Université De Lorraine

Faculté des Sciences
Département d'Informatique

Master I
Initiation à la recherche

Sujet

Dessiner c'est prouver : Quantomatic un logiciel graphique pour un raisonnement quantique

Réalisé par :

Antoine Baran Tarek Mokhtari

Encadrants :

Mr. Emmanuel Jeandel
Mr. Simon Perdrix

20 mai 2016

Table des matières

Liste des figures

Introduction

Le formalisme mathématique qui permet de calculer et de raisonner en informatique quantique – et plus généralement en mécanique quantique – est traditionnellement un calcul matriciel : les états quantiques sont représentés par des vecteurs et les évolutions par des matrices. Ces dernières années un langage graphique appelé ZX-calculus a été développé : le calcul matriciel y est remplacé par des diagrammes et une preuve consiste à transformer ces diagrammes suivant des règles graphiques.

Peut-on prouver autant de choses à l'aide de ce langage graphique qu'en utilisant des matrices ? Cette question est encore ouverte actuellement, la complétude du langage a seulement été démontrée pour des fragments de la mécanique quantique. Un logiciel appelé Quantomatic permet de construire des diagrammes du ZX-calculus sur ordinateur. Quantomatic permet également d'automatiser autant que possible l'application des transformations graphiques.

Chapitre 1

ZX-Calculus

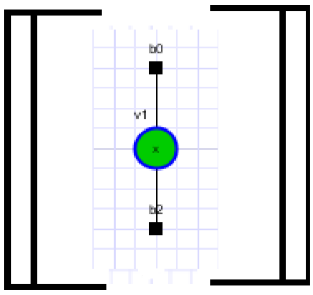
Sommaire

Le ZX-Calculus est un langage graphique permettant de raisonner sur des systèmes et processus quantiques.

Ce langage est représenté par des graphes composés de sommets, d'arêtes et d'entrées/sorties, il existe trois types de sommets :

1.1 Les composants du langage

1.1.1 Les sommets verts

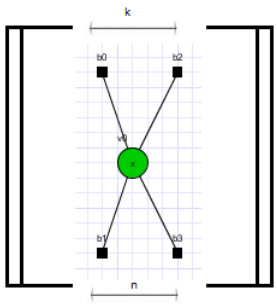


$$\begin{pmatrix} 1 & 0 \\ 0 & e^{ix} \end{pmatrix}$$

FIGURE 1.1 – Un sommet vert et sa matrice correspondante.

La figure 1.1 montre un sommet vert et la matrice qui lui correspond. C'est donc une matrice toute simple sur laquelle on peut retrouver la valeur x de l'angle du sommet.

Prenons l'exemple d'un sommet vert qui a plusieurs entrées/sorties (figure 1.2) :



$$\overset{2^k}{\longleftrightarrow} \quad \overset{2^n}{\updownarrow} \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ 0 & & & & \vdots \\ \vdots & & 0 & & \vdots \\ \vdots & & & & 0 \\ 0 & \dots & \dots & 0 & e^{ix} \end{pmatrix}$$

FIGURE 1.2 – Un sommet vert avec plusieurs E/S et sa matrice correspondante.

C'est presque la même matrice, sauf que le nombre d'entrées/sorties influe sur la matrice.

1.1.2 Les sommets rouges

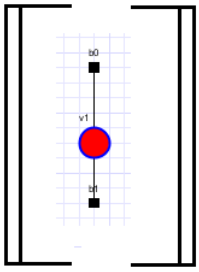
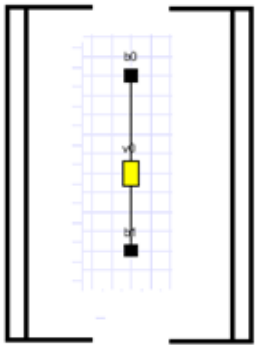


FIGURE 1.3 – Un sommet rouge.

La matrice d'un sommet rouge étant un peu compliquée, nous le simplifierons par le fait que c'est un sommet vert auquel on applique une transformation que nous verrons par la suite.

1.1.3 Les "carrés d'Hadamard"

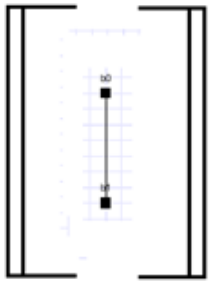


$$1/\sqrt{2}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

FIGURE 1.4 – Un "carré d'Hadamard" et sa matrice correspondante.

La matrice correspondant aux carrés d'Hadamard n'est pas très compliquée non plus.

1.1.4 Les arêtes



$$\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$$

FIGURE 1.5 – Une arête et sa matrice correspondante.

La matrice d'une arête correspond simplement à la matrice identité.

Comme on l'a vu, les sommets rouges et verts peuvent comporter des angles. Ces sommets peuvent être reliés par des arêtes afin de former un graphe et peuvent également comporter des « entrées-sorties ». La première partie de notre projet a donc été de nous familiariser avec ce langage graphique.

Dans le but de pouvoir faire des démonstrations dans ce langage, il existe des règles nous permettant de transformer ces graphes.

On sait que le langage est correct, si on utilise des règles de transformation sur les graphes, la matrice correspondante reste la même. C'est donc pour ça que nous allons travailler majoritairement par le biais des graphes. On remplace les calculs matriciels par des calculs graphiques.

L'objectif final de notre projet sera de savoir si, étant donné deux graphes, on peut passer de l'un à l'autre à l'aide de transformations.

1.2 Matrices

Pour parler un petit peu des matrices, voilà comment cela se présente :

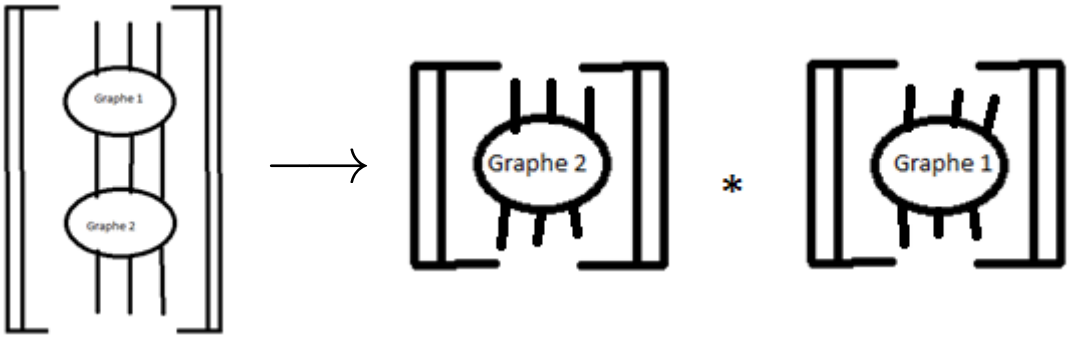


FIGURE 1.6 – Deux graphes connectés et produit matriciel.

Une matrice qui contient deux graphes reliés peut être développé en le produit de deux matrices, l'une contenant le premier graphe et l'autre contenant le deuxième (figure 1.6.

Si par contre les deux graphes ne sont pas connectés, l'opération entre les deux matrices devient un produit tensoriel (figure 1.7) :

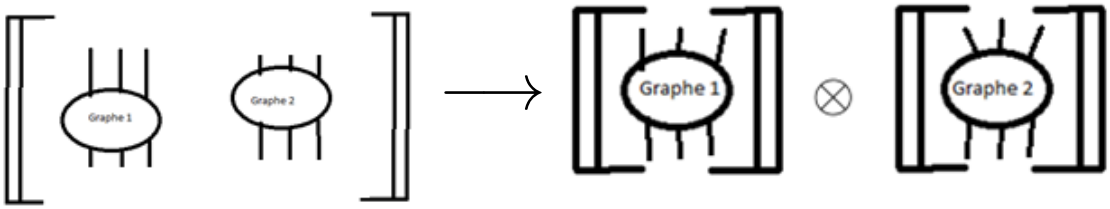
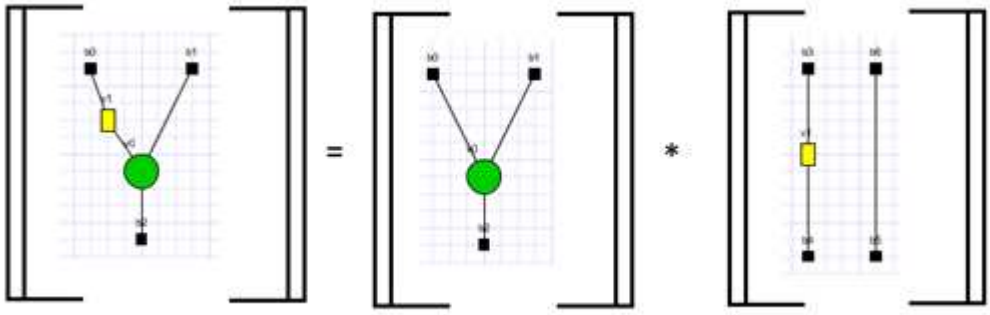


FIGURE 1.7 – Deux graphes non connectés et produit tensoriel.

Exemple

Ce qui nous donne sur un exemple simple (figure 1.8) :



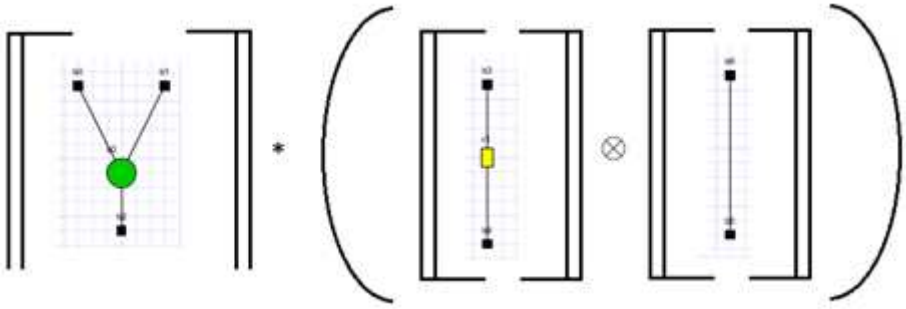


FIGURE 1.8 – Exemple correspondance graph $\longleftrightarrow$ matrice.

Les matrices correspondantes sont les suivantes (figure 1.9) :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & e^{ix} \end{pmatrix} \left(1/\sqrt{2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right)$$

FIGURE 1.9 – Matrice correspondante à l'exemple précédent.

Mais en pratique, le but reste de pouvoir raisonner avec les diagrammes, nous allons donc voir certaines règles permettant de transformer ces diagrammes.

1.3 Règles de démonstration

1.3.1 Règles de base

Pour mieux comprendre le langage ZX-Calculus, voici les cinq premières règles sur lesquelles nous avons travaillé :

1. spider :

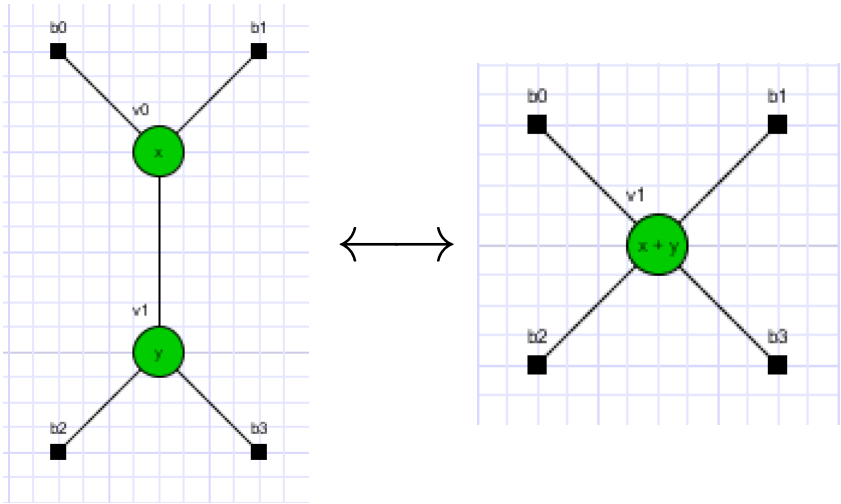


FIGURE 1.10 – La règle spider.

Deux sommets de mêmes couleurs peuvent être regroupés et leurs angles sont additionnés.

2. id :

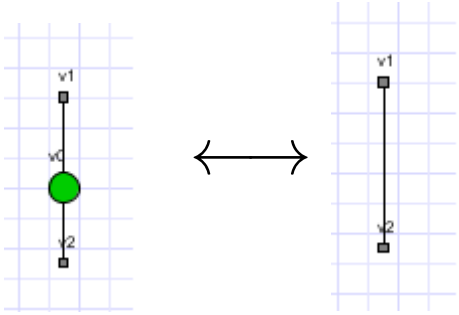


FIGURE 1.11 – la règle id.

3. sp :

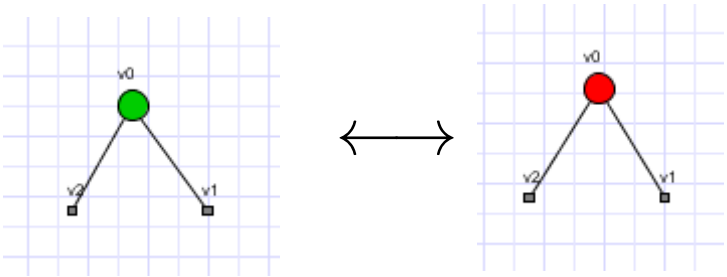


FIGURE 1.12 – La règle sp.

4. copy :

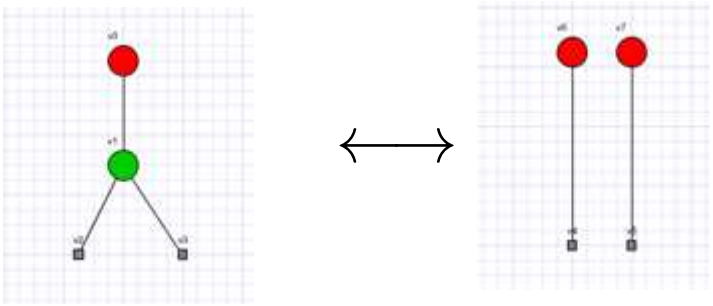


FIGURE 1.13 – La règle copy.

5. bialgebra :

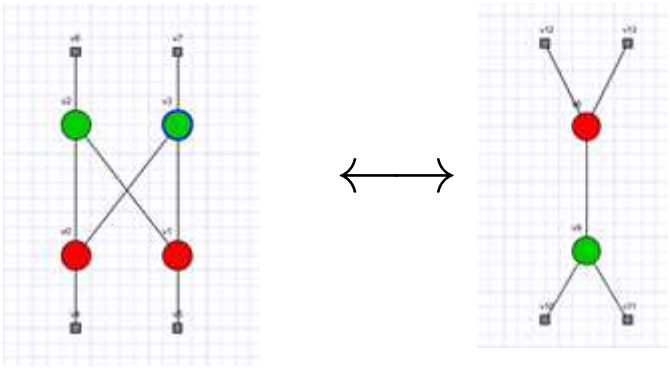


FIGURE 1.14 – La règle bialgebra.

Toutes ces règles sont possibles dans un sens comme dans l'autre, elles vont nous permettre de prouver de nouvelles règles plus complexes.

1.3.2 Règles avec angles

1. inversion :

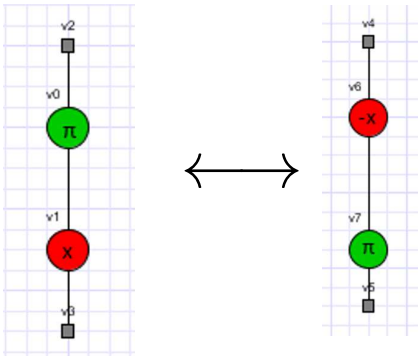


FIGURE 1.15 – La règle inversion.

2. dédoublement :

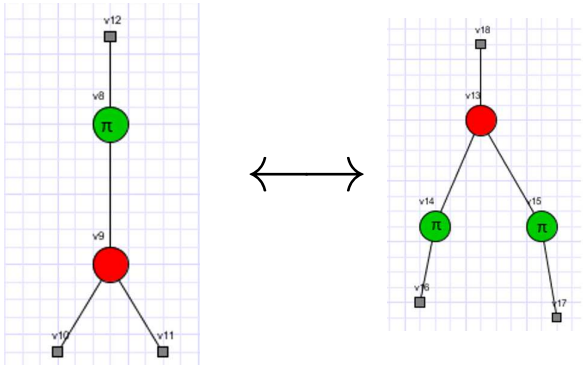


FIGURE 1.16 – la règle dédoublement.

3. changement couleur :

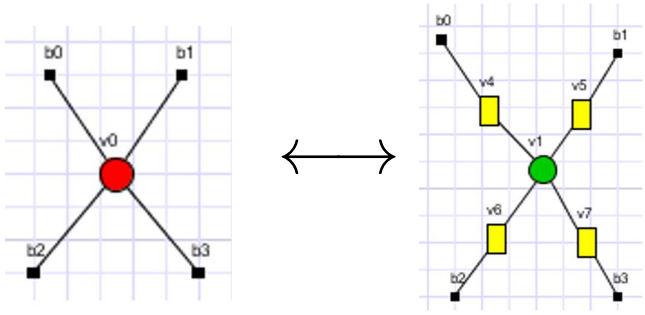


FIGURE 1.17 – La règle changement couleur.

C'est cette dernière règle qui nous permet de calculer la matrice d'un point rouge, en transformant un sommet rouge en vert, étant donné qu'on connaît la matrice d'un carré et celle d'un sommet vert.

4. décomposition Hadamard :

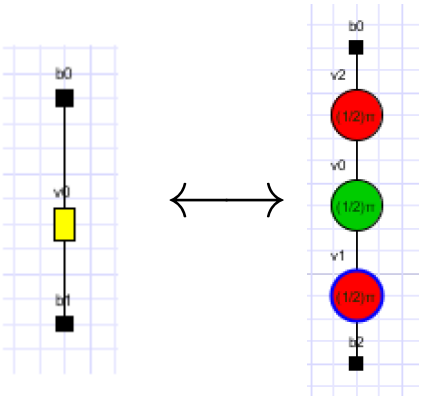


FIGURE 1.18 – La règle décomposition Hadamard.

5. isomorphe :
Si deux graphes sont isomorphes, ils vont représenter la même matrice.

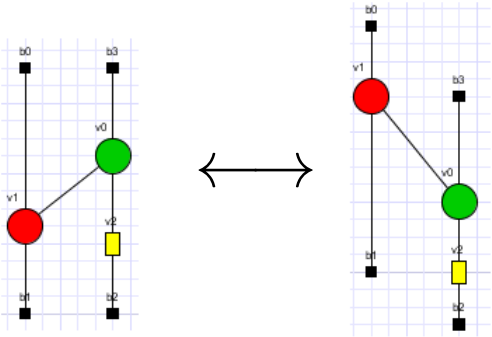


FIGURE 1.19 – La règle isomorphe.

Ce ne sont pas les mêmes graphes mais ils gardent une même matrice, c'est une "meta-règle".

1.3.3 Démonstration

Ces premières règles nous ont permis de prouver plusieurs nouvelles règles en partant d'un cas très simple. Au fur et à mesure des démonstrations il est possible de réutiliser les règles démontrées, de cette manière on a toujours de nouvelles possibilités pour faire des démonstrations.

Un petit exemple d'une règle que nous avons démontré (1.20) :

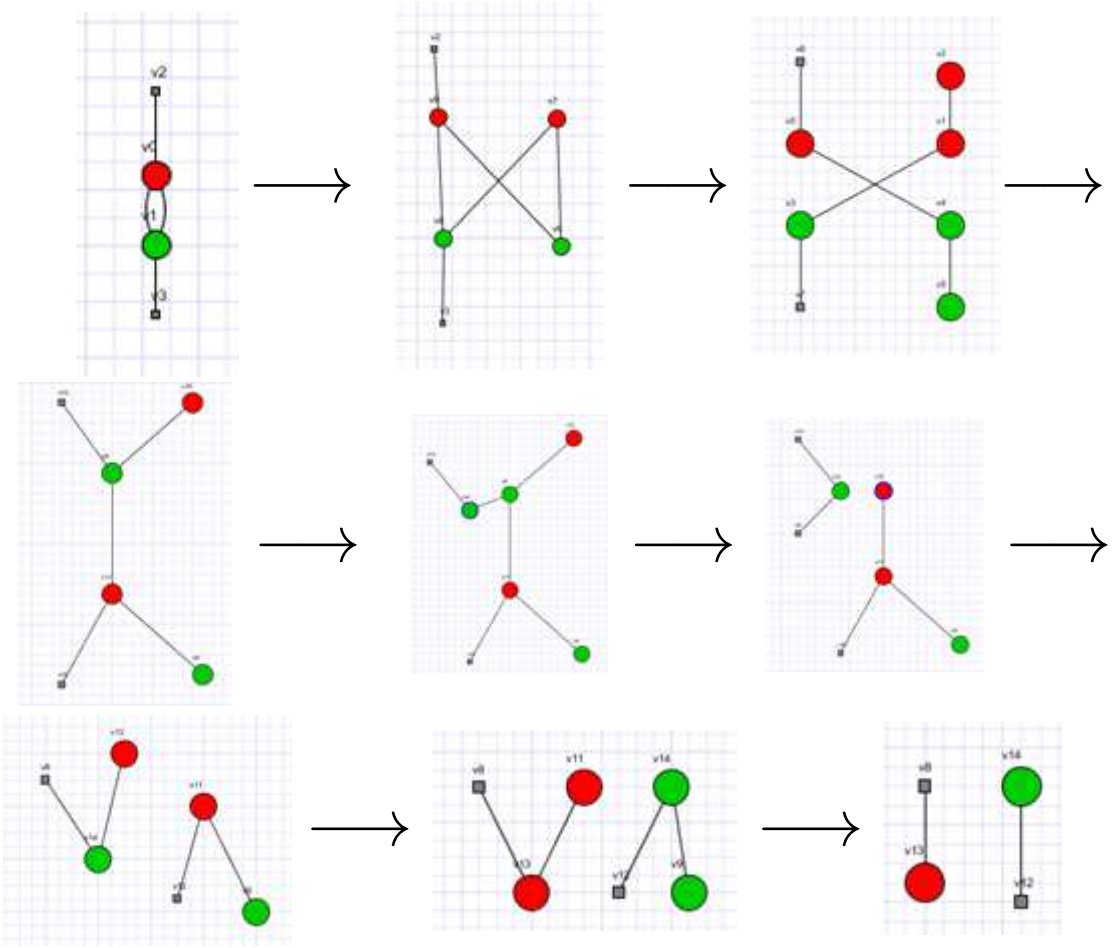


FIGURE 1.20 – Un exemple de démonstration d'une règle.

Cette démonstration nous permet maintenant de supprimer deux arêtes entre les deux mêmes sommets. Le but est donc de réussir à démontrer des règles qui vont nous faciliter d'autres dérivations.

Les règles de transformations des graphes préservent la sémantique et ne vont pas changer les matrices, les démonstrations se passent donc de la même manière, par exemple (1.21) :

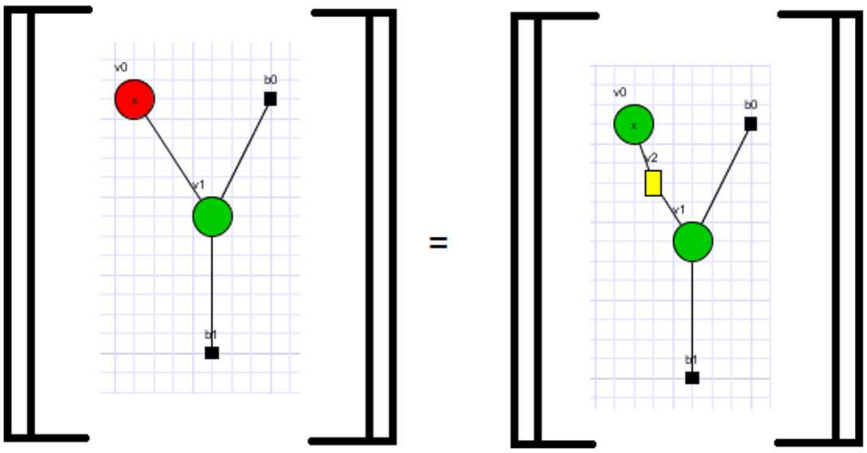


FIGURE 1.21 – Préservation de la sémantique.

C'est ce qui va nous permettre de travailler principalement avec ces règles.

1.4 Forme normale

Nous cherchons à savoir si deux graphes sont égaux, pour cela on les transforme en « graph state », que nous allons définir par la suite. On veut montrer que tous les graphes peuvent se transformer en graph state.

1.4.1 Parité

Il est possible de calculer la parité (rouge ou verte) d'un graphe de la manière suivante :

$$P_{red}(G) \;=\; (\#Hadamards + \textstyle\sum_{v \,\in\, red} \sigma(v)) \, mod \, 2$$

La parité (ici rouge) d'un graphe est l'addition du nombre de carré d'hadamard avec la somme des degrés des sommets rouges (respectivement verts pour la parité verte), le tout modulo deux. Lors de l'application de règle à un graphe, la parité ne change en aucun cas, par exemple (1.22) :

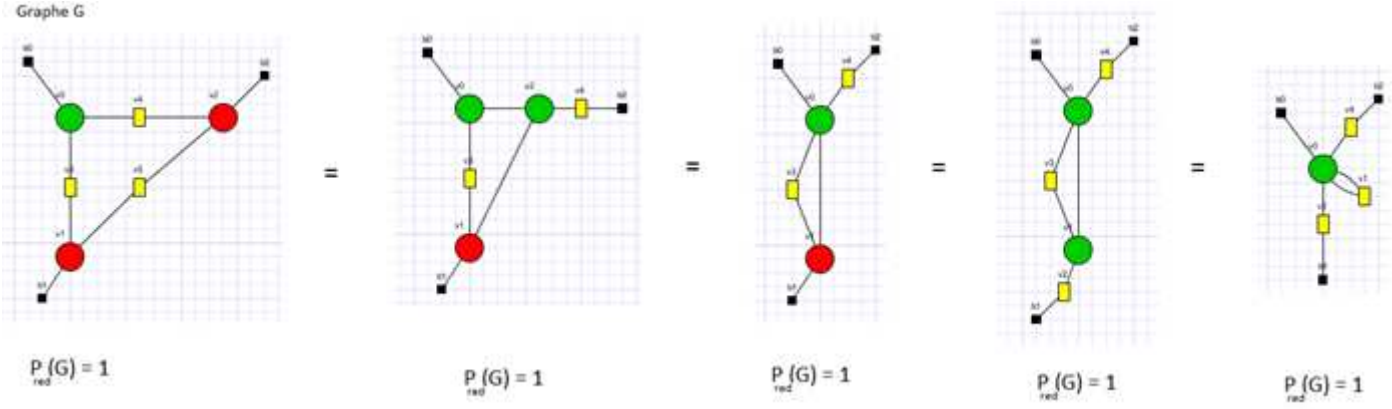


FIGURE 1.22 – Exemple sur la parité rouge.

La parité est une condition suffisante nous permettant de dire qu'on ne peut pas passer d'un graphe D à un graphe D'. Si deux graphes n'ont pas la même parité, ils ne sont pas égaux. De cette façon c'est une petite vérification qui nous permet d'éviter de perdre du temps inutilement.

1.4.2 Etat Graphe (Graph State)

Les Etats graphes sont des graphes dont les sommets sont uniquement des sommets verts et des carrés d'Hadamard, les carrés d'Hadamard et les sommets verts ne doivent pas être reliés entre eux respectivement. En partant d'un graphe normal, le but est donc d'éliminer tous les sommets rouges et de simplifier les arrêtes "verte-verte" et "hadamard-hadamard". Le but est d'arriver à une forme sur laquelle on aime bien travailler. On aimerait trouver un ensemble de règle permettant de partir d'un graphe normal et d'arriver à son graph state.
Nous allons par la suite travailler sur ces graphs state qui sont des versions "simplifiées" des graphs normaux. Dans le cadre de notre projet les graphs state vont nous permettre de savoir plus facilement si deux graphes sont égaux, cela nous permet de ramener un problème général à un problème avec les graphs state, donc plus simple.

Chapitre 2

Quantomatic

Sommaire

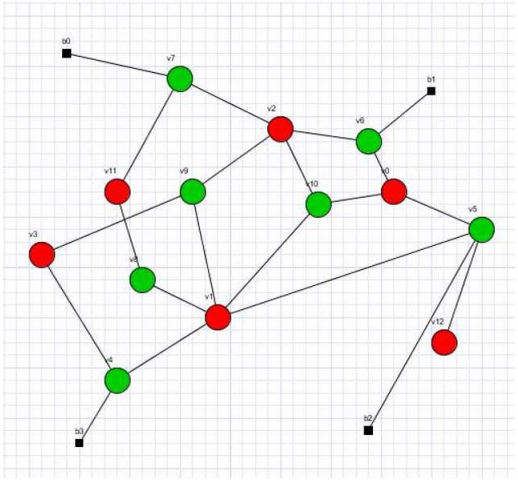
Quantomatic est ce qu'on appelle un assistant de preuve schématique, c'est-à-dire qu'il fournit le support pour raisonner avec les langages graphiques. Quantomatic nous permet de dessiner des graphes du ZX-Calculus et y appliquer des règles de dérivation. Quantomatic a été développé par des chercheurs et des étudiants en doctorat principalement dans les universités d'Oxford et Edinburgh.

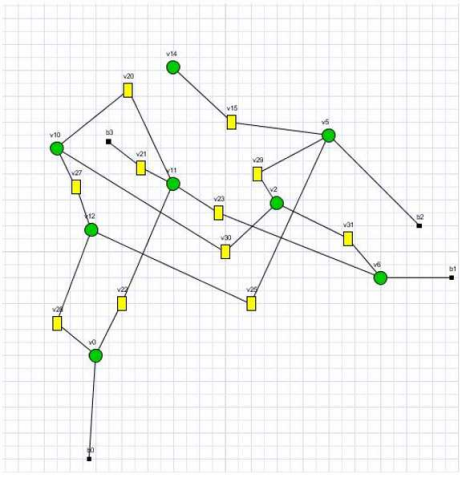
2.1 Ajout théorème

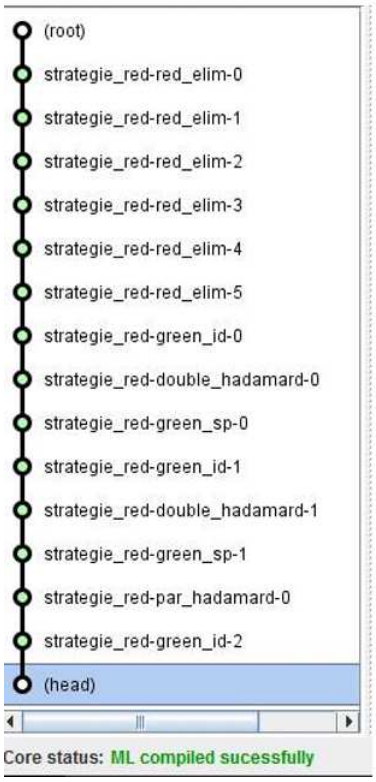
Quantomatic nous facilite la tâche lorsque nous travaillons sur le ZX-Calculus, il est possible d'ajouter de nouvelles règles pour les réutiliser par la suite. Nous partons d'un graphe simple, puis nous lui appliquons des règles de dérivation jusqu'à avoir une simplification qui nous intéresse. Nous sauvegardons ensuite la liste des règles utilisée pendant la démonstration, et à partir de cela Quantomatic nous permet d'ajouter le théorème correspondant, qui pourra maintenant être utilisé.

2.2 Implémentation stratégies

Ce Logiciel nous permet également de travailler sur des stratégies, il y a des fonctions implémentés en Poly/ML qui permettent d'appliquer une séquence de plusieurs règles dans un ordre précis à un graphe.
Nous avons donc travaillé sur ce langage pour, par exemple programmé une stratégie permettant, en partant d'un graphe, d'obtenir son "Etat graphe" grâce à une certaine séquence de règle parmi lesquels plusieurs que nous avions démontrés. Pour mieux visualiser cette stratégie, une petite illustration :







La stratégie crée nous permet, à partir d'un graph de départ comme le premier, de trouver son graphe state. La liste des règles appliquées par la stratégie est décrite par la 3ème image.

Chapitre 3

Développement Java

Sommaire

Le logiciel Quantomatic travaille avec des fichiers qgraph, dans le but de notre projet, nous avons développé une petite application en java nous permettant de manipuler ces graphes et d'avoir diverse informations. Les fichiers qgraph sont écrits en Json.

3.1 Informations

A partir d'un graph venant de Quantomatic nous avons implémentés diverses fonctionnalités, il nous est possible par exemple de connaitre le nombre de sommets, de savoir quels sommets sont reliés, calculer la parité rouge ou verte d'un graphe ainsi que diverses informations concernant les angles, les voisins et les types des sommets.

3.2 Manipulation

Nous pouvons créer ou modifier un fichier qgraph qui sera manipulable par la suite dans Quantomatic en y ajoutant ou modifiant des sommets où des arêtes. Nous avons également la possibilité d'ajouté un carré d'Hadamard entre deux sommets où vérifier si un graphe donné est valide ou non, c'est-à-dire qu'il doit avoir des carrés d'Hadamard de degré 2 et des sorties de degré 1.

3.3 Création (fonction "random")

Nous avons également implémenté une fonctionnalité nous permettant de créer des graphes de manière aléatoire avec par exemple le nombre de sommets et le nombre d'entrées/sorties paramétrables.

La connexion entre deux sommets est également aléatoire mais peut-être défini selon une probabilité : plus l'utilisateur choisi une probabilité élevé plus il y aura de chance d'avoir une arrête entre deux sommets.
Cette fonction nous a posé quelques problèmes puisqu'il y a plus de paramètres que ce que nous pensions à prendre en compte pour créer un graphe correcte surtout du côté des carrés d'hadamard qui sont des sommets assez spéciaux.
Etant donné que nous travaillons beaucoup sur des graphes, cela nous permet de gagner un temps considérable puisque nous n'avons plus à créer un graphe différent à chaque fois que nous voulons tester une fonctionnalité ou une stratégie sur un graphe.
Pour mieux visualiser voici un graphe 3.1 ouvert dans Quantomatic créé par notre fonction random avec les paramètres suivants :

- 25 sommets.
- 10 entrées/sorties.
- $P_g = 0.3$
- $P_r = 0.4$
- $P = 0.1$

avec :

- P_g : Probabilité pour qu'un sommet soit vert lors de sa création.
- P_r : Probabilité pour qu'un sommet soit rouge lors de sa création.
- P : Lors de la génération de arêtes, en prenant les sommets (non hadamard) deux à deux, la probabilité pour que ces deux sommets soient connectés.

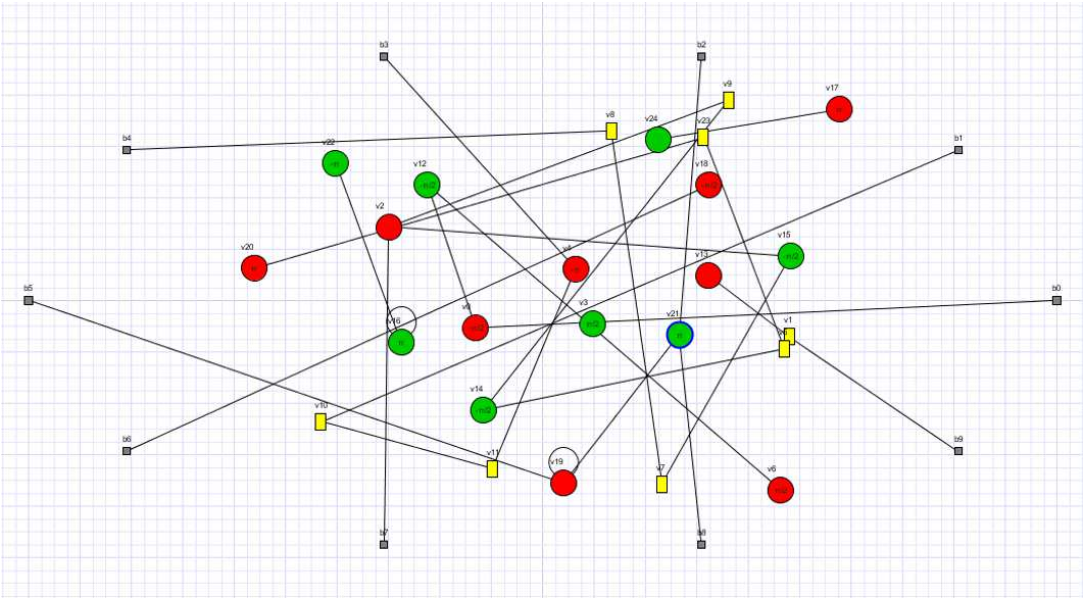


FIGURE 3.1 – Un graph généré par la fonction random.

3.4 Concaténation

La concaténation est une fonctionnalité que nous avons choisi d'implémenter pour assembler deux graphes. Cette fonction est exécutable de deux manières :

3.4.1 Avec deux graphes en paramètre

On prend simplement deux graphes et on les rassemble en un seul graphe sans connexion entre les deux.

3.4.2 Avec deux graphes et deux entrées/sorties en paramètre

Avec ces paramètres l'utilisateur donne deux graphes ainsi qu'une entrée/sortie du premier graphe et une entrée/sortie du deuxième graphe en paramètre. La fonction va dans ce cas assembler les deux graphes en les connectant par les entrées/sorties données en paramètre par l'utilisateur. Ce cas a été un peu plus difficile à réaliser en raison des noms des entrées/sorties qui peuvent être les mêmes pour les deux graphes. Voici une petite illustration (3.2) :

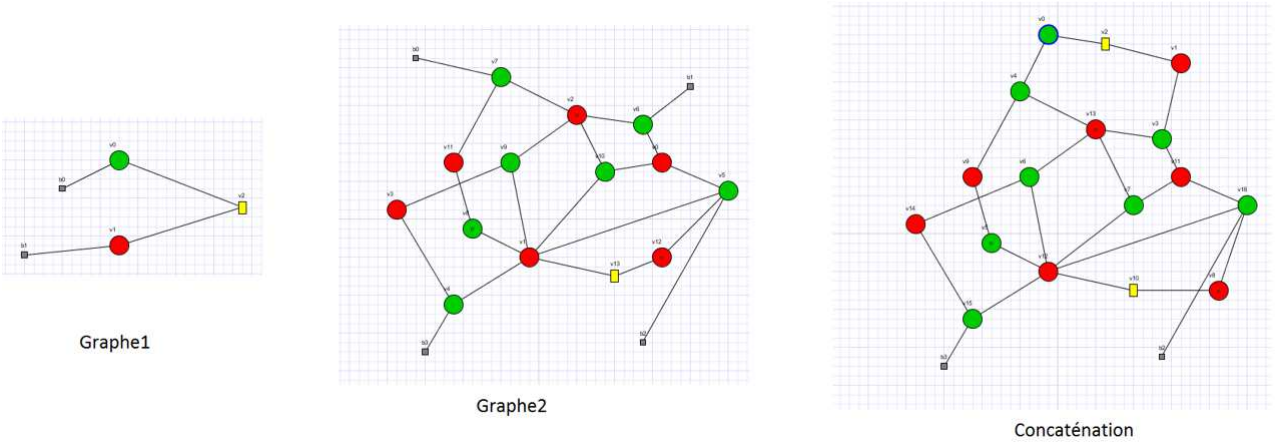


FIGURE 3.2 – Un exemple de concaténation

On voit bien que le graphe 1 a été ajouté en haut du deuxième graphe, on y retrouve ses trois sommets.

3.5 Vérification de Graphe

Pour qu'un graphe soit correct il y a certaines vérifications à faire : un graphe correcte est un graphe dont les carrés d'hadamard sont de degré deux et les entrées/sorties de degré un. Nous avons donc ajouté une fonctionnalité nous permettant de vérifier les graphes. Cette fonction va nous être très pratique par rapport à la création de graphe aléatoire. Du coup si on prend un cas particulier en créant un graphe avec uniquement des entrées/sorties, elles doivent toutes être reliées entre elles deux à deux. Si le nombre d'entrées/sorties est impair, c'est impossible.

Conclusion

Ce projet d'initiation à la recherche nous aide donc à découvrir un peu plus comment se passe la recherche. Ici nous avons découvert entièrement un nouveau langage et travaillé dessus tout au long du semestre. Notre projet a également assez vaste puisqu'il nous a permis de manipuler à la fois les graphes sur papier, dans le logiciel Quantomatic ainsi que dans notre application en java. Malgré quelques difficultés finalement résolus, ce projet nous a plu et nous avons aimé passer du temps dessus.

Annexe A

La fonction random

```
public static QGraph random (int nVertices, int nBoundaries, double
    green, double red, double p) {
// Checking parameters:
if (p > 1 || p < 0) {
        System.out.println("Random: P must be between 0 and 1.");
        return null;
}
if (nVertices < 0 || nBoundaries < 0) {
        System.out.println("Random: Number of vertices and
            boundaries must be positif.");
        return null;
}
if (green > 1 || green < 0 || red > 1 || red < 0) {
        System.out.println("Random: parameters green (3rd) and red
            (4th)  must be between 0 and 1.");
        return null;
}
if (green + red > 1) {
        System.out.println("The addition of the parameters: green
            (3rd) and red (4th) must be inferior to 1.");
        return null;
}
if (nVertices == 0 && nBoundaries % 2 == 1) {
        System.out.println("Random: No solution for the input: (" +
            nVertices + "," + nBoundaries + "," + green + "," + red
            + "," + p + ").");
        return null;
}
if (nBoundaries % 2 == 1 && green == 0 && red == 0) {
        System.out.println("Random: No solution for the input: (" +
            nVertices + "," + nBoundaries + "," + green + "," + red
            + "," + p + ").");
        return null;
}
```

```
// Creating an empty graph:
QGraph graph = new QGraph();

// Adding boundaries:
for (int i = 0; i < nBoundaries; i++) {
        String name = graph.addBoundary();
        graph.getBoundary(name).setX((float) (B_WIDTH * Math.cos((2
            * i * Math.PI)/nBoundaries)));
        graph.getBoundary(name).setY((float) (B_HIGHT * Math.sin((2
            * i * Math.PI)/nBoundaries)));

}

// Adding random N vertices:
double randomNumber;
for (int i = 0; i < nVertices; i++) {
        Random random = new Random();
        Random random2 = new Random();
        randomNumber  = Math.random();
        int v = random2.nextInt(5);
        String value = new String();
        switch (v) {
                case 0:
                        value = new String("-\\pi");
                        break;
                case 1:
                        value = new String("-\\pi/2");
                        break;
                case 2:
                        value = new String("");
                        break;
                case 3:
                        value = new String("\\pi/2");
                        break;
                case 4:
                        value = new String("\\pi");
                        break;
                default:
                        break;
        }
        if (randomNumber < green) {
                //graph.addVertex(Type.GREEN);
                graph.addVertex(Type.GREEN, value,
                    2*MAX_X*random.nextFloat() - MAX_X,
                    2*MAX_Y*random.nextFloat() - MAX_Y);
        }
        else if (randomNumber < (green + red)) {
                //graph.addVertex(Type.RED);
```

```
                graph.addVertex(Type.RED, value,
                    2*MAX_X*random.nextFloat() - MAX_X,
                    2*MAX_Y*random.nextFloat() - MAX_Y);
        }
        else {
                //graph.addVertex(Type.HADAMARD);
                graph.addVertex(Type.HADAMARD, "",
                    2*MAX_X*random.nextFloat() - MAX_X,
                    2*MAX_Y*random.nextFloat() - MAX_Y);
        }
}

// Handling a special case: nBoundaries impair and all generated
    vertices are Hadamards:
// Change one hadamard randomly to a green or a red vertex
if (nBoundaries % 2 == 1 && nVertices > 0){
        if (graph.getNReds() + graph.getNGreens() == 0) {
                Random random = new Random();
                int r = random.nextInt(2);
                if (r == 0)
                        graph.getVertex(new String ("v" +
                            random.nextInt(nVertices))).setType(Type.GREEN);
                else if (r == 1)
                        graph.getVertex(new String ("v" +
                            random.nextInt(nVertices))).setType(Type.RED);

        }
}

// Handling a special case: 1 Hadamard, nothing else:
if (nVertices == 1 && !graph.getHadamards().isEmpty())
        graph.addEdge(graph.getHadamards().get(0),
            graph.getHadamards().get(0));

// Connecting boundaries to random vertices:
ArrayList<Vertex> CV;
ArrayList<Boundary> CB;

Random random = new Random();
Iterator<Boundary> iterator = graph.getBoundaries().iterator();
Boundary current;
int rand;
while (iterator.hasNext()) {
        current = iterator.next();
        CV = graph.connectableVertices();
        CB = graph.connectableBoundaries();

        if (graph.degree (current.getName()) > 0) {
                continue;
        }
```

```
        if (!CV.isEmpty()) {
                        rand = random.nextInt(CV.size());
                graph.addEdge(current, CV.get(rand));
        }
        else {
                do {
                        rand = random.nextInt(CB.size());
                } while
                   (current.getName().equals(CB.get(rand).getName()));
                graph.addEdge(current, CB.get(rand));
        }

}

// Connecting Hadamards:
ArrayList<Vertex> hadamards = graph.getHadamards();
Iterator<Vertex> iterH;

for (int i = 0; i < 2; i++) {
        iterH = hadamards.iterator();
        random = new Random();
        while (iterH.hasNext()) {
                Vertex v = iterH.next();
                CV = graph.connectableVertices();
                if (graph.degree(v.getName()) == 2)
                        continue;
                else {
                        do {
                                rand = random.nextInt(CV.size());
                        } while
                           (v.getName().equals(CV.get(rand).getName()));
                        graph.addEdge(v, CV.get(rand));
                }
        }
}

// connecting other vertices according to P:
for (int i = 0; i < nVertices; i++)
        for (int j = i; j < nVertices; j++) {
                if (graph.getVertex(new String("v" + i)).getType()
                   == Type.HADAMARD ||
                        graph.getVertex(new String("v" +
                           j)).getType() == Type.HADAMARD)
                        continue;
                else if (Math.random() < p)
                        graph.addEdge(new String("v" + i), new
                           String("v" + j));
        }
return graph;
}
```

Annexe B

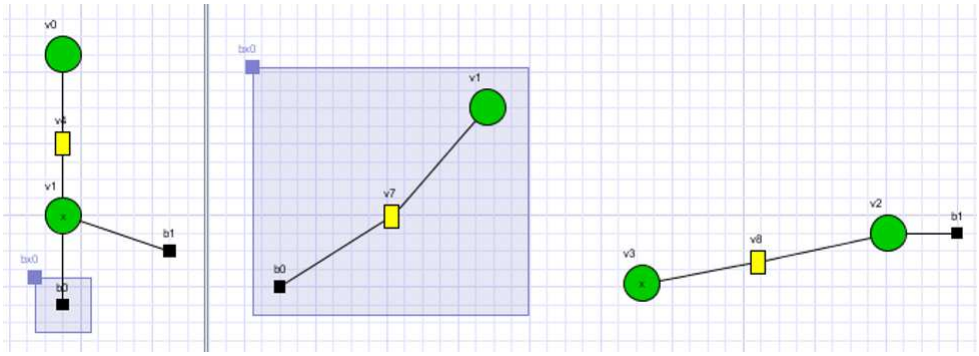
Stratégie permettant d'arriver vers un graph state

```
open RG_SimpUtil

val red_elim = load_releset [
        "theorems/red_elim"
];

val simps = load_ruleset [
        "axioms/green_id", "axioms/green_sp",
           "theorems/double_hadamard", "theorems/par_hadamard",
           "axioms/copy"
];

val simplification = load_ruleset [
        "theorems/regleV", "axioms/pi0-",
           "axioms/greeen_hadamard2", "axioms/3pirgreen"
];

register_simproc {"strategie_red", REDUCE_ALL red_elim ++
   REDUCE_ALL simps ++ REDUCE_ALL simplification ++ REDUCE_ALL
   simps};
```

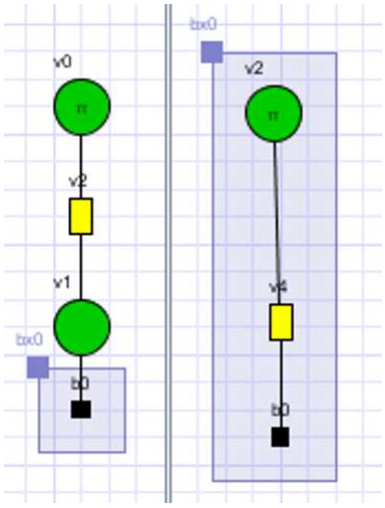
Annexe C

Quelques nouvelles règles du ZX-Calculus utilisées dans la stratégie

1. regleV :



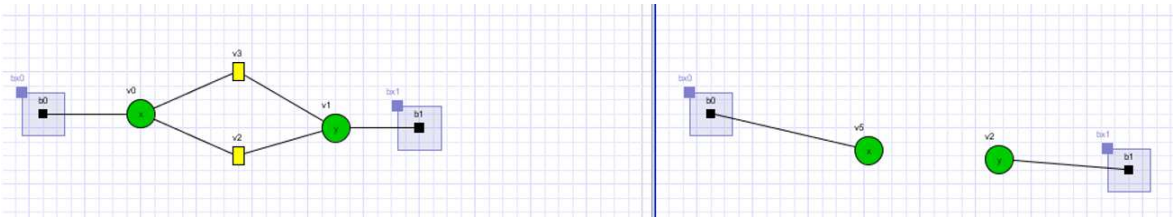
2. pi0- :



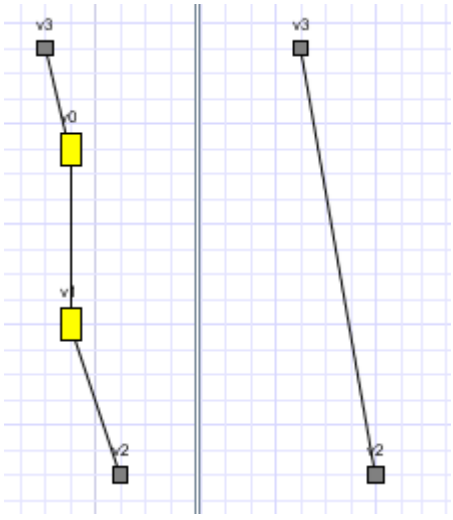
3. green_hadamard2 :

4. 3pirgreen :

5. par_hadamard :



6. double_hadamard :



Références bibliographiques

[1] Miriam backenin in new jornal of physics. vol. 16. no. 9. pages 093021. september, 2004 : The zx-calculus is complete for stabiliser quantum mechanics.

[2] Ross Duncan and Simon Perdrix(2009). Graph states and the necessity of euler decomposition. in : Mathematical theory and computational practice, 5635, springer berlin heidelberg, berlin, heidelberg, pp. 285-296, doi :10.1007/978-3-642-14162-1_24.

[3] Quantomatic. https ://sites.google.com/site/quantomatic/.