Nº d'ordre : 105/2023.C/Inf

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche
Université des **S**ciences et de la **T**echnologie **H**ouari **B**oumediene
Faculté d'Informatique

Thèse de DOCTORAT

Présentée pour l'obtention du grade de Docteur

En : **Informatique**

Spécialité : **Systèmes Informatiques**

Par

MAYATA Raouf

Thème

Contribution à l'optimisation du temps de réponse dans un entrepôt de données par les approches bio-inspirées

Soutenue publiquement, le 16/10/2023, devant le jury compose de :

Mme. L. MAHDAOUI	Professeur	à l'USTHB, Alger	Présidente
M. A. BOUKRA	Professeur	à l'USTHB, Alger	Directeur de thèse
M. K. ATIF	Maître de conférence/A	à l'USTHB, Alger	Examinateur
M. A.R. BABA ALI	Professeur	à l'USTHB, Alger	Examinateur
M. W.K. HIDOUCI	Professeur	à l'ESI, Alger	Examinateur
M. K. BOUKHALFA	Professeur	à l'ENSIA, Alger	Examinateur

Contribution à l'Optimisation du Temps de Réponse dans un Entrepôt de Données par les Approches Bio-Inspirées

MAYATA Raouf

Une thèse présentée pour l'obtention du diplôme de
Doctorat

Faculté de l'Informatique
Université de Science et de Technologie Houari Boumediene
Algerie
12 Décembre 2022

Remerciements

Je tiens tout d'abord à exprimer ma profonde gratitude envers mes parents, MAYATA Ahmed Tidjani et TOUATI BRAHIM Moufida. Leur amour inconditionnel a été le moteur qui m'a permis de persévérer face aux défis et aux obstacles rencontrés tout au long de cette aventure intellectuelle. Je leur suis infiniment reconnaissant d'avoir toujours cru en moi, même lorsque je doutais de moi-même. Merci du fond du cœur. Vous êtes mes héros, mes inspirations et mes modèles de vie, et je suis honoré de vous avoir comme parents.

Je tiens à exprimer ma sincère gratitude envers mon directeur de thèse, BOUKRA Abdelmadjid. Vos conseils éclairés et votre soutien indéfectible ont été essentiels à la réussite de cette recherche.

Je tiens à exprimer ma sincère reconnaissance envers les membres du jury. Leur engagement envers ma thèse et leur volonté de consacrer leur temps à son évaluation sont une preuve de leur générosité et de leur dévouement envers la recherche académique.

À ma petite famille, vous êtes ma plus grande richesse, ma motivation ultime et ma raison d'être. À ma bien-aimée épouse SAADI Ikram, je te suis infiniment reconnaissant pour tout ce que tu apportes à ma vie. À mon champion Ouais, ton rire contagieux et ta joie de vivre ont illuminé mes journées et m'ont rappelé l'importance de profiter de chaque instant. À ma précieuse Rym, qui nous a quittés bien trop tôt, ton souvenir restera à jamais gravé dans mon cœur. Ta présence a été un cadeau précieux, et je suis honoré d'avoir été ton père, même pour un court laps de temps. Et à mon fils Tamim, qui n'a pas encore vu le jour, tu es déjà une source infinie de bonheur et d'espoir pour notre famille. Je suis impatient de te rencontrer, de t'aimer et de te guider dans ton propre chemin.

À mes chers frères et soeurs, Nadhir, Soumia, Amina et Lamia. Nous avons partagé des moments précieux et des souvenirs inoubliables, et je suis reconnaissant de vous avoir comme frères et sœurs. Votre présence dans ma vie a été un soutien essentiel, et je vous suis infiniment reconnaissant pour votre amour et votre soutien.

J'aimerais exprimer ma profonde gratitude envers mes amis et ma famille qui ont

été présents à chaque étape de ce parcours doctoral. Je tiens également à remercier spécialement ASMA Younes, mon meilleur ami, pour sa présence constante, son écoute attentive et ses conseils précieux.

Merci tout le monde, Merci du fond du cœur.

Resumé

Un entrepôt de données est une structure qui stocke de grandes quantités de données. Ces données sont exploitées de manière optimale afin d'améliorer l'efficacité de la prise de décision. L'énorme volume de données rend les réponses aux requêtes complexes et consommatrices de temps. C'est pourquoi les vues matérialisées sont utilisées afin de réduire le temps de traitement des requêtes. Comme il n'est pas possible de matérialiser toutes les vues en raison de contraintes d'espace de stockage et de temps de maintenance, le choix du sous-ensemble de vues à matérialiser est devenu l'une des décisions cruciales dans la conception d'un entrepôt de données pour une efficacité optimale. Il existe deux variante du Problème de Sélection des Vues (PSV), à savoir le PSV à contrainte d'espace de stockage et le PSV à contrainte de temps de maintenance.La présente thèse cherche à améliorer les résultats obtenus par les méthodes de la littérature pour ces deux variantes du PSV.

Le PSV à contrainte d'espace de stockage était la première variante proposée dans la littérature.Comme le PSV est un problème NP-Difficile, plusieurs approches basées sur des heuristiques et des métaheuristiques ont été proposées. Pour cette variante, nous proposons deux approches basées sur les métaheuristiques. La première approche est basée sur la mécanique quantique (QEAM), tandis que la deuxième approche est basée sur l'évolution différentielle (QDE).

Quant à la deuxième variante du PSV, nous avons proposé une approche basée sur l'algorithme *Colliding Bodies Optimization*. Ainsi, plusieurs améliorations au niveau du calcul de la fonction objectif et la contrainte ont été proposées. Tout cela dans un nouvel algorithme que nous appelons *Quantum Colliding Bodies Optimization* (QCBO).

Après avoir proposé des approches pour les deux variantes du PSV. Nous avons traité le PSV dans un contexte multi-objectif. Pour ce faire, nous avons traité les contraintes comme des objectifs supplémentaires à améliorer avec le temps d'exécution. Nous proposons deux algorithmes pour résoudre le PSV bi-objectifs –Ces objectifs sont de réduire le coût d'exécution des requêtes et le coût de maintenance–. Ces deux algorithmes proposés sont inspirés de l'algorithme génétique, à savoir MOGA et NSGA2.

Une étude expérimentale est menée afin d'étudier l'efficacité des approches proposées. Cette expérimentation vise à mettre en relief les effets positifs des choix adoptés

dans les approches proposées et à situer les performances de ces approches par rapport aux méthodes existantes. Cette étude a conduit à dire que les approches proposées sont très compétitives par rapport aux approches existantes et que les choix adoptés dans ces approches ont un effet positif sur les résultats. De ce fait nous pensons que les solutions proposées dans cette thèse constituent une bonne base pour la résolution du Problème de Sélection des Vues.

Abstract

A data warehouse is a structure that stores large amounts of data. These data are optimally utilized to improve decision-making efficiency. The enormous volume of data makes query responses complex and time-consuming. That's why materialized views are used to reduce query processing time. Since it's not possible to materialize all views due to storage space and maintenance time constraints, the selection of the subset of views to materialize has become one of the crucial decisions in designing a data warehouse for optimal efficiency. There are two variants of the Materialized View Selection Problem (MVS), namely the storage space-constrained MVS and the maintenance time-constrained MVS. This thesis aims to improve the results obtained by existing methods for these two variants of the MVS.

The storage space-constrained MVS was the first variant proposed in the literature. Since the VSP is an NP-Hard problem, several heuristic and metaheuristic-based approaches have been proposed. For this variant, we propose two metaheuristic-based approaches. The first approach is based on Quantum-inspired Evolutionary Algorithm (QEAM), while the second approach is based on Differential Evolution (QDE).

As for the second variant of the MVS, we proposed an approach based on the Colliding Bodies Optimization algorithm. Several improvements in the calculation of the objective function and constraint have been proposed. All of this is incorporated into a new algorithm called Quantum Colliding Bodies Optimization (QCBO).

After proposing approaches for both variants of the MVS, we addressed the MVS in a multi-objective context. To do so, we treated the constraints as additional objectives to be improved along with the execution time. We propose two algorithms to solve the bi-objective VSP, where the objectives are to reduce query execution cost and maintenance cost. These two proposed algorithms are inspired by the Genetic Algorithm, namely MOGA and NSGA2.

An experimental study is conducted to evaluate the effectiveness of the proposed approaches. This experimentation aims to highlight the positive effects of the choices made in the proposed approaches and to compare their performance with existing methods. The study concludes that the proposed approaches are highly competitive compared to existing approaches and that the choices made in these approaches have a positive effect on the results. Therefore, we believe that the solutions proposed in this

thesis provide a solid foundation for solving the Materialized View Selection Problem.

Liste des tableaux

Table des figures

List of Algorithms

Table des matières

Chapitre 1

Introduction Générale

1.1 Contexte

L'entrepôt de données est l'une des diverses formes de construction et de stockage de données intégrées à utiliser dans des applications analytiques et d'aide à la décision. L'entrepôt de données contient des données collectées à partir d'une variété de ressources, souvent hétérogènes, dans un large intervalle de temps, pour répondre aux requêtes analytiques des utilisateurs. De fait, l'utilisation de l'entrepôt de données est un moyen d'intégrer des données provenant de bases de données volumineuses et incohérentes, souvent situées à différents endroits. Les rapports analytiques des entrepôts de données peuvent soutenir le processus de prise de décision des dirigeants de grandes organisations. Ce traitement analytique en ligne (OLAP) implique des défis importants tels que la grande taille de l'entrepôt de données, ainsi que la complexité des requêtes analytiques et, par conséquent, le temps de réponse très élevé de ces requêtes. Pour résoudre ce problème, plusieurs méthodes qui concernent les différentes phases de la construction de l'entrepôt de données ont été proposées. Ces méthodes sont : la fragmentation, l'indexation et la matérialisation des vues.

Une vue matérialisée n'est autre qu'une vue conventionnelle avec la particularité du stockage du résultat de la requête dans une table physique. L'exécution d'une requête sur cette table sera nettement plus rapide que son exécution sur les données brutes originales de l'entrepôt de données. Une fois matérialisée, la vue occupera un espace de stockage et devra être mise à jour périodiquement pour correspondre aux données de l'entrepôt original. Par conséquent, toutes les vues ne peuvent être matérialisées pour deux raisons principales : la contrainte d'*espace de stockage*, et la contrainte de *temps de maintenance*. Le choix du sous-ensemble de vues le plus bénéfique en termes de temps de réponse aux requêtes est connu sous le nom du problème de sélection des vues (PSV). Selon la contrainte considérée, il existe deux types de PSV. Le PSV à contrainte d'espace de stockage et le PSV à contrainte de temps de maintenance.

1.2 Problématiques de Recherche et Objectifs

La présente thèse a pour objectif de proposer des approches qui améliorent les résultats obtenus par les méthodes proposées dans la littérature, et ce pour les deux variantes du PSV.

Le PSV à contrainte d'espace de stockage était la première variante proposée pour le PSV. Dans un premier temps, plusieurs approches basées sur des heuristiques ont été proposées, notamment des approches basées sur l'algorithme glouton. Il s'agit de choisir progressivement des vues à matérialiser selon le « bénéfice par unité d'espace » de chaque vue. Malgré leur temps d'exécution plutôt rapide, les approches gloutonnes s'avèrent non extensibles. En effet, la qualité des solutions de ces approches se détériore considérablement en augmentant la taille du problème. Quand nous parlons d'extensibilité, les approches les plus répandues sont les méta-heuristiques. Elles sont généralement basées sur des phénomènes naturels ou physiques pour guider la recherche vers des régions intéressantes de l'espace de recherche. Dans cette thèse, nous proposons deux approches basées sur les méta-heuristiques pour résoudre le PSV à contrainte d'espace de stockage. La première approche est basée sur la mécanique quantique, tandis que la deuxième approche est basée sur l'évolution différentielle.

L'espace de stockage est une contrainte qui dépend fortement du *hardware*. Avec l'évolution des moyens de stockage, cette contrainte pose beaucoup moins de problèmes. Cette constatation a motivé la considération de la contrainte de temps de maintenance peu de temps après la première variante du PSV.

L'entrepôt de données se met à jour périodiquement et, étant donné que les vues matérialisées existent physiquement dans un espace séparé de l'entrepôt, elles ont besoin de maintenance à leur tour. Le temps dédié à la mise à jour de ces vues est restreint, ce qui rend le nombre de vues à matérialiser restreint également d'où la contrainte de temps de maintenance.

Cette contrainte n'est pas aussi simple que la contrainte d'espace de stockage. Même si les deux contraintes semblent similaires, la nature *non monotone* du coût de maintenance accroît la complexité des algorithmes. Nous proposons une approche basée sur l'algorithme *colliding bodies optimisation* (CBO) pour résoudre le PSV à contrainte de temps de maintenance. C'est un algorithme inspiré des lois physiques du phénomène de collision entre les objets.

La considération des contraintes lors de la résolution du PSV nécessite une connaissance a priori de la limite de ces contraintes avant même l'exécution de l'algorithme de résolution. Dans le contexte d'aide à la décision, les décideurs ne possèdent pas forcément cette information, ou seront peut-être prêts à faire des compromis s'ils estiment que c'est bénéfique. En d'autres termes, il est plus agréable de donner le choix aux décideurs plutôt que de leur demander de fixer une valeur pour chaque contrainte. Cette constatation nous oblige à voir le PSV d'un autre point de vue. Au lieu de consi-

dérer l'espace de stockage et le temps de maintenance comme des contraintes, nous les considérons comme des objectifs, tout comme le temps global d'exécution des requêtes.

Un autre objectif de cette thèse est de considérer le PSV dans un contexte multi-objectifs. Dans la littérature, rares sont les travaux qui ont essayé de résoudre le PSV multi-objectifs. Nous proposons deux algorithmes pour résoudre le PSV bi-objectifs. Ces objectifs sont de réduire le coût d'exécution des requêtes et le coût de maintenance. Les deux algorithmes proposés sont inspirés de l'algorithme génétique.

1.3 Structure de la thèse et contributions

La présente thèse est composée de six chapitres.

Le premier chapitre présente les généralités des entrepôts de données, ainsi que les différentes techniques d'optimisation dans ces entrepôts, comme l'utilisation des indexes, la fragmentation, ou les vues matérialisées qui représentent le centre d'intérêt de cette thèse.

Le deuxième chapitre introduit progressivement les différents éléments nécessaires pour expliquer le problème de sélection des vues (PSV). L'objectif du PSV est expliqué en détails, tout comme les deux contraintes. La complexité du PSV est également expliquée à la fin de ce chapitre.

Dans le troisième chapitre, nous présentons quelques notions de complexité, ainsi que les différentes techniques pour résoudre la classe NP-difficile, à laquelle appartient le PSV. Nous présentons par la suite les principales méta-heuristiques qui vont être utilisées dans nos contributions. Avant de conclure ce chapitre, nous présentons une vue d'ensemble sur les travaux réalisés dans la littérature pour résoudre le PSV.

Le quatrième chapitre se concentre sur la première variante du PSV. Nous expliquons en détails le premier travail de Harinarayan *et al.* qui a considéré le PSV en général, ainsi que le PSV à contrainte d'espace de stockage particulièrement. Quelques travaux de T.V. Vijay sont également présentés. Ensuite, nous proposons deux approches basées sur *quantum evolutionary algorithm* (QEA) et sur de l'évolution différentielle (DE). Nos deux algorithmes proposés sont nommés QEAM et QDE respectivement. Une étude expérimentale et une autre comparative sont également présentées dans ce chapitre.

Le cinquième chapitre concerne la deuxième variante du PSV, qui est le PSV à contrainte de temps de maintenance. Dans ce chapitre, nous expliquons le travail de Xu Yu *et al.*. Juste après, nous présontons notre contribution pour cette variante du PSV. Il s'agit d'une approche basée sur l'algorithme *colliding bodies optimisation* (CBO). Notre adaptation de ce dernier au PSV est appelé QCBO. Une étude expérimentale

qui justifie les différents choix et qui valide QCBO est présentée à la fin de ce chapitre.

Le sixième chapitre commence avec l'explication de la motivation qui nous a poussés à considérer le PSV dans le contexte multi-objectifs. Une brève présentation des notions de l'optimisation multi-objectives est présentée juste après. Nous appliquons deux célèbres algorithmes, MOGA et NSGA2, fortement inspirés de l'algorithme génétique pour résoudre le PSV multi-objectifs. Une comparaison entre ces deux algorithmes et QCBO sera détaillée à la fin de ce chapitre.

Enfin, nous clôturons la thèse avec une conclusion générale et quelques perspectives.

Chapitre 2

Les Entrepôts de Données

2.1 Introduction

Plusieurs études ont montré qu'une variété de professions existent depuis l'antiquité. En comparaison, la profession et les pratiques qui traitent l'information sont très récentes, voire immatures : elles datent en effet des années 1960. Une des preuves de leur immaturité est la tendance à valoriser les détails au détriment d'une vision globale et synthétique de l'information.

Pour tout sorte de business, les managers, les superviseurs et les exécutifs sont amenés à prendre des décisions. Ces décisions nécessitent un croisement d'informations précises provenant de plusieurs sources. Avec le temps, avoir une image globale rassemblant toutes ces informations est devenu un besoin incontournable, donnant naissance aux systèmes d'aide à la décision (*DSS : Decision Support Systems*). Dans un premier temps (1960), l'information était stockée dans des *fichiers maîtres*, sur des *disques magnétiques*. Le principal problème de ce moyen de stockage était l'accès à l'information ; il fallait en effet parcourir tout le disque, et ce quelque soit la position de l'information visée. Dans les années 1970, les *bases de données* (BDs) ont vu le jour avec l'avènement du *disque de stockage*, un nouveau moyen de stockage qui élimine le défaut des disques magnétiques, permettant un accès direct à la position de l'information recherchée. Une nouvelle ère a commencé avec la naissance des ordinateurs personnels (*PC : Personal Computer*) dans les années 1980. L'utilisateur final commence à avoir un rôle actif ; il a alors accès aux données et aux systèmes, ce qui était auparavant réservé aux informaticiens. Cette accessibilité a ouvert d'autres horizons : les systèmes de management d'information (*MIS : Management Information Systems*), qui étaient proposés comme méthodes d'exploitation des données pour la prise de décisions concernant des opérations détaillées. Néanmoins, une seule BD ne pouvait simultanément répondre aux besoins des opérations transactionnelles et à ceux des MIS, ce qui impose une nouvelle structure pour traiter les besoins (i.e., requêtes) analytiques.

De nos jours, les *Entrepôts de Données* sont une des structures les plus utilisées pour les DSS dirigés par l'information. Selon Bill Inmon, « *un entrepôt de données est une collection de données orientées sujet, intégrées, non volatiles, historisées, et utilisées pour supporter un processus d'aide à la décision* » [Inm02]. Contrairement à une BD, l'entrepôt de données est orienté sujet. Son objectif est donc de fournir une vue d'ensemble d'un aspect, ainsi que de simplifier l'information difficilement retrouvable au sein de sources de données hétérogènes. Ceci nécessite une *intégration* de ces sources dans une structure indépendante *non volatile*, et non sous la forme de requêtes stockées et calculées au moment de l'interrogation. L'*historisation* est un aspect nécessaire et important dans un entrepôt de données, car elle permet de suivre l'évolution de l'activité étudiée au cours du temps.

Le volume des entrepôts de données peut atteindre les téraoctets. Si un tel volume est interrogé avec des requêtes complexes, le temps de réponse deviendra très important. Pour réduire ce temps de réponse, plusieurs méthodes existent, telles que l'utilisation des *vues matérialisées*, l'*indexation*, et les *fragmentations verticale et horizontale*. Cependant, la manipulation de ces méthodes n'est pas évidente. Leurs applications évoquent souvent des problèmes combinatoires, ces derniers nécessitant des algorithmes pour choisir une meilleure solution garantissant un certain seuil d'optimisation.

Nous présentons dans ce chapitre, organisé en cinq sections, des généralités sur les entrepôts de données ainsi que les méthodes d'optimisation. La section 2.2 présente les différences entre les BDs et les entrepôts de données. La section 2.3 porte sur le modèle multidimensionnel. La section 2.4 traite des techniques d'optimisation utilisées dans les différentes phases de conception. Enfin, la section 2.5 conclue le chapitre.

2.2 Base de données *vs.* Entrepôt de données

Plusieurs entreprises utilisent les SGBDs (Systèmes de Gestion de Bases de Données) pour créer et gérer leurs BDs. Ces BDs sont considérées comme des systèmes opérationnels, utilisés pour traiter les transactions quotidiennes de l'entreprise. Les BDs contiennent des données actuelles et détaillées, provenant d'une source unique. Elles sont interrogées avec des requêtes de type OLTP (On-Line Transaction Processing), qui sont de courtes transactions de type « lecture », « insertion », « mise à jour », et « suppression ». Par souci d'efficacité, les SGBDs doivent assurer un accès facile aux données, ainsi qu'un temps de réponse court aux requêtes. Ils en reçoivent en effet un nombre important de la part des opérateurs, et ce de façon régulière. Tout en étant efficaces pour l'aspect métier, les BDs ne peuvent pas être utilisées pour assurer les requêtes analytiques dites OLAP (On-Line Analytical Processing), et ce pour plusieurs raisons. L'objectif des BDs est de servir les requêtes transactionnelles. Si une charge

supplémentaire de requêtes OLAP leur est imposée, le temps de réponse aux requêtes transactionnelles va augmenter, diminuant ainsi considérablement l'efficacité des BDs à servir leur objectif principal. Ainsi, le traitement des requêtes analytiques n'est pas évident dans l'environnement des BDs, et nécessite un administrateur expérimenté pour élaborer des requêtes OLAP exploitables par les décideurs. De plus, pour les entreprises ayant plus d'une BD, il n'est pas possible d'avoir une seule image synthétique rassemblant les informations visées dans les différentes BDs.

Ainsi, les entrepôts de données sont une structure dédiée à répondre à un petit nombre de requêtes complexes de type OLAP posées sur un grand volume de données historiques et agrégées. Ces données sont rassemblées à partir de plusieurs sources diverses à l'aide du processus ETL (Extract, Transform and Load) (Figure 1). Ce processus, comme son nom l'indique, extrait l'information brute de sa source (HTML, XML, EXCEL, BD, etc), la transforme en information exploitable d'un point de vue analytique, et la charge sur l'ED.

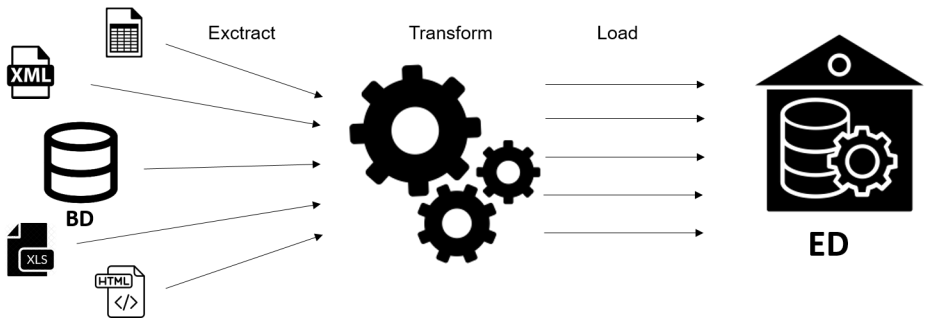


FIGURE 1 – Processus d'extraction, transformation, et chargement

La Table 1 récapitule les différences entre les BDs et les EDs.

Contrairement aux requêtes OLTP, les requêtes OLAP se basent principalement sur l'opération « sélection » afin d'analyser les données de toutes les manières possibles. Les autres opérations, e.g., « insertion » ou « mise à jour », sont exécutées périodiquement.

Tout comme pour la construction d'une BD, celle d'un ED doit suivre une architecture. Le *modèle multidimensionnel* est le mieux adapté pour la représentation des données dans un ED [KR13]. C'est un modèle qui facilite l'accès aux données, et permet de les observer selon plusieurs axes d'analyse. La section suivante explique en détails ce qu'est le modèle multidimensionnel.

2.3 Le Modèle Multidimensionnel

Le modèle multidimensionnel classe toute donnée dans une des deux catégories suivantes : *fait*, ou *dimension*. Les *faits* sont les données qui représentent le sujet analysé. Ils sont associés à des mesures, souvent numériques. Ces mesures sont extraites

TABLE 1 – *Différences entre les Bases de données et les Entrepôts de données*

	Base de données	**Entrepôt de données**
Objectif	Enregistrer l'information	Analyser l'information
Orientation	Application	Sujet
Types de données	Courantes (à jour)	Courantes et historiques (pas forcément à jour)
Types de requêtes	OLTP	OLAP
Architecture	Approche relationnelle directe	Approche multidimensionnelle
Source(s) de données	Une seule	Plusieurs
Alimentation	Directe par l'utilisateur	Par les outils d'ETL

suite à l'application d'opérations (e.g, somme, moyenne, minimum, maximum, etc) sur des données pendant la phase ETL. Les *dimensions* sont, quant à elles, les données qui caractérisent les faits. Plus particulièrement, ce sont les paramètres et les informations pouvant influencer la mesure des faits.

Pour mieux comprendre le modèle multidimensionnel, le *cube de données* est souvent utilisé. La Figure 2 représente l'exemple d'un cube de données décrivant une activité de vente de détail d'un ensemble de magasins appartenant à une entreprise.

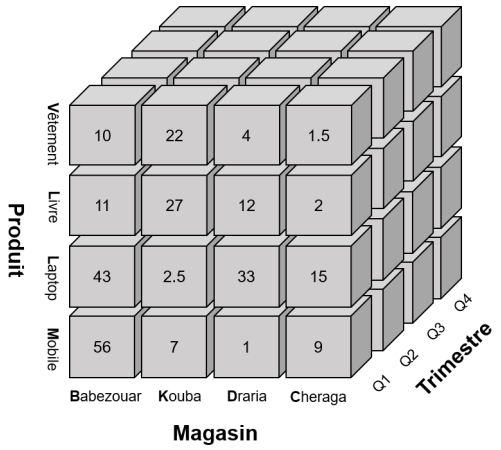


FIGURE 2 – Exemple de cube de données pour une entreprise de commerce de détail

Les valeurs inscrites sur le cube représentent les faits ; il s'agit des bénéfices de chaque vente en millions de Dinars. Les faits sont caractérisés selon trois dimensions (i.e., le temps, le produit, et le magasin). Si le nombre de dimensions dépasse 3, le cube devient un *hypercube*. Les opérations principales permettant l'analyse de ces données sont :

— *roll-up* (à ne pas confondre avec ROLAP) : cette opération consiste à effectuer une agrégation (un regroupement) et de grimper dans la hiérarchie des dimensions. La Figure 3 représente un roll-up effectué en rassemblant les informations de chaque Wilaya (Alger et Oran) de l'exemple de la Figure 2.

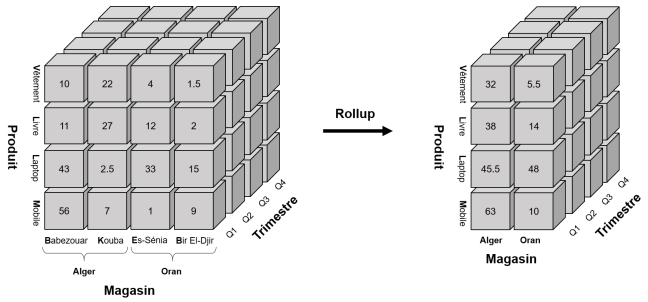


FIGURE 3 – Exemple de roll-up

— *drill-down :* c'est l'opération inverse du roll-up ; il s'agit de descendre dans la hiérarchie des dimensions pour avoir des informations plus détaillées. La Figure 4 représente un drill-down effectué sur le cube de la Figure 2, en séparant les informations qui concernent les « laptops » de celle qui concernent les « mobiles ».

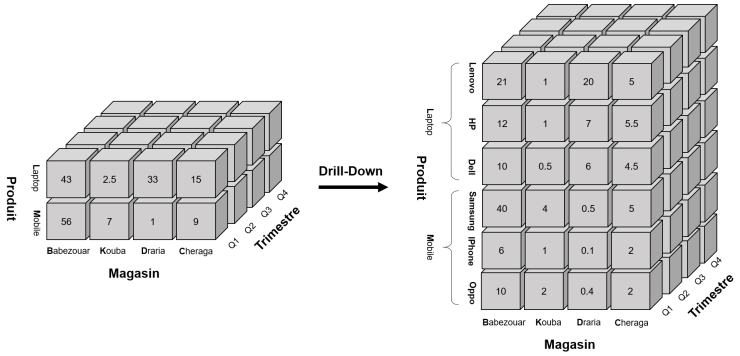


FIGURE 4 – Exemple de drill-down

— *dice :* cette opération permet d'extraire un sous-cube en fixant les dimensions sur un ensemble restreint. La Figure 5 décrit un dice effectué en fixant les valeurs de la dimension « temps » sur (Q1, Q2), de la dimension « produit » sur (laptop, mobile), et de la dimension « magasin » sur (Draria, Cheraga).
— *slice :* cette opération permet de récupérer une tranche du cube en fixant une valeur sur une dimension. Sur la Figure 6, le slice est effectué en fixant la valeur de la dimension « temps » sur (Q1).
— *rotation :* cette opération permet de pivoter une vue (cube/sous-cube/tranche) afin de proposer un nouveau point de vue.

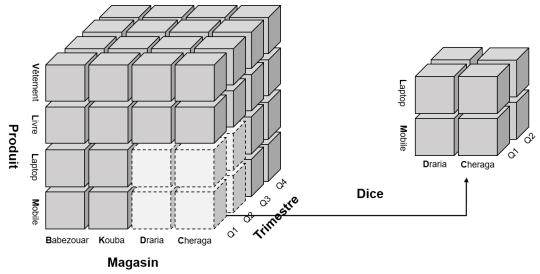


FIGURE 5 – Exemple de Dice

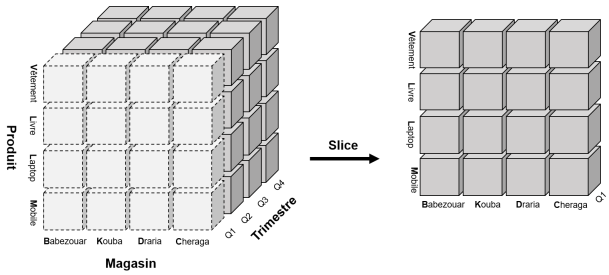


FIGURE 6 – Exemple de slice

Le modèle multidimensionnel peut être implémenté sur des SGBDs réels selon deux modèles : MOLAP(*Multidimensional On-Line Analytical Processing*) et ROLAP (*Relational On-Line Analytical Processing*).

2.3.1 Le Modèle MOLAP

Pour le modèle MOLAP, la vue multidimensionnelle donnée par l'hypercube est directement stockée dans un *tableau multidimensionnel*. En plus des données détaillées de l'ED, ce tableau stocke toutes les agrégations possibles après les avoir calculées. L'avantage de cette approche est la réduction considérable du temps d'exécution des requêtes, car l'accès aux données est direct (en O(1)). Cependant, le pré-calcul et la matérialisation (stockage) des agrégations pose un problème de ressources (temps de calcul, et espace de stockage) lors des mises à jour. Ce modèle est sollicité lorsque le volume de l'ED est réduit (c'est-à-dire, lorsqu'il ne dépasse pas les gigabytes), et la mise à jour est peu fréquente.

2.3.2 Le Modèle ROLAP

Comme son nom l'indique, le modèle ROLAP stocke les données dans des SGBDs relationnels. Les faits sont stockés dans une table appelée *table de faits*, et les dimensions sont stockées dans des *tables de dimensions*. Contrairement au modèle MOLAP, la vue multidimensionnelle du modèle ROLAP est donnée dynamiquement à travers

des requêtes sur les tables des faits/dimensions. Cette approche est moins rapide que l'approche MOLAP, mais elle élimine les problèmes des ressources lors des mises à jour.

Il existe trois variantes du modèle ROLAP : le *schéma en étoile*, le *schéma en flocon de neige*, et le *schéma en constellation*

Schéma en Étoile

Tout comme une étoile, ce schéma présente la table de faits au centre, entourée et liée à des tables de dimensions [Sil08]. La table de faits contient un nombre très important d'instances. Chaque instance se compose de clés étrangères vers les tables de dimensions, ainsi que des mesures (généralement numériques). Les requêtes d'analyse se font à travers des jointures entre la table de faits et la dimension visée en utilisant les clés étrangères présentes dans la table de faits.

En général, dans le schéma en étoile, chaque instance d'une table de dimensions contient la hiérarchie complète de ces dimensions. Par exemple, pour la dimension « temps », une instance doit contenir toute la hiérarchie d'une date (i.e., semaine, mois, trimestre, année, etc). Ceci permet l'application des opérations d'analyse (en particulier roll-up et drill-down), malgré le fait que ces dimensions soient *dénormalisées*. La Figure 7 propose un exemple d'un schéma en étoile d'une activité de vente.

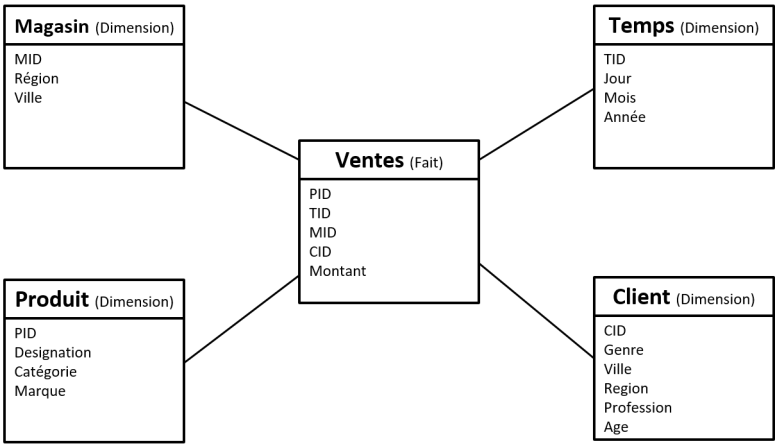


FIGURE 7 – Exemple d'un schéma en étoile

Schéma en Flocon de Neige

Parfois, une dimension - ou une partie d'une dimension - est tellement complexe ou volatile qu'elle ne peut être représentée correctement par une seule instance. Dans une telle situation, une partie de cette dimension est *normalisée* en dehors de la dimension même, résultant en une dimension de dimension, ou *sous-dimension*. Si l'on applique cette division dans les dimensions du schéma en étoile, le résultat est le *schéma en*

flocon de neige. Ce schéma a été déconseillé par Kimball [Kim97] car il est difficile à comprendre et à manipuler pour les utilisateurs.

La Figure 8 représente un exemple d'un schéma en flocon de neige, également pour une activité de vente.

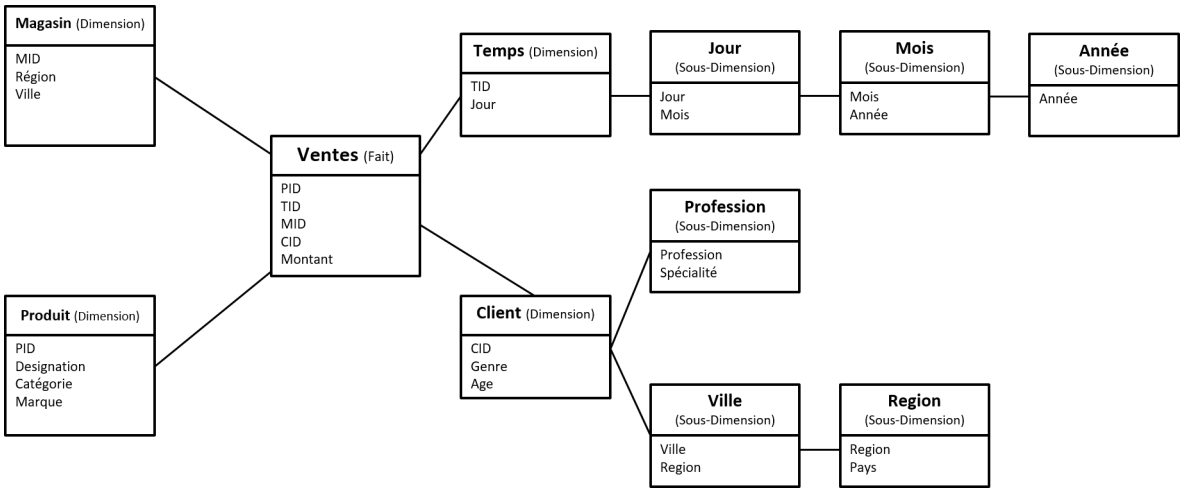


FIGURE 8 – Exemple d'un schéma en flocon de neige

Schéma en Constellation

Dans certains cas, les faits (mesures) étudiés ne sont pas exactement caractérisés par les mêmes dimensions. Il est alors possible de diviser les faits dans plusieurs tables de faits, où chacune possède son propre ensemble de dimensions. Ces dimensions sont parfois partagées entre plusieurs tables de fait, tout comme un ensemble d'étoiles, d'où l'appellation : *constellation de faits*.

2.4 Techniques d'optimisation dans les Entrepôts

Les entrepôts de données sont conçus pour répondre aux requêtes OLAP dans le but de bien analyser les données et de prendre de meilleures décisions. Étant posé sur une structure volumineuse, le temps de réponse des requêtes OLAP est important. Il est devenu le centre d'intérêt de plusieurs recherches. Il existe des techniques d'optimisation pour améliorer ce temps de réponse dans chaque phase de conception d'un entrepôt de données. Ces techniques sont : (1) *la matérialisation des vues*, (2) l'utilisation des *Index*, et (3) la *Fragmentation Verticale et Horizontale*.

2.4.1 Les Vues Matérialisées

Une vue est une requête nommée, dont le résultat est calculé lors de son exécution. Toutefois, une vue matérialisée est une vue pour laquelle nous stockons physiquement, en plus de la requête, son résultat dans une table [BDD+98]. En pré-calculant les opérations les plus coûteuses, comme la jointure et l'agrégation, les vues matérialisées améliorent considérablement le temps d'exécution des requêtes. Nous ferons une présentation détaillée sur les vues matérialisées dans le chapitre suivant.

2.4.2 Les Index

Les index sont des objets de BD, associés aux tables de cette BD, et créés pour améliorer le temps d'accès aux données de ces tables. Les techniques d'indexation ont toujours existé pour le contexte OLTP, mais quelques-unes de ces techniques sont difficilement applicables dans le contexte OLAP, car elles ne sont pas conçues pour traiter de grands volumes de données, ni les requêtes complexes et itératives souvent utilisées dans les applications OLAP. Le principe de base des index est simple. Quand un SGBD cherche un enregistrement dans une table, au lieu de faire un balayage de toute la table jusqu'à l'enregistrement voulu, il suffit de consulter la table des index qui sert comme un catalogue permettant de faciliter/accélérer l'accès aux données. Une des manières de classer les index est le classement selon le fait qu'ils soient appliqués sur une ou plusieurs tables. Selon ce principe, les indexes se divisent en deux grandes catégories :

— **Un index mono-table** est un index défini sur un ou plusieurs attribut(s) appartenant à une même table. Nous citons dans cette catégorie l'index *B-tree* (ou *B-arbre*), l'index de *hachage*, et l'index *binaire* (ou *bitmap*).

— **Un indexe multi-table** est un index défini sur plusieurs tables. Nous citons dans cette catégorie l'index de *jointure*, l'index de *jointure binaire*, et l'index de *jointure en étoile*.

2.4.3 La Fragmentation

La fragmentation est la division d'une table ou d'un objet en partitions de petite taille. Ainsi, la performance des requêtes s'améliore, car seules les données (partitions) utiles sont chargées lors des exécutions des requêtes. Il existe deux types de fragmentation :

— **La Fragmentation Verticale** consiste à partitionner la table en *fragments verticaux*, qui est une opération de projection sur cette table (vue ou index). Les fragments verticaux peuvent contenir des redondances selon le besoin. Ces

redondances sont généralement utilisés pour éviter les jointures coûteuses entre fragments.

— **La Fragmentation Horizontale** consiste à diviser une table (vue ou index) en plusieurs ensembles de lignes nommés *fragments horizontaux*, qui vérifient un même prédicat de sélection.
Il existe deux types de fragmentation horizontale : *primaire et dérivée*. La fragmentation horizontale dite *primaire* s'applique sur une table en se basant sur des prédicats de sélection de la même table. La fragmentation horizontale *dérivée* est quant à elle basée sur les prédicats de sélection d'une autre table avec laquelle il existe un lien (i.e., une clé étrangère). Une fragmentation horizontale primaire doit être appliquée sur la deuxième table. De cette façon, si un fragment primaire est sollicité, une opération de *semi-jointure* (coûtant moins qu'une jointure classique) avec le fragment dérivé sera appliquée.

2.5 Conclusion

Dans ce chapitre, nous avons introduit les notions de base des Entrepôts de Données, ainsi que les techniques d'optimisation du temps d'exécution des requêtes OLAP. L'application de chacune de ces techniques entraîne des problèmes d'optimisation, à savoir le *problème de sélection des index*, le *problème de sélection du schéma de fragmentation verticale*, et le *problème de sélection des vues matérialisées*, qui se positionne comme le centre d'intérêt de cette thèse. Le chapitre suivant explique en détails le *problème de sélection des vues matérialisées*, son origine, ses diverses variantes présentes dans la littérature, ainsi que les différences entre ces variantes.

Chapitre 3

Le Problème de Sélection des Vues

3.1 Introduction

Comme énoncé dans le chapitre précédent, les vues matérialisées sont une des méthodes utilisées pour l'optimisation du temps de réponse dans les entrepôts de données. Néanmoins, comme toute méthode d'optimisation, l'utilisation de ces vues n'est pas sans conséquence. Les vues matérialisées existant en effet comme des tables dans un SGBD, elles y occupent un espace de stockage supplémentaire, créant une contrainte de stockage lors de l'utilisation de cette méthode. Ainsi, pour correspondre aux données existantes dans l'entrepôt original, les vues matérialisées ont besoin d'être mises à jour périodiquement, ce qui crée une autre contrainte de maintenance (mise à jour) lors de l'utilisation des vues matérialisées.

Les contraintes susmentionnées rendent la matérialisation de toutes les vues possibles une solution difficile, voire impossible à réaliser. Il faut donc sélectionner un sous-ensemble à matérialiser à partir de l'univers des vues possibles. Ceci nous mène au problème de sélection des vues à matérialiser (PSV), qui consiste à choisir le sous-ensemble de vues le plus bénéfique en terme de performance de réponse aux requêtes sur l'entrepôt de données. Ainsi, ce chapitre introduit le PSV, ainsi que ses différentes variantes.

3.2 Contexte Multidimensionnel

Pour illustrer le problème de sélection des vues à matérialiser, nous allons utiliser l'exemple d'un schéma en étoile. Comme mentionné dans le chapitre 2.3, le modèle multidimensionnel divise les données en deux catégories : faits et dimensions. L'exemple de la Figure 9 est un schéma en étoile d'une table de faits (« ventes »), et de trois tables de dimensions (« magasin », « produit », et « client »). Chaque ligne de la table de faits contient la clé primaire de chaque dimension (PID, MID, et CID), ainsi que la mesure

« total ventes ».

Ce modèle calcule le total des ventes d'un produit p à un client c dans un magasin m dans le point (p, c, m). La requête SQL qui fournit le résultat du point (p, c, m) peut être écrite comme suit :

```
SELECT Produit, Client, Magasin sum TotalVentes
FROM Ventes
GROUP BY Produit, Client, Magasin
```

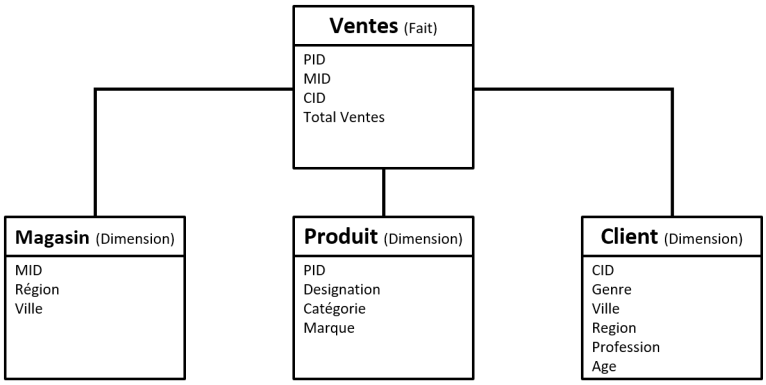


FIGURE 9 – Exemple d'un schéma en étoile à 3 dimensions

Pour obtenir plusieurs autres informations, nous utilisons des résultats agrégés, tel que le total des ventes d'un produit p dans un magasin m pour tous les clients. Pour représenter un tel point, Harinarayan *et al.* ont introduit la valeur *All* dans le domaine de chaque dimension [HRU96], ce qui rend le résultat de l'exemple agrégé retrouvable dans le point (p, *All*, m). La requête SQL qui fournit le résultat du point (p, *All*, m) peut être écrite comme suit :

```
SELECT Produit, Magasin sum TotalVentes
FROM Ventes
GROUP BY Produit, Magasin
```

Pour cet exemple, la liste de toutes les vues possibles est la suivante :

```
1. produit, client, magasin (PCM)    3. produit, magasin (PM)
2. produit, client (PC)              4. client, magasin (CM)
```

```
5. produit (P)                    7. magasin (M)
6. client (C)                     8. none
```

Nous avons ignoré les différents attributs qui existent dans chaque dimension, et considéré que les ID (clés) sont les seuls représentants des dimensions. Nous avons choisi de ne pas utiliser la valeur *All* dans la représentation des vues car elle n'apparaît pas dans les requêtes SQL. La valeur *None* correspond au total des ventes de tous les produits à tous les clients dans tous les magasins, soit le point (*All, All, All*).

Comme expliqué dans le chapitre 2.3, dans le contexte MOLAP, la vue donnée par le *cube de données* est stockée directement dans un *tableau multidimensionnel*. Dans ce contexte, la structure de *treillis* est utilisée pour représenter la hiérarchie entre les requêtes (ou les vues), où les vues candidates pour matérialisation sont représentées. La section suivante explique en détails la structure de *treillis*, proposée par Harinarayan *et al.* [HRU96].

Pour le contexte ROLAP, chaque requête est représentée par un arbre algébrique. Chaque noeud (non feuille) de cet arbre est considéré comme une vue candidate. Il existe deux modèles pour représenter l'arbre algébrique de la charge des requêtes, à savoir : le multi-view processing plan (MVPP) [YKL97], et le AND-OR view graph [GM05].

3.3 Le Treillis de Dépendance

Harinarayan *et al.* a introduit le *treillis de dépendance* (*dependent lattice*) pour exprimer la relation de dépendance entre les différentes vues d'un modèle multidimensionnel [HRU96]. Les sommets de ce *treillis* représentent les requêtes OLAP, ou bien les vues, et les arêtes représentent les liens entre ces vues.

Tout comme [HRU96], Nous définissons le *treillis de dépendance* avec un ensemble de vues V et la relation de dépendance $\preceq$ (dérivée de / pouvant être calculée à partir de). Pour deux éléments u et v appartenant à V, Nous disons que u dépend de v ($u \preceq v$), si la requête u peut être satisfaite en utilisant seulement le résultat de v. La relation d'*ancêtre* et de *descendant* sont donc définies comme suit :

$$\texttt{ancêtre(u)} = \{ v \mid u \preceq v\}$$

$$\texttt{descendant(u)} = \{ v \mid v \preceq u\}$$

Ainsi, pour une relation de dépendance ($u \preceq v$), nous disons que v est l'*ancêtre* de u, et que u est le *descendant* de v. En utilisant cette définition, nous remarquons que

chaque élément du *treillis* peut être considéré à la fois comme son propre *ancêtre*, et comme son propre *descendant*.

Le *treillis de dépendance* est représenté par un graphe orienté acyclique $G = (V, E)$. Les sommets V(G) représentent l'ensemble des requêtes (vues), tandis que les arêtes E(G) représentent les relations entre ces requêtes. Nous définissons la notion d'*ancêtre propre immédiat* comme suit :

$$\texttt{next(a) = \{ v | u} \preceq \texttt{v,} \nexists \texttt{ w, u} \preceq \texttt{w, w} \preceq \texttt{v, u}\neq\texttt{v}\neq\texttt{w \}}$$

Une arête entre deux requêtes u et v ($v \rightarrow u$) n'existe que si et seulement si v est l'*ancêtre propre immédiat* de u. De même, un chemin entre u et v n'existe que si u dépend de v.

Dans la Figure 10, nous présentons le *treillis de dépendance* de l'architecture de données présenté dans l'exemple de la Figure 9. Nous appelons $v+$ la racine (PMC) qui représente l'ED lui-même.

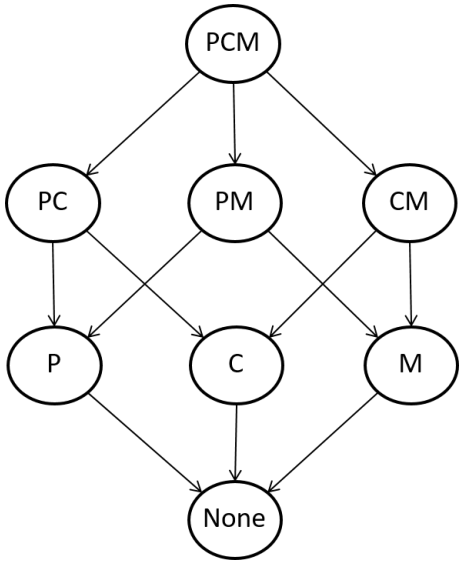


FIGURE 10 – Treillis de dépendance du schéma de la Figure 9

Pour simplifier la représentation, nous avons ignoré la hiérarchie à l'intérieur des dimensions, et considéré que les IDs sont les seuls représentants de chaque dimension.

3.4 Le Problème de Sélection des Vues

Quand nous parlons de la sélection des vues, nous nous trouvons devant trois options :

— **Matérialiser toutes les vues** : Dans le contexte MOLAP, cette approche consiste à matérialiser tous les noeuds du *treillis de dépendance*. Cette approche assure le meilleur temps de réponse pour toutes les requêtes (appelé *time optimal solution* par Harinarayan *et al.* [HRU96]). Malheureusement, cette approche n'est pas praticable. En effet, pour stocker toutes les vues, un grand espace de stockage est nécessaire ; de plus, il n'est pas possible de maintenir toutes ces vues (i.e., les mettre à jour pour correspondre à l'entrepôt original) car cela nécessite un très grand temps de maintenance.
— **Ne matérialiser aucune vue** : Pour cette approche, nous devons accéder à l'entrepôt original pour répondre aux différentes requêtes. Ceci ne présente aucun avantage en termes de performance des requêtes.
— **Matérialiser seulement une partie des vues** : Comme expliqué précédemment pour la structure de treillis, il existe une dépendance entre les différentes vues. Cette approche consiste à choisir la partie qui convient à la charge de requêtes appliquées sur l'entrepôt de données, tout en tenant compte de la dépendance qui existe entre les vues sollicitées par ces dernières.

Le PSV se résume donc au choix du sous-ensemble des vues à matérialiser à partir de l'univers des vues présentes dans le *treillis de dépendance*, qui assurent le maximum de bénéfices en termes de performance de la charge des requêtes appliquées sur l'ED, tout en respectant une contrainte de ressources S (espace de stockage ou temps de maintenance).

Nous pouvons ainsi présenter le PSV comme suit. Soient :
— T l'ensemble de tous les noeuds du treillis de dépendance.
— C une charge de requêtes.
— S une contrainte de ressources (espace de stockage ou temps de maintenance).
— TPC (Total Processing Cost) le coût total des évaluations de la charge C.

Le PSV consiste à choisir un sous-ensemble de T qui minimise le coût TPC, tout en respectant la contrainte S.

Il existe deux types de PSV :

— **Le PSV statique** : Le PSV statique consiste à choisir un sous-ensemble de T qui minimise le coût TPC, tout en respectant la contrainte S. Il faut noter que, pour le PSV statique, la charge des requêtes est connue *a priori* et n'est pas censée évoluer au cours du temps. Si un changement de la charge C survient, il faut reconsidérer le problème (reconstruire un nouveau sous-ensemble à matérialiser).

— **Le PSV dynamique** : Contrairement au PSV statique, le PSV dynamique prend en considération l'évolution de la charge des requêtes. Ce problème a été introduit par Kotidis *et al.* pour la première fois [KR99]. Ils ont proposé un algorithme intitulé *DynaMat* qui enregistre les évolutions de la charge des requêtes et matérialise dans chaque cas le meilleur ensemble de vues pour satisfaire cette charge.

Dans notre travail, nous essayons de résoudre le PSV Statique.

3.4.1 Poids Associés aux éléments du Treillis de Dépendance

Pour pouvoir calculer les différentes métriques (coût de stockage, coût total des évaluations des requêtes), nous associons des poids à chaque sommet et chaque arête du *treillis de dépendance* (Figure 11).

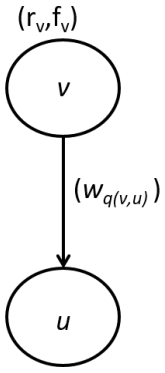


FIGURE 11 – Poids des sommets/arêtes (contrainte d'espace)

À chaque sommet $v \in V(G)$, nous associons deux poids :
— r_v : Le coût initial du scan.
— f_v : La fréquence des requêtes sur la vue v.

À chaque arête $(v, u) \in E(G)$, nous associons un poids :
— $w_{q(v,u)}$: Le coût d'évaluation de la requête u en utilisant v.

Le choix de ces poids est inspiré par le travail de Harinarayan *et al.*, qui ont proposé un modèle de coût linéaire [HRU96]. Ce modèle a largement été utilisé dans la résolution du PSV.

Il faux noter que les poids présents dans la Figure 11 concernent le PSV à *contrainte d'espace de stockage*. De même pour la *contrainte de temps de maintenance*, la Figure 12 représente les poids associés à chaque sommet/arête.

Ainsi, à chaque sommet $v \in V(G)$, nous associons trois poids :
— r_v : Le coût initial du scan.
— f_v : La fréquence des requêtes sur la vue v.
— g_v : La fréquence des mises à jour sur la vue v.

À chaque arête $(v, u) \in E(G)$, nous associons deux poids :
— $w_{q(v,u)}$: Le coût d'évaluation de la requête u en utilisant v.
— $w_{u(v,u)}$: Le coût de de la mise à jour de la vue u en utilisant la vue v.

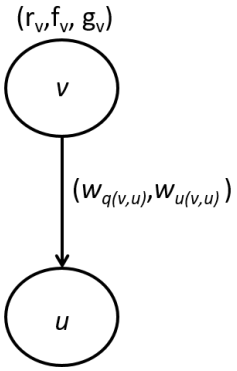


FIGURE 12 – Poids des sommets/arêtes (contrainte de temps de maintenance)

3.5 Objectifs et Contraintes

Les poids mentionnés dans la section précédente sont associés aux éléments du *treillis de dépendance* afin de calculer les différents coûts qui représentent l'objectif et la contrainte, à savoir : le TPC (*total processing cost*) des requêtes appliquées sur l'ED, le SC (*space cost*) occupé par les vues matérialisées, et le MC (*maintenance cost*) nécessaire pour la mise à jour.

3.5.1 Le Total Processing Cost

Le *total processing cost* est le coût total des évaluations d'une charge de requêtes sur l'ensemble des vues matérialisées. Pour calculer le TPC, nous définissons le DPC (*direct processing cost*) entre deux vues. Le DPC de la requête u utilisant v (noté $q(u, v)$) est la somme des coûts d'évaluation w_q de toutes les arêtes sur *le chemin le plus court* qui mène de u vers v, plus le coût initial du scan de la vue v (r_v). Si la vue v est incapable de répondre à la requête u, $q(u, v)$ sera calculé directement à partir de la vue $v+$. Le coût sera donc le *scan cost* de l'ED.

Comme mentionné auparavant, Harinarayan *et al.* ont proposé un modèle de coût linéaire en se basant sur l'idée que le temps nécessaire pour satisfaire une requête est le nombre de lignes présentes dans la vue correspondante [HRU96]. Beaucoup de modèles ont été inspirés par celui-ci, y compris le modèle TCP que nous allons implémenter dans notre travail.

Soit M (inclus dans $V(G)$) un sous-ensemble de vues à matérialiser, et $q(v, M)$ le DPC minimal de la vue v (appartient à $V(G)$) en présence du sous-ensemble M. Le TPC est défini par l'équation (3.1) :

$$TPC(G, M) = \sum_{v \in V(G)} f_v.q(v, M) \tag{3.1}$$

3.5.2 La contrainte d'espace de stockage

La première contrainte traitée pour le PSV était la contrainte de l'espace de stockage. Comme les supports ne pouvaient pas assurer le stockage de toutes les vues possibles, qui dépasse plusieurs fois la taille de l'entrepôt, une contrainte qui limite l'espace de stockage est introduite. La contrainte d'espace a été traitée de deux façons différentes :

— Fixer le nombre de vues à matérialiser, avec l'ensemble ne devant pas dépasser ce nombre. L'espace est donc identique au nombre de vues matérialisées, comme l'indique l'équation (3.2) :

$$SC_n(G, M) = k < S \tag{3.2}$$

Avec k le nombre de vues matérialisées, soit la cardinalité de M, et S le nombre de vues à ne pas dépasser pour la matérialisation.
La borne supérieure de la valeur k (pire cas pour la contrainte) est la cardinalité de $V(G)$ lors de la matérialisation de toutes les vues. Ainsi, la borne inférieure de la valeur k est 0, et ce quand aucune vue n'est matérialisée.

— Fixer la taille de stockage, ou le nombre de lignes à ne pas dépasser. L'espace de stockage est donc défini avec l'équation (3.3) :

$$SC(G, M) = \sum_{v \in M} r_v < S \tag{3.3}$$

Où l'on calcule la somme des tailles des vues présentes dans M. Dans cette équation S représente la taille (nombre de ligne) à ne pas dépasser.

Nous remarquons que la borne supérieure de la contrainte SC pour cette équation est la somme des tailles de toutes les vues de $V(G)$, et ce quand toutes les vues sont matérialisées. De même, quand aucune vue n'est matérialisée, la borne inférieure de SC est égale à 0.

3.5.3 La contrainte de temps de maintenance

Avec l'évolution des supports de stockage et la réduction de leurs coûts, la contrainte d'espace de stockage a perdu tout intérêt ; ce qui a attiré l'attention sur la deuxième contrainte, à savoir le temps nécessaire pour mettre à jour les vues pour correspondre aux données de l'ED original. Pour ce faire, il faut calculer le MC (maintenance cost), et s'assurer qu'il ne dépasse pas un seuil donné.

Le calcul du coût de maintenance (MC) ressemble au calcul du TPC. Nous définissons le *direct maintenance cost* DMC entre deux vues u et v (noté $m(u, v)$) comme la somme des coûts de maintenance (ou mise à jour) $w_u(v, u)$ de toutes les arêtes sur *le chemin le plus court* qui mène de u vers v.

Soit M (inclus dans $V(G)$) un sous-ensemble de vues à matérialiser, et $m(v, M)$ le DMC minimal de la vue v (appartient à $V(G)$) en présence du sous-ensemble M. Le MC est défini par l'équation (3.4) :

$$MC(G, M) = \sum_{v \in M} g_v.m(v, M) \tag{3.4}$$

Il faut noter que, s'il n'existe pas d'ancêtre matérialisé pour la vue v, la vue sera maintenue directement à partir de l'ED ($v+$).

3.5.4 La différence entre les contraintes

Les contraintes de l'espace de stockage et de temps de maintenance semblent être similaires. Cependant, en pratique, c'est la contrainte de temps de maintenance qui est considérée comme le véritable obstacle car elle est plus difficile a cause de la propriété *non monotone* expliquée dans la présente section.

Pour la contrainte de temps de maintenance, le coût de maintenance MC d'un sous-ensemble M dépend des vues matérialisées, et non de leurs cardinalités ou de leurs tailles. Ainsi, le coût de maintenance MC peut diminuer si l'on matérialise plus de vues, contrairement à la contrainte d'espace de stockage où le coût total ne peut qu'augmenter. Cet aspect est appelé la *propriété non-monotone*. Cette propriété a été introduite pour la première fois par Gupta *et al.*, mais elle a été traitée dans le contexte

ROLAP en utilisant le AND-OR view-graph [GM99]. Par la suite, Xu *et al.* a expliqué cette propriété dans le contexte MOLAP [YYCG03]. Tout comme Xu *et al.*, nous expliquons cette propriété avec l'exemple suivant :

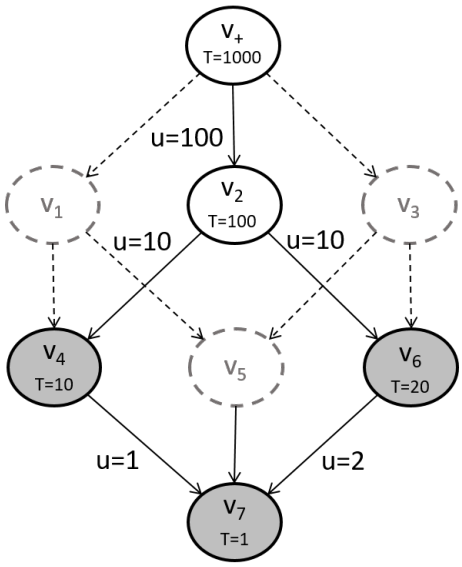


FIGURE 13 – Exemple illustrant la *propriété non monotone* du PSV à contrainte de maintenance

Soit le *treillis de dépendance* de la Figure 13. Pour un sommet v_i, T représente la taille (r_{vi}) de la vue v_i. Pour chaque arête (v_i, v_j), u représente le coût de maintenance de v_i en utilisant v_j ($w_{u(v,u)}$). Pour simplifier, nous supposons que la fréquence de mise à jour (maintenance) est égale à 1 pour toutes les vues. Les sommets en gris représentent les vues matérialisées. Ainsi, pour cet exemple, $M = \{v4, v6, v7\}$. Le coût d'espace SC de l'ensemble M est donc la somme des tailles des vues matérialisées, soit $1+10+20 = 31$. Cependant, le coût de maintenance MC est égal à $1+100+100 = 201$, car le coût de mise à jour de v_7 est de 1 (à partir de v_4), tandis que v_4 et v_6 sont mises à jour à partir du sommet $v+$ (100).

Si l'on matérialise le sommet v_2, le coût d'espace (SC) va certainement augmenter. Il sera égal à $1 + 10 + 20 + 100 = 131$. Néanmoins, le temps de mise à jour (MC) va considérablement diminuer ($1 + 10 + 10 + 100 = 121$) car, comme v_2 est matérialisé, v_4 et v_6 peuvent être maintenus à partir de v_2 avec un coût de maintenance réduit par rapport à $v+$.

Cet aspect de *non monotonie* rend le PSV à contrainte de temps de maintenance plus difficile que le PSV à contrainte d'espace de stockage.

3.6 Complexité du PSV

Après avoir introduit au préalable les objectifs et les contraintes, nous définissons ici le problème de sélection des vues formellement.

Le problème de sélection des vues (PSV) à contrainte d'espace de stockage consiste en la sélection d'un sous-ensemble de vues M qui minimise $TPC(G, M)$ sous la contrainte que le coût total de stockage soit inférieur à un seuil S. De même, le problème de sélection des vues (PSV) à contrainte de temps de maintenance ou mise à jour consiste en la sélection d'un sous-ensemble de vues M qui minimise $TPC(G, M)$ sous la contrainte que le coût total de maintenance soit inférieur à un seuil S.

Le PSV a été prouvé comme étant NP-difficile par Gupta & Mumick [GM99]. Le nombre de combinaisons possibles est très grand, avec une complexité de l'ordre de $O(2^n)$ tel que n est le nombre de toutes les agrégations possibles du schéma, i.e., si d est le nombre de dimensions du schéma, et qu'il n'existe pas de hiérarchie à l'intérieur des dimensions, $n = 2^d$. Ce très grand nombre de combinaisons rend les méthodes exactes cherchant la solution optimale non applicables, surtout pour les schémas à grand nombre de dimensions.

3.7 Conclusion

Dans ce chapitre, nous avons défini le problème de sélection des vues ainsi que sa position dans le contexte multidimensionnel. Nous avons introduit la structure de *treillis de dépendance* qui représente la hiérarchie existante entre les différentes vues d'un schéma. Ainsi, nous avons présenté les PSV statique et dynamique, et expliqué que notre travail se situe dans le cadre du PSV statique. Les différentes variantes du PSV ont également été présentées, à savoir : les PSV à contrainte d'espace de stockage, et à contrainte de temps de maintenance. La différence entre les deux contraintes (propriété de monotonie) a été expliquée à travers un exemple. Comme le PSV est NP-difficile, plusieurs techniques d'optimisation ont été appliquées pour le résoudre. Le chapitre suivant expliquera les différentes techniques existant dans la littérature pour traiter les problèmes NP-difficile.

Chapitre 4

Résolution des Problèmes d'optimisation

4.1 Introduction

Nous avons mentionné dans le chapitre précédent que le PSV est un problème *NP-difficile*. Nous allons expliquer dans ce chapitre les différentes catégories de problèmes d'optimisation, et où se situe la classe des problèmes NP-difficile parmi ces catégories. Nous allons également présenter les méthodes et approches connues dans la littérature pour traiter cette catégorie de problèmes. Nous présenterons par la suite quelques métaheuristiques qui seront utilisées dans notre contribution pour résoudre le PSV.

4.2 Complexité des Problèmes

Un problème d'optimisation est un problème pour lequel il existe plusieurs solutions. Ces solutions sont évaluées selon une *fonction objectif*. L'objectif que l'on cherche quand nous souhaitons résoudre un problème d'optimisation est de trouver la (ou les) meilleure(s) solution(s) dans l'univers des solutions possibles à travers un algorithme adéquat. Pour résoudre un problème, un algorithme a besoin de deux ressources importantes : le temps et l'espace d'exécution. La complexité du temps d'un algorithme est définie par le nombre d'étapes nécessaires pour résoudre un problème de taille n. La complexité d'un algorithme est souvent définie selon l'analyse du pire cas possible. L'annotation la plus utilisée pour décrire la complexité d'un tel cas pour un algorithme est O (grand O).

4.2.1 Problème de décision vs. Problème d'optimisation

Les problèmes d'optimisation sont des problèmes qui cherchent à trouver une solution optimale dans l'univers des solutions. Il répondent généralement à des questions comme : « Quelle est la (ou les) solution(s) optimale(s) selon une fonction objectif f pour résoudre un problème ? ». Cependant, les problèmes de décision sont des problèmes dont les questions associées sont du type : « vrai ou faux ». Tout problème d'optimisation peut être réduit à un problème de décision [Tal09]. Pour ce faire, il faut poser une question telle que : « La solution s est-elle optimale ? ».

Selon leurs complexités, les problèmes peuvent être divisés en plusieurs classes, à savoir : les classes P, NP, et NP-complet. Il faut noter que ce classement est propre aux problèmes de décision.

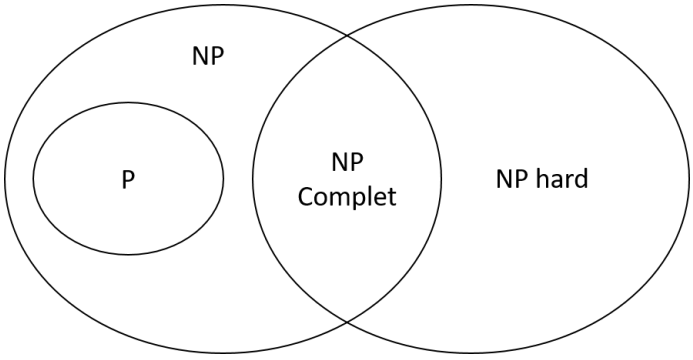


FIGURE 14 – Les classes P, NP, NP complet et NP difficile

4.2.2 La Classe de Complexité P

La classe de complexité P regroupe tous les problèmes pour lesquels un algorithme *déterministe*, de complexité $O(p(n))$ tel que $p(n)$ est une fonction polynomiale, existe pour les résoudre. Dans cette classe, l'évolution de la taille n du problème n'entraîne pas un grand changement du temps d'exécution des algorithmes. C'est pour cela que les problèmes de cette classe sont considérés comme des problèmes *faciles* à résoudre.

4.2.3 La Classe de Complexité NP

La classe de complexité NP regroupe les problèmes pour lesquels il n'existe pas d'algorithme déterministe polynomial, mais un algorithme *non-déterministe* polynomial pour les résoudre. Un algorithme *non-déterministe* est un algorithme dont une (ou plusieurs) instruction(s) consiste(nt) en un choix entre plusieurs *chemins* (ou continuations), sans spécifier quel chemin va prendre l'algorithme durant son exécution. S'il existe une machine capable d'exécuter ces instructions, la complexité de l'algorithme

sera polynomiale. Tout problème de la classe P est inclus dans la classe NP [Tal09] (Figure 14).

4.2.4 La Classe de Complexité NP Complet

Un problème de décision A est dit *réductible* en un problème de décision B s'il existe un algorithme qui peut transformer toute instance de A en une instance de B. Un problème A est dit NP-complet si tout problème de la classe NP peut être *réduit* à ce problème avec un algorithme *polynomial* [Tal09]. Si un problème NP-complet est résolu avec un algorithme déterministe polynomial, alors tous les problèmes de la classe NP seront résolus en un temps polynomial.

4.2.5 La classe NP-Difficile

Un problème est dit NP-difficile s'il est associé à un problème de décision faisant partie de la classe NP-complet. La majorité des problèmes d'optimisation du monde réel dans les différentes spécialités sont des problèmes NP-difficile. La section suivante présente les différentes méthodes de résolution de ces problèmes.

4.3 Les méthodes de résolution d'un problème NP-difficile

Les problèmes NP-difficile sont des problèmes dont les algorithmes de solution nécessitent un temps exponentiel pour trouver une solution optimale. Ainsi, pour résoudre ces problèmes, il faut donc parcourir tout l'espace de recherche si la solution optimale est visée en utilisant les méthodes exactes ; ou parcourir une sous-partie de cet espace pour chercher une solution quasi-optimale en utilisant les méthodes approchées.

4.3.1 Les Méthodes Exactes

Ces méthodes garantissent une solution optimale. Néanmoins, la complexité de ces méthodes est toujours *exponentielle*. Nous trouvons dans cette classe de méthodes quatre types d'algorithmes [Tal09] : la programmation dynamique, la programmation par contrainte, la famille *branch and X* (*branch and bound*, *branch and cut*, *branch and price*), et la famille des algorithmes A^*. Ces méthodes partagent le principe de la division du problème en sous-problèmes plus simples. Dans le domaine de l'intelligence artificielle, la méthode la plus sollicitée de cette classe était l'algorithme A^*. Mais, comme les autres méthodes, l'algorithme A^* ne peut pas être utilisé pour les grandes instances des problèmes d'optimisation.

4.3.2 Les Méthodes Approchées

Dans cette classe, nous cherchons à trouver une *bonne* solution en un temps raisonnable. Elle se divise principalement en deux catégories : les *algorithmes d'approximation* et les *heuristiques*. Les algorithmes d'approximation sont une conséquence à la conjecture répandue que $P \neq NP$. Ils cherchent à trouver une solution et à prouver sa qualité, i.e. : sa distance de la solution optimale, avec un algorithme polynomial. Les heuristiques trouvent des « bonnes » solutions en un temps raisonnable pour un grand nombre de problèmes d'optimisation [Tal09]. Elles sont également divisées en deux catégories : les *heuristiques spécifiques* et les *métaheuristiques*. Comme leur nom l'indique, les *heuristiques spécifiques* sont destinées à résoudre des problèmes spécifique. Elles ne sont généralement pas applicables pour plusieurs problèmes. Au contraire, les *métaheuristiques* sont à usage général : elles peuvent être appliquées sur tous les problèmes d'optimisation, ou presque.

4.4 Présentation de Quelques Métaheuristiques

Dans cette section, nous allons présenter quelques métaheuristiques utilisées dans la littérature pour résoudre le PSV, à savoir : l'algorithme glouton et l'algorithme génétique. Nous allons également présenter les métaheuristiques qui seront utilisées dans notre contribution à la résolution du PSV.

4.4.1 L'algorithme Glouton

L'algorithme glouton a été introduit pour la première fois en 1971 par J. Edmonds [Edm71]. Il commence avec une solution vide, et construit progressivement une solution en choisissant une variable à la fois jusqu'à avoir une solution complète. Le choix du nouveau élément à ajouter à la solution dans chaque étape est fait selon une *heuristique locale*, qui cherche à trouver l'élément avec le meilleur bénéfice à un instant t de l'exécution de l'algorithme. Une fois choisi, un élément ne peut être supprimé car l'algorithme glouton ne permet pas de retour en arrière. Généralement, l'algorithme glouton est déterministe, i.e. : nous obtenons le même résultat pour une même instance du problème quelque soit le nombre de fois que l'algorithme est exécuté. L'Algorithme 1 représente le pseudo code de l'algorithme glouton.

4.4.2 L'algorithme Génétique

L'algorithme génétique (GA) est l'un des algorithmes les plus connus et les plus implémentés des métaheuristiques. Il a été introduit au cours des années 1970 par John Holland et ses étudiants à l'Université du Michigan [Hol75]. GA fait partie de la

Algorithm 1: Pseudo code de l'algorithme glouton

$s = \{\}$; * solution initiale vide *\
repeat
  $e \leftarrow$ le nouveau élément sélectionné avec l'heuristique locale ;
  $s = s \cup e$;
until *une solution complète est obtenue*;

famille des *algorithmes évolutionnaires* (EA). C'est une famille de métaheuristiques à base de populations. Inspirés par la théorie de l'évolution de Darwin, les EAs essaient d'améliorer la qualité des *individus* (solutions) jugés par leur *fitness* (fonction objectif) au cours des *générations* (itérations de l'algorithme). Pour ce faire, ils utilisent principalement trois opérations pour produire une génération (Figure 15) : la *sélection*, la *reproduction*, et le *remplacement* [Tal09]. La sélection consiste à choisir, à partir de l'ensemble des parents, la liste des individus qui vont être utilisés pour la reproduction. Ces individus choisis se reproduisent en utilisant des opérateurs comme le *cross-over* et la *mutation*. Nous choisissons par la suite les individus qui vont survivre à partir de la liste des parents et des nouveau descendants, connus par le remplacement.

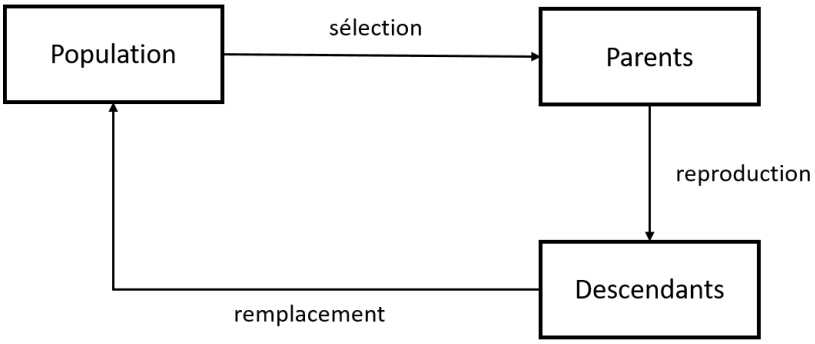


FIGURE 15 – Les opérateurs utilisés dans une génération d'EAs

Comme mentionné précédemment, les algorithmes génétiques (GAs) sont la sous-classe la plus populaire des EAs. Ils sont souvent utilisés pour les problèmes dont la représentation est binaire.

Le GA commence avec un ensemble d'individus appelé *population*. Chaque individu représente une solution au problème à résoudre. Un individu est caractérisé par des variables appelées *gènes*. Les gènes sont joints en une chaîne pour former un chromosome qui correspond à une solution (Figure 16).

La *fitness* est une fonction qui détermine la qualité de chaque individu, i.e. : dans quelle mesure un individu peut rivaliser avec le reste de la population. Un *fitness score*

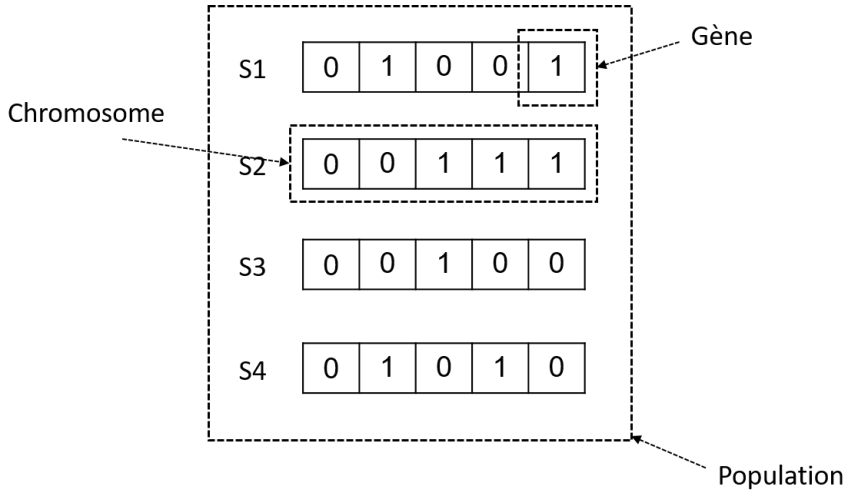


FIGURE 16 – Gène, chromosome, et population

est associé à chaque solution. Ce score est utilisé par la suite lors de la phase de *sélection*.

L'idée de la phase de *sélection* est de choisir les individus les plus aptes et de les laisser transmettre leurs gènes à la génération suivante. Deux individus (parents) sont sélectionnés en fonction de leurs *fitness scores*. Les individus avec un *fitness score* élevé ont plus de chances d'être sélectionnés pour la reproduction.

Le *cross-over*, ou *croisement*, est l'opération la plus importante d'un GA. Il existe plusieurs types de *croisements*. Un des types les plus utilisés est le *croisement uniforme*. Dans ce cas, chaque *gène* provient d'un des deux parents avec une probabilité égale, ou les probabilités sont ajustées en fonction du *fitness score*. Après croisement, de nouveaux descendants sont générés et ajoutés à la population.

Certains gènes des nouveaux descendants formés peuvent subir une *mutation* avec une faible probabilité aléatoire. Cela implique que ces gènes vont être modifiés (dans le cas d'une représentation binaire, les bits sont inversés).

Comme la population doit être de taille fixe, l'opération de *remplacement* est appliquée. Les parents avec un *score* faible sont remplacés par des descendants ayant un meilleur *score*, en appliquant le principe *« survival for the fittest »*.

Toutes les phases précédentes sont répétées jusqu'à ce qu'un critère d'arrêt soit atteint (par exemple : nombre de génération fixe). Le pseudo code du GA est donné dans l'Algorithme 2.

4.4.3 L'évolution Différentielle

L'algorithme de l'évolution différentielle (DE) est un des algorithmes les plus populaires pour les problèmes d'optimisation à espace de recherche continu. Il a été proposé

Algorithm 2: Pseudo code de l'algorithme génétique

Génération de la population initiale ;
Calcul des *fitness* ;
repeat
  Sélection ;
  Croisement ;
  Mutation ;
  Calcul des *fitness* ;
  Remplacement des parents de moindre qualité ;
until *critère d'arrêt est atteint*;

par Storn et Price dans les années 90 [SP97]. Tout comme l'algorithme génétique, le DE fait partie de la classe des algorithmes évolutionnaires. Dans un premier temps, il n'a pas été introduit en tant que nouvel algorithme évolutionnaire, mais comme variante du GA [Sim13].

Dans le DE, les individus sont des vecteurs à n-dimensions qui représentent une solution du problème. Le principe du DE est de prendre trois vecteurs aléatoires, de calculer la différence entre deux vecteurs, et de rajouter une version mise à l'échelle de cette différence au premier vecteur pour obtenir une solution candidate. L'exemple de la Figure 17 [Sim13] représente le principe du DE appliqué sur un vecteur à deux dimensions.

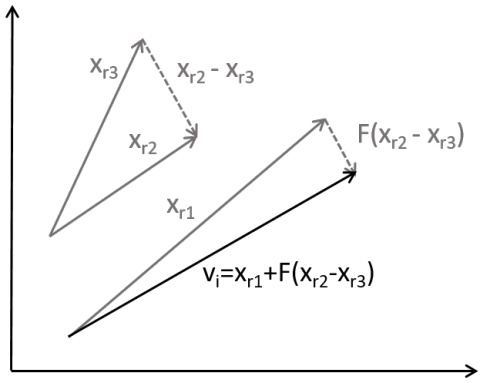


FIGURE 17 – Exemple du DE sur des vecteurs à deux dimensions [Sim13]

Comme tout algorithme évolutionnaire, le DE commence avec une population initiale $X = \{x_1, x_2, x_3, ..., x_N\}$. Pour chaque individu dans cette population, trois phases sont exécutées :

— **Génération du vecteur mutant :**

Pour chaque individu x_i, un vecteur mutant v_i est généré en utilisant trois individus aléatoires différents x_{r1}, x_{r2}, et x_{r3} autres que x_i, suivant la formule de l'équation 4.1 :

$$v_i = x_{r1} + F(x_{r2} - x_{r3}) \tag{4.1}$$

tel que F est un nombre réel utilisé pour la mise à l'échelle de la différence entre x_{r2} et x_{r3}.

— **Génération du vecteur d'essai :**

Le vecteur mutant v_i est ensuite combiné (croisé) avec l'individu x_i lui-même pour générer un vecteur d'essai u_i. Le croisement se fait comme suit (formule 4.2) :

$$u_{ij} = \begin{cases} v_{ij} & : if(rand < c) or(j = J_r) \\ x_{ij} & : sinon \end{cases} \tag{4.2}$$

Pour chaque j dans $[1, n]$, où n est le nombre de dimensions des vecteur u_i, v_i et x_i ; u_{ij} est la $j^{ème}$ dimension de u_i ; v_{ij} est la $j^{ème}$ dimension de v_i ; x_{ij} est la $j^{ème}$ dimension de x_i ; $rand$ est un nombre aléatoire tiré d'une distribution uniforme dans $[0, 1]$; c est le taux de croisement constant (nombre réel entre $[0, 1]$) ; et J_r est un entier aléatoire issu d'une distribution uniforme dans $[1, n]$. J_r est utilisé pour s'assurer que le nouveau vecteur d'essai n'est pas un clone de x_i.

— **La sélection :**

Une fois le vecteur d'essai u_i créé, il est comparé à l'individu x_i, et le vecteur le plus adapté de chaque paire (x_i, u_i) est conservé à la prochaine génération d'algorithme DE. Le moins *fit* est rejeté (*survival for the fittest*).

L'algorithme basique du DE d'un problème à n-dimensions est donné dans l'Algorithme 3 [Sim13].

4.4.4 QEA "Quantum Evolutionary Algorithm"

Le *quantum computing* est une nouvelle théorie issue de la fusion de l'informatique et de la mécanique quantique. Son objectif principal est d'étudier toutes les possibilités qu'un ordinateur pourrait avoir s'il suivait les lois de la mécanique quantique. Comme certains effets de la mécanique quantique ne peuvent pas être simulés efficacement sur un ordinateur, plusieurs recherches sur la fusion de l'informatique évolutionnaire et l'informatique quantique ont été menées depuis la fin des années 1990. Ceci a donné naissance à des algorithmes évolutionnaires, caractérisés par certains principes de la

Algorithm 3: Pseudo code de l'algorithme DE

F = step size parameter $\in [0.4; 0.9]$;
c = cross over rate $\in [0.1; 1]$;
Initialize a population of candidate solutions x_i for $i \in [1, N]$;
while *stop criterion not reached* **do**
    foreach *individual* $x_i, i \in [1, N]$ **do**
        $r_1 \leftarrow$ random integer $\in [1, N] : r_1 \neq i$;
        $r_2 \leftarrow$ random integer $\in [1, N] : r_2 \notin \{i, r_1\}$;
        $r_2 \leftarrow$ random integer $\in [1, N] : r_3 \notin \{i, r_1, r_2\}$;
        $v_i \leftarrow x_{r1} + F(x_{r2} - x_{r3})$; * mutant vector *\
        $J_r \leftarrow$ random integer $\in [1, n]$;
        foreach *dimension* $j \in [1, n]$ **do**
            $r_c j \leftarrow$ random number $\in [0, 1]$;
            if *(*$r_c j < c$*) or (*$j = J_r$*)* **then**
                $u_{ij} \leftarrow v_{ij}$;
            else
                $u_{ij} \leftarrow x_{ij}$;
            end
        end
    end
    foreach *index* $i \in [1, N]$ **do**
        if $fitness(u_i) < fitness(x_i)$ **then**
            $x_i \leftarrow u_i$;
        end
    end
end

mécanique quantique. Un des algorithmes les plus utilisés est le QEA (*quantum-inspired evolutionary algorithm*) proposé par Han *et al.* [HK02].

La plus petite unité d'information du quantum computing est appelée *q-bit*. Chaque q-bit est défini par deux nombres complexes α et β vérifiant : $\alpha^2+\beta^2 = 1$. α^2 représente la probabilité que le q-bit soit à l'état 0, tandis que β^2 représente la probabilité que le q-bit soit à l'état 1. Puisque QEA fait partie des EAs, il utilise une population pour rechercher la solution optimale. Cependant, contrairement à d'autres EAs, les individus de la population ne sont pas des solutions candidates mais des *quantum vectors*. Un *quantum vector* est un vecteur de longueur n (n étant la longueur des solutions), et est composé de q-bits comme illustré sur la Figure 18.

Les vecteurs de solutions sont générés à l'aide de l'opérateur de *measurement*, qui sera discuté dans la section suivante. Cela donne à QEA l'avantage de pouvoir représenter une superposition linéaire d'états, puisqu'un seul quantum vector peut générer plusieurs solutions, d'où une bonne caractéristique de diversité de la population.

$$\text{Quantum vector} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \dots & \alpha_n \\ \beta_1 & \beta_2 & \beta_3 & \dots & \beta_n \end{bmatrix}$$

q-bit

FIGURE 18 – Les individus de la population de QEA (*quantum vectors*)

Les opérateurs quantiques

Afin d'assurer les différentes phases d'un algorithme évolutionnaire, le QEA utilise les opérateurs quantiques.

— **Le measurement :**
L'opérateur *measurement* (aussi appelé *observation*) génère un vecteur de solutions en en choisissant une parmi l'ensemble des solutions de la superposition existante dans le *quantum vector*. Pour chaque q-bit, un nombre aléatoire *rand* est généré à partir d'une distribution uniforme entre [0, 1] pour chaque position *j*. Si *rand* est supérieur à $|\beta|^2$, alors la position de la solution j (correspondante à ce β) prend la valeur 1, sinon elle prend la valeur 0. Cette étape est appliquée sur tous les q-bits d'un *quantum vector* afin d'obtenir le vecteur solution (Figure 19). L'opérateur de mesure peut être considéré comme un opérateur d'exploration, car il peut générer des solutions différentes lorsqu'il est utilisé plusieurs fois sur le même *quantum vector*.

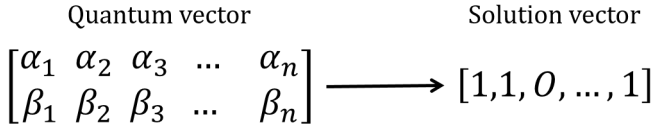


FIGURE 19 – L'opérateur de *measurement* ou d'*observation*

— **Q-gate :**
Cet opérateur est utilisé pour améliorer la qualité des individus en les orientant vers la meilleure solution globale ou locale. Ceci est accompli en effectuant une rotation dont l'angle est déduit à partir des deux valeurs α et β (Figure 20). Le choix de l'angle de rotation $\delta\theta$ est très important. Une valeur mal choisie peut en effet conduire à une convergence prématurée ou divergence très tardive. La direction de l'angle de rotation dépend de deux facteurs : la valeur du bit correspondant à la meilleure solution, et le signe des valeurs α et β. La Table 2 de [Lay11] résume tous les cas possibles des sens de rotation de l'angle $\delta\theta$.

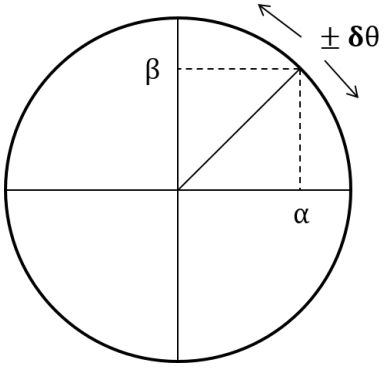


FIGURE 20 – *Angle de Rotation*

— **La Migration :**
Comme la meilleure solution de chaque individu est stockée, la Migration est le processus d'échange des meilleurs solution entre les individus. Elle se divise en : *Migration Globale* et *Migration Locale*. Dans la *Migration Globale*, toutes les meilleures solutions locales trouvées pour chaque individu sont remplacées par la meilleure solution trouvée depuis le début de l'algorithme (*global best solution*). Quant à la *Migration Locale*, quelques meilleurs solutions sont remplacées par la meilleure d'entre elles.

— **La mutation :**
Layeb [Lay11] a introduit l'opérateur de mutation au QEA pour assurer un bon taux d'exploration. Avec une certaine probabilité *inter q-bit* mutation ou *intra*

α	β	Best solution bit value	Angle
> 0	> 0	1	$+ \delta\theta$
> 0	> 0	0	- $\delta\theta$
> 0	< 0	1	- $\delta\theta$
> 0	< 0	0	$+ \delta\theta$
< 0	> 0	1	- $\delta\theta$
< 0	> 0	0	$+ \delta\theta$
< 0	< 0	1	$+ \delta\theta$
< 0	< 0	0	- $\delta\theta$

TABLE 2 – Table de consultation des orientations possibles de l'angle $\delta\theta$

q-bit mutation est appliquée (Figure 21).

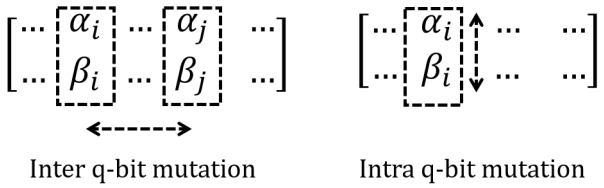


FIGURE 21 – Mutation inter et intra q-bit

Pour le *quantum vector* choisi, nous appliquons la mutation *inter q-bit* en sélectionnant deux q-bits aléatoires et en les échangeant. Cependant, pour la mutation *l'intra q-bit*, un q-bit aléatoire est sélectionné et une permutation entre ses amplitudes α et β est effectuée.

Le pseudo code de l'algorithme QEA est donné par l'Algorithme 4.

4.4.5 CBO « Colliding Bodies Optimisation »

L'algorithme Colliding Bodies Optimisation (CBO) est une métaheuristique à base de population. Inspiré du concept de la collision entre les objets, il a été proposé pour la première fois par Kaveh et Mahdavi [KM14] pour des problèmes d'optimisation à espace de recherche continu. Après la collision de deux objets en mouvement avec des masses et des vitesses spécifiées, ils sont séparés par de nouvelles vitesses (Figure 22). Ce phénomène physique obéit à des lois spécifiques. Par exemple, l'élan (*momentum*) doit rester le même après la collision.

Algorithm 4: Pseudo code de l'algorithme QEA

$t \leftarrow 0$;
Initialize a population of *quantum vectors* $Q(t)$;
Initialize a population of candidate solutions $P(t)$ by applying *measurement* operator on $Q(t)$;
Evaluate $P(t)$;
Add the best solution among $P(t)$ to $B(t)$;
while *stop criterion not reached* **do**
    $t \leftarrow t + 1$;
    Evaluate $P(t)$;
    Update $Q(t)$ using *Q-gate* ;
    Add the best solution among $P(t)$ and $B(t-1)$ to $B(t)$;
    $b \leftarrow$ global best solution ;
    if *Migration Condition* **then**
        Apply global or local migration ;
    end
end

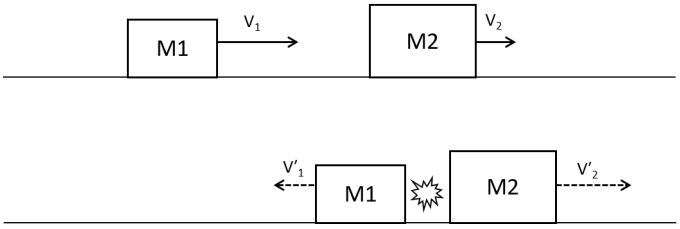


FIGURE 22 – Collision entre deux objects

Quelques principes physiques utilisés dans CBO

Si l'on considère deux objets O_1 (de masse m_1 et de vitesse v_1) et O_2 (de masse m_2 et de vitesse v_2), l'élan total des deux objets avant et après collision doit être le même. Il est donné par l'équation suivante (4.3) :

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2' \tag{4.3}$$

De même, la conservation de l'énergie cinétique totale est exprimée comme suit (4.4) :

$$\frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 = \frac{1}{2} m_1 v_1'^2 + \frac{1}{2} m_2 v_2'^2 + Q \tag{4.4}$$

tel que v_1' est la nouvelle vitesse de l'objet O_1 après collision ; v_2' est la nouvelle vitesse de l'objet O_2 après collision ; et Q est la perte en énergie cinétique due à l'impact. Les nouvelles vitesses v_1' et v_2' sont données par les équations 4.5 et 4.6 :

$$v_1' = \frac{(m_1 - \varepsilon m_2)v_1 + (m_2 + \varepsilon m_2)v_2}{m_1 + m_2} \tag{4.5}$$

$$v_2' = \frac{(m_2 - \varepsilon m_1)v_2 + (m_1 + \varepsilon m_1)v_1}{m_1 + m_2} \tag{4.6}$$

tel que ε est le coefficient de restitution (CoR) des deux objets de collision, défini comme le rapport de la vitesse relative de séparation sur la vitesse relative d'approche (équation 4.7) :

$$\varepsilon = \frac{|v_2' - v_1'|}{v_2 - v_1} = \frac{v'}{v} \tag{4.7}$$

Si la valeur de ε est a 1, La collision est dite *élastique* et la perte en énergie cinétique $Q = 0$. Néanmoins, pour la majorité des objects reels, la collision est *inélastique*. En effet, une collision est dite *inélastique* si : $\varepsilon \leq 1$ et $Q \neq 0$. L'équation 4.8 est utilisée dans l'algorithme du CBO.

$$\varepsilon = 1 - \frac{iter}{iter_{max}} \tag{4.8}$$

Tel que *iter* est l'itération en cours, et $iter_{max}$ est le nombre total des itérations prédéfini. Cette équation assure que ε est presque élastique dans un premier temps, et inélastique à la fin de l'exécution.

Le principe de CBO

Dans CBO, chaque solution candidate x_i est considérée en tant que corps en collision (CB : *colliding body*). La population de ces CBs est divisée en deux groupes principaux de cardinalité égale : le groupe *stationnaire* et le groupe *en mouvement*. Les CBs en mouvement se déplacent pour suivre les CBs du groupe stationnaire et entrer en collision par paire avec ces derniers (Figure 23). Ces collisions ont pour buts (1) d'améliorer la position des CBs du groupe en mouvement, et (2) faire bouger les CBs du groupe stationnaire vers de meilleures positions. Après la collision, les nouvelles positions des CBs sont mises à jour en fonction de la nouvelle vitesse en utilisant les lois de collision.

Selon [KM14], la procédure CBO peut être décrite comme suit :

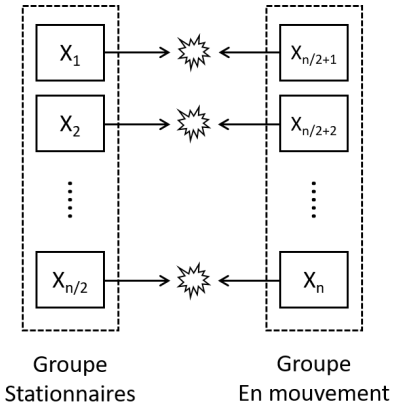


FIGURE 23 – Collision entre le groupe stationnaire et le groupe en mouvement

1. Les positions initiales des CBs sont déterminées avec une initialisation aléatoire d'une population d'individus de taille n.
2. L'ampleur de la masse corporelle pour chaque CB est définie avec l'équation suivante (4.9) :

$$m_k = \frac{\frac{1}{fit(k)}}{\sum_{i=1}^{n} \frac{1}{fit(i)}} \quad , k = 1, 2, 3...n \tag{4.9}$$

   où $fit(i)$ représente la valeur de la fonction objectif du corps i, et n est la taille de la population. Il semble qu'un CB avec une bonne valeur de *fitness* exerce une masse plus importante qu'un CB avec une mauvaise valeur. On note que la formule (4.9) concerne les problèmes de minimisation. Pour les problèmes de maximisation, la fonction objectif $fit(i)$ sera remplacée par $\frac{1}{fit(i)}$.
3. Les CBs sont triés dans un ordre croissant selon leurs masses, et divisés en deux groupes de taille égale à savoir :
   — La moitié inférieure des CBs (CBs stationnaires) : ces CB sont de bonnes solutions qui sont stationnaires. La vitesse de ces objets avant collision est nulle, donc :

$$v_i = 0 \quad , i = 1, 2, 3, ..., \frac{n}{2} \tag{4.10}$$

   — La moitié supérieure des CB (CB en mouvement) : ces CBs se déplacent vers la moitié inférieure. Ensuite, les meilleurs et les pires CBs de chaque groupe entreront en collision. Le changement de position des CBs représente la vitesse de ces CBs avant la collision comme :

$$v_i = x_i - x_{i-\frac{n}{2}} \quad , i = \frac{n}{2} + 1, ..., n \tag{4.11}$$

où v_i et x_i sont respectivement la vitesse et le vecteur de position du $i^{e`me}$ CB dans ce groupe, et $x_{i-\frac{n}{2}}$ est le vecteur de position du $i^{e`me}$ CB stationnaire, soit le vis-à-vis du CB.

4. Après la collision, les vitesses des objets en collision dans chaque groupe sont évaluées à l'aide des équations (4.5) et (4.6), et de la vitesse avant collision. La vitesse de chaque CB en mouvement après la collision est obtenue par :

$$v'_i = \frac{(m_i - \varepsilon m_{i-\frac{n}{2}})v_i}{m_i + m_{i-\frac{n}{2}}} \quad , i = \frac{n}{2} + 1, ..., n \tag{4.12}$$

Ainsi, la nouvelle vitesse de chaque CB stationnaire est obtenue par :

$$v'_i = \frac{(m_{i+\frac{n}{2}} + \varepsilon m_{i+\frac{n}{2}})v_{i+\frac{n}{2}}}{m_i + m_{i+\frac{n}{2}}} \quad , i = 1, ..., \frac{n}{2} \tag{4.13}$$

5. Les nouvelles positions des CBs sont définies en utilisant leurs nouvelles vitesses. La nouvelle position du CB en mouvement est donnée par :

$$x_i^{new} = x_{i-\frac{n}{2}} + rand \circ v'_i \quad , i = \frac{n}{2} + 1, ..., n \tag{4.14}$$

où x_i^{new} et v'_i sont respectivement les nouvelles position et la vitesse après la collision du $i^{e`me}$ CB en mouvement, et $x_{i-\frac{n}{2}}$ est l'ancienne position du vis-à-vis du CB dans le groupe stationnaire. Les nouvelles positions des CBs stationnaires sont obtenues par :

$$x_i^{new} = x_i + rand \circ v'_i \quad , i = 1, ..., \frac{n}{2} \tag{4.15}$$

tel que *rand* est un vecteur aléatoire uniformément distribué dans l'intervalle $[-1, 1]$, et le signe « $\circ$ » dénote une multiplication élément par élément.

6. Les étapes 2 à 5 sont répétées jusqu'à ce qu'une condition d'arrêt soit atteinte. On note que l'état d'un CB (stationnaire ou en mouvement) peut changer d'une itération à une autre.

4.5 Aspects communs entre les métaheuristiques

Quelque soit la métaheuristique utilisée pour régler un problème d'optimisation, il existe plusieurs points communs qu'il faut bien prendre en charge pour la bonne exploitation de cette métaheuristique.

4.5.1 La Représentation de la solution

La conception de toute métaheuristique nécessite une représentation des solutions. C'est une question de conception fondamentale dans le développement des métaheuristiques [Tal09]. La représentation joue un rôle majeur dans l'efficience et l'efficacité de toute métaheuristique. La représentation doit être pertinente et adaptée au problème d'optimisation abordé. Un mauvais choix de représentation peut pénaliser la métaheuristique dans la façon d'aborder l'espace de recherche, voire en temps d'exécution car il s'agit de la fonction la plus sollicitée par toute métaheurisque. Les représentations peuvent être linéaires ou non-linéaire. Dans le cas du PSV, nous utilisons une représentation linéaire binaire, comme expliqué dans le chapitre suivant.

4.5.2 L'équilibre Exploration/Exploitation

Un des aspects les plus importants lors de l'utilisation d'une métaheuristique est l'équilibre entre l'exploration et l'exploitation. Dans l'exploitation (aussi appelée intensification), les régions prometteuses de l'espace de recherche sont explorées plus en profondeur dans l'espoir de trouver de meilleures solutions. Pour l'exploration (ou la diversification), les régions non explorées doivent être visitées pour assurer que toutes les régions de l'espace de recherche sont explorées de manière égale et que la recherche ne se limite pas à un nombre réduit de régions. Le bon choix des paramètres des métaheuristique assure l'équilibre exploration/exploitation et évite de tomber dans un *optimum local*, i.e., se limiter à la meilleure solution d'une seule région de l'espace de recherche.

4.5.3 Gestion des contraintes

La gestion des contraintes dans les problèmes d'optimisation est un autre sujet important pour la conception efficace de métaheuristiques [Tal09]. En effet, de nombreux problèmes d'optimisation continus et discrets ont des contraintes, qui ne sont pas triviales à traiter. Plusieurs stratégies existent pour traiter les solutions qui violent ces contraintes. Quelques-unes des ces stratégies sont :

— Les stratégies du rejet : pour ces stratégies, les solutions infaisables sont rejetées et non utilisées lors du reste du processus de recherche. Ces stratégies sont

peu utilisées, car les solutions infaisables ne sont pas exploitées. En effet, ces dernières peuvent contenir des informations intéressantes concernant l'espace de recherche. Si, dans l'espace de recherche, les régions des solutions faisables sont discontinues, les solutions infaisables peuvent servir comme ponts entre ces régions.

— Les stratégies de pénalisation : pour cette classe de stratégies, les solutions infaisables sont utilisées comme les solutions faisables mais avec une pénalité. Cette pénalité est appliquée sur la fonction objectif pour défavoriser ces solutions. Ces stratégies sont les plus populaires.

— Les stratégies de réparation : les stratégies de réparation consistent en des heuristiques transformant une solution infaisable en une solution faisable. Une procédure d'appariement est appliquée aux solutions infaisables pour générer des solutions faisables.

— Les stratégies de préservation : dans cette classe, une représentation et des opérateurs intégrant des connaissances spécifiques aux problèmes sont appliqués de façon à assurer que seules les solutions faisables seront générées lors de l'application de la métaheuristique.

4.6 Vue D'ensembles Sur les Travaux Réalisés dans la Littérature Pour Resoudre le PSV

Si nous classifions les travaux réalisés sur le Problème de Sélection des vues selon le framework utilisé pour représenter la hiérarchie entre les vues, nous allons avoir trois classes : les travaux utilisant les treillis, les AND-OR graphes et les MVPP.

4.6.1 Travaux qui utilisent le treillis

Dans cette classe nous trouvons le travail de Harinarayan et.al [HRU96] qui est le tout premier travail qui à posé la problématique du PSV. Harinarayan a proposé un algorithme glouton nommé HRUA pour résoudre le PSV à contrainte d'espace de stockage. Cet algorithme choisi a chaque fois la vue la plus bénéfique en terme de *benefit per unit space* pour matérialisation, jusqu'à ce qu'il ne reste plus d'espace pour stocker des vues.

T.V.VIJAY a proposé plusieurs algorithmes basé sur la recherche aléatoire pour résoudre le PSV. Le point commun entre tous ces travaux est le fait d'impliquer la contrainte dans la représentation des solutions. Ce qui rend ces travaux non-adaptables à la contrainte de temps de maintenance. Nous étudions dans le chapitre 5 en détails ces travaux ensemble avec le travail de Harinarayan.

Nous trouvons également pour la contrainte d'espace de stockage le travail de Gang Gou et.al[GYL06] qui a proposé un Algorithme A* très efficace pour résoudre le PSV, mais comme toute méthode exacte, le temps d'exécution leurs pose un problème, surtout quand les instances sont grandes.

Lin et.al [LK04] a proposé une approche appelé *genetic selection* dans laquelle il a utilisé l'algorithme génétique comme méthode de recherche. La particularité de ce travail est l'utilisation de la fonction *greedy repair* ou la réparation gloutonne. Il s'agit d'utiliser le principe inverse de l'algorithme HRUA lors de la réparation des solutions infaisables trouvées.

Pour la contrainte de temps de maintenance, nous trouvons deux travaux qui utilise le treillis comme framework de représentation de hiérarchie des vues. Le premier est le travail de Kalnis et.al [KMP02] dans lequel les auteurs ont proposé quatre algorithme de recherche aléatoire, à savoir *Iterative Improvment*, *Simulated Annealing*, *Random Sampling* et *Two-phase Optimisation*. Le deuxième est le travail de XU YU et.al [YYCG03] qui a proposé une approche basée sur l'algorithme génétique. La particularité de ce travail est l'utilisation de *stockastic ranking* comme méthode de gestion des contraintes. Ce travail sera détaillé dans le chapitre 6.

4.6.2 Travaux qui utilisent le AND-OR Graph

Le premier travail qui a proposé l'utilisation du AND-OR *view graph* est celui de Gupta [GM05]. Dans ce travail Gupta a proposé plusieurs algorithmes gloutons comme extension du travail de Harinarayan [HRU96] sur la sélection des vues et des indexes dans le cas d'une contrainte d'espace de stockage. Juste après, Gupta et Mumik [GM99] ont introduit le PSV a contrainte de temps de maintenance. Ils ont proposé un algorithme glouton appelé *Inverted Tree Greedy Algorithm* et un algorithme A*.

Horng et.al [HCLK99] a proposé un algorithme hybride nommé GLS « Genetic Local Search » qui combine l'algorithme génétique avec l'algorithme local search pour résoudre le PSV à contrainte d'espace de stockage. Lee et Hammer [LH01] ont proposé le premier algorithme génétique qui traite le PSV à contrainte de temps de maintenance dans le contexte AND-OR view graph.

Nous pouvons trouver également plusieurs autres travaux. Nous citons le travail de Sun et.al [SW09] dans lequel les auteurs ont utilisé l'algorithme Particle Swarm Optimization (PSO) pour résoudre le PSV à contrainte d'espace. De la même manière, Xin et.al [LQJW10] ont proposé l'algorithme Shuffled Frog Leaping (SFL) pour traiter la même variante du PSV.

4.6.3 Travaux qui utilisent le MVPP

Le framework MVPP (Multiple View Processing Plan) a été introduit dans le travail de Zhang et.al [ZY99]. Dans [ZYY01], Zhang et.al ont appliqué un algorithme hybride (évolutionnaire/heuristique) composé de deux niveaux de traitement de l'algorithme. L'algorithme de niveau supérieur recherche un bon plan de traitement global à partir des plans de traitement locaux basés sur des requêtes. L'algorithme de niveau inférieur sélectionne le meilleur ensemble de vues matérialisées avec le coût total minimal pour un plan de traitement global particulier. Aucune contrainte n'était considérée dans ce travail. Par contre, le coût de maintenance était inclus dans la fonction objectif, comme la plupart des travaux utilisant le MVPP. Ce même travail à utilisé une fonction de réparation comme méthode de gestion de contrainte. Dans [HCL03] les auteurs ont appliqué l'algorithme hybride GLSA(Genetic Local Search Algorithm). Derakhshan et.al [DDKS06] a proposé un algorithme basé sur Simulated Annealing. Jiratta et.al [PoA07] ont proposé l'algorithme 2PO (Two Phases Optimization) qui est la combinaison de Iterative Improvment et Simulated Annealing pour résoudre le PSV. Nous avons remarqué que la majorité des travaux utilisant MVPP pour représenter la hiérarchie entre les vues utilisent le même modèle de coût de [ZYY01] et n'applique aucune contrainte lors du traitement du PSV.

4.7 Conclusion

Dans ce chapitre, nous avons défini les différentes classes des problèmes d'optimisation ainsi que les méthodes pour résoudre chacune de ces classes. Une des méthodes les plus populaires pour résoudre les problèmes NP-difficile est l'utilisation des métaheuristique. Nous avons présenté quelques métaheuristiques que nous allons étudier/utiliser dans le reste de cette thèse, à savoir : l'algorithme glouton, l'algorithme génétique, l'évolution différentielle, le quantum evolutionary et le colliding bodies. Dans le chapitre suivant, nous présenterons les principaux travaux réalisés pour résoudre le PSV à contrainte d'espace de stockage, ainsi que notre propre contribution pour résoudre ce problème.

Chapitre 5

Algorithmes pour Résoudre le PSV à Contrainte d'Espace

5.1 Introduction

Dans le chapitre précédent, nous avons défini les problèmes NP-Difficile ainsi que leur position dans les différentes classes de problèmes d'optimisation. De plus, nous avons présenté les méthodes pour résoudre ce type de problèmes. Plusieurs travaux qui utilisent ces méthodes ont été proposées dans la littérature pour résoudre le PSV à contrainte d'espace de stockage.

Kalnis et.al [KMP02] a mentionné que les metaheuristiques sont plus adaptées que les autres méthodes pour résoudre le PSV parce qu'elles peuvent être appliquées sur de grandes instances de problèmes, leurs paramètres peuvent assurer le compromis entre la qualité de la solution proposée et le temps d'exécution. Ainsi, elles peuvent être adaptées à plusieurs versions du PSV.

Dans ce chapitre, nous allons présenter les travaux les plus connus pour la résolution du PSV à contrainte d'espace de stockage. Ensuite, nous proposerons nos propres approches fondées sur les algorithmes « Quantum Evolutionary » et « Differential Evolution » présentés dans le chapitre précédent. Enfin, plusieurs tests expérimentaux sont effectués afin de valider les approches proposées.

5.2 Travaux de Harinarayan *et al.*

Quand on veut parler du PSV, quelque soit sa variante, les travaux de Harinarayan *et al.* sont incontournables [HRU96]. Ce sont les premiers qui ont présenté la problématique du PSV, ainsi que le *treillis de dépendance* et le modèle de coût. Le *treillis de dépendance* et le modèle de coût ont été expliqués au préalable dans le chapitre 3 – sections 3.3 et 3.5.1, respectivement. Harinarayan *et al.* ont proposé un algorithme

glouton pour résoudre le PSV à contrainte d'espace de stockage HRUA. Cet algorithme HRUA sélectionne à chaque étape la meilleure vue en terme de *bénéfice* pour matérialisation, jusqu'à atteindre le nombre maximal de vues à matérialiser. Le bénéfice de la vue v par rapport à S $B(v, S)$ est défini comme suit :

1. Pour chaque vue $w \preceq v$, une quantité B_w est définie :
   — Soit u l'ancêtre de w ($w \preceq u$) le moins coûteux qui existe dans l'ensemble S.
   — Si $C(v) < C(u)$, alors : $B_w = C(u) - C(v)$. Sinon : $B_w = 0$.
2. $B(v, S) = \sum_{w \preceq v} B_w$.

L'algorithme 5 présente le pseudo-code de la solution gloutonne de Harinarayan (HRUA).

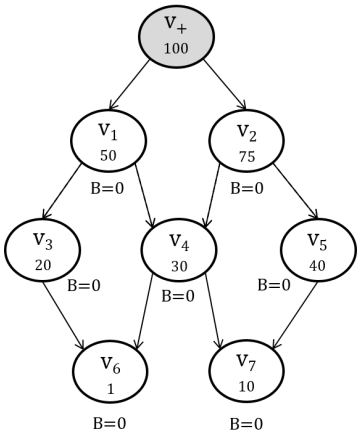
Algorithm 5: L'algorithme glouton de Harinarayan [HRU96]

```
S = {v+} ;
for i = 0; i < k; i = i + 1 do
    v ← sélectionner une vue pas dans S dont le Bénéfice B(v, S) est maximisé ;
    S = S ∪ {v};
end
Return S ;
```

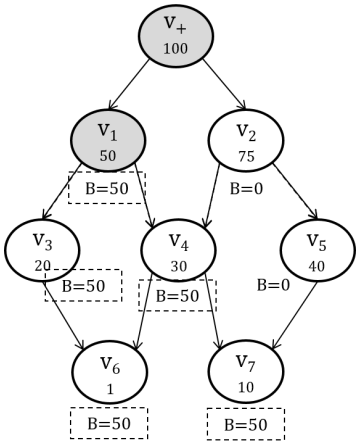
Afin de mieux comprendre l'algorithme HRUA, nous allons reconduire l'exemple de Harinarayan [HRU96] ci-après.

Soit le *treillis de dépendance* de la Figure 24 sur lequel nous allons appliquer l'algorithme HRUA pour sélectionner les meilleures vues à matérialiser ($k = 3$). Comme le sommet représente l'ED, il est toujours matérialisé. L'objectif de HRUA est de choisir les trois vues (autres que v_+) les plus bénéfiques pour la réponse aux requêtes. La Table 3 est divisée en trois colonnes (premier choix, deuxième choix, et troisième choix) ; chaque colonne justifie le choix de la vue à matérialiser en comparant les bénéfices de matérialisation de toute vue inexistante dans S.

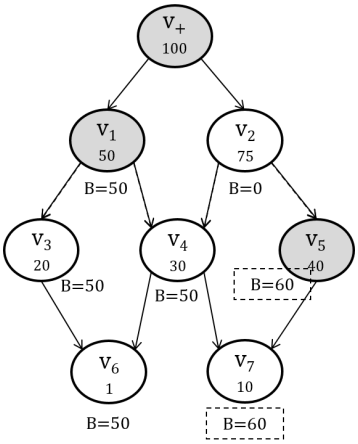
Nous remarquons que, pour le premier choix, v_1 est choisie pour la matérialisation avec un bénéfice de 250. En effet, en matérialisant v_1, tous ses descendants l'utiliseront pour répondre aux requêtes au lieu d'utiliser v_+. Le gain est alors de $100-50 = 50$ pour les cinq descendants de v_1, à savoir : v_1,v_3,v_4,v_6,v_7, d'où un bénéfice égal à 250 (50×5) (Figure 24b). Pour le deuxième choix, v_5 est la vue la plus bénéfique avec un bénéfice de 70. Il existe deux descendants de v_5, qui sont : v_7 et v_5 elle-même. Le bénéfice pour la vue v_5 jusqu'à cette étape est de 0. Après sa matérialisation, le bénéfice de v_5 deviendra $100 - 40 = 60$. Néanmoins, pour v_7, le bénéfice en présence de l'ensemble S à cette étape est de 50. Après la matérialisation de v_5, le bénéfice deviendra $60 - 50 = 10$. Ainsi, le total des bénéfices de la matérialisation de v_5 est de 70 (Figure 24c). De la



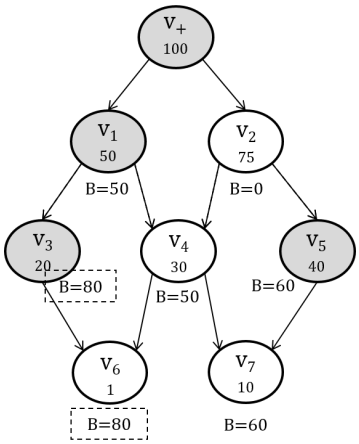
(a) avant sélection : $S = \{v_+\}$



(b) sélection de v_1 : $S = \{v_+; v_1\}$



(c) sélection de v_5 : $S = \{v_+; v_1; v_5\}$



(d) sélection de v_3 : $S = \{v_+; v_1; v_5; v_3\}$

FIGURE 24 – Exemple de l'application de HRUA

	Premier choix	Deuxième choix	Troisième choix
v_1	$50 \times 5 = 250$		
v_2	$25 \times 5 = 125$	$25 \times 2 = 50$	$25 \times 1 = 25$
v_3	$80 \times 2 = 160$	$30 \times 2 = 60$	$30 \times 2 = 60$
v_4	$70 \times 3 = 210$	$20 \times 3 = 60$	$20 + 20 + 10 = 50$
v_5	$60 \times 2 = 120$	$60 + 10 = 70$	
v_6	$99 \times 1 = 99$	$49 \times 1 = 49$	$49 \times 1 = 49$
v_7	$90 \times 1 = 90$	$40 \times 1 = 40$	$30 \times 1 = 30$

TABLE 3 – Exemple de l'exécution de HRUA

même façon, pour le troisième et dernier choix, la vue v_3 est choisie avec un bénéfice de 60. Le bénéfice deviendra 80 pour les vues v_3 et v_6 (descendants de v_3), au lieu de 50 jusqu'à cette étape de l'algorithme (Figure 24d).

En matérialisant trois vues, l'algorithme HRUA s'arrête et retourne le résultat : $S = \{v_+; v_1; v_5; v_3\}$.

5.3 Travaux de Vijay Kumar *et al.*

L'équipe de Vijay Kumar s'est beaucoup intéressée au PSV. Ils ont en effet proposé plusieurs approches basées sur diverses métaheuristiques. Tout comme Harinarayan *et al.* [HRU96], ils considèrent le PSV à contrainte d'espace de stockage en fixant le nombre de vues à matérialiser (qu'ils appellent Top-K views). Le premier travail proposé est un algorithme génétique, le GVSA [VKK12a]. Ils ont représenté les solutions sous forme de vecteurs de taille k (k étant le nombre de vues à sélectionner). Dans chaque case de ce vecteur, un nombre entre 1 et N (N : la cardinalité de l'ensemble des vues) représente la présence de la vue correspondante au nombre dans l'ensemble des vues matérialisées. Quant à l'algorithme GVSA, c'est une application simple de l'algorithme génétique ; la sélection se fait entre des paires aléatoirement choisies avec une probabilité de prendre la moins bonne solution. Les auteurs ont choisi de faire le *single point crossover*, où le point de croisement est également aléatoirement choisi. Les résultats expérimentaux ont montré que GVSA donne de meilleurs résultats que le HRUA de Harinarayan *et al.* [HRU96].

Une autre approche proposée par l'équipe de Vijay Kumar est l'algorithme mémétique MVSA [KK13]. L'algorithme mémétique est une extension de l'algorithme génétique traditionnel. Il rajoute juste une phase de *recherche locale* pour mieux exploiter le voisinage de l'espace de recherche traversé. Ceci a pour bénéfice d'éviter la

convergence prématurée de l'algorithme. Dans l'algorithme MVSA, les auteurs ont reproduit le même algorithme génétique GVSA auquel ils ont ajouté une procédure de *recherche locale*. Ils ont utilisé l'algorithme *Iterative Improvement* pour exploiter le voisinage de chaque individu et le remplacer si une meilleure solution est trouvée. Les résultats expérimentaux ont montré que MVSA donne également de meilleurs résultats que HRUA de Harinarayan *et al.* [HRU96].

Plusieurs autres travaux ont été également proposés par la même équipe pour résoudre le PSV. Nous citons l'algorithme Recuit Simulé (Simulated Annealing) [VKK12b], l'algorithme Iterative Improvement [VKK13], ainsi que plusieurs algorithmes utilisant la Bee Colony Optimization [AK15] [AK17]. Le point commun entre toutes ces méthodes est qu'elles sont comparées avec HRUA lors des expérimentations. Cependant, ceci ne peut pas nécessairement être considéré comme une preuve de performance, étant donné que HRUA n'est lui-même pas très performant comme solution au PSV. Deux travaux en particulier ont été proposés et comparés à d'autres approches ; il s'agit de l'algorithme Evolution Différentielle [VKK14], et Quantum Evolutionary [KVK18], qui vont être détaillés dans la section suivante.

5.3.1 L'algorithme DEVSA

Comme mentionné dans la section précédente, les auteurs ont utilisé l'algorithme Evolution Différentielle pour résoudre le PSV. L'algorithme DE a été présenté dans la section 4.4.3. Dans cette section, nous allons expliquer l'adaptation de DE au problème PSV dans l'algorithme DEVSA.

Tout comme leurs autres algorithmes, les auteurs ont choisi de présenter les solutions par des vecteurs de taille k (k étant la limite en nombre des vues à ne pas dépasser). La valeur de chaque élément de ce vecteur est un entier qui représente l'index d'une vue appartenant à l'univers des vues (qui sont numérotées). Le vecteur, qui s'appelle $Top - k$, représente l'ensemble des vues choisies pour matérialisation. On note que les éléments du vecteur sont distincts puisqu'on ne peut pas matérialiser une même vue plusieurs fois. On note également que les vues sont triées dans un ordre croissant. La Figure 25 présente un exemple d'un vecteur de $Top - 5$ vues (une solution d'une instance du PSV avec la contrainte $SC_n = 5$).

Top - 5	2	4	5	6	9

FIGURE 25 – Exemple de la représentation de Top-5 vues

Comme expliqué dans la section 4.4.3, l'algorithme DE est composé de trois phases : *génération du vecteur mutant*, *génération du vecteur d'essai*, et *sélection*. L'algorithme

DE a été conçu pour traiter des problèmes d'espace de recherche continu. Les auteurs ont fait une adaptation pour l'utiliser afin de résoudre le PSV, qui est un problème à domaine de recherche discret.

Pour la phase de *génération de vecteurs mutants*, l'algorithme DEVSA choisit trois vecteurs aléatoires à partir de la population sur lesquels il applique l'équation 4.1, comme dans l'algorithme DE original. Cependant, au lieu de faire la soustraction directe entre les éléments, dans DEVSA le signe « $-$ » est appliqué par le choix du minimum entre chaque paire d'éléments des deux vecteurs sur lesquels la soustraction est appliquée. De même, le signe « $+$ » est appliqué en choisissant le maximum entre les éléments des deux vecteurs sur lesquels l'addition est appliquée. De plus, le facteur F est la probabilité d'appliquer une mutation sur le dernier élément du vecteur de soustraction.

Pour la phase de *génération du vecteur d'essai*, dans la section 4.4.3, l'équation 4.2 représente l'opérateur *uniform cross over*. Cette méthode n'est pas nécessairement la seule à appliquer pour la génération du vecteur d'essai. En effet, d'autres formes de croisement peuvent être appliquées comme le *single point cross over*, le *two points cross over*, ou encore le *n-points cross over*. Pour l'algorithme DEVSA, les auteurs ont utilisé un croisement particulier qui s'appelle *Modified Cross Over*. Ce croisement a été proposé par L. Davis [Dav85]. Il s'agit de comparer un nombre aléatoire r au paramètre CR (Cross over Rate). Si $r < CR$, l'élément de la première solution est pris, et le nombre r est régénéré une autre fois et re-comparé au CR, jusqu'à avoir une valeur supérieure au CR. Quand $r > CR$, le reste des éléments est pris dans la deuxième solution, et ce, sans répétition.

Pour la *sélection*, le principe « *survival for the fittest* » est appliqué entre le vecteur d'essai et l'individu-même, comme dans DE original. Pour illustrer l'algorithme DEVSA, nous allons reconduire le même exemple que Vijay Kumar dans [VKK14].

Soit une population de dix individus qui sont des vecteurs des Top-5 vues présentées dans la Figure 26.

Nous allons appliquer les trois phases principales de DEVSA sur le premier individu de la population.

La première phase consiste à construire le vecteur mutant en utilisant l'équation (4.1). On choisit dans un premier temps trois individus aléatoires (autres que l'individu 1). On suppose que ces trois individus sont les individus 3, 4, et 9. On ajoute la différence entre 4 et 9, multipliée par le facteur $F = 0.2$ à l'individu 3 pour construire le vecteur mutant u_1 de la façon suivante :

1. $u_1 = \{2, 5, 6, 12, 14\} + F(\{3, 5, 14, 15, 16\} - \{4, 7, 11, 14, 15\})$
2. $u_1 = \{2, 5, 6, 12, 14\} + 0.2(\{3, 5, 11, 14, 15\})$
3. $u_1 = \{2, 5, 6, 12, 14\} + (\{3, 5, 9, 11, 14\})$

	Top-5 Vues					TPC
1	2	8	12	14	15	1134
2	3	9	11	12	13	1073
3	2	5	6	12	14	1134
4	3	5	14	15	16	1201
5	2	4	9	11	14	1066
6	2	4	7	8	10	1079
7	2	3	5	7	9	1122
8	2	9	12	14	16	1114
9	4	7	11	14	15	1074
10	4	5	11	14	16	1120

FIGURE 26 – Population de taille 10 des Top-5 vues

4. $u_1 = (\{3, 5, 9, 12, 14\})$

Nous avons précédemment expliqué que la soustraction est appliquée par le choix de la plus petite valeur entre deux vecteurs, élément par élément. La soustraction entre les deux vecteurs 4 et 9 a donné le vecteur $\{3, 5, 11, 14, 15\}$. Le *Scale Factor* F est la probabilité que le dernier élément soit remplacé par une vue aléatoire. Le résultat de son application est le remplacement de la vue « 15 » par la vue « 9 ». La liste est de nouveau triée dans l'ordre croissant. Le vecteur résultant après application du *Scale Factor* est donc $\{3, 5, 9, 11, 14\}$. Finalement, l'addition entre ce dernier vecteur obtenu et le vecteur 3 consiste à choisir la plus grande valeur entre ces deux vecteurs, élément par élément. Le vecteur mutant est donc : $\{3, 5, 9, 12, 14\}$.

La deuxième phase de l'algorithme DEVSA est de construire le vecteur d'essai en appliquant le croisement entre le vecteur choisi (vecteur N°1) et le vecteur mutant récemment généré. Comme expliqué auparavant, un nombre aléatoire r est généré et comparé au taux CR pour chaque élément des deux vecteurs de croisement. Le point de croisement est choisi quand la valeur de r est supérieure à CR. Dans cet exemple, ça s'est produit pour le deuxième élément. Le vecteur d'essai contiendra le premier élément du vecteur 1 ($\{2, 8, 12, 14, 15\}$), soit l'élément « 2 », ainsi que le reste du vecteur mutant ($\{3, 5, 9, 12, 14\}$), soit les éléments « 3,5,9,12 ». Le résultat sera le vecteur d'essai : $\{2, 3, 5, 9, 12\}$.

La dernière phase est la sélection. Le TPC est calculé pour le vecteur 1 et le vecteur d'essai. S'il s'avère que le vecteur d'essai est meilleur que l'individu 1, le vecteur 1 sera remplacé par le vecteur d'essai.

Ces trois phases sont appliquées pour chaque individu de la population, et ce pour toutes les itérations de l'algorithme.

Les résultats expérimentaux ont montré que DEVSA donne de bons résultats, non seulement comparé à HRUA, mais également comparé à GVSA et MVSA. Leur façon d'adapter DE à un problème discret est fortement liée au choix de représentation des individus/solutions, qui est issu de la contrainte d'espace de stockage. Ce travail a été comparé avec un autre algorithme (QIEVSA), qui s'est avéré meilleur que DEVSA. La section suivante expliquera l'algorithme inspiré de l'algorithme Quantum Evolutionary, soit le QIEVSA.

5.3.2 L'algorithme QIEVSA

Vijay Kumar et son équipe ont développé un algorithme inspiré de QEA pour résoudre le PSV. L'algorithme QEA a été présenté dans la section 4.4.4. Dans la présente section, nous allons expliquer l'utilisation des principes de QEA pour la résolution du PSV dans l'algorithme QIEVSA.

Avant de parler de l'algorithme QIEVSA, il faut attirer l'attention sur le fait que l'algorithme QEA originel est un algorithme dédié à résoudre des problèmes dont l'espace de recherche est binaire. Cependant, la représentation des Top-K vues utilisée dans la quasi-totalité des algorithmes de Vijay Kumar *et al.* contient des nombres décimaux. Ceci les a obligé d'adapter cette représentation au QEA de la façon suivante : au lieu de prendre des décimaux pour représenter les Top-K vues, il ont opté pour la forme binaire de ces derniers. Ainsi, un vecteur des Top-K vues n'est plus de taille « K », mais plutôt de taille « $K \times d$ », tel que d est le nombre de dimensions du schéma en étoile. En effet, pour pouvoir représenter toutes les vues en forme binaire, on aura besoin de d bits. La Figure 27 explique cette nouvelle représentation.

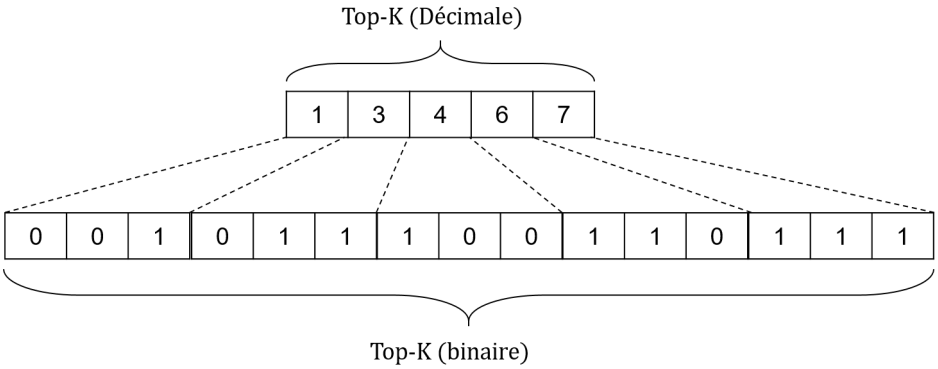


FIGURE 27 – Représentation binaire des Top-5 vues d'un treillis de trois dimensions

L'algorithme QIEVSA peut se résumer en les étapes suivantes :

1. Comme tout algorithme évolutionnaire, la première étape est l'initialisation de

la population. Dans le cas de l'algorithme QIEVSA, et de tout algorithme inspiré du QEA, la population n'est pas constituée des solutions, mais des quantum vectors (section 4.4.4). Dans leur travail [KVK18], les auteurs ont choisi d'initialiser la population en démarrant d'un point où chaque q-bit de chaque quantum vector est à la même distance des valeurs 0 et 1. En respectant la formule $\alpha^2 + \beta^2 = 1$, les valeurs de α et de β pouvant assurer une telle distance sont : $\alpha = \beta = \frac{1}{\sqrt{2}}$.

2. La phase suivante est l'application du *measurement*. Il s'agit de générer un vecteur de solutions à partir de chaque individu. Le fait de choisir des points situés à la même distance de 0 et 1 assurera la diversité de la population pour la première génération g(0).
3. L'évaluation se fait en calculant le TPC de chaque vecteur de solution généré au *measurement*, et ce après la conversion du vecteur binaire en vecteur décimal.
4. Dans cette étape, les meilleures solutions trouvées dans la génération g(0) sont stockées sous formes binaire et décimale dans $B(0)$ et $DB(0)$, respectivement. La meilleure solution de toute la génération et sa forme décimale sont également stockées dans B et DB, respectivement. Comme les auteurs ont décidé de garder trace des N meilleures solutions de chaque génération (N étant la taille de la population), toutes les solutions de la première génération sont stockées dans $B(0)$ et $DB(0)$. L'indice de génération g est donc incrémenté de 1.
5. Obtention des solutions $P(g)$ en appliquant le *measurement* sur les vecteurs $Q(g-1)$, tout comme l'étape 2.
6. Évaluation des solutions $P(g)$ en calculant leur TPC, comme dans l'étape 3.
7. Génération des vecteurs $Q(g)$ en appliquant l'opérateur *Q-Gate* (section 4.4.4) aux vecteurs $Q(g-1)$.
8. Les N meilleures solutions à partir de $P(g)$ et $B(g-1)$ sont stockées dans $B(g)$, ainsi que leur forme binaire dans $DB(g)$. La meilleure solution globale trouvée jusqu'à cette étape et sa forme binaire sont stockées dans B et DB, respectivement.
9. Dans cette étape, l'opérateur de *migration* est appliqué. Après un nombre prédéfini de générations, QIEVSA applique une migration locale ou globale. Il s'agit de remplacer les meilleures solutions d'une génération $B(g)$ par les meilleures d'une autre génération ou par la meilleure solution globale B.
10. QIEVSA fonctionne pendant un nombre prédéfini de générations durant lesquelles les étapes de 5 à 9 sont reconduites. La finalité sera de retourner la meilleure solution trouvée de toutes les générations, soit B.

Les résultats expérimentaux ont montré que QIEVSA donne de bons résultats comparé à HRUA, GVSA, MVSA et même DEVSA. Mais, comme pour DEVSA, le choix

de la représentation des solutions est lié à la contrainte d'espace de stockage. Dans le cas de QIEVSA, les composantes de la solution (les bits) ne sont pas significatifs si elles sont séparées du reste des bits qui forment la valeur décimale (la vue). Ainsi, l'évolution de ces derniers séparément favorise clairement la diversification, même lors de l'utilisation de l'opérateur Q-Gate.

5.3.3 Discussion

Vijay Kumar et son équipe ont réalisé des solutions au PSV basées sur diverses métaheuristiques. Le point commun entre tous ces travaux est l'application de la contrainte d'espace de stockage qu'ils ont réalisée en fixant le nombre de vues à sélectionner dans ce qu'ils ont appelé *Top-K views*. De plus, la représentation qu'ils ont choisi est fortement liée à la contrainte d'espace de stockage. Cette représentation n'est pas très utilisée dans la résolution du PSV, ce qui rend leurs travaux difficilement comparables à d'autres travaux de résolution du PSV (à part HRUA). À partir des diverses méthodes utilisées par Vijay Kumar *et al.*, deux méthodes ont particulièrement donné de meilleurs résultats comparées aux autres. Ces méthodes sont DEVSA et QIEVSA.

Dans la section suivante, nous allons présenter notre propre contribution à la résolution du PSV en utilisant les mêmes métaheuristiques (Évolution Différentielle et Quantum Evolutionary) avec la représentation binaire, plus utilisée et plus adaptée aux deux contraintes.

5.4 Contribution à la résolution du PSV à contrainte d'espace de stockage

Dans cette section, nous allons présenter nos contributions à la résolution du PSV à contrainte d'espace de stockage. Ces contributions sont principalement basées sur deux métaheuristiques, i.e., « Évolution Différentielle » et « Quantum Evolutionary », car elles ont donné de bons résultats dans les travaux de Vijay Kumar *et al.*. Cependant, notre adaptation de ces deux métaheuristiques va utiliser leurs opérateurs originaux, sans interprétation. Dans un premier temps, nous allons présenter un algorithme nommé QEAM qui représente l'application directe du QEA au PSV. Nous allons également présenter l'adaptation de l'Evolution Différentielle selon deux méthodes : l'utilisation de la fonction de transformation *Sigmoid* pour l'algorithme BDE, et l'utilisation de *Quantum Binary approach* dans QDE.

5.4.1 QEAM

Dans cette section, nous allons présenter l'algorithme QEAM (Quantum Evolutionary Algorithm for MVS problem) pour résoudre le PSV à contrainte d'espace de stockage [MB20b]. Contrairement aux travaux de Vijay Kumar *et al.*, la contrainte ne va pas s'appliquer en fixant le nombre de vues, mais en fixant la taille du disque à ne pas dépasser lors de la matérialisation de l'ensemble de vues en utilisant l'équation 3.3.

Représentation des Q-bits

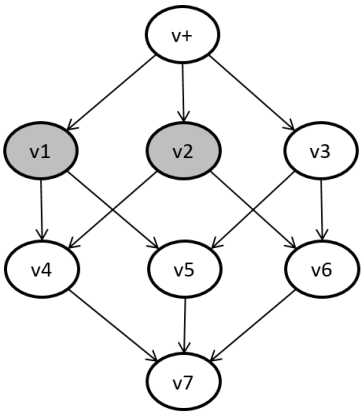
L'équation $\alpha^2 + \beta^2 = 1$ représente essentiellement l'équation d'un cercle unitaire. Chaque point du périmètre de ce cercle peut être représenté par une seule variable θ. Les coordonnées cartésiennes de θ son représentées par cos_θ et sin_θ. Si on prend θ dans $[0, 2\pi]$, tous les points du cercle sont couverts. Pour cela, au lieu d'utiliser les variables α et β pour représenter un Q-bit comme dans le QEA original, nous allons utiliser la variable θ, à partir de laquelle les valeurs de α et β peuvent être induites. Ceci va non seulement diviser l'espace occupé par la population par deux, mais également rendre l'application de l'opérateur Q-Gate plus simple, étant donné que l'angle de rotation va être ajouté directement sur le Q-bit.

Représentation des solutions

Dans QEAM et le reste de nos travaux, chaque solution est représentée par un vecteur binaire $X = \{x_1, x_2, x_3, ..., x_n\}$, tel que n représente le nombre total des vues (à part le sommet). Une variable x_i prend la valeur 1 si la vue v_i est matérialisée, sinon x_i prend la valeur 0. Comme le montre la Figure 28, le vecteur binaire X est la représentation d'une solution dont la hiérarchie des vues est donnée par le *treillis* à trois dimensions présent dans la même Figure et dans lequel les vues matérialisées sont en gris. Dans cet exemple, les vues v_1 et v_2 sont matérialisées, ce qui correspond à la valeur 1 en deuxième et troisième positions du vecteur X. Le reste des vues n'est pas matérialisé, d'où la valeur 0 des positions restantes. On rappelle que le sommet du *treillis* représente l'ED lui-même et est toujours matérialisé. C'est la raison pour laquelle nous avons commencé le vecteur X par l'indice 1.

Solutions violant la contrainte

L'utilisation de l'opérateur *mesurement* pour générer des solutions ne peut garantir que ces solutions ne violent la contrainte. Lors de l'exploration de l'espace de recherche, l'algorithme rencontrera certainement des solutions infaisables ; ce sont des solutions dont la taille dépasse le seuil S. L'une des méthodes les plus couramment utilisées pour résoudre ce problème consiste à implémenter une fonction de pénalité pénalisant ces



Solution Vector "*X*"

0	1	2	3	4	5	6	7
0	1	1	0	0	0	0	0

FIGURE 28 – Exemple de représentation d'une solution à un PSV pour *treillis* à trois dimensions

solutions infaisables et garantissant que, même si elles ont une bonne *fitness*, elles ne seront pas hautement évaluées parmi d'autres solutions. Dans notre travail, aucune fonction de pénalité n'est utilisée. Au lieu de cela, nous avons opté pour la stratégie de réparation (section 4.5.3). Nous appliquons cette stratégie de la façon suivante : nous prenons la plus petite vue matérialisée et nous la *dématérialisons* pour réduire l'espace occupé par l'ensemble choisi. Si la contrainte n'est plus violée, elle est adoptée ; sinon, les plus petites vues sont éliminées une à une jusqu'à ce que la contrainte ne soit plus violée. Cela garantit qu'après l'application du *measurement*, aucune solution ne sera infaisable.

Reset Operator

Comme expliqué dans la section 4.5.2, un des soucis les plus flagrants dans l'application de toute métaheuristique est la convergence prématurée vers un *optimum local*. Il s'agit de la meilleure solution d'une région limitée de l'espace de recherche. Pour parer à ce blocage, nous avons utilisé le « Reset Operator » (aussi connu sous le nom de *reseeding*, ou encore *extinction*) pour échapper à l'*optimum local*. Si la meilleure solution globale ne change pas pendant un certain nombre d'itérations, le « Reset Operator » est appliqué en remplaçant tous les individus par des nouveaux. En faisant cela, et en conservant la meilleure solution globale trouvée jusqu'à présent, nous pouvons nous assurer que ces vecteurs rechercheront la meilleure solution à partir de points de départ différents, couvrant de nouvelles régions dans l'espace de recherche et garantissant la diversité pour l'ensemble de la population. Le « Reset Operator » fonctionne

d'une manière différente en tant qu'opérateur de diversification que l'opérateur de mutation régulier. Contrairement à la mutation qui est appliquée avec une probabilité, le « Reset Operator » n'est appliqué à l'échelle globale que si l'algorithme ne parvient pas à améliorer la meilleure solution. Dans QEAM, le nombre d'itérations nécessaires pour appliquer le « Reset Operator » est fixé à 100 itérations, et ce après plusieurs tests.

Le Pseudo code du QEAM

L'Algorithme 6 représente le pseudo-code du QEAM [MB20b].

Algorithm 6: Pseudo-code de l'algorithme QEAM

N : population size ;
m_p : mutation probability $\in [0.001; 0.1]$;
t : space constraint threshold $\in [0.1; 0.9]$;
C_{max} : max space cost when all vertices are materialized ;
ra : rotation angle $\in [0.01\pi; 0.1\pi]$;
Initialize a population of quantums Q_i where $i \in [1, N]$;
repeat
  foreach $Q_i, i \in [1, N]$ **do**
    X_i = measurement(Q_i) ;
    if *(*$SC(G, X_i) > C_{max} \times t$*)* **then**
      Repair X_i ;
    end
    if *(*$TPC(G, X_i) < TPC(G, Best)$*)* **then**
      Best = X_i ;
    end
    update Q_i using Q-Gate and ra ;
    if *(*$rand < m_p$*)* **then**
      Apply mutation ;
    end
  end
  if *(no update on Best for 100 iterations)* **then**
    Apply Reset Operator ;
  end
until *stop criterion*;

L'algorithme commence par l'initialisation de de la population (les *quantum vectors*). Pour assurer la bonne distribution des solutions dans l'espace de recherche, il faut que la position de chaque vecteur soit aléatoirement choisie parmi 0 et 1, i.e., il faut que la probabilité d'avoir la valeur 0 soit la même que d'avoir la valeur 1. Un tel

aspect peut être assuré en fixant la valeur de chaque Q-bit de chaque *quantum vector* à $\frac{\pi}{4}$. Les valeurs de α et β qui vont être induites de ce Q-bit sont : $\alpha = \beta = \frac{1}{\sqrt{2}}$.

Après l'initialisation, l'opérateur *measurement* est appliqué sur toute la population pour générer des solutions. Si une solution s'avère de taille plus grande que le seuil S (représenté par la multiplication $(C_{max} \times t)$), cette solution est réparée comme expliqué antérieurement. Si cette solution est meilleure que la meilleure solution globale *Best*, elle la remplace.

Ensuite, la position de l'individu Q_i est mise à jour en utilisant l'opérateur Q-Gate et en utilisant lookup table donnée par la Table 2. Nous rappelons que l'angle de rotation ra est ajouté directement sur le Q-bit suivant la nouvelle représentation expliquée dans 5.4.1. L'étape qui suit est l'application de la mutation avec une probabilité m_p.

Enfin, si la population se trouve dans une convergence prématurée, le *Reset Operator* est appliqué pour assurer un nouveau départ et éviter le blocage.

Résultats Expérimentaux

Afin de tester les performances de l'algorithme QEAM proposé, nous avons choisi de le comparer à HRUA, et à l'algorithme génétique GEA. QEAM et GEA ont été implémentés avec HRUA en utilisant JDK 1.8 sous Windows 10. Les tests ont été effectués sur un PC dont le proceseur est un Intel i-5 2,4 GHz et doté de 8 Go de RAM. Après plusieurs tests, les paramètres ont été définis comme indiqués dans la Table 4.

TABLE 4 – *Paramètres utilisés pour QEAM/GEA*

Parameters	**QEAM values**	**GEA values**
Population size	100	100
Number of generations	1000	1000
Query frequencies	Suivant la distribution Zipf	
Cross over rate	-	0.5
Rotation angle	0.02π	-
Mutation problem	0.001	0.001

Après plus de trente tests sur les paramètres, nous avons fixé la taille des deux algorithmes (QEAM et GEA) à 100 individus, et le nombre de générations à 1000. Ainsi, les fréquences de requêtes suivent la distribution Zipf. Par défaut, les fréquences de requêtes élevées sont plus susceptibles d'être attribuées au niveau élevé (proche du sommet) [YYCG03]. Le taux du croisement (Cross Over rate) est fixé à 0.5 pour GEA et l'angle de rotation (Rotation angle) est fixé à 0.02π pour QEAM. Finalement, la

probabilité de mutation est de 0.001 pour les deux algorithmes. Le seuil est attribué en fixant un pourcentage t sur C_{max} (la taille maximale lors la matérialisation de toutes les vues).

Le premier test effectué compare QEAM avec HRUA pour deux instances du PSV (treillis de cinq et six dimensions). À partir de la Table 5, nous pouvons observer la différence entre les deux algorithmes en faveur de QEAM. Pour le treillis à cinq dimensions, QEAM trouve de meilleures solutions pour tous les seuils sauf 0.7, 0.2 et 0.1, pour lesquels HRU et QEAM trouvent la même solution. En ce qui concerne le treillis à six dimensions, QEAM surpasse HRU, et ce pour tous les seuils.

TABLE 5 – *QEAM vs HRU pour les treillis de 5-dimensions et 6-dimensions*

	5 dimensions		**6 dimensions**	
t	**HRU**	**QEAM**	**HRU**	**QEAM**
0.9	4943892	**4939772**	5091526	**5090329**
0.8	4967327	**4967200**	5114865	**5113650**
0.7	**5000283**	**5000283**	5156049	**5142962**
0.6	5052126	**5051328**	5186824	**5179914**
0.5	5106501	**5099592**	5245149	**5229696**
0.4	5228176	**5190686**	5311518	**5295998**
0.3	5337020	**5299322**	5429357	**5385457**
0.2	**5479742**	**5479742**	5554820	**5552444**
0.1	**5864309**	**5864309**	5890921	**5866430**

Pour approfondir les tests de performance de QEAM, nous l'avons comparé à GEA pour trois ensembles (treillis de cinq, six, et sept dimensions). Les Tables 6, 7, et 8 contiennent les résultats de trente exécutions indépendantes de QEAM et GEA sur des treillis à cinq, six, et sept dimensions, respectivement. Pour comparer les deux algorithmes, nous comparons des indicateurs statistiques tels que min TPC, max TPC, et TPC moy. En observant la Table 6, on peut remarquer que QEAM surpasse GEA pour tous les seuils appliqués sur le treillis à cinq dimensions sauf pour $t = 0.7$. Ce qui semble être la meilleure solution trouvée par GEA (min TPC) est presque toujours le TPC moyen de QEAM pour les valeurs de la contrainte t entre 0.2 et 0.6, c'est-à-dire que QEAM a trouvé cette solution pour chaque exécution puisque TPC moy = min TPC = max TPC. Autrement, pour le reste des valeurs de t (0.8 et 0.9), nous remarquons que QEAM surpasse GEA, sauf pour le min TPC de 0.8 où les deux algorithmes ont

TABLE 6 – *QEAM vs GEA pour un treillis de 5-dimensions*

t	**0.2**			**0.3**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	6329699	6329699	6329699	**5832429**	**5832429**	**5832429**
GEA	6329699	6329699	6329699	5833793	**5832429**	5836932
t	**0.4**			**0.5**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5517559**	**5517559**	**5517559**	**5363293**	**5363293**	**5363293**
GEA	5550006	**5517559**	5603151	5370767	**5363293**	5408040
t	**0.6**			**0.7**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5211504**	**5211504**	**5211504**	5131318	**5129229**	5141761
GEA	5222253	**5211504**	5261172	**5129990**	**5129229**	**5134056**
t	**0.8**			**0.9**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5057897**	**5057886**	**5057992**	**5005221**	**5003802**	**5005982**
GEA	5059288	**5057886**	5066340	5007811	5004902	5018762

trouvé les mêmes valeurs.

Pour le treillis à six dimensions, la Table 7 montre que QEAM trouve de meilleures solutions puisque ses TPS moyen, minimum et maximum sont meilleurs que ceux de GEA, sauf pour le seuil 0.9 où GEA trouve un TPC min similaire et des TPC moyen et max meilleurs. Enfin, pour le treillis à sept dimensions (Table 8), la suprématie de QEAM sur GEA devient très claire puisque QEAM surpasse GEA pour tous les indicateurs statistiques (min, max, et moy), et ce pour tous les seuils sans exception. Cela prouve l'extensibilité du QEAM.

5.4.2 Adaptation de l'Evolution Différentielle

La première méthode que nous avons implémenté est basée sur l'algorithme de l'Evolution Différentielle. Notre adaptation de cet algorithme à la résolution du PSV doit passer par une *discrétisation*. En effet, comme mentionné dans la section 4.4.3, l'Evolution Différentielle est destinée aux problèmes dont l'espace de recherche est continu. Néanmoins, l'espace de recherche du PSV est discret/binaire. Il faut donc trouver une méthode pour rendre cet algorithme applicable sur les problèmes à espace de recherche discret. Il existe plusieurs méthodes de discrétisation dans la littérature. La méthode la plus utilisée est l'utilisation des fonctions de transfert telles que *Sigmoid*

TABLE 7 – *QEAM vs GEA pour un treillis 6-dimensions*

t	**0.2**			**0.3**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**6071667**	**6071139**	**6075149**	**5729729**	**5729290**	**5732187**
GEA	6092932	6074163	6132835	5742271	5732009	5771425
t	**0.4**			**0.5**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5529680**	**5528380**	**5542924**	**5401846**	**5400238**	**5409800**
GEA	5548545	5531299	5573185	5415950	5404208	5448819
t	**0.6**			**0.7**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5302157**	**5299431**	**5305142**	**5223241**	**5222921**	**5224470**
GEA	5309566	5302211	5331387	5231954	5224645	5244760
t	**0.8**			**0.9**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5169821**	**5169548**	**5171193**	5127587	**5124723**	5132179
GEA	5172474	5170130	5177511	**5126254**	**5124723**	**5129101**

et *V-shaped* [CSA+17]. Une des méthodes également utilisée est « Quantum Binary Approach » [CSA+17]. Cette méthode consiste à utiliser les quantums comme des intermédiaires qui représentent des probabilités. Ces probabilités décident si la valeur de chaque élément doit être à 0 ou bien à 1. Comme les q-bits sont des nombres réels, les métaheuristiques à espace de recherche continu peuvent être appliquées sur une population de quantums. Ces métaheuristiques manipuleront les quantums au lieu de manipuler les vecteurs de solutions. Dans les sections suivantes, nous allons présenter l'algorithme BDE (Binary Differential Evolution) et QDE (Quantum Differential Evolution algorithm) pour résoudre le PSV à contrainte d'espace de stockage [MB21]. Tout comme l'algorithme QEAM, nous avons appliqué la contrainte en fixant la taille du disque à ne pas dépasser lors de la matérialisation de l'ensemble de vues en utilisant l'équation 3.3.

BDE

L'Evolution Différentielle est basée sur les opérations à valeurs réelles et ne résout pas les problèmes d'optimisation de combinaisons. Une façon simple de traiter ce problème consiste à arrondir les variables de v_i générées par l'équation (4.1) à 0 ou 1 en fonction de leur valeur. Mais cette méthode est grossière et manque d'équité. Pour

TABLE 8 – *QEAM vs GEA pour un treillis de 7-dimensions*

t	**0.2**			**0.3**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5961218**	**5958769**	**5968507**	**5693475**	**5690025**	**5698758**
GEA	6008725	5968508	6067438	5720542	5697404	5745253
t	**0.4**			**0.5**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5541371**	**5537741**	**5544734**	**5429844**	**5425179**	**5439560**
GEA	5553033	5543841	5564611	5439568	5433807	5451920
t	**0.6**			**0.7**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5347854**	**5345455**	**5350717**	**5288682**	**5287320**	**5290813**
GEA	5355728	5350362	5363415	5292569	5288728	5303858
t	**0.8**			**0.9**		
	TPC moy	min TPC	max TPC	TPC moy	min TPC	max TPC
QEAM	**5242071**	**5240336**	**5245413**	**5205145**	**5203746**	**5208078**
GEA	5245077	5241625	5249824	5207119	5205017	5209752

remédier à ce problème, les fonctions de transformation ont été utilisées. Une des ces fonctions est *Sigmoid*, présentée par l'équation (5.1).

$$Sig(x) = \frac{1}{(1 + e^{-x})} \tag{5.1}$$

L'idée principale de notre algorithme, nommé BDE, est d'utiliser la fonction de transformation lors de la phase de la génération du vecteur mutant v_i pour s'assurer que ses composants vont avoir la valeur 1 ou 0. Le reste de l'algorithme est identique à l'Evolution Différentielle originale. une nouvelle équation (5.2) est donc ajoutée juste après l'application de l'équation (4.1). L'équation (5.2) est décrite comme suit.

$$v_{ij} = \begin{cases} 1 & : \ if(rand \leq Sig(v_{ij})) \\ 0 & : \ sinon \end{cases} \tag{5.2}$$

Comme mentionné auparavant, la représentation des solutions se fait par des vecteurs binaires. Dans ces vecteurs, la valeur 1 signifie que la vue correspondante est matérialisée, tandis que la valeur 0 signifie qu'elle ne l'est pas. Quant aux solutions violant la contrainte, nous avons utilisé la méthode de réparation, tout comme QEAM.

Les résultats expérimentaux de cet algorithme serons présentés en même temps que celles de l'algorithme QDE, qui sera introduit dans la section suivante.

QDE

La deuxième méthode de *discrétisation* de l'Evolution Différentielle est la « Quantum Binary Approach ». Il s'agit de manipuler une population de quantums au lieu de manipuler une population de solutions, tout comme l'algorithme QEAM. Notre adaptation de cette méthode pour résoudre le PSV se résume en l'algorithme QDE [MB21]. Dans cette adaptation, nous essayons de combiner quelque aspects de QEA pour arriver à appliquer l'Evolution Différentielle pour l'espace de recherche binaire du PSV. Pour ce faire, nous avons réutilisé plusieurs aspects de QEAM, tels que la représentation des Q-bits, la représentation des solutions, le traitement des solutions qui violent la contrainte, et l'application du *Reset Operator*.

Pour les Q-bits, nous avons utilisé une seule variable θ. À partir de θ, les probabilités α^2 et β^2 peuvent être induites. Les solutions sont représentées par des vecteurs de bits, la valeur 1 signifie qu'une vue est matérialisée, tandis que la valeur 0 signifie qu'elle ne l'est pas (voir l'exemple de la Figure 28). Pour les solutions qui violent la contrainte, une fonction de réparation est utilisée pour s'assurer que toutes les solutions proposées par l'algorithme sont faisables. Pour des raisons d'équité lors des comparaisons avec le QEAM, le Reset Operator est également utilisé pour le QDE. Il s'agit de remplacer la population par un autre ensemble de quantums si l'algorithme n'arrive pas à améliorer les solutions pour un nombre déterminé d'itérations.

L'initialisation de la population est également inspirée de QEAM. Nous avons initialisé chaque Q-bit de chaque quantum de la population à la valeur $\frac{\pi}{4}$. En faisant cela, nous assurons que la population explorera un maximum de l'espace de recherche car chaque Q-bit sera à la même distance de 0 et de 1 lors du commencement de l'algorithme. Le pseudo-code du QDE est donné par l'Algorithme 7.

L'Algorithme 7 commence avec l'initialisation de la population des quantums avec la valeur $\frac{\pi}{4}$ de chaque q-bit de ces quantums. Après, nous générons des solutions X_i à partir de ces quantums, en utilisant l'opérateur *measurement*, et en réparant cette solution si elle s'avère infaisable. Ensuite, nous appliquons les phases de *génération du vecteur mutant* et *génération du vecteur trial* sur tous les quantums de la population. Après cela, nous générons de nouvelles solutions X'_i à partir du vecteur trial en utilisant d'opérateur *measurement*. Cette solution sera réparée si elle viole la contrainte. Le quantum Q_i n'est remplacé par le vecteur trial dans la population que si la solution correspondante à ce dernier X'_i est meilleure que la solution X_i du quantum Q_i. Nous appliquons ensuite l'opérateur *quantum interference*, qui n'est autre que le Q-Gate qui suit la meilleure solution globale en utilisant la Table 2. Enfin, le Reset Operator est appliqué si une convergence prématurée est détectée (i.e., la meilleure solution globale

est fixe pour plus de cent itérations).

Algorithm 7: QDE pseudo-code [MB21]

N **:** population size;
n **:** number of views;
f **:** stepsize parameter $\in [0.4, 0.9]$;
c **:** crossover rate $\in [0.1, 1]$;
t **:** maintenance constraint threshold $\in [0.1, 1]$;
C_{max} **:** max space-cost when all vertices are materialized;
ra **:** rotation angle $\in [0.01\pi, 0.1\pi]$;
begin
  Initialize a population of quantums Q_i where $i \in [1, N]$;
  foreach Q_i *(*$i \in [1, N]$*)* **do**
    X_i= measurement(Q_i);
    if *(*$SC(X_i) > C_{max} \times t$*)* **then** Repair ($X_i$);
  end
  repeat
    foreach Q_i *(*$i \in [1, N]$*)* **do**
      r_1 = random int $\in [1, N] : r_1 \neq i$;
      r_2 = random int $\in [1, N] : r_2 \notin \{i, r_1\}$;
      r_3 = random int $\in [1, N] : r_3 \notin \{i, r_1, r_2\}$;
      $v_i = Q_{r1} + f(Q_{r2} - Q_{r3})$; /* v_i : mutant vector */
      J_r = random integer $\in [1, n]$;
      foreach Q_i *(*$i \in [1, N]$*)* **do**
        /* u_i : trial vector */
        if *(*$rand < c$*)or(*$j = J_r$*)* **then** $u_{ij} = v_{ij}$;
        else $u_{ij} = Q_{ij}$;
      end
    end
    foreach $i \in [1, N]$ **do**
      X'_i= measurement(u_i);
      if *(*$SC(X'_i) > C_{max} \times t$*)* **then** Repair ($X'_i$);
      if *(*$TPC(X'_i) < TPC(X_i)$*)* **then** $Q_i = u_i$;
      apply *Q-Gate* on Q_i;
    end
    if *no update on global best for 100 iterations* **then** Apply reset operator;
  until *stop criteria*;
end

Résultats Expérimentaux

Pour tester la performance des deux algorithmes proposés, à savoir BDE et QDE, nous les avons appliqués sur le benchmark TPC-DS. QDE, BDE, et GEA ont été implémentés en utilisant JDK 1.8 sous Windows 10. Les tests ont été effectués sur un PC dont le processeur est un Intel i-5 2,4 GHz, doté de 8 Go de RAM. Suite à plusieurs tests, les paramètres ont été définis comme indiqués dans la Table 9.

TABLE 9 – *Paramètres utilisés pour QDE/GEA*

Parameters	**QDE values**	**BDE values**	**GEA values**
Population size	100	100	100
Nb of generations	1000	1000	1000
Query frequencies	Suivant la distribution Zipf		
Scale factor	0.5	0.5	-
Cross Over rate	0.3	0.3	0.5
Rotation angle γ	0.02π	0.02π	-
Mutation Pb	-	-	0.001

La Table 10 représente les résultats de trente tests séparés de chaque algorithme sur un treillis de cinq dimensions. Dans cette table, nous présentons les indicateurs *min, max*, et *moy*. En observant les résultats de la Table 10, nous pouvons dire que, pour la contrainte $t = 0.2$, les trois algorithmes trouvent le même résultat à chaque exécution. Pour le reste des valeurs de la contrainte t, nous observons que les deux algorithmes proposés (BDE et QDE) surpassent GEA, sauf pour la valeur $t = 0.7$ pour laquelle GEA trouve une meilleure moyenne que QDE. BDE surpasse GEA pour toutes les valeurs de t sans exception. Par ailleurs, si nous comparons BDE avec QDE, nous trouvons qu'ils sont presque à égalité pour ce treillis de cinq dimensions. En effet, ces deux algorithmes trouvent les mêmes valeurs pour chacun des trente tests, et ceci pour les valeurs $t = 0.2$, 0.3, 0.5, et 0.8. Nous observons que QDE surpasse BDE à deux occasions, pour $t = 0.4$ et 0.6 ; tandis que BDE dépasse QDE à deux occasions également, pour $t = 0.7$ et 0.9.

La Table 11 représente les résultats des mêmes tests que ceux de la Table 10 sur un treillis de six dimensions. Nous observons que QDE surpasse GEA pour toutes les contraintes, tout comme sur le treillis à cinq dimensions. Ceci parce que le TCP moyen de QDE est toujours inférieur à celui de GEA, même quand le TPC max trouvé par QDE est plus grand que celui de GEA pour $t = 0.9$. De même, BDE surpasse GEA

pour tous les indicateurs (min, max, et moy), sauf pour $t = 0.6$ où le TPC min du GEA est meilleur que celui du BDE.

Si nous comparons les deux algorithmes BDE et QDE, nous remarquons que ce dernier prend l'avantage pour le treillis de six dimensions, puisque QDE trouve de meilleurs *min*, *max* et *moy*, et ceci pour toutes les valeurs de t, sauf 0.9.

TABLE 10 – *QDE vs. BDE vs. GEA pour un treillis de 5-dimensions*

t	**0.2**			**0.3**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	6329699	6329699	6329699	**5832429**	5832429	**5832429**
BDE	6329699	6329699	6329699	**5832429**	5832429	**5832429**
GEA	6329699	6329699	6329699	5833792	5832429	5836932
t	**0.4**			**0.5**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5517559**	5517559	**5517559**	**5363293**	5363293	**5363293**
BDE	5519138	5517559	5532226	**5363293**	5363293	**5363293**
GEA	5550006	5517559	5603151	5370767	5363293	5408040
t	**0.6**			**0.7**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5211504**	5211504	**5211504**	5130064	5129229	5141761
BDE	5211800	5211504	5213251	**5129229**	5129229	**5129229**
GEA	5222253	5211504	5261172	5129990	5129229	5134056
t	**0.8**			**0.9**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5057886**	5057886	**5057886**	5004711	5003802	5005982
BDE	**5057886**	5057886	**5057886**	**5004499**	5003802	**5004902**
GEA	5059288	5057886	5066340	5007811	5004902	5018762

Pour la Table 12, il s'agit des résultats des trois algorithmes sur un treillis de sept dimensions. Comme pour les deux autres treillis, QDE trouve de meilleurs résultats que GEA, mais cette fois-ci pour touts les indicateurs (min, max, et moy) et pour toutes les valeurs de la contrainte t sans aucune exception. En contre-partie, la performance du BDE diminue considérablement. Si nous le comparons au GEA, nous trouvons que GEA le dépasse pour toutes les valeurs de t sauf pour 0.8 et 0.9, et ce pour tous les indicateurs statistiques.

Comme il est devenu le moins performant des trois algorithmes, BDE est surpassé par QDE pour tous les tests du treillis à sept dimensions.

TABLE 11 – *QDE vs. BDE vs. GEA pour un treillis de 6-dimensions*

t	**0.2**			**0.3**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**6071299**	**6071139**	**6073979**	**5729540**	**5729290**	**5731392**
BDE	6079089	6072632	6087488	5740595	5732461	5763760
GEA	6092932	6074163	6132835	5742271	5732009	5771425
t	**0.4**			**0.5**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5529029**	**5528380**	**5540565**	**5402090**	**5400238**	**5408384**
BDE	5536860	5530630	5541022	5404763	5402261	5408960
GEA	5548545	5531299	5573185	5415950	5404208	5448819
t	**0.6**			**0.7**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5301954**	**5297950**	**5305142**	**5223311**	**5222921**	**5224470**
BDE	5305126	5302508	5306411	5224237	5223830	5224892
GEA	5309566	5302211	5331387	5231954	5224645	5244760
t	**0.8**			**0.9**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5169688**	**5169548**	**5169934**	5125737	5124723	5130635
BDE	5169995	5169810	5170424	**5124895**	5124723	**5125354**
GEA	5172474	5170130	5177511	5126254	5124723	5129101

Pour conclure, nous avons pu constater que les deux algorithmes proposés se comportent différemment. Pour QDE, nous pouvons dire que son rendement est stable vu qu'il a dépassé GEA pour tous les treillis sauf quelques occurrences qui diminuent à chaque fois que la taille du treillis augmente. Néanmoins, pour l'algorithme BDE, il commence avec de bons résultats pour le treillis à cinq dimensions pour lequel il donne des résultats proches de ceux de QDE. Cependant, nous avons constaté qu'il manque d'extensibilité car ses résultats deviennent moins performants en augmentant la taille du treillis.

5.4.3 QEAM vs QDE

Comme les deux algorithmes QEAM et QDE ont surpassé GEA et ont montré une extensibilité assez satisfaisante, nous cherchons maintenant à savoir si l'un de ces deux algorithmes est plus performant que l'autre. Pour ce faire, nous appliquons des tests statistiques pour savoir si la différence des résultats entre les deux algorithmes est

TABLE 12 – *QDE vs. BDE vs. GEA pour un treillis de 7-dimensions*

t	**0.2**			**0.3**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5963869**	**5959376**	**5979263**	**5696082**	**5691748**	**5703421**
BDE	6125139	6068094	6163802	5765242	5739749	5786772
GEA	6008725	5968508	6067438	5720542	5697404	5745253
t	**0.4**			**0.5**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5543028**	**5543144**	**5546942**	**5430002**	**5425390**	**5435547**
BDE	5571943	5560316	5578936	5446191	5442317	5450619
GEA	5553033	5543841	5564611	5439568	5433807	5451920
t	**0.6**			**0.7**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5348563**	**5346448**	**5351116**	**5289462**	**5288078**	**5294044**
BDE	5357407	5353297	5361673	5293094	5290974	5295431
GEA	5355728	5350362	5363415	5292569	5288728	5303858
t	**0.8**			**0.9**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QDE	**5241739**	**5240377**	**5243298**	**5204691**	**5202796**	5207240
BDE	5243932	5241940	5246190	5204890	5203808	**5206042**
GEA	5245077	5241625	5249824	5207119	5205017	5209752

significative ou non. Nous avons choisi d'appliquer le test de Wilcoxon avec un seuil de signification $\alpha = 0.05$.

Dans la Table 13, nous trouvons les résultats des tests de Wilcoxon sur trois treillis (cinq, six, et sept dimensions). Pour chaque valeur de la contrainte t, qui va de 0.3 à 0.8, nous avons utilisé le résultat de trente exécutions séparées des deux algorithmes QEAM et QDE. R^+ représente la *somme des rangs positifs* où QEAM surpasse QDE, tandis que R^- représente la *somme des rangs négatifs* où QDE surpasse QEAM. Nous concluons que la différence est significative (rejet de l'hypothèse nulle) si la valeur *p-value* calculée est inférieure au niveau de signification α.

Pour le treillis à cinq dimensions, le test de Wilcoxon s'avère non applicable, ce qui est dû au fait que les résultats des trente exécutions sont identiques pour toutes les valeurs de la contrainte t, sauf la valeur 0.7 pour laquelle les deux algorithmes donne des résultats différents à sept occasions. La valeur p n'a pas pu être calculée pour ce petit nombre de résultats non identiques. Ainsi, nous pouvons dire que, pour le treillis à

cinq dimensions, les deux algorithmes QEAM et QDE sont à égalité quant à la qualité des solutions proposées.

Si nous observons les résultats de la deuxième partie de la Table 13, qui concerne l'exécution du test de Wilcoxon sur un treillis de six dimensions, nous pouvons constater que, contrairement au treillis de cinq dimensions, les paires de tests sont assez diversifiées pour calculer la valeur p. Néanmoins, la différence entre les résultats des deux algorithmes s'est avérée non significative pour la majorité des valeurs de la contrainte t, sauf pour la valeur $t = 0.4$ pour laquelle nous avons trouvé une valeur $p = 0.008$ en faveur du QDE car R^- est plus grand que R^+. Ainsi, pour le treillis à six dimensions, nous ne pouvons pas conclure si un des deux algorithmes est plus performant que l'autre.

Les résultats qui concernent le treillis à sept dimensions dans la Table 13 sont différents de ceux des deux autres treillis. En effet, la majorité des tests indiquent une différence significative de la qualité des solutions en faveur de QEAM (car R^+ est plus grand que R^-), et ce pour les valeurs de contrainte $t = 0.3$, 0.4, 0.6, et 0.7. Quant aux valeurs restantes (0.5 et 0.8), le test a indiqué que la différence des résultats n'est pas significative ($p > \alpha$). En conséquence, pour le treillis à sept dimensions, nous pouvons dire que QEAM donne de meilleurs résultats que QDE.

Un autre facteur important quand deux algorithmes sont comparés est le calcul du temps nécessaire pour effectuer une exécution dans les mêmes conditions, à savoir un même nombre de générations qui est fixé à 1000 itérations pour les deux algorithmes QEAM et QDE.

La Figure 29 représente un graphe qui compare le temps nécessaire pour effectuer une exécution des deux algorithmes QEAM et QDE sur des treillis de cinq, six, sept, et huit dimensions. En observant la Figure 29, nous pouvons dire que le temps d'exécution nécessaire pour effectuer 1000 itérations par les deux algorithmes est presque identique pour le treillis à cinq dimensions. Ensuite, pour le treillis à six dimensions, nous observons que QDE prend plus de temps. De même pour le treillis à sept dimensions, avec un peu plus de distance que celui de six dimensions. Pour le treillis à huit dimensions, la différence est presque la même que celle des dimensions précédentes. En conclusion, nous pouvons dire que le temps d'exécution des deux algorithmes favorise légèrement QEAM sur QDE.

Compte-tenu des résultats expérimentaux obtenus sur les deux algorithmes QEAM et QDE, il semblerait que QEAM soit plus avantagux que QDE, vu qu'il présente de meilleures solutions pour le treillis à sept dimensions, ce qui prouve son extensibilité face au QDE. Ainsi, QEAM consomme légèrement moins de temps pour effectuer des exécutions.

TABLE 13 – *Test de Wilcoxon entre QEAM et QDE avec un niveau de signification* $\alpha = 0.05$

	5-dimension lattice					
	t=0.3	t=0.4	t=0.5	t=0.6	t=0.7	t=0.8
R^+	-	-	-	-	8	-
R^-	-	-	-	-	20	-
p-value	-	-	-	-	-	-
conclusion	-	-	-	-	NS	-
	6-dimension lattice					
	t=0.3	t=0.4	t=0.5	t=0.6	t=0.7	t=0.8
R^+	27	74	169.5	198.5	40	104.5
R^-	51	251	181.5	236.5	26	195.5
p-value	0.17	0.008	0.44	0.34	0.26	0.09
conclusion	NS	Significatif	NS	NS	NS	NS
	7-dimension lattice					
	t=0.3	t=0.4	t=0.5	t=0.6	t=0.7	t=0.8
R^+	380	386	259	338	369	186
R^-	85	79	206	127	96	279
p-value	0.001	<0.001	0.29	0.015	0.002	0.16
conclusion	Significatif	Significatif	NS	Significatif	Significatif	NS

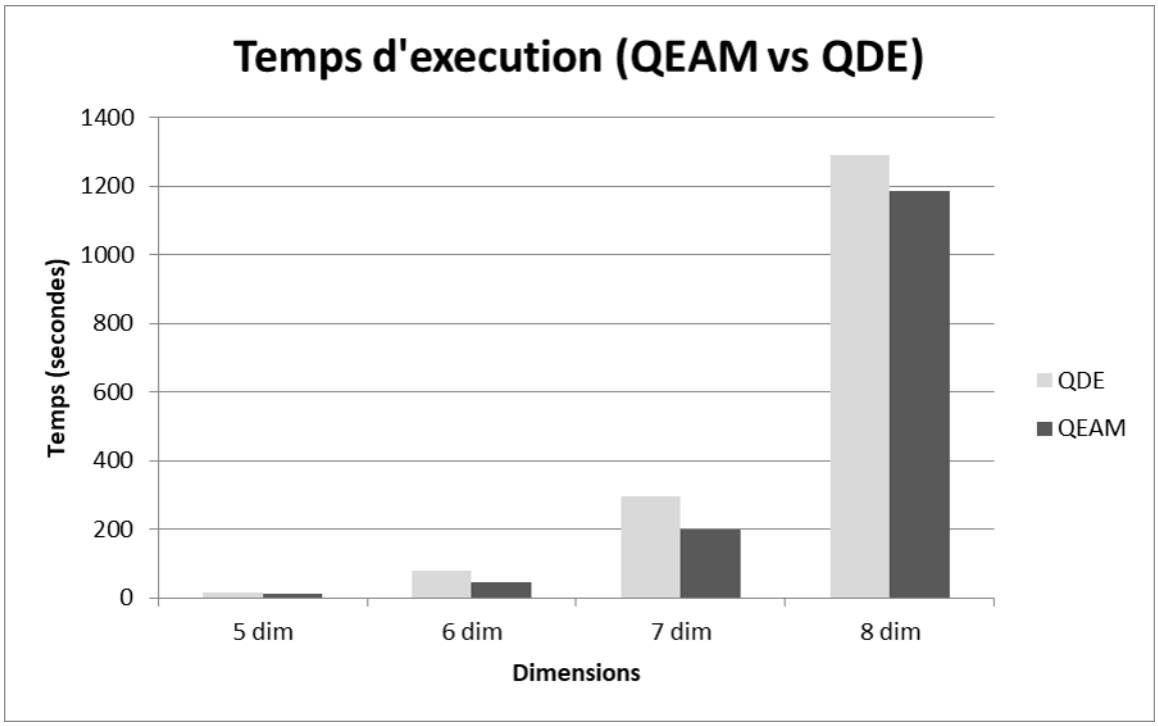


FIGURE 29 – Comparaison du temps d'exécution du QEAM et QDE pour des treillis de cinq à huit dimensions

5.5 Conclusion

Dans le présent chapitre, nous avons proposé deux approches basées sur les algorithmes QEA et DE, expliqués dans le chapitre précédent. La raison du choix de ces deux algorithmes est due aux résultats obtenus par ces derniers dans les travaux de Vijay Kumar *et al.*. Néanmoins, leur adaptation était fortement liée à la contrainte d'espace de stockage, ce qui nous a inspiré à utiliser les mêmes algorithmes en changeant la représentation pour les rendre indépendants de toute contrainte (e.g., espace de stockage ou temps de maintenance). Nos deux algorithmes proposés sont : QEAM et QDE.

Comparé au QEA original, notre adaptation QEAM utilise une représentation différente des q-bits. En effet, nous avons effectué une modification en représentant les q-bits avec une seule variable (l'angle θ) au lieu des deux variables α et β. Cela a allégé la représentation des quantums, et a facilité l'application du Q-Gate.

Contrairement au QEA, DE est destiné à des problèmes d'optimisation dont l'espace de recherche est continu. Comme l'espace de recherche du PSV est discret/binaire, nous avons essayé deux méthodes de *discrétisasion*. La première était l'utilisation des fonctions de transformation dans l'algorithme BDE, et la deuxième était utilisation de

l'approche *quantum binary* dans l'algorithme QDE. Les résultats expérimentaux ont montré que QDE est plus extensible que BDE.

Pour éviter l'optimum local, nous avons utilisé le « Reset Operator » pour les deux algorithmes proposés. Il constitue un nouveau point de départ si les algorithmes n'arrivent pas à améliorer les résultats en un nombre défini d'itérations. Aussi, nous avons traité les solutions qui violent la contrainte par la méthode de réparation. Nous attirons l'attention sur le fait que cette méthode n'est pas coûteuse dans le contexte de la contrainte d'espace de stockage, car son application est simple et directe et ne consomme pas de temps de calcul. Néanmoins, si nous comptons opter pour la même méthode lors du traitement du PSV à contrainte de temps de maintenance, cela va créer un problème de temps de calcul.

Les résultats expérimentaux ont montré que, comparé à l'algorithme génétique, nos deux algorithmes proposés QEAM et QDE donnent des solutions de meilleure qualité. Quand ces derniers sont comparés entre eux, les résultats ont montré un léger avantage pour le QEAM en terme de temps d'exécution, ainsi que pour la qualité des solutions fournies pour les grandes instances.

Dans le chapitre suivant, nous allons étudier le PSV à contrainte de temps et de maintenance, et nous présenterons nos contributions pour cette variante du PSV.

Chapitre 6

Algorithmes pour Résoudre le PSV à Contrainte de Temps de Maintenance

6.1 Introduction

Dans les chapitres précédents, nous avons étudié le PSV à contrainte d'espace de stockage. Après avoir constaté que l'espace de stockage pose moins de problèmes en tant que contrainte à cause de l'évolution de la technologie des disques (notamment en terme d'espace), la contrainte du temps de maintenance est devenue le centre d'intérêt des recherches. Comme nous l'avons mentionné auparavant, l'ED se met à jour périodiquement et, étant donné que les vues matérialisées existent physiquement dans un espace séparé de l'ED, elles ont besoin de maintenance (mise à jour) à leur tour. Ce temps de mise à jour nécessite une ressource de calcul, ce qui rend le défi beaucoup plus difficile que la contrainte de l'espace de stockage, qui ne nécessite que des ressources de stockage.

De plus, si nous observons l'équation du calcul de la contrainte de l'espace de stockage (équation 3.2/3.3), nous pouvons constater qu'elle est simple et facile à appliquer face à l'équation de calcul du temps de maintenance (équation 3.4). La dernière ressemble en effet plus à l'équation du calcul du TPC (équation 3.1). La difficulté du calcul de la contrainte entraîne des conséquences sur le temps d'exécution des algorithmes de solution, surtout si la méthode de réparation est adoptée lors du traitement des solutions infaisables – ce qui est le cas dans nos approches proposées. Pour limiter le problème du temps de calcul, nous avons proposé l'utilisation de la mémoire lors du calcul des MC de chaque solution. Une extension a également été proposée pour le calcul du TPC.

Dans un premier temps, nous avons essayé de résoudre le PSV à contrainte de temps de maintenance avec les algorithmes QEAM et QDE, proposés pour le PSV à contrainte d'espace de stockage. Les résultats n'étaient pas aussi satisfaisants que ceux du chapitre

précédent, notamment pour le QEAM. Nous avons donc proposé une nouvelle approche basée sur l'algorithme CBO (présenté dans le chapitre 4). Les résultats expérimentaux valident l'algorithme proposé, nommé QCBO, face aux autres approches.

Le présent chapitre sera organisé comme suit :

Nous allons dans un premier temps présenter les travaux de Xu Yu *et al.*. Ensuite, nous allons présenter en détails notre algorithme QCBO. Enfin, une étude expérimentale pour valider le QCBO sera présente, juste avant la conclusion du chapitre.

6.2 Travaux de Xu Yu *et al.*

Un des rares travaux ayant traité le PSV à contrainte de temps de maintenance en utilisant le treillis comme framework de représentation de hiérarchie entre les vues est le travail de Xu Yu *et al.* [YYCG03]. Dans ce travail, les auteurs ont appliqué un algorithme évolutionnaire. Pour l'opérateur *crossover*, ils ont utilisé l'opérateur *Uniform Cross Over*, expliqué dans le chapitre 4. La *mutation* a été appliquée avec une certaine probabilité P_m sur chaque élément de chaque vecteur de solution.

La particularité de ce travail réside dans l'opérateur de *sélection*. Dans cette phase, l'algorithme doit choisir l'ensemble qui va survivre à la génération suivante. Cependant, comme expliqué dans le chapitre précédent, des solutions infaisables peuvent survenir. Xu Yu *et al.* ont indiqué que l'utilisation des fonctions de pénalité n'était pas un bon choix, et ne produisait pas une bonne qualité de solutions. Ainsi, ils ont choisi de gérer les solutions infaisables par la méthode *stochastic ranking*, inspirée du travail de Runarsson *et al.* [RY00].

L'idée principale du *stochastic ranking*, présentée dans [RY00], est l'introduction d'une probabilité P_f pour assurer une *sélection* basée sur le classement. Au cours du classement, il faut comparer des paires de deux individus adjacents. S'ils ont tous deux des solutions faisables, ils sont naturellement comparés selon la fonction objectif. Cependant, si l'un d'entre eux est infaisable, la probabilité de les comparer selon la fonction objectif sera P_f, tandis que la probabilité de les comparer selon la fonction de pénalité sera $1 - P_f$. Puisque P_f est une probabilité, elle donne l'occasion aux deux fonctions (objectif et pénalité) de classer une paire. Quand $P_f > \frac{1}{2}$, le classement est biaisé vers la fonction objectif. Sinon, il est biaisé vers la fonction de pénalité. Il faut donc ajuster la probabilité P_f pour assurer l'équilibre entre la fonction objectif et la fonction de pénalité.

Xu Yu *et al.* ont adapté le principe de *stochastic ranking* dans un algorithme génétique nommé EA. Comme pour tout algorithme évolutionnaire, le *crossover* et la *mutation* ont été utilisés. L'opérateur de crossover employé est le *uniform crossover*. L'opérateur de mutation utilisé est similaire à celui le plus souvent utilisé. La principale différence par rapport à la plupart des algorithmes évolutionnaires est l'utilisation

de la *sélection* basée sur le *stochastic ranking*. Il s'agit de trier l'union des individus de l'ancienne et de la nouvelle génération d'une façon similaire à l'algorithme de tri à bulles, tout en suivant le principe de *stochastic*, expliqué dans [RY00].

Les résultats expérimentaux ont validé cette approche comparée à l'algorithme LEE de [LH01] qui a utilisé un algorithme évolutionnaire avec fonction de pénalité. Nous avons implémenté cette méthode et nous avons constaté qu'en effet, elle présente des solutions de bonne qualité, surtout pour le cas de la contrainte de temps de maintenance.

6.3 Contribution à la résolution du PSV à contrainte de Temps de Maintenance

La présente section présentera les différentes tentatives pour proposer une approche qui traite le PSV à contrainte de temps de maintenance. La première idée triviale est d'essayer les deux algorithmes QEAM et QDE précédemment proposés pour la contrainte d'espace de stockage, et de voir si ces deux approches vont donner des résultats satisfaisants pour la contrainte de temps de maintenance.

6.3.1 QEAM vs. QDE vs. EA

Pour tester les algorithmes QEAM et QDE, nous les avons comparés à l'algorithme EA de Xu Yu *et al.* [YYCG03]. La Table 14 contient la moyenne de trente exécutions séparées de chaque algorithme sur deux treillis à six et sept dimensions, avec variation du seuil t entre 20% et 90%.

En observant la Table 14, nous pouvons constater que QEAM ne rivalise plus avec QDE comme pour la contrainte d'espace. En effet, la qualité des solutions de l'algorithme QEAM a considérablement baissé si on le compare à l'algorithme QDE. L'écart entre le TPC des deux algorithmes est très grand, et ce pour toutes les valeurs de la contrainte t pour les deux treillis. D'autre part, nous observons que QDE est plus compétitif, mais il n'arrive pas à donner des résultats de qualité face au EA qui fonctionne très bien pour cette variante de PSV. QDE n'arrive à dépasser EA que pour deux valeurs de la contrainte t, i.e., 0.2 et 0.3 du treillis à six dimensions. Concernant le treillis à sept dimensions, EA dépasse QDE pour toutes les valeurs de t.

En résumé, les deux algorithmes QEAM et QDE, qui fonctionnaient bien pour le PSV à contrainte d'espace de stockage, ne sont pas aussi performants lors de la résolution du PSV à contrainte de temps de maintenance. En effet, cette variante du PSV n'est pas aussi simple que celle à contrainte d'espace. Kalnis *et al.* [KMP02] expliquent

TABLE 14 – *QEAM vs. EA vs. QDE pour les treillis de 6-dimensions et 7-dimensions*

	6 dimensions			**7 dimensions**		
t	**QEAM**	**EA**	**QDE**	**QEAM**	**EA**	**QDE**
0.2	5691385	5584828	**5565062**	5883976	**5619766**	5675884
0.3	5500915	5415318	**5413497**	5648981	**5474364**	5516468
0.4	5374072	**5309944**	5314370	5503857	**5380535**	5420541
0.5	5287898	**5241880**	5243970	5413442	**5310349**	5346004
0.6	5230521	**5186115**	5191452	5358712	**5265796**	5294389
0.7	5187462	**5146883**	5149982	5335813	**5229829**	5253182
0.8	5163581	**5115814**	5118813	5339943	**5202572**	5222635
0.9	5158503	**5091245**	5094025	5339363	**5181035**	5216139

que, même si les deux variantes du PSV semblent similaires, il existe une grande différence entre les deux contraintes. Ils attestent que la nature non monotone du coût de maintenance accroît la complexité des algorithmes de sélection des vues. Dans notre cas, les deux approches proposées pour résoudre le PSV à contrainte d'espace ne sont pas efficaces pour le cas de la contrainte de temps de maintenance. Cette constatation nous pousse à proposer une nouvelle approche pour laquelle nous devons assurer un bon rapport exploration/exploitation, ce qui permettra une meilleure recherche dans l'espace de recherche.

6.3.2 QCBO

Une nouvelle approche basée sur l'algorithme CBO [KM14] - détaillé dans la section 4.4.5 - est proposée dans un algorithme nommé QCBO [MB20a]. CBO est une métaheuristique créée en 2014. La raison principale du choix de cette métaheuristique est son indépendance des paramètres. En effet, elle ne contient qu'un seul paramètre, i.e., le coefficient de restitution ε, auto-ajustable au cours du processus. Tout comme le DE, l'algorithme CBO est destiné aux problèmes dont l'espace de recherche est continu. Comme nous l'avons conclu lors de la proposition de la version binaire du DE, nous allons appliquer l'approche « Quantum Binary » pour *discrétiser* le CBO. Nous avons réutilisé plusieurs aspects du QDE lors de l'application du QCBO, notamment la représentation des Q-bits, le vecteur binaire qui représente les solutions, et le *Reset Operator*. Par la représentation des Q-bits, nous voulons signifier l'utilisation d'une seule variable θ pour représenter les deux probabilités α^2 et β^2. Ainsi, nous avons uti-

lisé la même représentation des solutions que nous utilisons pour toutes les approches proposées jusqu'à présent : pour un vecteur binaire de taille N (N étant la cardinalité de l'ensemble des vues), la valeur 1 signifie la matérialisation de la vue de l'indice correspondant (Figure 28 illustre un exemple).

L'initialisation de la population se fait également de la même façon que pour le QDE. Nous donnons la valeur $\frac{\pi}{4}$ pour avoir la même probabilité d'obtenir 0 ou 1 à chaque Q-bit de chaque *quantum vector*.

L'idée principale de QCBO se résume comme suit.

La population est un ensemble de *quantum vectors*. Chaque *quantum vector* Q_i est considéré comme un objet de collision (CB). Après l'application de l'opérateur *measurement*, une solution X_i est générée à partir de chaque objet correspondant CB_i. Les CBs sont triés dans un ordre croissant suivant la masse (fonction objectif) de leurs solutions correspondantes. Ensuite, les CB sont divisés en deux groupes : le groupe stationnaire, et le groupe mobile. Le groupe mobile entre en collision avec le groupe stationnaire par paires en utilisant les équations (4.12) et (4.14), comme dans CBO original. Le groupe stationnaire, quant à lui, se déplace vers la meilleure solution globale en utilisant l'opérateur Q-gate de l'algorithme QEA. La mise à jour des positions des CBs est donnée par la Figure 30. Les étapes précédentes sont répétées jusqu'à ce que certains critères d'arrêt soient satisfaits.

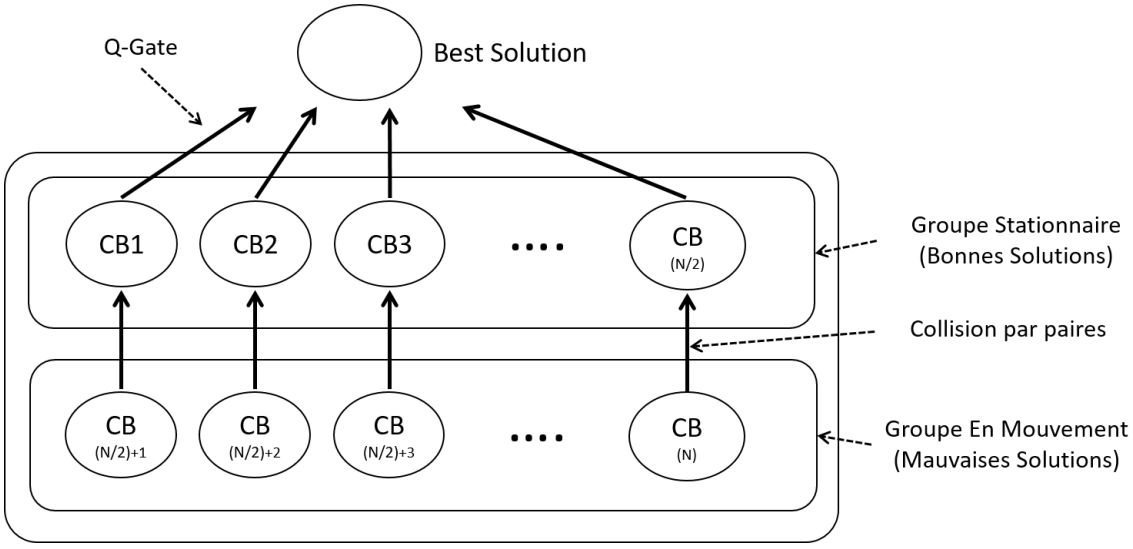


FIGURE 30 – Mise à jour de la position des groupes (stationnaire et mobile) dans QCBO

Une des différences entre le QDE et le QCBO quant à la discrétisation en utilisant l'approche du quantum est le besoin du tri des *quantum vectors* (ou des CBs). Dans notre approche, comme les CBs ne sont pas des vecteurs de solution, ils ne peuvent pas

être évalués directement. Par conséquent, nous considérons l'évaluation du *fitness* d'un CB, qui est un *quantum vector*, comme le *fitness* de son vecteur solution correspondant après application de l'opérateur *measurement*. En faisant cela, un même CB peut être évalué/classé différemment durant deux générations distinctes, ce qui favorise la diversification dans un premier temps.

Une autre différence entre QDE et QCBO est l'utilisation de la fonction de réparation. Dans un premier temps, pour le QDE, la fonction de réparation choisissait les éléments à dématérialiser à chaque fois d'une manière aléatoire. Pour QCBO, nous voulons choisir soigneusement les éléments à enlever de façon à avoir une des meilleures solutions possibles après réparation. Néanmoins, l'application de la fonction de réparation n'est pas aussi facile que pour la contrainte d'espace. La section suivante explique la difficulté de l'évaluation de la contrainte du temps de maintenance.

Problème de l'évaluation de la contrainte

L'évaluation de la contrainte est une fonction très sollicitée par n'importe quel algorithme utilisé. Il s'agit de classer une solution comme faisable, ou non faisable, selon la valeur calculée par la fonction d'évaluation de contrainte. Dans le chapitre précédent, nous avons fait face à la contrainte d'espace de stockage. La fonction d'évaluation de cette contrainte était très simple et facile à appliquer. Il s'agissait d'une simple somme des tailles des vues matérialisées (équation 3.3). Cependant, pour ce chapitre, nous traitons la contrainte du temps de maintenance où la fonction d'évaluation de la contrainte est compliquée (équation 3.4). Cela va consommer un grand temps de calcul, particulièrement lors de l'exécution des métaheuristiques à base de population, puisque cette fonction est appliquée sur chaque individu de la population pour le valider comme faisable, ou non faisable, avant de passer à l'itération suivante. De plus, l'utilisation de la fonction de réparation aggrave encore plus ce cas. En effet, dans la fonction de réparation, nous enlevons un élément à la fois de la liste des vues matérialisées, puis nous calculons juste après la contrainte pour savoir s'il est réparé ou pas. Cette phase risque d'être répétée plusieurs fois jusqu'à la réparation de la solution. Cette répétition entraîne des appels supplémentaires à la fonction d'évaluation de la contrainte.

Pour parer à ce problème du temps consommé par la fonction d'évaluation de la contrainte, nous avons utilisé la *mémoire d'évaluation*. Cette méthode sera expliquée en détaille dans la section suivante.

La mémoire d'évaluation

En informatique, la mémoire est une structure ou un dispositif qui permet de recueillir et de conserver des informations. Nous voulons utiliser ce principe pour garder des informations sur les solutions déjà passées lors d'une itération. En effet, la *mémoire d'évaluation* est une structure qui enregistre l'information qu'une solution a déjà passé

par le calcul de la fonction d'évaluation et, au lieu de recalculer la contrainte pour cette solution, le résultat stocké dans la mémoire est utilisé directement. En faisant cela, nous pouvons assurer qu'une solution ne consommera un temps de calcul pour évaluation qu'une seule fois durant toute l'exécution. La meilleure structure pouvant assurer ce principe de mémoire est la *table de hachage*. La table de hachage est une structure de données qui permet une association clé–valeur. L'accès aux données devient très rapide lorsque nous connaissons la clé de la donnée désirée. Dans notre cas, nous devons trouver une clé unique pour chaque solution, ce que nous avons trouvé facile grâce à la représentation que nous utilisons. Il s'agit d'une représentation binaire où chaque combinaison correspond à un nombre unique au système binaire (numérotation à base 2). Ce nombre unique de chaque combinaison (solution dans notre algorithme) peut jouer le rôle d'un identifiant de cette solution dans la table de hachage. L'idée est donc de prendre le code binaire de la solution, de le convertir en un chiffre décimal, et de chercher s'il existe dans la table de hachage. Si nous trouvons ce chiffre, nous déduisons que cette solution à été déjà évaluée auparavant, et nous utilisons directement la valeur de l'évaluation stockée dans la table. Autrement, nous calculons la valeur MC de la présente solution, et nous insérons cette valeur dans la table de hachage en utilisant la solution comme identifiant/clé. La Figure 31 explique le principe de la mémoire d'évaluation.

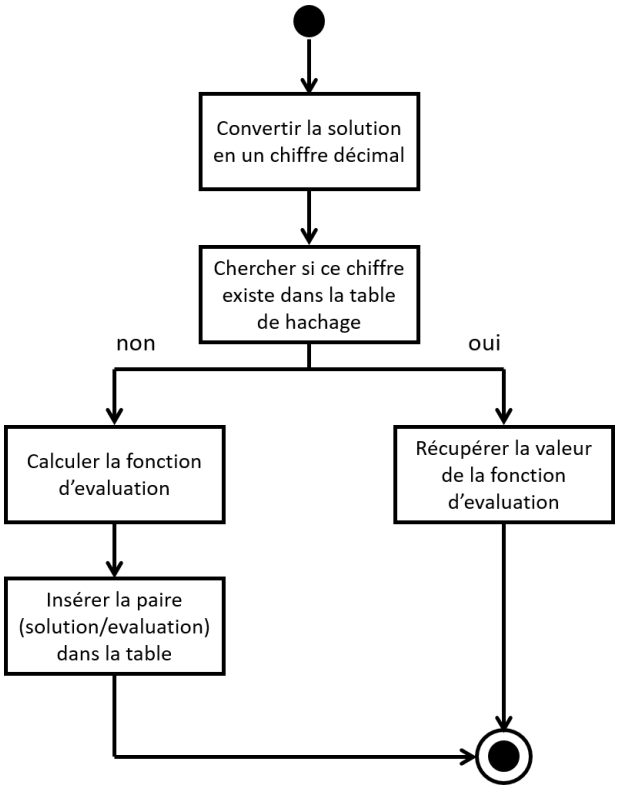


FIGURE 31 – Principe de la mémoire d'évaluation

Comme le principe de la fonction objectif est proche de celui de la fonction d'évaluation de la contrainte du temps de maintenance, nous pouvons également utiliser la mémoire d'évaluation pour la fonction objectif. L'application de la mémoire d'évaluation sur les algorithmes nous fait gagner davantage de temps d'exécution. Cela nous permettra d'utiliser une méthode plus compliquée qui nécessite plusieurs utilisations des fonctions objectif et d'évaluation. La nouvelle fonction de réparation sera expliquée dans la section suivante.

Nouvelle fonction de réparation

La fonction de réparation est une fonction qui transforme une solution infaisable en une solution faisable de son entourage. Dans un premier temps, nous avons utilisé une fonction de réparation qui choisit une solution voisine de manière aléatoire. Pour rendre cette fonction plus efficace, nous avons réfléchi à prospecter le voisinage des solutions infaisables pour choisir la meilleure d'entre elles. Pour ce faire, une des meilleures façons est d'utiliser le principe inverse de l'algorithme HRUA. Ce choix a été inspiré du travail de Lin *et al.* [LK04], qui ont proposé un algorithme de réparation nommé *greedy repair* pour réparer les solutions infaisables dans le cadre de la résolution du PSV à contrainte d'espace. Mais, encore une fois, l'utilisation d'une telle méthode ne pose pas de problème pour la contrainte d'espace. Quant à la contrainte de temps de maintenance, nous avons décidé d'utiliser cette méthode après avoir optimisé le temps d'exécution en utilisant la *mémoire d'évaluation*. La fonction de réparation est donnée par l'Algorithme 8.

Algorithm 8: Greedy Repairing Function

```
Input: X : Ineasable Solution ;
Output: X' : Repaired Solution ;
Function Greedy Repair(X) :
    X' = X ;
    while MC(X') > constraint threshold do
        foreach Materialized View in X' (column value 1) do
            Dematerialize (set value to 0);
            Calculate TPC(X');
            Rematerialize;
        end
        Dematerialize the view with least benefit from X';
    end
    return X';
End Function
```

Le Pseudo-code de QCBO

L'Algorithme 9 représente le pseudo-code de QCBO. La première partie ressemble beaucoup à QDE. Il commence avec l'initialisation de la population des CBs avec la valeur $\frac{\pi}{4}$ pour chaque q-bit des quantums. Ainsi, des solutions sont générées en utilisant l'opérateur *measurement* sur les CBs. Ces solutions sont réparées en utilisant la fonction *Greedy Repair* de l'Algorithme 8 si elles s'avèrent infaisables. Ensuite, les CBs sont triés dans un ordre croissant selon les *masses* (équation 4.9) de leurs solutions correspondantes. La première moitié, qui constitue le groupe stationnaire, utilise le Q-gate de QEA pour essayer de suivre la meilleure solution globale avec un angle de rotation ra ; tandis que le deuxième groupe, qui constitue le groupe en mouvement, entre en collision par paires avec le groupe stationnaire en utilisant les équations (4.12) et (4.14). Enfin, le *Reset Opeator* est utilisé en cas de convergence prématurée.

Algorithm 9: QCBO pseudo-code [MB20a]

N **:** population size;
t **:** maintenance constraint threshold $\in [0.1, 1]$;
C_{min} **:** max space-cost when all vertices are materialized;
ra **:** rotation angle $\in [0.01\pi, 0.1\pi]$;
begin
  Initialize a population of Colliding Bodies CB= { cb_i / $i \in [1; N]$ } with the value $\frac{\pi}{4}$ in every cb's column;
  while *stop criteria not reached* **do**
    foreach $cb_i / i \in [1; N]$ **do**
      X_i= measurement(cb_i);
      if *(*$MC(X_i) > C_{min} \times t$*)* **then** Greedy Repair (X_i);
    end
    Sort the population in an ascending oreder based on the masses of cbs (equation 4.9);
    foreach $cb_i / i \in [1; \frac{N}{2}]$ **do**
      // stationary group
      Apply Q-gate on cb_i ;
    end
    foreach $cb_i / i \in [\frac{N}{2} + 1; N]$ **do**
      // moving group
      cb_i collides with $cb_{i-\frac{N}{2}}$ using equations (4.12) and (4.14) ;
    end
    if *no update on global best for 100 iterations* **then** Apply reset operator;
  end
end

6.3.3 Aperçu de l'exploration/exploitation de QCBO

Comme mentionné dans le travail de Črepinšek *et al.* [vLM13], il est difficile de distinguer le rôle d'exploration et d'exploitation, même pour les opérateurs les plus basiques (sélection, croisement, et mutation) pour les algorithmes évolutionnaires. Vu de différents points de vue, chaque opérateur peut être considéré comme un opérateur d'exploitation ou d'exploration selon la façon dont il est utilisé. Dans cette section, nous allons clarifier le rôle de chaque opérateur et concept utilisé dans QCBO. Črepinšek *et al.* [vLM13] ont également expliqué une bonne façon de traiter les opérateurs, en indiquant que, généralement, l'algorithme devrait commencer par l'exploration et passer progressivement à l'exploitation. Dans ce travail, nous essayons d'appliquer ce concept sur tous les opérateurs quand cela est possible. Nous utilisons une approche uni-processus, où chaque opérateur est indépendamment responsable de l'équilibre entre l'exploration et l'exploitation. Cependant, certains opérateurs peuvent avoir tendance à privilégier l'exploration plutôt que l'exploitation, et vice-versa.

Représentation des CBs

Dans QCBO, la représentation de la population, qui est un ensemble de *quantum vectors*, contribue clairement à l'équilibre exploration/exploitation. Puisque nous avons à faire à des probabilités plutôt qu'à des solutions, un même CB peut générer des solutions différentes, surtout lorsque les probabilités d'avoir 0 et 1 sont presque les mêmes dans les composants qui le constituent. Mais, au fur et à mesure que nous avançons dans le processus, ces probabilités vont pencher d'un côté, résultant en des solutions très similaires générées à chaque fois par un même CB. Ainsi, la représentation elle-même est considérée comme un opérateur d'exploration au début (puisque nous commençons à $\frac{\pi}{4}$ pour chaque q-bit), et elle se transforme en opérateur d'exploitation au fur et à mesure que nous avançons dans le processus.

Mise à jour des positions des CBs

En gardant la même logique utilisée dans l'algorithme CBO originel, à chaque génération, QCBO divise la population en deux groupes. Les meilleures solutions sont considérées comme le groupe stationnaire, tandis que le reste est considéré comme le groupe mobile.

Tout d'abord, le groupe mobile entre en collision par paires avec le groupe stationnaire en utilisant les équations (4.12) et (4.14). La formule (4.12) utilise ε (coefficient de restitution), un coefficient dynamique qui est calculé à chaque génération en utilisant la formule (4.8). Kaveh et Mahdavi [KM14] ont expliqué qu'en utilisant la formule (4.8), ε garantit une collision inélastique, favorisant la recherche globale (exploration) au début, et la recherche locale (exploitation) par la suite.

Quant au groupe stationnaire, chaque CB utilise l'angle de rotation (ra), pour se déplacer vers la meilleure solution globale en utilisant le Q-gate. Nous avons choisi de guider cet opérateur plutôt vers l'exploitation pour créer l'équilibre, car nous avons utilisé certains opérateurs orientés vers l'exploration seulement, comme il sera mentionné plus loin.

Dans QCBO, la position de chaque CB est mise à jour, même si sa nouvelle position est pire que l'ancienne. Ce concept de non-sélection peut lui-même être considéré comme un opérateur d'exploration, car nous continuons à explorer de nouvelles zones de l'espace de recherche, même si nous perdons de bonnes positions de CB.

La fonction *Greedy Repair* et le *Reset Operator*

Lors de la réparation d'une solution, QCBO utilise la fonction *Greedy Repair* donnée par l'Algorithme 8, qui représente une version inverse de l'algorithme glouton HRUA. La fonction *Greedy Repair* s'assure que la nouvelle "solution réparée" est une des meilleures solutions dans le voisinage de l'ancienne solution infaisable. Par conséquent, la fonction de réparation peut être considérée comme un concept d'exploitation.

De l'autre coté, nous avons utilisé le *Reset Operator* mentionné précédemment, qui est considéré comme un opérateur orienté vers l'exploration uniquement, utilisé lorsque nous soupçonnons être bloqués autour de l'optimum local.

Récapitulation

Dans l'ensemble, QCBO utilise deux opérateurs orientés vers l'exploitation (i.e., le Q-gate et la fonction de réparation), deux opérateurs orientés vers l'exploration (i.e., l'opérateur de non-sélection et le *Reset Operator*), et deux opérateurs adaptatifs (i.e., la représentation des Objets de Collision et le coefficient de restitution ε). Tout ceci conduit à un juste équilibre entre l'exploration et l'exploitation, et explique les bons résultats donnés par la suite dans la section expérimentale.

6.3.4 Résultats Expérimentaux

Cette section étudie les performances de l'algorithme QCBO proposé, en utilisant le benchmark TPC-DS.

Dans un premier temps, nous allons présenter une analyse de comportement sur QCBO afin de justifier les différents choix adoptés pour assurer le bon équilibre entre exploration et exploitation. Ensuite, nous allons effectuer une étude comparative entre QCBO et d'autres algorithmes, tels que HRUA, CBO qui utilise les fonctions de transformation, et EA, qui est un des algorithmes les plus performants lors du traitement du PSV à contrainte de temps de maintenance.

Tous les tests ont été effectués en utilisant JDK 1.8 sous Windows 10 sur un PC dont le processeur est un Intel i-5 2,4 GHz, doté de 8 Go de RAM.

Analyse de comportement

Le premier paramètre que nous avons étudié pour le QCBO est le nombre de solutions infaisables. Selon Schoenauer et Michalewicz [SM96], la meilleure solution globale se trouve très souvent à la limite des régions faisables. Par conséquent, la plupart des travaux traitant des problèmes combinatoires contraints effectuent des tests de faisabilité. Dans nos travaux, comme mentionné précédemment, nous ne permettons pas des solutions infaisables. Néanmoins, nous pouvons utiliser le nombre de réparations effectuées pendant l'exécution de l'algorithme pour repérer les solutions infaisables.

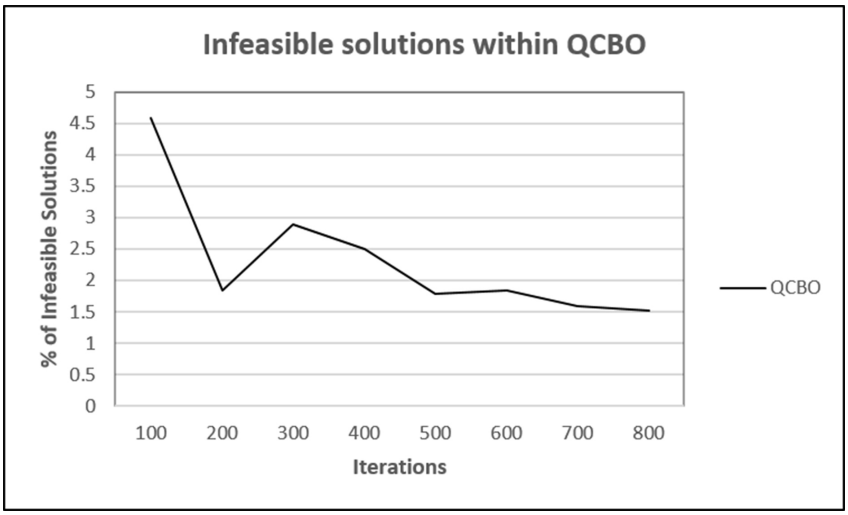


FIGURE 32 – Pourcentage des solutions infaisables sur 800 itérations

La Figure 32 est un graphe présentant le pourcentage de solutions infaisables pour chaque 100 itérations de QCBO. Nous avons exécuté QCBO sur un treillis à cinq dimensions avec un seuil de maintenance $t = 0.8$ sur 800 itérations. Le pourcentage de solutions infaisables est donné en divisant le nombre de réparations effectuées dans les 100 itérations sur 10000 (la taille de la population étant de 100). On peut noter que le pourcentage de solutions infaisables dans les 100 premières itérations est d'environ $4,5\%$, ce qui est relativement faible. Ceci s'explique par le seuil de maintenance élevé (i.e., $t = 0.8$). Cependant, au fur et à mesure que l'on avance, le pourcentage diminue considérablement, jusqu'à atteindre $1,5\%$ (moins que la moitié) à la 800ème itération. De ce comportement, nous pouvons déduire qu'au cours des itérations, QCBO présente des solutions plus proches des régions faisables de l'espace de recherche. En cherchant la meilleure solution dans les limites des régions faisables, QCBO a plus de chance de trouver des solutions intéressantes. De plus, dans le cas de cette approche, éviter une

solution infaisable signifierait un gain de temps d'exécution puisqu'il n'y aura pas de temps perdu à faire des réparations inutiles.

TABLE 15 – *Le gain en temps grâce à l'utilisation du ME*

(a) *ME sur le treillis à 5-dimensions*

	5 dimensions		
	ME	**no ME**	**% gain**
QCBO	5.38	12.38	56.57%
QEAM	2.63	15.50	83.06%
QDE	9.13	41.88	78.21%

(b) *ME sur le treillis à 6-dimensions*

	6 dimensions		
	ME	**no ME**	**% gain**
QCBO	32	60.25	46.89%
QEAM	28.63	49.63	42.32%
QDE	91	132.13	31.13%

(c) *ME sur le treillis à 7-dimensions*

	7 dimensions		
	ME	**no ME**	**% gain**
QCBO	122	192.88	36.75%
QEAM	140	237.75	41.11%
QDE	405.13	753.38	46.23%

Le deuxième paramètre que nous avons étudié est l'effet de la mémoire d'évaluation (ME). La Table 15 présente le résultat de l'exécution de QCBO, QEAM, et QDE avec et sans l'utilisation du ME, pour dix exécutions distinctes de chaque algorithme en changeant le taux de la contrainte. En observant la première partie de la Table 15a, nous pouvons constater que l'exécution des trois algorithmes en utilisant la mémoire d'évaluation fait gagner un temps d'exécution remarquable qui est estimé à 56.5% pour QCBO. Ce grand pourcentage est explicable pour le treillis à cinq dimensions qui présente un espace de recherche relativement petit, ce qui signifie que les algorithmes rencontrent la majorité des solutions pendant les premières itérations. Une fois remplie, la mémoire d'évaluation sera beaucoup plus sollicité en lecture qu'en écriture pour le reste du processus. En observant les deux autres parties de la Table 15b et 15c, nous pouvons constater que le gain en temps ne baisse que de 10% à la fois pour QCBO. Cependant, l'espace de recherche double, ce qui signifie que, sur la globalité, l'utilisation de la mémoire d'évaluation est bénéfique en terme de réduction du temps d'exécution, et ce même pour des grandes instances du PSV.

Le troisième paramètre étudié est l'effet du *Reset Operator*. L'objectif principal du *Reset Operator* est d'éviter la convergence prématurée, où l'algorithme reste bloqué dans un optimum local.

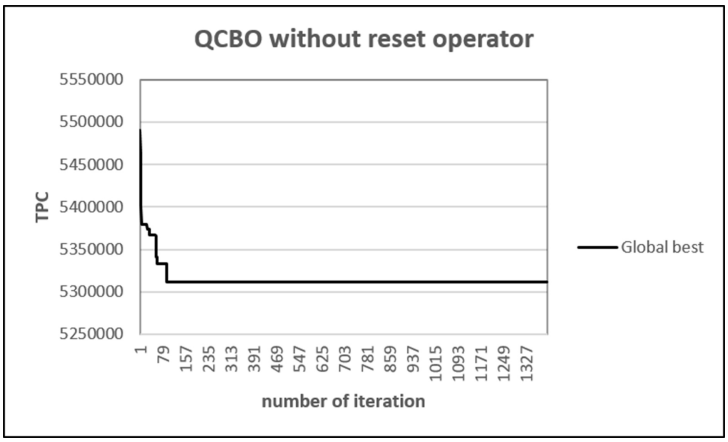


FIGURE 33 – Graphe du développement du TPC dans QCBO sans Reset Operator

Les Figures 33 et 34 représentent deux graphes qui nous permettent de garder trace de l'évolution de la meilleure solution lors de l'exécution de QCBO sur un treillis à sept dimensions avec un seuil de maintenance de 0.6 pour 1400 itérations. La différence est que, pour la Figure 33, QCBO est exécuté sans le *Reset Operator*, contrairement à la Figure 34, dans laquelle la meilleure solution locale de chaque génération est représentée dans le graphe avec la meilleure solution globale. La *génération* dans ce test représente chaque nouvelle population générée par le *Reset Operator*.

En examinant les deux figures, on peut clairement observer qu'en appliquant le *Reset Operator*, QCBO évite la convergence prématurée. Ceci peut être observé dans la courbe de la meilleure solution globale de la Figure 34 (QCBO avec *Reset Operator*) qui a continué à améliorer la meilleure solution globale jusqu'à des étapes tardives de la durée de vie de l'algorithme (environ 1300 itérations). En ce qui concerne la Figure 33 (QCBO sans *Reset Operator*), l'amélioration de la meilleure solution globale s'est arrêtée très tôt (la dernière amélioration a eu lieu autour de l'itération 100). L'algorithme est alors piégé dans un optimum local.

De plus, en observant les courbes des générations de la Figure 33, on peut également déduire qu'en appliquant le *Reset Operator*, l'algorithme gagne en diversité. Ceci peut être confirmé en observant les valeurs des meilleures solutions locales de chaque génération. Elles commencent toujours par des solutions relativement mauvaises, ce qui signifie qu'elles couvrent davantage de zones dans l'espace de recherche, puis elles améliorent ces solutions jusqu'à trouver une nouvelle meilleure solution globale, comme on le voit clairement dans les générations G1, G2, G3, et G6.

Étude comparative

Dans cette section, nous voulons prouver l'efficacité de l'algorithme QCBO comparé à d'autres algorithmes.

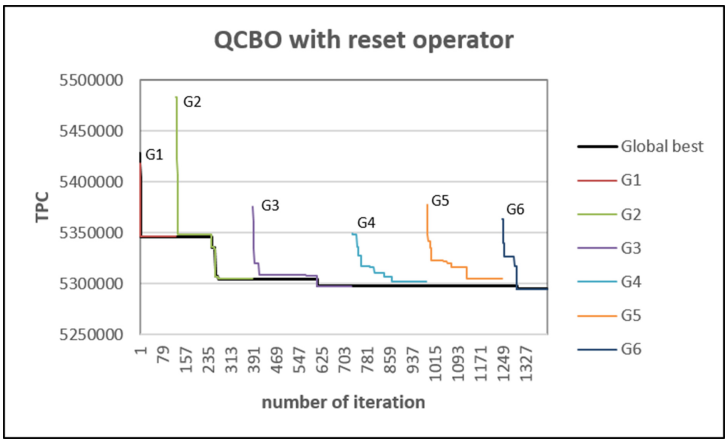


FIGURE 34 – Graphe de l'évolution du TPC dans QCBO avec *Reset Operator*

Nous allons commencer en comparant QCBO avec le CBO original qui utilise les fonctions de transfert, notamment la fonction *Sigmoid* et la fonction *VShaped*. Comme expliqué précédemment, dans QCBO, nous avons traité un problème ayant un espace de recherche discret binaire en utilisant l'approche *quantum binary* au lieu des fonctions de transformation régulières. La Table 16 contient les résultats (TPC) de plusieurs exécutions de QCBO avec CBO discret utilisant la fonction *Sigmoid* et la fonction *VShaped*, sur des treillis à cinq et à six dimensions respectivement, avec différents seuils de maintenance ($t = 0.9$ à 0.5).

TABLE 16 – *Sig-CBO vs. v-CBO vs. QCBO pour les treillis de 5-dimensions et 6-dimensions*

	5 dimensions			**6 dimensions**		
t	**sig-CBO**	**v-CBO**	**QCBO**	**sig-CBO**	**v-CBO**	**QCBO**
0.5	5143496	5143338	**5099592**	5290768	5295052	**5230059**
0.6	5071567	5075609	**5052126**	5269407	5234922	**5185738**
0.7	5019558	5018245	**5000283**	5236233	5162533	**5143775**
0.8	4969372	4971594	**4967200**	5206006	5130173	**5114870**
0.9	4988978	**4939772**	**4939772**	5196699	5094838	**5090698**

En observant la Table 16, nous pouvons noter que QCBO surpasse constamment les fonctions *sigmoid* et *v* dans les treillis à cinq et six dimensions et pour chaque seuil de maintenance. La raison est que QCBO assure un assez bon équilibre entre l'exploration et l'exploitation par rapport aux fonctions de transformation régulières. Contrairement

aux fonctions de transformation régulières, la représentation des quantums utilise à la fois les vitesses et les positions (qui sont des probabilités précédentes) pour mettre à jour un CB. Les fonctions de transformation, en revanche, dépendent entièrement des vitesses, puisque les positions dans leur cas sont soit 1, soit 0. Cela permet à la représentation des quantums d'explorer la zone de recherche plus efficacement. De plus, le fait que le groupe stationnaire tourne vers le meilleur global en utilisant le Q-gate donne également à QCBO un avantage concernant l'exploitation selon les résultats de la table, meilleur que de simplement entrer en collision avec le groupe mobile par paires.

La deuxième comparaison que nous allons effectuer est entre le QCBO et le EA de Xu Yu *et al.* [YYCG03], qui est une des meilleures solutions proposées pour le problème MVS sous contrainte de temps de maintenance. Après plusieurs tests, les paramètres utilisés dans les deux algorithmes sont indiqués dans la Table 17. Il faut noter que *FE* représente *Fitness Evaluation*, qui correspond aux nombres d'utilisations de la fonction objectif ; *SR* la probabilité de classement stochastique ; et *CO* la probabilité de croisement.

TABLE 17 – *Les paramètres utilisés dans les algorithmes QCBO et EA*

	Pop.size	*FE*	*Rotation*	*Mutation*	*SR*	*CO*
QCBO	100	200 000	0.04π	-	-	-
EA	100	200 000	-	0.001	0.4	0.5

Afin d'assurer une comparaison équitable entre les deux algorithmes implémentés, nous avons choisi de les comparer selon FE plutôt que sur le nombre d'itérations. En effet, en utilisant le nombre d'itérations, tout algorithme peut être avantagé s'il étudie plus de solutions en une itération – il consomme plus de temps d'exécution en conséquence – d'où l'utilisation de FE comme critère d'arrêt dans tous les algorithmes implémentés.

Les Tables 18, 19, et 20 contiennent les résultats de trente exécutions indépendantes de QCBO et EA sur des treillis à cinq, six, et sept dimensions, respectivement. Nous avons gardé trace de la moyenne, du minimum, et du maximum de TPC trouvés pendant ces exécutions. En observant la Table 18, on peut remarquer que QCBO surpasse EA pour tous les seuils. Ce qui semble être la meilleure solution trouvée par EA (TPC min) est presque toujours le TPC moyen de QCBO, i.e., QCBO a trouvé cette solution pour chaque exécution puisque moyenne = min = max pour tous les seuils sauf pour $t = 0.6$, comme le montre la Table 18.

Quant au treillis à six dimensions, la Table 19 montre que QCBO trouve de meilleures solutions puisque ses TPC moyen, min, et max sont meilleur que EA, sauf pour le seuil $t = 0.3$ où EA trouve un TPC min similaire.

TABLE 18 – *QCBO vs. EA pour un treillis de 5-dimensions*

t	**0.2**			**0.3**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5479742**	**5479742**	**5479742**	**5299322**	**5299322**	**5299322**
EA	5499864	**5479742**	5580018	5312721	**5299322**	5354626
t	**0.4**			**0.5**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5190686**	**5190686**	**5190686**	**5099592**	**5099592**	**5099592**
EA	5203998	**5190686**	5236239	5106157	**5099592**	5127608
t	**0.6**			**0.7**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5051780**	**5051328**	**5052126**	**5000283**	**5000283**	**5000283**
EA	5077675	5064453	5092808	5064874	**5000283**	5014292
t	**0.8**			**0.9**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**4967200**	**4967200**	**4967200**	**4939772**	**4939772**	**4939772**
EA	4969717	**4967200**	4975976	4940959	**4939772**	4951742

TABLE 19 – *QCBO vs. EA pour un treillis de 6-dimensions*

t	**0.2**			**0.3**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5554662**	**5552444**	**5560668**	**5387938**	**5385457**	**5402687**
EA	5584828	5555631	5662251	5415318	**5385457**	5463672
t	**0.4**			**0.5**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5299202**	**5296467**	**5306912**	**5232913**	**5229696**	**5235725**
EA	5309944	5301737	5327368	5241880	5233912	5251542
t	**0.6**			**0.7**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5181550**	**5179914**	**5184983**	**5144057**	**5142962**	**5145885**
EA	5186115	5181298	5194346	5146883	5143262	5152941
t	**0.8**			**0.9**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5114291**	**5113650**	**5115352**	**5090813**	**5090329**	**5091678**
EA	5115814	5113974	5118915	5091245	5090332	5092409

TABLE 20 – *QCBO vs. EA pour un treillis de 7-dimensions*

t	**0.2**			**0.3**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5593888**	**5586915**	**5608846**	**5456020**	**5451857**	**5464648**
EA	5619766	5591987	5692837	5474364	5460181	5505542
t	**0.4**			**0.5**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5367365**	**5364562**	**5371814**	**5304013**	**5300337**	**5308860**
EA	5380535	5369855	5397451	5310349	5303681	5319140
t	**0.6**			**0.7**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5259925**	**5257843**	**5262592**	**5226661**	**5225302**	**5228580**
EA	5265796	5261428	5274166	5229829	5226855	5233312
t	**0.8**			**0.9**		
	moy TPC	min TPC	max TPC	moy TPC	min TPC	max TPC
QCBO	**5200848**	**5200104**	**5202698**	**5180875**	**5179729**	**5183695**
EA	5202572	5200888	5205480	5181035	5180033	5185424

De même pour le treillis à sept dimensions, la Table 20 montre que QCBO a de meilleurs TPC moyen, min, et max comparés à EA. Ceci prouve l'extensibilité de QCBO puisque la qualité de la solution n'est pas compromise par la taille du problème. Les résultats montrent l'efficacité de QCBO par rapport à EA. Sa meilleure solution est meilleure que celle de EA, et même sa pire solution est presque toujours meilleure que la pire de EA.

Afin de déterminer statistiquement si la différence entre les résultats est significative, le test non-paramétrique de Wilcoxon a été effectué entre QCBO et EA. Toujours en essayant de prouver l'extensibilité de QCBO, le test a été effectué sur trente exécutions séparés de treillis à six, sept, et huit dimensions, avec un seuil de maintenance $t = 0.1$ à 0.9. Le niveau de signification *alpha* a été fixé à 0.01 (1%). R^+ est la somme des rangs positifs où QCBO surpasse EA, alors que R^- est la somme des rangs négatifs où EA surpasse QCBO. En observant la Table 21, on peut déduire que la différence entre les résultats donnés par QCBO et EA est très significative, et ce pour tous les cas. Alors que le niveau de signification *alpha* était fixé à 0.01 (1%), la valeur p observée n'a même pas atteint 0.001 pour tous les tests (toutes les dimensions et tous les seuils). Le R^- calculé est égal à zéro pour la plupart des cas, ce qui signifie que QCBO n'a pas été surpassé une seule fois sur l'ensemble des trente exécutions. Cependant, il est juste de mentionner que R^- augmente significativement lorsque le seuil de maintenance t est fixé

à 0.9, et ce pour toutes les dimensions. Cela signifie que QCBO a été surpassé quelques fois pendant les trente exécutions. Néanmoins, les valeurs p calculées pour $t = 0.9$ sont toujours inférieures à 0.001 pour ces occurrences, ce qui signifie que l'augmentation de R^- n'a pas compromis la signification.

Compte-tenu des résultats expérimentaux obtenus sur QCBO, il apparaît clairement que ce dernier soit plus avantageux que tous les algorithmes que nous avons implémentés pour la résolution du PSV à contrainte de temps de maintenance. Son équilibre exploration/exploitation assure une bonne qualité des solutions. Ainsi, la *mémoire d'évaluation* nous a permis de bénéficier de l'utilisation de plusieurs moyens coûteux en temps d'exécution, sans être sévèrement pénalisés.

6.4 Conclusion

Dans ce chapitre, nous avons essayé de traiter le PSV à contrainte de temps de maintenance. Comme nous l'avons expliqué, cette contrainte est plus compliquée que celle de l'espace de stockage. Ceci est dû à la propriété de *non monotonie* de la contrainte de temps de maintenance. Cette propriété augmente considérablement la complexité des algorithmes de sélection des vues [KMP02]. Dans un premier temps, nous avons essayé d'implémenter QEAM et QDE (nos deux algorithmes proposés pour la résolution du PSV à contrainte d'espace de stockage), mais ces derniers ne donnaient plus des résultats de qualité. En effet, face au EA de Xu Yu *et al.* [YYCG03], QEAM était très loin en terme de qualité de solution. QDE était lui mieux que QEAM, mais toujours pas aussi performant que EA. Nous avons donc cherché une autre approche pour résoudre le PSV à contrainte de temps de maintenance avec efficacité.

Notre choix s'est ainsi porté sur l'algorithme CBO (Colliding Bodies Optimization), une nouvelle métaheuristique qui utilise le principe des lois physiques de la collision entre les objets. Ce choix est justifié par le fait qu'elle ne possède pas de paramètre ajustable au début de l'algorithme. Le seul paramètre ε est auto-ajustable de façon à favoriser l'exploration au début du processus, et l'exploitation à sa fin.

Comme CBO est destiné aux problèmes dont l'espace de recherche est continu, il fallait utiliser une méthode de *discrétisation* pour l'adapter au PSV. Nous avons choisi la méthode « Quantum binary » dans un nouvel algorithme nommé QCBO [MB20a], dans lequel nous avons réutilisé plusieurs aspects de QDE, notamment : la représentation des quantums et des solutions, les valeurs initiales, l'utilisation du Q-gate, et l'utilisation du *Reset Operator* et de la fonction de réparation.

La reconduite des aspects utilisés dans QDE n'était pas difficile dans le cadre du QCBO, sauf pour la fonction de réparation. En effet, l'utilisation de celle-ci sous la contrainte de temps de maintenance est très coûteuse, comme mentionné dans la conclusion du chapitre précédent. Pour limiter la consommation supplémentaire du temps

TABLE 21 – *Test de Wilcoxon entre QCBO et EA avec un niveau de signification* $\alpha = 0.05$

	5-dimension lattice					
	t=0.1	t=0.2	t=0.3	t=0.4	t=0.5	t=0.6
R^+	465	465	464	465	464	453
R^-	0	0	1	0	1	12
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
conclusion	significant	significant	significant	significant	significant	significant
	t=0.7	t=0.8	t=0.9			
R^+	465	452	398			
R^-	0	13	67			
p-value	< 0.001	< 0.001	< 0.001			
conclusion	significant	significant	significant			
	6-dimension lattice					
	t=0.1	t=0.2	t=0.3	t=0.4	t=0.5	t=0.6
R^+	464	465	465	465	462	464
R^-	1	0	0	0	3	1
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
conclusion	significant	significant	significant	significant	significant	significant
	t=0.7	t=0.8	t=0.9			
R^+	459	452	396			
R^-	6	13	69			
p-value	< 0.001	< 0.001	< 0.001			
conclusion	significant	significant	significant			
	7-dimension lattice					
	t=0.1	t=0.2	t=0.3	t=0.4	t=0.5	t=0.6
R^+	437	465	465	465	462	465
R^-	28	0	0	0	3	0
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
conclusion	significant	significant	significant	significant	significant	significant
	t=0.7	t=0.8	t=0.9			
R^+	465	462	378			
R^-	0	3	87			
p-value	< 0.001	< 0.001	< 0.001			
conclusion	significant	significant	significant			

d'exécution causée par la fonction de réparation, nous avons introduit la *mémoire d'évaluation* lors du calcul de la contrainte, et de la fonction objectif. Les résultats expérimentaux prouvent l'efficacité de cette méthode qui fait gagner de 30% à 40% de temps d'exécution, et ce même pour une grande instance du PSV (i.e., treillis à sept dimensions), ce qui nous a incités à être plus ambitieux quant à la manière d'appliquer la réparation. En effet, nous avons utilisé une nouvelle fonction, *Greedy Repair*, qui se base sur le principe inverse de l'algorithme HRUA [HRU96] pour réparer les solutions infaisables.

Deux types de tests ont été effectués pour évaluer la performance du QCBO. Le premier type est celui qui étudie et analyse le comportement de l'algorithme. Il s'agit d'évaluer et de justifier les différents choix des méthodes et des aspects utilisés dans QCBO. Le deuxième type est l'étude comparative entre QCBO et d'autres algorithmes. Tout d'abord, nous avons comparé QCBO avec deux autres versions du CBO, i.e., sig-CBO qui utilise la fonction de transformation *Sigmoid*, et v-CBO qui utilise la fonction de transformation *VShaped*. Encore une fois, la méthode de « Quantum Binary » a confirmé sa suprématie par rapport aux fonctions de transformation car les résultats de QCBO surpassent ceux de sig-CBO et de v-CBO avec un écart important.

De plus, une comparaison directe des indicateurs statistiques (min, max, moy) entre QCBO et EA de Xu Yu *et al.* [YYCG03] est effectuée sur des treillis à cinq, six, et sept dimensions. Ces tests démontrent que QCBO présente des meilleurs résultats comparé à EA. Pour confirmer si cette différence est significative, un test supplémentaire de Wilcoxon a été appliqué. Le résultat de ce dernier confirme la suprématie du QCBO par rapport à EA. Cela signifie que les choix effectués lors de la proposition de notre approche assurent un bon équilibre exploration/exploitation ; d'où la bonne qualité des résultats.

Chapitre 7

Résoudre le PSV en Utilisant les Métaheuristiques Multi-Objectifs

7.1 Introduction

Dans les chapitres précédents, nous avons essayé de résoudre le PSV avec un seul objectif, qui est l'optimisation du temps de réponse aux requêtes. Jusqu'à présent, nous avons considéré l'espace de stockage et le temps de maintenance comme des obstacles qui empêchent la totalité des vues d'être matérialisées. Pour ce faire, nous avons essayé de simuler des valeurs de ces "obstacles" pour tester l'efficacité des méthodes proposées. Mais, dans le contexte "aide à la décision", le décideur ne possède a priori pas forcément d'informations aussi précises sur la valeur de la contrainte. Il pourrait même être prêt à faire des compromis s'il estime que c'est bénéfique. Ceci peut de plus varier d'un individu à un autre. En résumé, il est préférable de donner le choix au décideur et de le laisser estimer le meilleur chemin, plutôt que de lui demander de fixer une contrainte en amont. En prenant cela en considération, une nouvelle vision s'installe. Ce que nous avons jusqu'à présent considéré comme contrainte ou obstacle peut être vu comme un objectif. Cela change la nature du problème en un problème d'optimisation multi-objectifs. Dans ce chapitre, nous allons présenter les problèmes d'optimisation multi-objectifs, ainsi que les principales différences entre eux et les problèmes d'optimisation mono-objectif.

7.2 Les Problèmes D'optimisation Multi-objectifs

L'optimisation multi-objectifs trouve ses racines au 19^{eme} siècle, dans les travaux d'Edgeworth et de Pareto en économie [Edg81, Moo97]. Elle a été utilisée en économie et en sciences de gestion depuis plusieurs décennies, puis progressivement en sciences de l'ingénierie [Tal09]. Pour qu'un problème soit considéré comme problème d'optimisa-

tion multi-objectifs, il faut que ses “objectifs" soient conflictuels, i.e., l'optimisation d'un objectif entraîne la détérioration d'au moins un autre. Contrairement aux problèmes d'optimisation mono-objectif, la solution aux problèmes d'optimisation multi-objectifs (POM) n'est pas une seule solution, mais plutôt un ensemble qui s'appelle « Ensemble Pareto Optimal ». Une solution est dite *Pareto Optimale* si nous ne pouvons pas améliorer un de ses objectifs sans en détériorer au moins un autre [Tal09]. Cet ensemble représente le compromis entre les objectifs conflictuels. L'objectif principal de la résolution d'un problème multi-objectifs est d'obtenir l'ensemble Pareto Optimal. Ainsi, quand une méta-heuristique est appliquée, son but devient d'obtenir une approximation de l'ensemble *Pareto Optimal* ayant deux propriétés : la *convergence* vers le front *Pareto Optimal* et la *diversité uniforme*. La première propriété assure la génération d'un ensemble Pareto presque optimal de solutions, tandis que la deuxième propriété indique la bonne distribution des solutions obtenues autour du front Pareto optimal, pour qu'aucune information précieuse ne soit perdue.

En plus des concepts communs des méta-heuristiques mono-objectif, une méta-heuristique multi-objectifs contient trois composantes de recherche principales [Tal09] :

— **Affectation des *Fitness***
Comme les POM possèdent plusieurs fonctions objectifs, il devient très difficile de trier des solutions qui appartiennent au même ensemble pareto. Le choix d'une bonne méthode, prenant en considération tous les objectifs, est responsable de la convergence vers le front pareto optimal.

— **Préservation de la Diversité**
L'enjeu de cet aspect est de trouver des solutions diverses, i.e., un ensemble dispersé tout au long de l'ensemble pareto.

— **L'élitisme**
La préservation du meilleur ensemble pareto durant le processus permet une évolution rapide et robuste vers un bon ensemble pareto global.

Avant d'appliquer des approches multi-objectifs, nous allons d'abord présenter quelques notions indispensables dans le jargon POM.

7.2.1 Espace décision vs. Espace Objectif

Pour les problèmes d'optimisation multi-objectifs, il existe deux espaces de recherche : l'espace décision dans lequel nous trouvons les vecteurs de solution $x(x_1, x_2, ..., x_N)$ (N étant la taille du problème), et l'espace objectif dans lequel nous trouvons les vecteurs objectifs. Ce sont des vecteurs qui contiennent, pour chaque solution, l'ensemble de ses valeurs objectifs $x_{obj}(obj_1, obj_2, ..., obj_M)$ (M étant le nombre d'objectifs dans le POM). À chaque vecteur de solution X est associé un vecteur objectif Y, et le pas-

sage entre ces deux vecteurs est possible en utilisant la fonction de transfert F i.e., $Y = F(X)$. L'enjeu dans la résolution des POM n'est plus l'optimisation dans l'espace de décision, mais également l'optimisation dans l'espace objectif.

7.2.2 La Pareto Dominance

Nous allons ci-après reconduire deux définitions de[Tal09] (dans le cas des problèmes de minimisation) :

— **La pareto dominance**
Nous disons qu'un vecteur objectif $u = (u_1, u_2, ..., u_n)$ domine un autre vecteur objectif $v = (v_1, v_2, ..., v_n)$ ($u \prec v$) si, et seulement si, aucune composante de v n'est plus petite que la composante correspondante de u, et si au moins une composante de u est strictement plus petite, soit :

$$\forall i \in \{1, 2, ..., n\} : u_i \leq v_i \wedge \exists i \in \{1, 2, ..., n\} : u_i < v_i$$

— **La pareto optimalité**
Une solution $x^* \in S$ est dite *pareto optimale* si, pour toute solution $x \in S$, $F(x)$ ne domine pas $F(x^*)$, soit :

$$\forall x \in S : F(x) \not\prec F(x^*)$$

7.3 Le PSV dans le Contexte Multi-Objectifs

Le PSV à été traité comme un problème d'optimisation mono-objectif à contraintes dans la plupart des travaux de la littérature. En effet, le but de la sélection des vues est de minimiser le TPC sans dépasser un certain seuil de contraintes. Ces contraintes peuvent être considérées comme des objectifs dans le contexte multi-objectifs.

Contrairement à la contrainte d'espace, la contrainte du temps de maintenance a déjà été traitée comme objectif dans plusieurs travaux [HCLK99, ZYY01, PoA07] (notamment la majorité des travaux qui utilisent le MVPP). Ils ont utilisé la combinaison linéaire des deux objectifs (TPC et MC), et ils ont visé à minimiser cette somme dans un contexte mono-objectif. Cependant, la question de savoir si cette solution est la meilleure solution globale, ou si une meilleure solution globale existe, dépend de la signification des quantités exprimées par les objectifs et de leur échelle relative, et nécessite que le compromis entre les objectifs soit connu a priori afin de choisir les constantes appropriées devant chaque objectif. Si l'on ne parvient pas à trouver de bonnes constantes, un algorithme qui minimise le coût global peut en fait minimiser le plus coûteux des deux objectifs sans tenir compte de l'autre, ce qui va à l'encontre de l'objectif même de la prise en compte des deux objectifs en premier lieu.

Le PSV a été traité dans un contexte multi-objectifs la première fois dans le travail

de Lawrence [Law06], où il a considéré deux objectifs : minimiser le TPC et le MC. Deux algorithmes ont été appliqués : MOGA de Fonseca et Fleming [FF93], et NPGA de *Horn et al.* [HNG94]. Il cite que les résultats de ces deux algorithmes dépassent ceux des meilleures approches gloutonnes. Le deuxième travail ayant traité le PSV dans le contexte multi-objectifs est celui de Goswami *et al.* [GBD13]. Dans ce travail, les mêmes objectifs ont été considérés. Les auteurs ont appliqué l'algorithme de l'évolution Différentielle Multi-Objectifs (MED). Après expérimentation, les auteurs affirment que les résultats de cet algorithme sont similaires à ceux de l'algorithme NSGA2 [DPAM02].

Plus récemment, l'équipe de Vijay a utilisé plusieurs approches multi-objectifs pour résoudre le PSV [PK19b, PK19a, PVK20, PK20]. Dans ces travaux, les deux objectifs considérés sont le coût d'évaluation des vues qui sont matérialisées, et le coût d'évaluation des vues qui ne le sont pas. Ils ont divisé le coût d'évaluation des requêtes en deux, ce qui ne s'avère pas très significatif dans le contexte “aide à la décision".

Dans la prochaine section, nous allons présenter deux algorithmes, i.e., NSGA2 et MOGA, qui seront par la suite appliqués pour résoudre le PSV bi-objectifs.

Nous allons par la suite comparer les vecteurs objectifs de ces deux algorithmes avec les vecteurs objectifs des solutions données par QCBO dans le chapitre précédent.

7.3.1 MOGA

Comme expliqué précédemment, le corps de l'algorithme MOGA est le même que celui de l'algorithme génétique. La seule difficulté que nous rencontrons est lors du choix des individus qui vont persister (remplacement). Il faut trouver une manière de trier les solutions pour pouvoir choisir les meilleures qui vont survivre à la prochaine génération. Ceci nous oblige à associer des *fitness* aux vecteurs objectifs pour pouvoir trier ces solutions ; c'est la façon de calculer la *fitness* qui fait la différence entre MOGA et NSGA.

Le calcul de la *fitness* dans le MOGA consiste à trouver le nombre d'individus de la population P qui dominent chaque individu a. L'ajout de 1 à cette valeur donne le rang de a. Les rangs sont ensuite interpolés du meilleur au pire, de façon à ce que a obtienne une *fitness* $f'(a)$ proportionnelle à son rang et à la somme des rangs de tous les individus de P (les *fitness* les plus élevées étant les meilleures). La *fitness* de chaque individu est ensuite partagée tel que définit dans l'équation (7.1) :

$$f(a) = \frac{f'(a)}{\sum_{b \in P} sh(d(a,b))} \tag{7.1}$$

tel que $d(a,b)$ est la distance entre a et b dans l'espace objectif donné par l'équation suivante :

Algorithm 10: Pseudo-code de l'algorithme MOVSGA

N : population size ;
m_p : mutation probability $\in [0.001; 0.1]$;
P_E : probability of elite replacement $\in [0.3, 0.7]$;
Initialize a population P of X_i where $i \in [1, N]$;
repeat
  foreach $X_i, i \in [1, N]$ **do**
    Evaluate X_i using equation (7.1) ;
  end
  O $= \emptyset$;
  while $|O| < |P|$ **do**
    Select two individuals a and b using roulette wheel ;
    Apply uniform crossover on a and b to generate a' and b' ;
    Apply mutation on a' and b' with respect to m_p ;
    $O = O \cup \{a', b'\}$;
  end
  P = Best individuals of $P \cup O$;
  if $rand < P_E$ **then**
    Replace P's current pareto set with global pareto set ;
  end
until *stop criteria is hit*;

$$d(a,b) = \sqrt{\left(\frac{TPC(a) - TPC(b)}{TPC_{max} - TPC_{min}}\right)^2 + \left(\frac{MC(a) - MC(b)}{MC_{max} - MC_{min}}\right)^2} \tag{7.2}$$

TPC_{max} est la valeur TPC quand aucune vue n'est matérialisée, tandis que TPC_{min} est la valeur TPC quand toutes les vues sont matérialisées. MC_{max} est la valeur MC quand toutes les vues sont matérialisées, alors que MC_{min} est la valeur MC quand aucune vue n'est matérialisée.

$sh(d)$ est la fonction de partage donnée par l'équation suivante :

$$sh(d) = \begin{cases} 1 - (d/\sigma_{niche}) & : if(d \leq \sigma_{niche}) \\ 0 & : sinon \end{cases} \tag{7.3}$$

σ_{niche} définit le rayon de la niche, un paramètre qui est fixé sur la base d'une estimation de la distance phénotypique minimale souhaitée entre les individus. Une fois que les *fitness* ont été calculées, n'importe quel mécanisme de sélection peut être utilisé.

Nous avons utilisé l'algorithme MOGA pour resoudre le PSV bi-objectifs dans un algorithme nommé MOVSGA (Multi Objectif View Selection Genetic Algorithm). En résumé, notre algorithme ressemble à l'algorithme MOGA de [Law06] avec une différence, qui est l'utilisation de l'élitisme. Nous remplaçons d'une manière aléatoire un nombre d'individus de l'élite la population actuelle par des individus de l'élite globale. Le pseudo-code de MOVSGA est donné par l'Algorithme 10.

7.3.2 NSGA2

Le deuxième algorithme que nous avons appliqué pour résoudre de PSV bi-objectifs est le NSGA2. Tout comme MOGA, NSGA2 contient les mêmes étapes qu'un algorithme génétique. La seule différence réside dans la façon d'associer des *fitness* aux vecteurs objectifs afin de pouvoir trier les individus lors de la sélection et de la préservation. NSGA2 utilise une procédure nommée *fast non-dominated sort* pour trier l'ensemble des individus. L'idée principale de cette fonction est de diviser la population en *couches* pareto. Tout d'abord, pour chaque solution, nous calculons deux entités : 1) le nombre de dominations n_p, c'est-à-dire le nombre de solutions qui dominent la solution p, et 2) S_p, l'ensemble des solutions que la solution p domine. Toutes les solutions du premier front non dominé auront leur compte de dominations égal à zéro. Ensuite, pour chaque solution p avec $n_p = 0$, nous visitons chaque membre (q) de son ensemble S_p et réduisons son compte de dominations de un. En faisant cela, si, pour un membre q, le nombre de dominations devient nul, nous le plaçons dans une liste séparée Q. Ces membres appartiennent au deuxième front non dominé. On continue

la procédure ci-dessus avec chaque membre de Q et on identifie le troisième front. Ce processus se poursuit jusqu'à ce que tous les fronts soient couverts. Le pseudo-code de la fonction *fast non-dominated sort* est donné dans l'Algorithme 11.

La fonction *fast non-dominated sort* regroupe la population sous forme de plusieurs couches pareto, où chaque couche contient des solutions qui dominent toutes les autres couches inférieures. Mais cela n'est pas suffisant, car les individus d'une même couche ne sont pas triés. Pour pouvoir trier les individus à l'intérieur de chaque couche, nous avons besoin d'une fonction similaire à celle utilisée dans l'algorithme MOGA. Les auteurs ne voulaient utiliser la fonction de partage pour les raisons suivantes :

1. La performance de la méthode de la fonction de partage pour maintenir la diversité des solutions dépend largement de la valeur σ_{niche} choisie.
2. Comme chaque solution doit être comparée à toutes les autres solutions de la population, la complexité globale de l'approche de la fonction de partage est de $O(N^2)$.

Pour essayer d'éliminer ces deux problèmes, les auteurs ont proposé une nouvelle fonction qu'ils ont appelé *crowding distance assignment*, ou *la distance d'encombrement.*

Le calcul de la distance d'encombrement nécessite de trier la population en fonction de la valeur de chaque fonction objectif par ordre croissant de grandeur. Ensuite, pour chaque fonction objectif, les solutions limites (solutions ayant la plus petite et la plus grande valeur de fonction) se font attribuer une valeur de distance infinie. Toutes les autres solutions intermédiaires se font attribuer une valeur de distance égale à la différence absolue normalisée entre les valeurs de fonction de deux solutions adjacentes. Ce calcul est poursuivi avec les autres fonctions objectifs. La valeur globale de la distance d'encombrement est calculée comme la somme des valeurs de distances individuelles correspondant à chaque objectif. Chaque fonction objectif est normalisée avant de calculer la distance d'encombrement. L'Algorithme 12 décrit la procédure de calcul de la distance d'encombrement de toutes les solutions d'un ensemble non dominé T.

T$[i].m$ représente la valeur de la m^{ieme} fonction objectif de l'i^{eme} individu de l'ensemble T. Les paramètres f_m^{max} et f_m^{min} représentent les valeurs max et min de la m^{ieme} fonction objectif. Après avoir attribué une mesure de distance à tous les membres de la population de l'ensemble T, nous pouvons comparer deux solutions pour leur degré de proximité avec d'autres solutions. Une solution avec une valeur plus petite de cette mesure de distance est, dans un certain sens, plus encombrée par d'autres solutions.

7.3.3 Résultats Expérimentaux

Pour tester l'efficacité de l'approche multi-objectifs par rapport à l'approche standard mono-objectif avec contraintes, nous avons implémenté les deux algorithmes NSGA2 et MOVSGA et nous les avons comparé à dix points distincts de QCBO en variant la

Algorithm 11: Pseudo-code de la fonction fast non-dominated sort [DPAM02]

```
P : Population to be sorted ;
foreach p ∈ P do
    S_p = ∅ ;
    n_p = 0 ;
    foreach q ∈ P do
        if (p ≺ q) then
            S_p = S_p ∪ {q} ;
        else
            if (q ≺ p) then
            end
            n_p = n_p + 1 ;
        end
        if (n_p = 0) then
            p_rank = 1 ;
            F_1 = F_1 ∪ {p} ;
        end
    end
end
i = 1 ;
while F_i ≠ ∅ do
    Q = ∅ ;
    foreach p ∈ F_i do
        foreach q ∈ S_p do
            n_q = n_q − 1 ;
            if (n − q = 0) then
                q_rank = i + 1 ;
                Q = Q ∪ {q} ;
            end
        end
    end
    i = i + 1 ;
    F_i = Q ;
end
```

Algorithm 12: Pseudo-code de la procédure *crowding distance assignment* [DPAM02]

```
T : non-dominated set (pareto layer) ;
l = |T| ;
foreach i do
  | set T[i]_distance = 0 ;
end
foreach objective m do
  | T = sort (T, m) ;
  | T[1]_distance = T[l]_distance = ∞ ;
  | for i = 2 to l − 1 do
  |   | T[i]_distance = T[i]_distance + (T[i + 1].m − T[i − 1].m)/(f_m^max − f_m^min) ;
  | end
end
```

contrainte t de 0.1 à 0.9. Nous savons davantage que cette comparaison est biaisée vers l'approche mono-objectif pour plusieurs raisons. Une raison évidente est le temps de calcul nécessaire pour effectuer dix exécutions séparées contre le temps de calcul d'une seule exécution de l'approche multi-objectifs. Une autre raison est l'espace de recherche qui se restreint lors de l'attribution d'une valeur fixe de contrainte dans chaque exécution de l'approche mono-objectif. Néanmoins, nous souhaitons avoir un repère lors de l'évaluation des résultats expérimentaux des deux algorithmes appliqués, d'où la comparaison avec QCBO.

Comme tous les autres tests, ceux de l'approche multi-objectifs ont été effectués en utilisant JDK 1.8 sous Windows 10 sur un PC dont le processeur est un Intel i-5 2,4 GHz, doté de 8 Go de RAM.

La Figure 35 montre la distribution des solutions des algorithmes NSGA2, MOVSGA, et QCBO sur l'espace objectif d'un treillis à cinq dimensions. En observant cette figure, nous pouvons constater que les solutions de NSGA2 sont bien éparpillées sur le front pareto, et la majorité des solutions de QCBO s'aligne avec le même front, contrairement aux solutions de MOVSGA dont la plupart est visuellement dominée par le front de NSGA2 – sauf pour les petites valeurs de l'objectif *coût de maintenance* (MC), où MOVSGA présente quelque solutions dominantes.

Pour mieux estimer la performance des algorithmes, nous avons utilisé quelques indicateurs de performance. La Table 22 compare la cardinalité du front pareto des deux algorithmes NSGA2 et MOVSGA, le nombre de solutions dominantes, le nombre de solutions dominées, ainsi que le nombre de solutions non-dominées.

En regardant la Table 22, nous pouvons confirmer la qualité des solutions de NSGA2 par rapport à MOVSGA. En effet, pour NSGA2, le nombre de solutions dominées ne

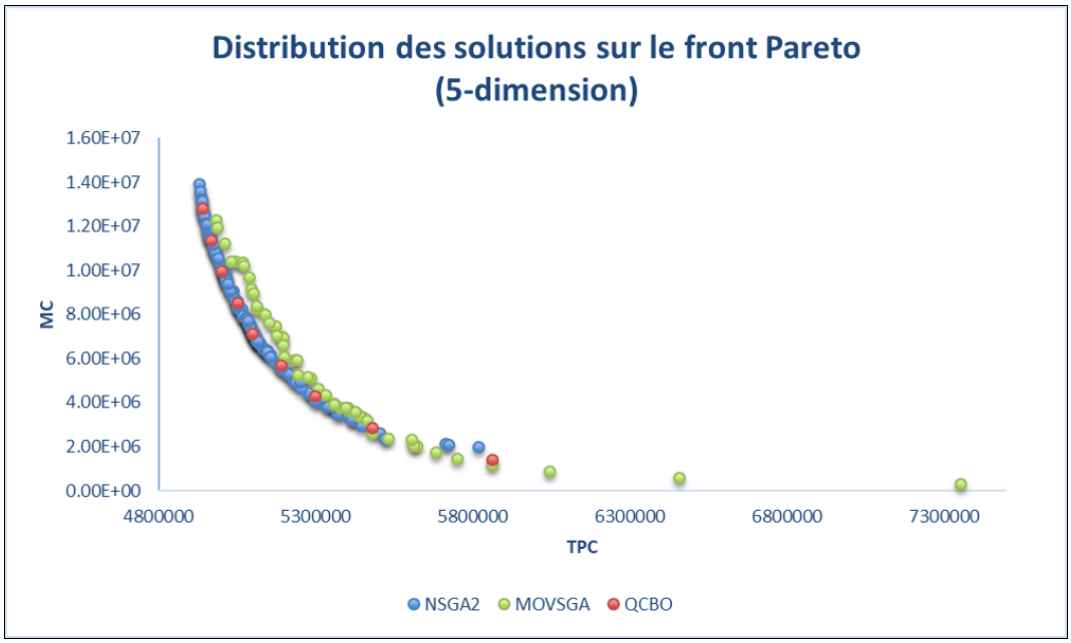


FIGURE 35 – Distribution des solutions des algorithmes NSGA2, MOVSGA, et QCBO sur l'espace objectif (treillis à 5 dimensions).

TABLE 22 – *Indicateurs sur NSGA2 et MOVSGA (5-dimensions)*

	Card.	*Nb. non-dominated*	*nb. dominated*	*nb. dominant*
NSGA2	132	125	7 (5%)	92 (69%)
MOVSGA	51	13	38 (74%)	8 (15%)

dépasse pas les 5%, alors qu'il atteint 74% pour le MOVSGA. D'autre part, nous constatons que 69% des solutions de NSGA2 sont dominantes, contre un faible pourcentage de 15% pour MOVSGA.

La Figure 36 montre la distribution des solutions des algorithmes NSGA2, MOVSGA, et QCBO sur l'espace objectif d'un treillis à six dimensions. En observant cette figure, nous constatons que la distribution des solutions de NSGA2 est toujours dans le front avec les solutions de QCBO. Mais ceci n'arrive que pour les grandes valeurs de l'objectif *coût de maintenance* (MC). Pour le reste (petites valeurs de MC), NSGA2 n'est plus performant comme pour le treillis à cinq dimensions. D'autre part, MOVSGA semble être dans la même position par rapport à NSGA pour les grandes valeurs de MC. Cependant, pour les petites valeurs, nous trouvons MOVSGA présent contrairement à NSGA2, qui ne couvre plus le front pareto entier.

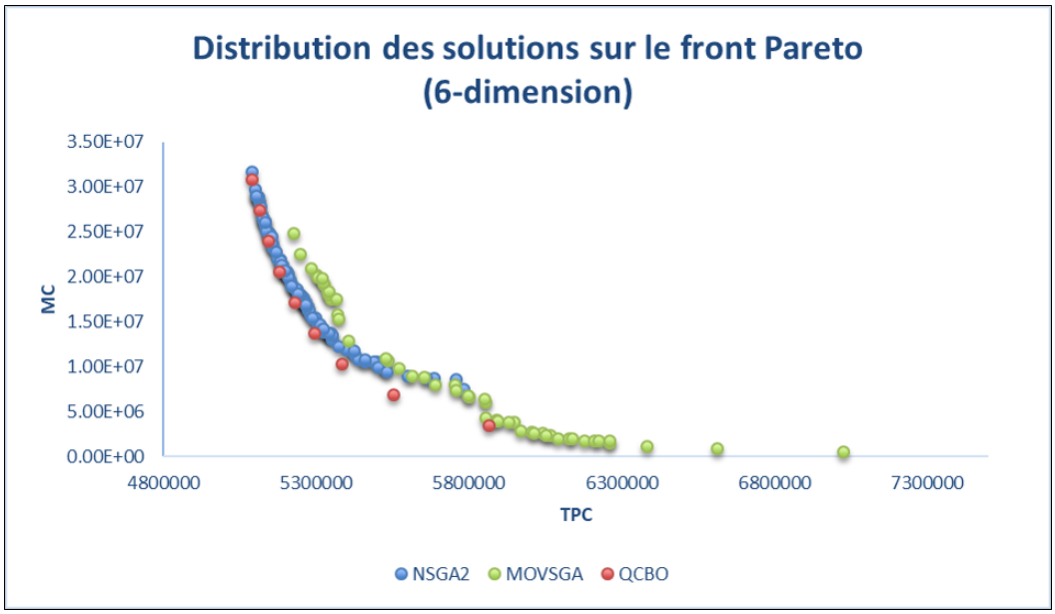


FIGURE 36 – Distribution des solutions des algorithmes NSGA2, MOVSGA, et QCBO sur l'espace objectif (treillis à 6-dimensions).

TABLE 23 – *Indicateurs sur NSGA2, MOVSGA (6-dimensions)*

	Card.	*Nb. non-dominated*	*nb. dominated*	*nb. dominant*
NSGA2	111	107	4 (3%)	81 (72%)
MOVSGA	54	30	24 (44%)	19 (35%)

En observant la Table 23, nous pouvons constater que les pourcentages des solutions dominantes/dominées n'a pas trop changé pour NSGA2, contrairement à MOVSGA pour lequel le pourcentage des solutions dominantes a augmenté (35%), et celui des solutions dominées a diminué (44%). La raison de cette amélioration des chiffres face à NSGA2 est due à l'absence de couverture de NSGA2 du front pareto au-delà de 5.8×10^6 TPC, où nous trouvons uniquement des solutions de MOVSGA.

La Figure 37 montre la distribution des solutions des algorithmes NSGA2, MOVSGA, et QCBO sur l'espace objectif d'un treillis à sept dimensions. Sur cette figure, nous remarquons le même comportement qu'avec le treillis à six dimensions. NSGA s'éloigne un peu du repère (QCBO) pour les petites valeurs de la contrainte *coût de processing* (TPC), et il est toujours absent pour les grandes valeurs de TPC, pour lesquels nous trouvons MOVSGA présent seul.

Si nous observons la Table 24, nous pouvons constater que le nombre de solutions

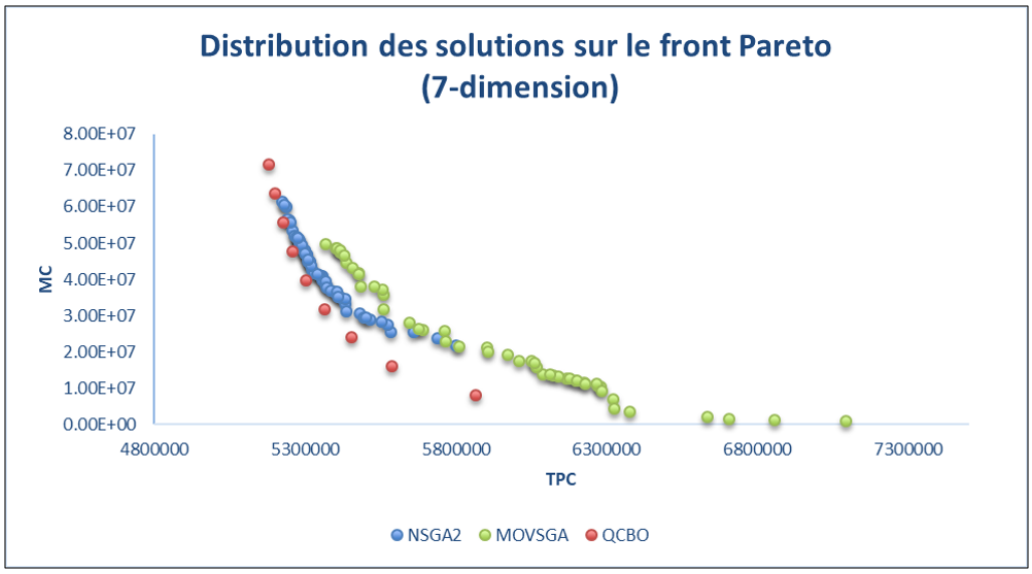


FIGURE 37 – Distribution des solutions des algorithmes NSGA2, MOVSGA, et QCBO sur l'espace objectif (treillis à 7-dimensions).

TABLE 24 – *Indicateurs sur NSGA2, MOVSGA (7-dimensions)*

	Card.	*Nb. non-dominated*	*nb. dominated*	*nb. dominant*
NSGA2	75	74	1 (1%)	40 (53%)
MOVSGA	62	47	15 (24%)	11 (17%)

trouvées dans le front pareto (cardinalité) de NSGA2 a diminué d'une façon drastique par rapport au treillis à cinq dimensions (presque la moitié). Quant au reste des indicateurs, le pourcentage des solutions dominé a remarquablement diminué (1%), avec celui des solutions dominantes (53%). La même chose est remarquée pour MOVSGA. Le taux des solutions dominées et dominantes a également diminué (24% et 17%, respectivement). Cela signifie que les deux algorithmes ne couvrent plus les mêmes parties du front pareto. La majorité de leurs solutions seront donc non-dominées.

Compte-tenu des résultats observés, il semblerait que NSGA2 présente de meilleures solutions que MOVSGA. Ce qui nous a permis de faire cette déduction est le fait que NSGA2 trouve toujours des solutions proches du repère QCBO, contrairement à MOVSGA. Ainsi, l'amélioration des indicateurs de MOVSGA ne reflète pas l'amélioration de la qualité des solutions de ce dernier, mais plutôt le manque de diversité du coté de NSGA2, qui converge vers de bonnes solutions, mais pas pour tout le front pareto.

7.4 Conclusion

Dans ce chapitre, nous avons tenté de résoudre le PSV dans le contexte multi-objectifs. Le but est de donner le choix aux décideurs de ne pas fixer la valeur des contraintes a priori, mais plutôt d'avoir un ensemble de solutions à partir duquel ils pourront choisir, en faisant les compromis qu'ils estiment convenables. Lors de l'étude de l'existant, nous avons remarqué que cette approche n'est pas très populaire ; peu de travaux ont abordé le PSV du point de vue multi-objectifs.

Nous avons réalisé une introduction à l'optimisation multi-objectifs, en expliquant les principales différences entre celle-ci et l'optimisation mono-objectif, tels que l'espace objectif et l'ensemble pareto dominant. Nous avons appliqué deux des algorithmes les plus connus dans le contexte multi-objectifs, à savoir MOGA et NSGA2. Ce ne sont autres que des adaptations de l'algorithme génétique à l'optimisation multi-objectifs. La principale différence entre ces deux algorithmes réside dans la fonction de classement des solutions, qui n'est pas évident quand il s'agit de plusieurs critères de tri (objectifs). MOGA utilise la fonction de partage (équation 7.1) guidée par le paramètre σ_{niche}, tandis que NSGA2 fait le tri en deux étapes. Dans un premier temps, il désigne les couches pareto. Ensuite, à l'intérieur de chaque couche, il utilise la *distance d'encombrement* pour trier les solutions d'une même couche. Le résultat sera un tri global des solutions. Nous avons implémenté NSGA2 et une version du MOGA qui utilise l'*élitisme* (MOSVGA), et nous avons comparé leurs résultats avec dix solutions du QCBO distribuées équitablement sur l'espace objectif.

Les résultats expérimentaux ont montré que NSGA2 donne de bonnes solutions face à MOVSGA. Dans la plupart des cas, ses solutions sont proches de celles de QCBO. Cependant, en augmentant la taille du problème (treillis à six et sept dimensions), la diversité des solutions de NSGA2 diminue considérablement. Cela ne reflète pas forcément la non-extensibilité de NSGA2 car les solutions ne s'éloignent pas du repère QCBO, ce qui veut dire que la détérioration n'a pas atteint la convergence. La baisse de diversité peut être due au choix de sélection (qui est la roulette). Un changement de cet opérateur est donc envisagé dans les prochains travaux pour connaître son impact sur la diversité des solutions dans l'espace objectif.

Chapitre 8

Conclusion Générale

L'énorme quantité de données, la complexité des requêtes dans les entrepôts de données, ainsi que l'importance d'une réponse rapide aux requêtes analytiques dites OLAP (*online analytical processing*) ont conduit au rôle essentiel de la sélection des vues matérialisées dans la conception des entrepôts de données. Puisque différentes contraintes – tels que la quantité d'espace de stockage disponible et le coût de maintenance des vues – rendent impossible le stockage de l'ensemble de toutes les vues possibles, il est nécessaire de sélectionner un ensemble optimal de vues à matérialiser pour réduire le temps de réponse des requêtes analytiques complexes et augmenter l'efficacité de l'entrepôt de données. Le problème de sélection des vues (PSV) a pour objectif de choisir le sous-ensemble le plus bénéfique si matérialisé. Selon la contrainte, il existe deux variantes du PSV : le PSV à contrainte d'espace de stockage, et le PSV à contrainte de temps de maintenance.

À travers cette thèse, nous avons proposé deux approches pour résoudre le PSV à contrainte d'espace de stockage : QEAM et QDE. QEAM est basée sur l'algorithme QEA (*quantum evolutionary algorithm*), tandis que QDE est basée sur l'algorithme de l'évolution différentielle (DE). L'application du QEAM était simple et directe, car QEA est destiné à des espaces de recherche binaire – ce qui est le cas du PSV – contrairement au DE. Comme évolution différentielle est destinée à des espaces de recherche continus, une méthode de discrétisation est obligatoire lors de son utilisation. Nous avons appliqué deux méthodes, à savoir les fonctions de transfert et l'approche *quantum binary*. Une comparaison entre les deux a montré que l'approche *quantum binary* était plus efficace, d'où la proposition de notre algorithme *quantum differential evolution* (QDE). Nous avons traité les solutions qui violent la contrainte par la méthode de réparation, et ce pour nos deux contributions. Les résultats expérimentaux ont montré que QEAM et QDE présentent des solutions de meilleure qualité, et ce en les comparant à l'algorithme génétique qui est l'approche la plus utilisée pour résoudre le PSV. Une comparaison entre QEAM et QDE a montré un léger avantage pour QEAM.

Une approche de résolution du PSV à contrainte de temps de maintenance a également été proposée dans cette thèse, suite à l'échec des algorithmes QEAM et QDE sur cette variante du PSV. En effet, à cause de la propriété non monotone de la contrainte du temps de maintenance, la complexité des algorithmes de sélection des vues augmente considérablement. Ainsi, le calcul de la contrainte est aussi complexe que le calcul de la fonction objectif, ce qui signifie un temps d'exécution additionnel. Nous avons proposé une approche basée sur l'algorithme *colliding bodies optimisation* (CBO) pour résoudre le PSV à contrainte de temps de maintenance. Tout comme l'évolution differentielle, CBO est destiné à des espaces de recherche continus. Cette fois, nous avons directement opté pour l'approche *quantum binary* dans un algorithme nommé *quantum colliding bodies optimisation* (QCBO). Comme les autres approches proposées, QCBO traite les solutions infaisables par la méthode de réparation. Cette méthode est améliorée dans le cas du QCBO. Il s'agit d'une réparation gloutonne basée sur le principe inverse du célèbre algorithme HRUA. Ainsi, pour limiter la consommation supplémentaire du temps d'exécution causée par la nouvelle fonction de réparation, nous avons introduit la mémoire d'évaluation lors du calcul de la contrainte et de la fonction objectif. Cet aspect nous fait gagner de 30% à 40% de temps d'exécution.

Enfin, une approche qui traite le PSV dans le contexte multi-objectifs a été proposée à la fin de cette thèse. Nous avons implémenté le célèbre *non-dominated sorting genetic algorithm* (NSGA2), et MOSVGA, une version du *multi-objective genetic algorithm* (MOGA), qui utilise l'élitisme pour résoudre le PSV bi-objectifs. Les résultats expérimentaux ont montré que NSGA2 donne de bonnes solutions face à MOVSGA. Cependant, en augmentant la taille du problème, NSGA2 perd en diversité de son ensemble pareto.

En termes de perspectives, il serait intéressant de poursuivre la recherche des approches pour traiter le PSV multi-objectifs. Un premier pas serait d'essayer de régler le problème de perte en diversité du NSGA2. Ainsi, d'autres algorithmes multi-objectifs mieux adaptés au PSV sont à trouver. Il serait également intéressant d'étudier comment la sélection des vues pourrait être prolongée dans le contexte du *big data*. En effet, la taille des données dans ce contexte est plus importante que dans le cas des entrepôts des données. Étant que le PSV est un problème relativement ancien, il serait avantageux d'utiliser les efforts déployés par la communauté pour résoudre le PSV dans le but de optimiser le temps de requêtes posées dans le contexte du *big data*.

Bibliographie

[AK15] Biri Arun and T.V. Vijay Kumar. Materialized view selection using improvement based bee colony optimization. *International Journal of Software Science and Computational Intelligence (IJSSCI)*, 7(4) :35–61, 2015.

[AK17] Biri Arun and T.V. Vijay Kumar. Materialized view selection using artificial bee colony optimization. *International Journal of Intelligent Information Technologies (IJIIT)*, 13(1) :26–49, 2017.

[BDD+98] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan, James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in oracle. In *VLDB*, 1998.

[CSA+17] Broderick Crawford, Ricardo Soto, Gino Astorga, José García, Carlos Castro, and Fernando Paredes. Putting continuous metaheuristics to work in binary search spaces. *Complexity*, 2017 :8404231, May 2017.

[Dav85] Lawrence Davis. Applying adaptive algorithms to epistatic domains. IJCAI'85, page 162–164, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

[DDKS06] Roozbeh Derakhshan, Frank Dehne, Othmar Korn, and Bela Stantic. Simulated annealing for materialized view selection in data warehousing environment. DBA'06, page 89–94, USA, 2006. ACTA Press.

[DPAM02] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm : Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2) :182–197, 2002.

[Edg81] F.Y. Edgeworth. *Mathematical Psychics : An Essay on the Application of Mathematics to the Moral Sciences*. Reprints of economic classics. C. K. Paul, 1881.

[Edm71] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1(1) :127–136, Dec 1971.

[FF93] Carlos M. Fonseca and Peter John Fleming. Genetic algorithms for multiobjective optimization : Formulationdiscussion and generalization. In *International Conference on Genetic Algorithms*, 1993.

[GBD13] Rajib Goswami, Dhruba Kumar Bhattacharyya, and Malayananda Dutta. Multiobjective differential evolution algorithm using binary encoded data in selecting views for materializing in data warehouse. In Bijaya Ketan Panigrahi, Ponnuthurai Nagaratnam Suganthan, Swagatam Das, and Shubhransu Sekhar Dash, editors, *Swarm, Evolutionary, and Memetic Computing*, pages 95–106, Cham, 2013. Springer International Publishing.

[GM99] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, page 453–470, Berlin, Heidelberg, 1999. Springer-Verlag.

[GM05] H. Gupta and I.S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 17(1) :24–43, 2005.

[GYL06] Gang Gou, J.X. Yu, and Hongjun Lu. A/sup */ search : an efficient and flexible approach to materialized view selection. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 36(3) :411–425, 2006.

[HCL03] J.-T. Horng, Y.-J. Chang, and B.-J. Liu. Applying evolutionary algorithms to materialized view selection in a data warehouse. *Soft Computing*, 7(8) :574–581, Aug 2003.

[HCLK99] Jorng-Tzong Horng, Yu-Jan Chang, Baw-Jhiune Liu, and Cheng-Yan Kao. Materialized view selection using genetic algorithms in a data warehouse system. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 3, pages 2221–2227 Vol. 3, 1999.

[HK02] Kuk-Hyun Han and Jong-Hwan Kim. Quantum-inspired evolutionary algorithm for a class of combinatorial optimization. *IEEE Transactions on Evolutionary Computation*, 6(6) :580–593, 2002.

[HNG94] J. Horn, N. Nafpliotis, and D.E. Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 82–87 vol.1, 1994.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. *ACM SIGMOD Record*, 25(2) :205–216, 1996.

[Inm02] W.H. Inmon. *Building the Data Warehouse*. Wiley, 2002.

[Kim97] Ralph Kimball. A dimensional modeling manifesto. 1997.

[KK13] T. V. Vijay Kumar and Santosh Kumar. Materialized view selection using memetic algorithm. In Rajendra Prasath and T. Kathirvalavakumar, editors, *Mining Intelligence and Knowledge Exploration*, pages 316–327, Cham, 2013. Springer International Publishing.

[KM14] A. Kaveh and V.R. Mahdavi. Colliding bodies optimization : A novel meta-heuristic method. *Computers and Structures*, 139 :18–27, 2014.

[KMP02] Panos Kalnis, Nikos Mamoulis, and Dimitris Papadias. View selection using randomized search. *Data Knowledge Engineering*, 42(1) :89–111, 2002.

[KR99] Yannis Kotidis and Nick Roussopoulos. Dynamat : A dynamic view management system for data warehouses. *SIGMOD Rec.*, 28(2) :371–382, jun 1999.

[KR13] R. Kimball and M. Ross. *The Data Warehouse Toolkit : The Definitive Guide to Dimensional Modeling*. Wiley, 2013.

[KVK18] Santosh Kumar and T. V. Vijay Kumar. A novel quantum-inspired evolutionary view selection algorithm. *Sādhanā*, 43(10) :166, Aug 2018.

[Law06] Michael Lawrence. Multiobjective genetic algorithms for materialized view selection in olap data warehouses. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, page 699–706, New York, NY, USA, 2006. Association for Computing Machinery.

[Lay11] Abdesslem Layeb. A novel quantum inspired cuckoo search for knapsack problems. 3(5) :297–305, sep 2011.

[LH01] MINSOO LEE and JOACHIM HAMMER. Speeding up materialized view selection in data warehouses using a randomized algorithm. *International Journal of Cooperative Information Systems*, 10(03) :327–353, 2001.

[LK04] Wen-Yang Lin and I-Chung Kuo. A genetic selection algorithm for olap data cubes. *Knowledge and Information Systems*, 6(1) :83–102, Jan 2004.

[LQJW10] Xin Li, Xu Qian, Junlin Jiang, and Ziqiang Wang. Shuffled frog leaping algorithm for materialized views selection. In *2010 Second International Workshop on Education Technology and Computer Science*, volume 3, pages 7–10, 2010.

[MB20a] Raouf Mayata and Abdelmadjid Boukra. Hybrid algorithm for materialised view selection. *International Journal of Innovative Computing and Applications*, 11(4) :167–180, 2020.

[MB20b] Raouf Mayata and Abdelmadjid Boukra. Using quantum evolutionary based algorithm to solve materialized view selection problem. DTUC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[MB21] Raouf Mayata and Abdelmadjid Boukra. Materialized view selection using discrete quantum based differential evolution algorithm. In Salim Chikhi, Abdelmalek Amine, Allaoua Chaoui, Djamel Eddine Saidouni, and Mohamed Khireddine Kholladi, editors, *Modelling and Implementation of Complex Systems*, pages 203–216, Cham, 2021. Springer International Publishing.

[Moo97] H.L. Moore. Cours d'Économie politique. by vilfredo pareto, professeur à l'université de lausanne. vol. i. pp. 430. i896. vol. ii. pp. 426. i897. lausanne : F. rouge. *The ANNALS of the American Academy of Political and Social Science*, 9(3) :128–131, 1897.

[PK19a] Jay Prakash and T. V. Vijay Kumar. Multi-objective materialized view selection using improved strength pareto evolutionary algorithm. *International Journal of Artificial Intelligence and Machine Learning*, 9, 2019.

[PK19b] Jay Prakash and T.V. Vijay Kumar. A multi-objective approach for materialized view selection. *International Journal of Operations Research and Information Systems*, 10, 2019.

[PK20] Jay Prakash and T. V. Vijay Kumar. Multi-objective materialized view selection using nsga-ii. *International Journal of System Assurance Engineering and Management*, 11(5) :972–984, Oct 2020.

[PoA07] Jiratta Phuboon-ob and Raweewan Auepanwiriyakul. Selecting Materialized Views Using Two-Phase Optimization with Multiple View Processing Plan, March 2007.

[PVK20] Jay Prakash and T. V. Vijay Kumar. Multi-objective materialized view selection using moga. *International Journal of System Assurance Engineering and Management*, 11(2) :220–231, Jul 2020.

[RY00] T.P. Runarsson and Xin Yao. Stochastic ranking for constrained evolutionary optimization. *IEEE Transactions on Evolutionary Computation*, 4(3) :284–294, 2000.

[Sil08] F. Silvers. *Building and Maintaining a Data Warehouse*. CRC Press, 2008.

[Sim13] D. Simon. *Evolutionary Optimization Algorithms*. Wiley, 2013.

[SM96] Marc Schoenauer and Zbigniew Michalewicz. Evolutionary computation at the edge of feasibility. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature — PPSN IV*, pages 245–254, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[SP97] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4) :341–359, Dec 1997.

[SW09] Xia Sun and Ziqiang Wang. An efficient materialized views selection algorithm based on pso. In *2009 International Workshop on Intelligent Systems and Applications*, pages 1–4, 2009.

[Tal09] E.G. Talbi. *Metaheuristics : From Design to Implementation.* Wiley Series on Parallel and Distributed Computing. Wiley, 2009.

[VKK12a] T. V. Vijay Kumar and Santosh Kumar. Materialized view selection using genetic algorithm. In Manish Parashar, Dinesh Kaushik, Omer F. Rana, Ravi Samtaney, Yuanyuan Yang, and Albert Zomaya, editors, *Contemporary Computing*, pages 225–237, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[VKK12b] T. V. Vijay Kumar and Santosh Kumar. Materialized view selection using simulated annealing. In Srinath Srinivasa and Vasudha Bhatnagar, editors, *Big Data Analytics*, pages 168–179, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[VKK13] T. V. Vijay Kumar and Santosh Kumar. Materialized view selection using iterative improvement. In Natarajan Meghanathan, Dhinaharan Nagamalai, and Nabendu Chaki, editors, *Advances in Computing and Information Technology*, pages 205–213, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[VKK14] T.V. Vijay Kumar and Santosh Kumar. Materialised view selection using differential evolution. *International Journal of Innovative Computing and Applications*, 6(2) :102–113, 2014.

[vLM13] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms : A survey. *ACM Comput. Surv.*, 45(3), jul 2013.

[YKL97] Jian Yang, Kamalakar Karlapalem, and Qing Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 136–145, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[YYCG03] Jeffrey Xu Yu, Xin Yao, Chi-Hon Choi, and Gang Gou. Materialized view selection as constrained evolutionary optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 33(4) :458–467, 2003.

[ZY99] Chuan Zhang and Jian Yang. Genetic algorithm for materialized view selection in data warehouse environments. In Mukesh Mohania and A. Min Tjoa, editors, *DataWarehousing and Knowledge Discovery*, pages 116–125, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[ZYY01] Chuan Zhang, Xin Yao, and Jian Yang. An evolutionary approach to materialized views selection in a data warehouse environment. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 31(3) :282–294, 2001.