porte logiche più complesse e rimuovere questa ipotesi diventerebbe necessaria la conoscenza di ulteriori informazioni, ad esempio sulla funzione implementata dalla cella.
- Il circuito può essere formato da sole celle e non da altri circuiti. Questa ipotesi è stata introdotta per semplificare il problema. Ciononostante il programma creato sarebbe in grado di gestire il problema anche senza questa ipotesi se il circuito che si vuole inserire venisse descritto come una cella (non sono posti limiti al numero di ingressi e/o uscite che queste possono avere).
- Gli ingressi del circuito hanno una transizione nulla e le uscite hanno una capacità nulla.

Come si è visto nessuna di queste pone dei vincoli ad sviluppi futuri.

Capitolo 2

Algoritmo genetico

L'idea sulla quale si basa un algoritmo genetico consiste nel far evolvere N soluzioni, ricombinandole tra loro, al fine di trovarne una che sia frutto di molteplici combinazioni ottenute attraverso diverse generazioni. Esistono molti tipi di algoritmi genetici e evolutivi (i primi sono un sottoinsieme dei secondi). Per risolvere il problema si è dovuto scegliere quale tra quelli esistenti sarebbe potuto essere adatto al problema ma sopratutto applicabile. L'applicabilità è condizionata dal costo computazionale totale della soluzione: il tempo di esecuzione del programma deve essere tale da fornire i risultati in tempi utili. Ad esempio gli algoritmi di ottimizzazione multi-oggetto (che si adatterebbero bene a questo problema al fine di minimizzare sia i ritardi che l'area) hanno complessità che spesso risulta essere più che quadratica rispetto N individui della popolazione[1], mentre quelli a singola funzione oggetto hanno complessità molto inferiori (lineare rispetto al numero di individui). Nel problema in esame la valutazione di ogni individuo della popolazione richiede un costo pari a $O(KLog(K))$, con K numero di celle nel circuito. Per evitare tempi di elaborazione troppo grandi si è scelto di utilizzare un algoritmo genetico a singola funzione oggetto che presenti una complessità poco più che lineare in funzione del numero di individui nella popolazione.

2.1 BRKGA

I Biased Random-Key Genetic Algorithm hanno spesso dimostrato di essere più efficienti rispetto ad altri algoritmi dello stesso tipo [Gonçalves and Resende(2011)].

Come in altri algoritmi genetici, ogni individuo è rappresentato da un DNA formato da un insieme di cromosomi[2]. Nel algoritmo in questione ogni cromosoma corrisponde ad un numero in virgola mobile compreso tra 0 e 1 ed il DNA è semplicemente un vettore di questi valori. All'inizio del processo evolutivo viene creato certo numero N di DNA, che corrisponde alla popolazione della generazione zero. Ogni DNA viene quindi associato ad una soluzione del problema della quale verrà calcolato valore di fitness che nel caso in esame è correlato al tempo

[1] Molto spesso questa complessità è data dal fatto che questi algoritmi utilizzano un ordinamento chiamato *non-dominated sorting* che richiede una complessità più che quadratica per la sua risoluzione.

[2] Si farà riferimento ai singoli elementi del vettore anche tramite altri termini biologici quali allele, gene, genotipo...

di propagazione peggiore del circuito. Gli individui vengono quindi classificati in due categorie: *elite* e *non elite*. A questo punto si procede con la fase di *crossover* ovvero la riproduzione degli elementi della popolazione al fine di creare una nuova generazione [3]. Questa fase è rappresentata in Figura 2.1. La popolazione *elite* (di dimensione p_e) viene copiata senza variazioni nella nuova

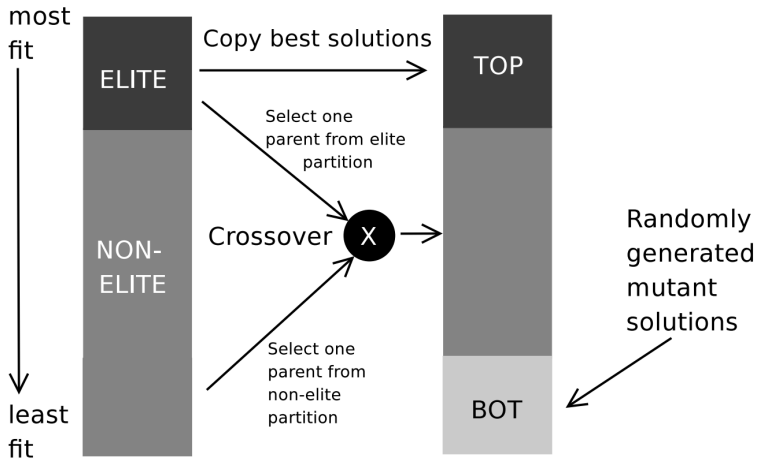

Figura 2.1:

generazione, mentre il resto della popolazione viene formato da p_m *mutanti* e $N - p_m - p_e$ *figli* della attuale generazione. Come si vede in figura, questi ultimi sono generati dalla combinazione di un elemento *elite* ed un altro preso dal resto della popolazione. La combinazione di queste due soluzioni avviene prendendo una certa percentuale di cromosomi ρ_e della soluzione elite, e la rimanente $1 - \rho_e$ dalla quella non elite. I mutanti invece vengono creati in maniera completamente casuale. Il processo quindi riparte dalla decodifica della popolazione e si ripete fino a che non viene raggiunto un certo valore di fitness o, più in generale, fino a quando non è soddisfatto lo *stop criteria*. In Figura 2.2 viene proposto la raffigurazione dell'esecuzione dell'algoritmo.

[3] Il numero di elementi tra una generazione e quella successiva non cambia e neanche il numero di elementi *elite* e *mutanti*.

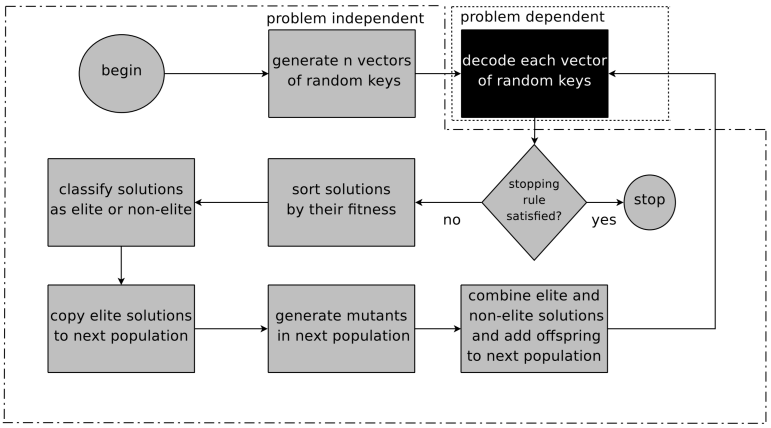

Figura 2.2:

2.2 Settaggio dei parametri

Dalla sezione precedente emerge che l'algoritmo genetico assegna all'utente la responsabilità di fissare i parametri N, p_e, p_m e ρ_e. Per il problema in esame ci si è affidati ai valori consigliati [Gonçalves and Resende(2011)]:

$$0.1N < \quad p_e \quad < 0.25N \tag{2.1}$$

$$0.1N < \quad p_m \quad < 0.3N \tag{2.2}$$

$$0.5 < \quad \rho_e \quad < 0.8 \tag{2.3}$$

dove N rappresenta il numero di individui nella popolazione. Benché spesso questi valori portino sempre a buoni risultati, la scelta non è semplice e dipende molto dall'applicazione e dal circuito in analisi.

Capitolo 3

Impostazione del problema e idea risolutiva

Il problema è stato affrontato dividendolo in due parti: una *problem-dependent* e una *problem-independent*. La prima si occupa di:

- leggere i file che descrivono celle e circuito
- organizzare questi in una struttura dati adatta al tipo di problema
- fornire delle funzioni per il calcolo dei delay delle celle e del circuito

La seconda parte, invece, risolve un problema di ottimizzazione combinatorio generico che verrà poi specializzata per risolvere il caso in esame.

3.1 Grafo del circuito

Inizialmente era stato deciso di rappresentare il circuito come un grafo diretto aciclico, dove ogni cella corrispondeva un nodo del grafo. Questa scelta però si è mostrata inadeguata poiché attraverso ognuna di queste ci sono più percorsi che possono portare a risultati diversi. Infatti poiché ogni pin ha una specifica capacità un collegamento su due pin di una stessa cella può causare ritardi differenti. Lo stesso ragionamento può essere fatto con i pin in uscita. Si è deciso

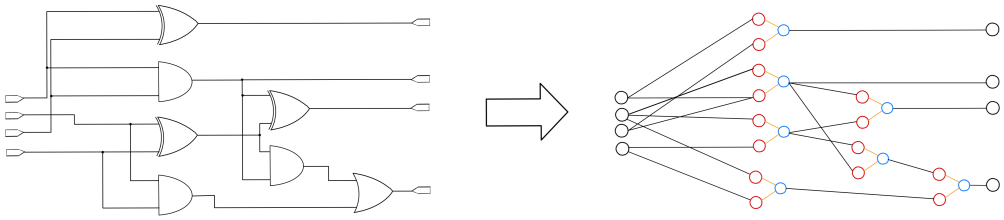

Figura 3.1:

quindi di rappresentare il circuito attraverso un DAG dove ogni nodo corrisponde semplicemente ad un ingresso o ad un uscita di una cella o, più in generale, ad un pin. In figura 3.1 vengono mostrare le due rappresentazioni del circuito: a destra la rappresentazione "fisica" e a sinistra il grafo associato. In quest'ultimo i nodi di colore azzurro rappresentano i pin di uscita, mentre quelli rossi i pin di ingresso. Le interconnessioni arancioni sono quelle che permettono ad una cella di essere interconnessa nella rappresentazione a grafo. Il modo in cui queste vengono create si basa sull'ipotesi che ogni ingresso è collegato ad ogni uscita (in figura la verifica dell'ipotesi è banale poiché ogni cella ha solo un uscita, ma nei circuiti assegnati ci sono i casi in cui è possibile avere anche più uscite).

3.2 Ordinamento topologico

Come introdotto durante la spiegazione del problema, per calcolare il ritardo complessivo è necessario attraversare ordinatamente i nodi del grafo associato al circuito e per ogni nodo aggiornare i valori di quello successivo indicando il valore di transizione che vedrà in ingresso. Poiché ogni nodo può avere più archi entranti che forniscono ritardi differenti, verrà memorizzato quello peggiore. In questo modo alla fine del processo iterativo si potrà ricavare il tempo di propagazione peggiore all'interno del circuito.

Per fare ciò, è necessario avere una struttura che ordini i nodi in modo tale che il ritardo venga calcolato correttamente. Un algoritmo di ordinamento topologico salva i nodi ordinatamente all'interno di un vettore. Questo algoritmo sfrutta uno heap binario che esegue l'ordinamento in funzione del numero di archi entranti per ogni nodo. La *chiave* dello heap è rappresentata quindi dal numero di archi entranti "non risolti", ovvero gli archi che provengono da nodi non ancora visitati. Una volta creato lo heap, viene iterativamente estratto il nodo in cima, lo si inserisce nell'array ordinato e si decrementa la chiave di tutti i nodi ai quali arrivavano i suoi archi in uscita. Così facendo, se il grafo è aciclico, ogni volta che viene estratto un nodo questo avrà una chiave con valore zero, che equivale a dire che tutti i nodi prima di esso sono già stati posizionati nell'arra ordinato.

3.3 Calcolo percorso peggiore

Una fase di aggiornamento delle capacità di uscita precede l'operazione di calcolo del ritardo. Questa non necessita un attraversamento ordinato delle celle del circuito. Per calcolare il ritardo, invece, si sfrutta la struttura ordinata ottenuta precedentemente. Una volta creata una tabella di hash che associa ogni nodo al ritardo peggiore fino a quel nodo, si procede iterativamente secondo i seguenti passi:

- Selezione dell'elemento i-esimo nel vettore
- Calcolo del ritardo per ogni nodo al quale è collegato (in uscita): quando un nodo presenta più archi in uscita significa che si sta trattando un pin di uscita di una cella. Questo vuol dire che il ritardo sarà uguale per tutti i nodi che ha in uscita, di conseguenza basta calcolarlo una sola volta. L'iterazione attraverso tutti i noti successivi è però necessaria

poiché a ognuno di essi deve eventualmente registrare la nuova transizione in ingresso. Quindi l'algoritmo attraversa tutti i nodi in uscita, ma solo per il primo di questi calcolerà effettivamente il delay. Il valore viene salvato e restituito poi come risultato anche per le altre iterazioni.

- Si somma il valore di distanza ottenuto rispetto al noto j-esimo e lo si paragona con il valore memorizzato rispetto al valore in tabella. Se quello calcolato è maggiore, l'entrata in tabella viene aggiornata al nuovo valore.

Una volta che sono stati visitati tutti gli elementi del vettore è possibile cercare all'interno della tabella il valore più alto. Questo corrisponderà al ritardo peggiore del circuito.

Capitolo 4

Dettagli implementativi

4.1 BRKGA

La `class BRKGA` rappresenta un semplice Framework O-O per la l'esecuzione dell'algoritmo genetico descritto. Di seguito è presente la definizione della classe. In relazione all'algoritmo descritto in Figura 2.2, la classe implementa i seguenti metodi:

```
template<class Input, class Output, class FType>
class BRKGA
{
public:
    BRKGA(const Input& input):input(input), output(input){ bias = -1;}
    BRKGA(const Input& input, const BRKGAParams &p);
    BRKGA(const Input& input, size_t popSize, size_t nAlleles, size_t pe,
        size_t pm, double rho_e);
    virtual ~BRKGA(){}

    // ******************** User Interaction Members ******************/
    bool Evolve();
    virtual FType BestFitness() const {return fitVect.front().first;}
    void SimpleTest();  // performs only one cycle of evolution
    void SetParams(const BRKGAParams &p);
    void Reset();
    size_t PopSize() const {return p.popSize;}
    // ************************************************************

    double getBias() const;
    void setBias(double value);

protected:
    // ************************** Hotspots  ***************************
    virtual void _Reset() = 0; //called by Reset(). Reset derived class data
    virtual void Decode() = 0;
    virtual bool StopCriteria() = 0;
    // ****************************************************************

    void InitializePopulation();
```

```
30      void Classify();
31
32      // ********** Algorithm data **************
33      std::vector<DNA> hypercube;
34      std::vector<std::pair<FType, size_t>> fitVect;
35
36      BRKGAParams p;
37      size_t generations;
38      double bias;
39      // ****************************************
40
41      const Input& input;
42      Output output;
43  };
```

- `InitializePopulation()`: genera N vettori contenenti k elementi causali, ovvero inizializza N `DNA` con cromosomi random.
- `Decode()`: la funzione deve decodificare ogni elemento della popolazione e salvare il valore di fitness nel primo elemento di ogni entrata del vettore `fitVect`. Quest'ultimo verrà poi ordinato in funzione dei valori salvati.
- `StopCriteria()`: una volta implementata dovrà restituire il valore `true` se l'algoritmo deve fermarsi, `false` altrimenti.
- `Classify()`: ordina il vettore in funzione del fitness decrescente. Per ridurre il costo computazionale si esegue un ordinamento parziale: l'ordinamento di ferma una volta ordinati i primi p_e elementi, in questo modo si avrà la classificazione in *elite* e *non elite*.

L'algoritmo non esegue mai una copia degli individui da una popolazione a quella successiva ma lavora sempre su quella corrente per ridurre il costo computazionale.

4.2 Graph

La classe `Graph`, viene utilizzata come interfaccia per eseguire gli algoritmi di ordinamento topologico e calcolo del percorso peggiore.

```
1   template <class M>
2   class GraphNode
3   {
4       friend bool operator==(const GraphNode& a, const GraphNode& b) { return
            a.item == b.item; }
5       template<class N>
6       friend std::ostream& operator<<(std::ostream& os, const GraphNode<N>&
            gn);
7
8       friend class Graph<M>;
9
10  public:
```

```
11      GraphNode(std::string name = ""):name(name){}
12      virtual ~GraphNode(){}
13      virtual M Distance(GraphNode* a) = 0;
14      std::string getName() const {return name;}
15      virtual void AddNeighbor(GraphNode* a){adj.push_back(a);}
16
17  protected:
18      std::string name;
19      std::vector<GraphNode*> adj;
20
21  private:
22      BinHeapNode<GraphNode*, M>* heapPtr;
23  };
24
25  template <class M>
26  class Graph
27  {
28      template<class N>
29      friend std::ostream& operator<<(std::ostream& os, const Graph<N>& g);
30
31      public:
32      Graph(){}
33      virtual ~Graph(){}
34
35      // ********************************************
36      void TopologicalSort();
37      M getWorstPathDistance();
38
39      virtual void DeleteNode(int i);
40
41      GraphNode<M>* getNode(int node);
42      // ********************************************
43
44      protected:
45      std::vector<GraphNode<M>*> adjList;
46      std::vector<GraphNode<M>*> sorted;
47  };
```

La funzione `TopologicalSort()` agisce come descritto nel paragrafo 3.2: inserisce tutti i nodi in un *binary heap* assegnando come chiave il numero di archi entranti. estrae gli elementi e li salva in un vettore. Inoltre la funzione controlla che il grafo sia aciclico.

Anche la funzione `getWorstPathDistance()` si comporta come già descritto sopra: chiama per ogni nodo la funzione `Distance(GraphNode* a)` che restituisce la distanza rispetto al nodo passato come argomento. Alla fine dell'esecuzione avrà salvato in una tabella (`std::unordered_map`) le distanze peggiori per ogni nodo del grafo. Restituisce quindi il ritardo peggiore.

4.3 Binary Heap

Come appena visto la classe `Graph` fa uso di uno heap binario per ordinare gli elementi. Poiché la *Standard Library* del C++ non fornisce le funzioni `DecreaseKey` e `IncreaseKey`, si è deciso di implementare una struttura apposita:

```
template <class T, class K>
class BinHeapNode
{
    public:
    BinHeapNode(T item = T(), K key = K()):item(item), key(key) {}
    ~BinHeapNode(){}
    K getKey() const {return key;}
    void setKey(const K &value) {key = value;}

    T getItem() const {return item;}
    void setItem(const T &value) {item = value;}

    private:
    friend class BinHeap<T,K>;
    T item;
    K key;
};


template <class T, class K>
class BinHeap
{
    public:
    BinHeap(int reserve = 12, const std::function<bool(K, K)>& compare =
        [](K k1, K k2)
    {return k1 > k2;});

    std::pair<T,K> Pop();
    bool DecreaseKey(BinHeapNode<T,K>* n, K newKey);
    bool IncreaseKey(BinHeapNode<T,K>* n, K newKey);
    bool Empty() const {return vect.size() == 0;}
    bool Contains(BinHeapNode<T,K>* n);
    bool DeleteElement(BinHeapNode<T,K>* n);

    std::pair<T,K> Front();
    BinHeapNode<T,K>* Push(T item, K key);

    bool CheckCollisions();

    void SetCompare(const std::function<bool(K, K)> &value);

    inline static const std::function<bool(K, K)> minHeap = [](K k1, K
        k2){return k1 > k2;};
    inline static const std::function<bool(K, K)> maxHeap = [](K k1, K
        k2){return k1 < k2;};
    std::vector<BinHeapNode<T,K>*> vect;
    private:
```

```
45      int parent(int i) { return (i-1)/2; }
46
47      int left(int i) { return (2*i + 1); }
48
49      int right(int i) { return (2*i + 2); }
50      void Heapify(size_t i);
51
52      bool Comparator();
53      std::function<bool(K, K)> compare;
54
55      std::unordered_map<BinHeapNode<T,K>*, int> pos;
56  };
```

La classe `BinHeap`, rappresenta l'implementazione di uno heap binario. Non presenta caratteristiche particolari e i costi delle funzioni sono quelli classici degli heap binari riassunti nella seguente tabella:

Funzione	Costo
Inserimento	$O(log(n))$
Estrazione minimo	$O(log(n))$
Decremento chiave	$O(log(n))$

4.4 Classe Cell

Ogni cella del circuito viene rappresentata attraverso la classe `Cell` che ne contiene tutte le informazioni: nome, tipo, dati di temporizzazione, area, pin di ingresso e uscita ed un id che la identifica univocamente.

```
1   class Cell
2   {
3       [...]
4   public:
5       Cell(const std::string& cellName = "") : name(cellName){}
6
7       // **********SETTERS AND GETTERS*************
8       [...]
9       // ****************************************
10
11      // special copy
12      void CopyParams(const Cell* c);
13
14      // ******************GETTERS OF CELL
            PARAMETERS*****************************
15      double GetInPinCapacity(int i) const {return input[i].capacity;}
16      double GetOutPinCapacity(int i) const {return  output[i].capacity;}
17      double GetTimingInfo(double intransit, double outCap, size_t output,
            size_t type) const;
18      size_t GetNumOfInputs() const {return input.size();}
19      size_t GetNumOfOutputs() const {return output.size();}
```

```
20      std::string GetInputName(int i) const {return input[i].name;}
21      std::string GetOutputName(int i) const {return output[i].name;}
22      double getArea() const {return area;}
23      //
            **********************************************************************
24
25      // ********************CONST PUBLIC
            ATTRIBUTES*****************************
26      static const size_t cell_fall = 0;
27      static const size_t cell_rise = 1;
28      static const size_t fall_transition = 2;
29      static const size_t rise_transition = 3;
30      //
            **********************************************************************
31
32      void setId(const std::string &value);
33      std::string getId() const;
34      void TestTimingInfo(){timingInfo[0][0].Test();}
35
36  private:
37      std::string name;
38      std::string type;
39      std::string id;
40      double area;
41      std::vector<std::vector<CellTimingInfo>> timingInfo;
42      std::vector<Pin> input;
43      std::vector<Pin> output;
44  };
```

Un oggetto `Pin` descrive un pin di una cella memorizzandone capacità, direzione (input o output) e il nome. Quest'ultimo serve per costruire il circuito: un collegamento tra due celle viene rilevato cercando se queste hanno pin con lo stesso nome.

`CellTimingInfo` è un contenitore di tutti i dati di temporizzazione di un uscita della cella. Tramite la funzione `double GetDelay (double inputTransition, double outCapacitance)` è possibile ricevere il valore di ritardo che corrisponde al valore di transizione in ingresso e alla capacità in output passati come argomento.

4.5 Classe Circuit

Questa classe costituisce la struttura per descrivere il circuito nonché la classe di input dell'algoritmo genetico.

```
1   class Circuit
2   {
3       [...]
4   public:
5       Circuit(const std::vector<std::vector<Cell>> *selection)
6       : cellSelection(selection)  {}
7
8       void AssignRandom();
9       void AssignAll(double p) const;
```

```
10
11      size_t GetNumOfCells() const {return adjList.size();}
12      const Cell* GetCell(int i) const {return inputLists[i].first;}
13
14      std::vector<const Cell*> GetInputsOfCell(int cell) const;
15      std::vector<const Cell*> GetInputsOfCell(const Cell* cell) const;
16
17      std::vector<const Cell*> GetOutputsOfCell(int cell) const;
18      std::vector<const Cell*> GetOutputsOfCell(const Cell* c) const;
19
20      void ChangeCell(const Cell *c) const;
21      bool ChangeCell(const Cell *c, double p) const;
22      void ChangeCell(size_t i) const;
23      bool ChangeCell(size_t i, double p) const;
24
25      double GetAreaOccupation() const;
26
27  private:
28      mutable std::list<std::pair<Cell, std::vector<Cell*>>> adjList;
29      std::vector<std::pair<Cell*, std::vector<Cell*>>> inputLists;
30
31      const std::vector<std::vector<Cell>>* cellSelection;
32
33      bool readInstruction1(std::string line);
34      bool readInstruction2(std::string line);
35      bool readInstruction3(std::string line);
36
37      std::list<std::pair<Cell, std::vector<Cell*>>>::iterator
            searchOutputSingal(const std::string &name);
38      std::vector<std::vector<Cell>>::const_iterator searchCellType(const
            std::string &name);
39
40  };
```

Oltre agli standard *setters* e *getters* fornisce le seguenti funzioni

- `AssignRandom()`
- `AssignAll()`
- `ChangeCell(...)`

Le prime due sono funzioni che servono per il debug. La terza invece da la possibilità di cambiare una cella in una dello stesso tipo e l'argomento `double p` indica quale cella. Questo valore è compreso tra 0 e 1 e viene mappato in un valore intero che indicherà una posizione all'interno del vettore che contiene le celle dello stesso tipo.

4.6 Classe CircuitGraph

Rappresenta la classe di output dell'algoritmo genetico e gestisce il grafo associato al circuito. Deriva dalla classe `Graph`. Questa classe lavora assieme alla `CircuitNode` che

deriva da `GraphNode`. `CircuitNode` specializza il `Distance()` usato dalla funzione `getWorstPathDistance()` e fornisce altri metodi per calcolare i vari ritardi e capacità associate ad ogni nodo. Le classi `InputCircuitNode` e `OutputCircuitNode` derivano da questa e specializzano i metodi che servono per calcolare delay e capacità.

```
class CircuitNode : public GraphNode<double>
{
    [...]
public:
    CircuitNode(CircuitGraph* it, double capacity, const std::string& name =
        "")
    : GraphNode<double>(name), it(it), outCapacity(capacity)
    {worstInRTransit = 0; worstOutRTransit = 0;}

    CircuitNode(CircuitGraph* it, const std::string& name)
    : GraphNode<double>(name), it(it)
    {worstInRTransit = 0; worstOutRTransit = 0;}
    virtual ~CircuitNode() override {adj.clear();}
    [...]
    virtual double Distance(GraphNode* a) override;
    virtual void CalcOutputCap() = 0;
    virtual double GetWorstDelay() = 0;
    virtual double GetWorstTransition() = 0;

    std::string getName() const {return name;}
    void SetInTransition(double value) {if(value > worstInRTransit)
        worstInRTransit = value;}
    void SetCapacity(double value) {outCapacity = value;}

protected:
    CircuitGraph* it;
    double worstInRTransit;
    double worstOutRTransit;
    double delay;
    double outCapacity;
};

class InputCircuitNode : public CircuitNode
{
public:
    InputCircuitNode(CircuitGraph *it, double capacity = 0, const
        std::string& name = "")
    : CircuitNode(it, capacity, name) {}
    InputCircuitNode(CircuitGraph* it, const std::string &name)
    : CircuitNode(it, name){}
    ~InputCircuitNode() override {}

    double GetWorstDelay() override {return 0;}
    double GetWorstTransition() override{return worstInRTransit;}
    void CalcOutputCap() override {outCapacity = 0;}
};

class OutputCircuitNode : public CircuitNode
```

```
{
public:
    OutputCircuitNode(CircuitGraph* it, double capacity, const std::string&
        name = "")
    : CircuitNode(it, capacity, name)
    {delayUpdated = transitionUpdated = false;}
    OutputCircuitNode(CircuitGraph* it, const std::string& name)
    : CircuitNode(it, name){}
    ~OutputCircuitNode() override {}

    double GetWorstDelay() override;
    double GetWorstTransition() override;
    void CalcOutputCap() override;

private:
    bool delayUpdated;
    bool transitionUpdated;
};

class CircuitGraph : public Graph<double>
{
    [...]
public:
    CircuitGraph(const Circuit &c);
    ~CircuitGraph();

    // circuit
    void SetupCaps(); // for each cell, updates the output capacitance
    void SetAllCells(std::vector<double> v);

    std::pair<const Cell*, int> GetCell(CircuitNode *n) const;
    double GetAreaOccupation() {return circuit.GetAreaOccupation();}
    size_t GetNumOfNodes() const {return map2.size();}

private:
    void CreateEdges();
    double GetTimingParam(CircuitNode*, ParamType p);
    void CellSetup(const Cell *cell); // update out cap for a signle cell

    //Cell {input_1, input_2 ... input_n, output_1 ... output_n}
    std::unordered_map<CircuitNode*, std::pair<const Cell*, int>>  map1;
    std::unordered_map<const Cell*, std::vector<CircuitNode*>> map2;
    const Circuit& circuit;
};
```

4.7 CircuitSolver

Questa classe specializza la classe `BRKGA` creando così un algoritmo specializzato per la risoluzione del problema.

```
class CircuitSolver : public BRKGA<CircuitInput, CircuitOutput, double>
```

```
{
public:
    CircuitSolver(const CircuitInput& input, const  BRKGAParams& params);;
    ~CircuitSolver() override {if(dataOut.is_open())dataOut.close();}

    double BestFitness() const override;
    void TrackEvolution(bool track, const std::string& fileName, bool
        timeTracking = 1);
    size_t getDeadlock() const;
    void setDeadlock(size_t value);
    size_t getMaxGens() const;
    void setMaxGens(const size_t &value);
    double getDelayGoal() const;
    void setDelayGoal(double value);
    CriteriaPolicy getCriteriaPolicy() const;
    void setCriteriaPolicy(const CriteriaPolicy &value);
    double getFitnessExp() const;
    void setFitnessExp(double value);

private:
    // ********** Solver methods *************
    void Decode() override;
    bool StopCriteria() override;
    void _Reset() override;
    // ***************************************

    bool IsDeadlockCritReached();
    bool IsMaxGensCritReached();
    bool IsDelayCritReached();

    // *********Stop Criterias*************
    size_t counter;
    double lastBest;
    size_t deadlock;

    size_t maxGens;
    double fitGoal;
    CriteriaPolicy criteriaPolicy;
    // ***********************************

    double fitnessExp, k;
    double f(double delay) const { return pow(k/delay, fitnessExp); };
    double f_1(double fitness) const { return k/pow(fitness,
        1.0/fitnessExp); };

    bool tracking;
    bool timeTracking;
    time_t lastTimeRec;

    std::ofstream dataOut;
};
```

Le funzioni principali sono le seguenti:

- `Decode()`: per ogni elemento all'interno del vettore `hypercube`, imposta le celle nel circuito, ricalcola la capacità di output di queste e poi esegue `getWorstPathDistance()`. Il valore ottenuto viene passato alla funzione `f()` che lo *trasforma* in fitness. Questa trasformazione serve per dare un valore di fitness più alto ai circuiti che hanno ritardo minore. La trasformazione può non essere lineare e dipende dai parametri `fitnessExp` e `k` che vengono impostati dall'utente.

- `StopCriteria()`: in funzione dei 3 possibili criteri di arresto restituisce `true` o `false`. L'utente può specificare uno o più dei seguenti criteri:

  - numero massimo di generazioni: viene verificato dalla funzione `IsMaxGensCritReached()` che verificare se l'attributo della classe base `generations` ha superato il valore indicato da `maxGens`.
  - Fitness deadlock: si considera situazione di *deadlock* quando l'algoritmo non ottiene miglioramenti per k generazioni successive. Viene verificato dalla funzione `IsDeadLockCritReached()` che utilizza le variabili `counter` e `lastBest` per memorizzare quando è avvenuto l'ultimo incremento.
  - Ritardo preimpostato raggiunto: la funzione `IsDelayCritReached()` verifica se è stato raggiunto il valore di delay preimpostato. Questo criterio è utile se abbinato agli altri due.

Questi criteri possono essere abbinati secondo le politiche *and* o *or*, specificate dalla variabile `criteriaPolicy`. In particolare nel caso di una `AndPolicy` tutti i criteri dovranno essere soddisfatti per causare l'uscita dell'algoritmo. Altrimenti, nel caso della `OrPolicy`, basterà che soltanto un criterio si verifichi.

Capitolo 5

Risultati

Le varie parti del problema sono state testate singolarmente e assieme effettuando vari test al fine di verificarne il corretto funzionamento. Oltre agli strumenti di debug sono stati utilizzati due programmi: il primo utilizza la libreria `graphviz` per disegnare schematicamente il circuito, permettendo di controllare che il circuito venisse costruito correttamente; il secondo invece fornisce una semplice interfaccia per eseguire l'algoritmo in più istanze ed ottenere un grafico real-time per ognuna di queste al fine di poter paragonare le diverse scelte sui parametri. Questi due programmi hanno aiutato a controllare che il progetto intero funzionasse correttamente.

5.1 Applicazioni Greedy

Al fine di verificare le dipendenze tra i dati sono stati registrati i risultati di alcune semplici tecniche *greedy*. Si nota una certa dipendenza tra area e delay del circuito: al crescere dell'area

Tecnica di slezione delle celle	Risultati sommatore 64 bit	Risultati decoder 6 bit	Risultati moltiplicatore 11 bit
Celle più grandi	Delay: 5.84936 Area: 5419.1	Delay: 2.01184 Area: 3200.8	Delay: 3.06200 Area: 4178.69
Celle più piccole	Delay: 6.29817 Area: 2122.67	Delay: 2.74827 Area: 360.9	Delay: 3.07560 Area: 1553.04
Random	Delay: 6.12198 Area: 3440.83	Delay: 3.22608 Area: 1076.87	Delay: 3.13217 Area: 2690.02

Figura 5.1:

sembra calare il delay. Questi risultati sono in linea con le aspettative. Quello che però non viene tenuto in considerazione è che al crescere dell'area, cresce anche la capacità dei pin della cella, quindi si avrà un peggioramento del delay. Come si vedrà nei risultati successivi, si può partire dal circuito con area maggiore in modo da avere già un buon ritardo e lo si fa evolvere tramite algoritmo genetico. Questo permette di partire da una soluzione e migliorare sia i tempi che l'area del circuito.

5.2 Variazione della popolazione

Sono state fatte eseguite 4 istanze di risoluzione di un circuito facendo variare il numero di individui totali, e si sono osservati i risultati. In relazione alla Figura 5.2, si possono fare le

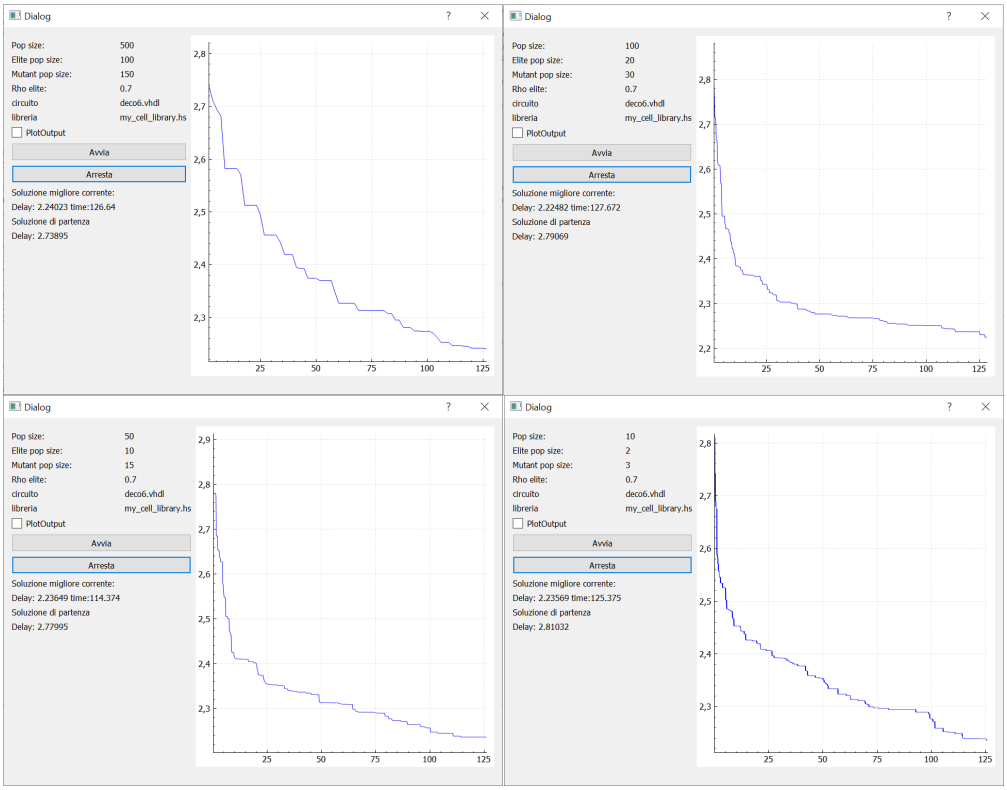

Figura 5.2:

seguenti considerazioni

- popolazioni molto grandi e popolazioni molto piccole portano ad una convergenza più lenta rispetto a popolazioni con un numero di individui adatto alla dimensione del problema
- per tempi di elaborazioni sufficientemente grandi tutti i e 4 casi ottengono buoni miglioramenti

In particolare se non per casi estremi non si è notata una forte dipendenza dalla dimensione della popolazione, così come l'algoritmo converge sempre abbastanza rapidamente se i valori p_e e p_m rimangono all'interno dei valori consigliati. Questi risultati sono simili a quelli proposti in [Gonçalves and Resende(2011)].

5.3 Impostazione del bias sull'area

Come anticipato l'algoritmo offre la possibilità di partire da una soluzione trovata tramite un criterio greedy sull'area. In figura si vedono le differenze tra 3 possibilità:

- Bias nullo: si nota che l'andamento dell'area è abbastanza aleatorio. Si possono notare 2

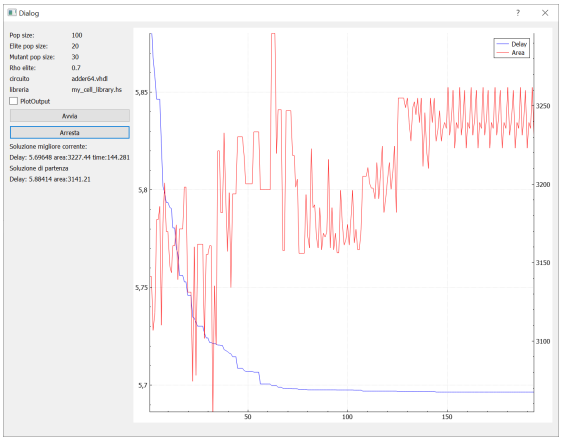

Figura 5.3:

  fatti rilevanti:

  nella parte finale anche piccole minime variazioni nel delay possono portare a grossi incrementi di area. Questo è in linea con le aspettative poiché l'algoritmo minimizza solamente il ritardo.

  sempre nella parte finale del grafico si nota che l'andamento dell'area tende a diventare periodico. Questo vuol dire che ci sono più soluzioni con lo stesso delay ma con aree diverse.

- Bias 1: la prima generazione ha tutti circuiti con area massima. Il grafico mette in evidenza come il fatto di utilizzare celle con aree grandi non porti necessariamente a buoni risultati in termini di delay. Il fatto però di utilizzare questo sistema ha portato l'algoritmo a convergere ad una soluzione leggermente migliore di quella del caso precedente.

- Bias 0: la prima generazione ha tutti circuiti con area minima. Questa potrebbe essere considerata la soluzione intermedia tra le due precedenti. L'area trovata risulta essere minore rispetto ad entrambi i casi precedenti mentre il delay non si discosta di molto da quello trovato con le altre tecniche.

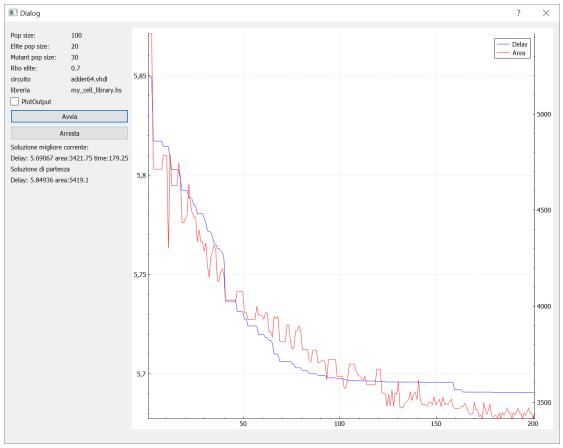

Figura 5.4:

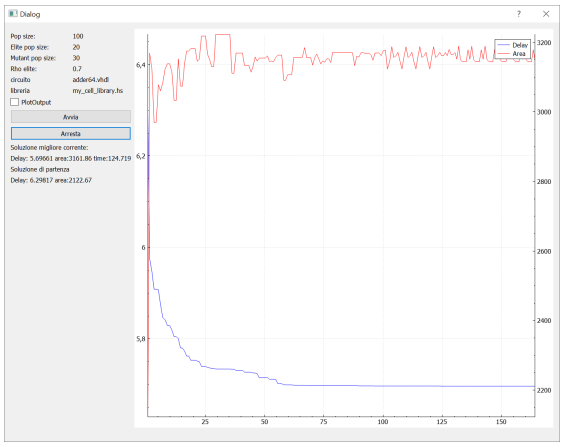

Figura 5.5:

5.4 Variazione di ρ_e

Allo stesso modo dei test sulla dimensione della popolazione, è possibile farne su gli altri parametri. In particolare si vuole mettere in evidenza come può cambiare la convergenza in funzione di ρ_e. I risultati ottenuti sono sempre in linea con quelli proposti in [Gonçalves and Resende(2011)]. In Figura 5.6 si nota come valori esterni al range consigliato $0.5 < \rho_e < 0.8$ fanno convergere l'algoritmo più lentamente. Questo effetto potrebbe dimostrarsi ancora più accentuato su circuiti più grandi.

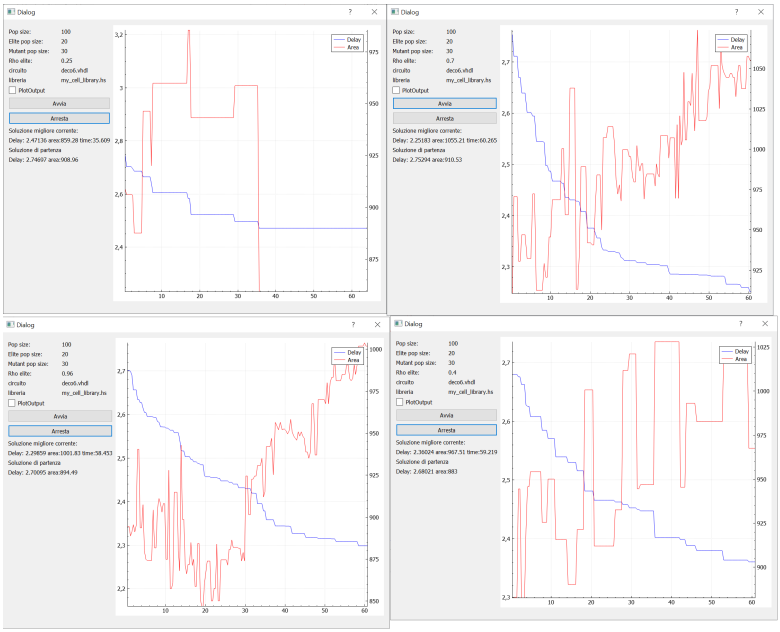
Dialog
Pop size: 100
Elite pop size: 20
Mutant pop size: 30
Rho elite: 0.25
circuito: dec6.vhdl
libreria: my_cell_library.lis
PlotOutput
Avvia
Arresta
Soluzione migliore corrente:
Delay: 2.47136 area:859.28 time:35.609
Soluzione di partenza
Delay: 2.74697 area:908.96
Delay
Area
Dialog
Pop size: 100
Elite pop size: 20
Mutant pop size: 30
Rho elite: 0.2
circuito: dec6.vhdl
libreria: my_cell_library.lis
PlotOutput
Avvia
Arresta
Soluzione migliore corrente:
Delay: 2.25183 area:1055.21 time:60.265
Soluzione di partenza
Delay: 2.75294 area:910.53
Delay
Area
Dialog
Pop size: 100
Elite pop size: 20
Mutant pop size: 30
Rho elite: 0.96
circuito: dec6.vhdl
libreria: my_cell_library.lis
PlotOutput
Avvia
Arresta
Soluzione migliore corrente:
Delay: 2.29859 area:1001.83 time:58.453
Soluzione di partenza
Delay: 2.70095 area:894.49
Delay
Area
Dialog
Pop size: 100
Elite pop size: 20
Mutant pop size: 30
Rho elite: 0.4
circuito: dec6.vhdl
libreria: my_cell_library.lis
PlotOutput
Avvia
Arresta
Soluzione migliore corrente:
Delay: 2.36024 area:967.51 time:59.219
Soluzione di partenza
Delay: 2.68021 area:883
Delay
Area

Figura 5.6:

Capitolo 6

Conclusioni

Il progetto ha messo in luce l'efficacia (già comprovata) dell'algoritmo genetico nonostante la sua semplice implementazione. Si è visto, inoltre, che è necessario scegliere opportunamente i parametri che descrivono il processo di evoluzione al fine di far convergere l'algoritmo più velocemente. Ciononostante, rimanendo all'interno dei valori consigliati per ognuno di questi parametri, l'algoritmo converge sempre a soluzioni piuttosto buone.

I risultati ottenuti, tuttavia, non negano il fatto che molti sono i miglioramenti che possono essere fatti al fine di rendere l'algoritmo più rapido. Ad esempio, si possono pensare tecniche che facciano evolvere i parametri durante la stessa evoluzione. Oppure si potrebbe strutturare l'O-O Framework in modo da aggiungere gradi di liberà all'utilizzatore in modo da costruire algoritmi più specifici e adatti al problema. Questi, o altri miglioramenti, possono essere oggetto di sviluppi futuri.

Bibliografia

[Gonçalves and Resende(2011)] José Fernando Gonçalves and Mauricio G. C. Resende. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17 (5), Oct 2011. doi: 10.1007/s10732-010-9143-1. URL https://doi.org/10.1007/s10732-010-9143-1.