Université Libre de Bruxelles

INFO-F-524
Optimisation continue

Stochastic Unit Commitment

Antoine Passemiers
Cédric Simar

26 mai 2018

Table des matières

1 Utilisation du programme

1.1 Prérequis

Afin de faire tourner correctement le projet, les librairies suivantes nécessitent d'être installées.

— NumPy ($\geq$ 1.13.3)

— Cython ($\geq$ 0.25.2)

— PuLP

1.2 Appel au programme

```
$ # Résolution du primal (pour trouver la valeur de la solution optimale)
$ python main.py <path_to_instance>
$ # Résolution de la relaxation linéaire
$ python main.py <path_to_instance> --relax
$ # Résolution de la relaxation linéaire + arrondi
$ python main.py <path_to_instance> --relax --round
$ # Décomposition lagrangienne et méthode du sous-gradient
$ python main.py <path_to_instance> --decompose
$ python main.py <path_to_instance> --decompose --nar 6 --epsilon 0.01 --alpha0
   2000 --rho 0.96
```

Les différents paramètres de l'algorithme du sous-gradient sont les suivants :

— n_{ar} (nar) : le nombre d'itérations du sous-gradient à effectuer avant de commencer à appliquer l'heuristique et obtenir des solutions primales faisables.

— ϵ (epsilon) : Seuil de convergence / saut de dualité en dessous duquel l'algorithme est considéré comme ayant convergé. Lorsque $(UB - LB)/UB < \epsilon$, l'algorithme s'arrête.

— α_0 (alpha0) : Le pas initial du sous-gradient.

— ρ (rho) : Le facteur de diminution du pas (si aucune solution primale admissible n'a encore été trouvée).

2 Considérations générales sur l'implémentation

Dans le cadre de ce projet, nous avons fait le choix d'utiliser PuLP (Mitchell et al., 2011) en tant qu'interface entre Cbc et Python. PuLP est une bibliothèque de modélisation de programmes linéaires et permet la résolution de ces derniers au moyen de solveurs tels que GLPK, CBC, CLP ou autre. L'API est stable et largement utilisée, mais contrairement à celle de la bibliothèque CVXPY, elle ne permet pas de manipuler des variables ou des contraintes de manière vectorisée à l'aide de matrices ou d'autres structures de données dédiées.

L'implémentation d'une formulation peut de fait devenir fastidieuse et les erreurs difficiles à retracer lorsque l'on vient à définir des variables multi-indices (implémentables sous forme de matrices de variables) et indicer ces dernières au sein de boucles. En effet l'absence d'erreur du programme n'implique pas que les contraintes ont été correctement implémentées et toutes ajoutées au modèle. Numpy apporte une sécurité supplémentaire relative à la dimensionalité : si par exemple un produit matriciel est effectué entre une matrice de constantes et un vecteur de variables PuLP et que les dimensions sont incompatibles, une exception est automatiquement levée. Ceci tient également pour la construction de contraintes où le membre de gauche ne possède pas la même dimensionalité que le membre de droite, pour la somme de variables, etc.

2.1 Extension de PuLP

Pour les raisons que nous venons de donner, nous avons étendu Numpy afin de rendre l'indiçage fantaisiste ("fancy indexing") comptatible avec les variables PuLP. Pour ce faire, nous avons simplement sous-classé les tableaux *numpy.ndarray* et notamment réécrit les méthodes de comparaison. La classe fille s'appelle *LpVarArray*. Nous avons entre autres veillé à ce que :

— Une combinaison linéaire d'objets *LpVarArray* de dimensions compatibles résultent en un un nouvel objet *LpVarArray*.

— Une comparaison entre deux *LpVarArray* donne un tableau de contraintes PuLP.

— Les tableaux de contraintes PuLP puissent être ajoutés à une instance de problème. Pour ce faire, nous avons sous-classé *pulp.LpProblem* en *ArrayCompatibleLpProblem* et ajouté les méthodes requises.

Motivons nos choix d'implémentation par un exemple concret d'utilisation de ces nouvelles classes. Supposons que p soit un *LpVarArray* tridimensionnel et R_plus un *numpy.ndarray* contenant des constantes. La modélisation peut donc se faire selon deux styles, et nous avons opté pour la seconde version :

```
    # Ajout séquentiel des contraintes
    for g in range(G):
        for s in range(S):
            for t in range(1, T):
                problem += (p[g, s, t] - p[g, s, t-1] <= R_plus[g])

    # Ajout des contraintes par fancy indexing
    problem += (np.swapaxes(p[:, :, 1:] - p[:, :, :-1], 0, 2) <= R_plus)
```

2.2 Contenu du dossier source

Voici comment l'implémentation du projet a été répartie :

— *decomposition.py* : Contient l'implémentation de l'unique fonction *decompose_problem*, permettant sur base d'une instance de type *SUPInstance* de générer les instances des problèmes PuLP nécessaires, dont le problème original (PP), les sous problèmes issus de la décomposition lagrangienne ($P1_s$ et $P2$), ainsi que les problèmes de répartition économique pour chaque scénario (ED_s). Tous les problèmes/sous-problèmes partagent les variables PuLP, ce qui permet à la résolution d'un problème de **directement mettre à jour les valeurs des variables** dans les problèmes partageant ces mêmes variables, sans que nous ayons explicitement à le faire.

— *dive_and_fix.py* : Implémentation de l'heuristique dive-and-fix, que nous avons gardé malgré les mauvaises performances de celle-ci.

— *experimental.py* : Fichier bac à sable

— *genetic.pyx* : Implémentation en "pur C" et parallèle d'un algorithme génétique permettant de minimiser le nombre de contraintes non satisfaites pour une instance du problème.

— *heuristics.py* : Implémentation de *evolve-and-fix* (voir tâche 2).

— *instance.py* : Structure de données pour les constantes et indices du problème, accompagnée de fonctions utilitaires pour parser les différents fichiers d'instance. Notez la différence entre une instance *SUPInstance* telle que définie dans ce fichier, et les instances des problèmes PuLP tels que définis à l'aide de la classe *SUCLpProblem* dans le fichier *utils.py*.

— *lp_relaxation.py* : Formulations du problème d'origine ainsi que de sa relaxation linéaire. Selon la formulation désirée, le problème est renvoyé sous forme d'une instance de la classe *SUCLpProblem*.

— *main.py* : Point d'entrée du programme et parseur de commandes

— *subgradient.py* : Algorithme du sous-gradient permettant de trouver une solution duale sur base des sous-problèmes $P1_s$ et $P2$.

— *utils.py* : Problèmes PuLP et intégration de NumPy dans PuLP telle que décrite dans la section précédente.

— *variables.py* : Initialisation des variables nécessaires à la formulation du SUC. Les contraintes spéciales (bornes sur les variables) sont implictement ajoutées au problème. Par exemple, les variables u_{gst} sont supposées entre comprises entre 0 et 1, et ce peu importe si la formulation choisie est le problème d'origine, la relaxation linéaire ou même la décomposition lagrangienne.

3 Tâche 1

Résoudre la relaxation linéaire de la Formulation SUC

3.1 Relaxation linéaire

La relaxation linéaire du modèle utilisé [1] peut être formulée ainsi :

$$\text{(RL)} \qquad \min \sum_{g\in G}\sum_{s\in S}\sum_{t\in T} \pi_s(K_g u_{gst} + S_g v_{gst} + C_g p_{gst}) \tag{3.20}$$

t.q.

$$\sum_{l\in LI_n} e_{lst} + \sum_{g\in G_n} p_{gst} = D_{nst} + \sum_{l\in LO_n} e_{lst} \qquad \forall n\in N, s\in S, t\in T \tag{3.21}$$

$$e_{lst} = B_{ls}(\theta_{nst} - \theta_{mst}) \qquad \forall l=(m,n)\in L, s\in S, t\in T \tag{3.22}$$

$$e_{lst} \le TC_l \qquad \forall l\in L, s\in S, t\in T \tag{3.23}$$

$$-TC_l \le e_{lst} \qquad \forall l\in L, s\in S, t\in T \tag{3.24}$$

$$p_{gst} \le P^+_{gs} u_{gst} \qquad \forall g\in G, s\in S, t\in T \tag{3.25}$$

$$P^-_{sg} u_{gst} \le p_{gst} \qquad \forall g\in G, s\in S, t\in T \tag{3.26}$$

$$p_{gst} - p_{gs,t-1} \le R^+_g \qquad \forall g\in G, s\in S, t\in T \tag{3.27}$$

$$p_{gs,t-1} - p_{gst} \le R^-_g \qquad \forall g\in G, s\in S, t\in T \tag{3.28}$$

$$\sum_{q=t-UT_g+1}^{t} z_{gq} \le w_{gt} \qquad \forall g\in G_s, t\ge UT_g \tag{3.29}$$

$$\sum_{q=t+1}^{t+DT_g} z_{gq} \le 1 - w_{gt} \qquad \forall g\in G_s, t\le N - DT_g \tag{3.30}$$

$$\sum_{q=t-UT_g+1}^{t} v_{gsq} \le u_{gst} \qquad \forall g\in G_f, t\ge UT_g \tag{3.31}$$

$$\sum_{q=t+1}^{t+DT_g} v_{gsq} \le 1 - u_{gst} \qquad \forall g\in G_f, t\le N - DT_g \tag{3.32}$$

$$z_{gt} \le 1 \qquad \forall g\in G_s, t\in T \tag{3.33}$$

$$v_{gst} \le 1 \qquad \forall s\in S, t\in T \tag{3.34}$$

$$z_{gt} \ge w_{gt} - w_{g,t-1} \qquad \forall g\in G_s, t\in T \tag{3.35}$$

$$v_{gst} \ge u_{gst} - u_{gs,t-1} \qquad \forall g\in G_f, s\in S, t\in T \tag{3.36}$$

$$\pi_s u_{gst} = \pi_s w_{gt} \qquad \forall g\in G_s, s\in S, t\in T \tag{3.37}$$

$$\pi_s v_{gst} = \pi_s z_{gt} \qquad \forall g\in G_s, s\in S, t\in T \tag{3.38}$$

$$p_{gst} \ge 0,\ 0 \le u_{gst} \le 1 \qquad \forall g\in G, s\in S, t\in T \tag{3.39}$$

$$z_{gt} \ge 0,\ 0 \le w_{gt} \le 1 \qquad \forall g\in G_s, t\in T \tag{3.40}$$

1. A. Papavasiliou. Coupling Renewable Energy Supply with Deferrable Demand. PhD thesis, University of California, Berkeley, 2011, pg 22

3.1.1 Incertitudes du modèle

— Les contraintes (3.21) d'équilibre de marché imposent que la production d'énergie satisfasse la demande étant donné les niveaux de puissance assignés aux lignes de transmission.

— Les contraintes (3.22) sont obtenues à partir de la première loi de Kirchhoff (loi des noeuds) et de la seconde loi de Kirchhoff (loi des mailles). La susceptance y sert à modéliser la puissance lorsque les lignes sont parcourues par un courant continue. Les lignes du réseau peuvent être soumises à des contingences, symbolisées par les susceptances des lignes. Lorsqu'une ligne de transmission l est mise hors service dans un scénario s, la susceptance associée B_{ls} est fixée à 0.

— Les contraintes de types (3.25) et (3.26) tiennent compte des contingences liées aux générateurs eux-mêmes. Dans tout scénario s où un générateur g est hors usage, les constantes P^{+}_{gs} et P^{-}_{gs} sont fixées à 0 afin de forcer la production à 0.

3.1.2 Non-anticipativité

Les contraintes (3.37) et (3.38) de non-anticipativité permettent de forcer la planification des générateurs lents établie lors de la seconde phase à respecter celle effectuée pour ces mêmes générateurs lors de la première phase. En effet les contingences peuvent être énumérées (d'où la présence de scénarios) mais ne peuvent avoir de caratère certain. L'engagement des générateurs lents doit donc être le même dans tous les scénarios, d'où les contraintes d'égalité. Étant donné que les variables w_{gt} et zgt sont définies pour les générateurs lents et les variables u_{gst} et v_{gst} pour tous les générateurs, u_{gst} et v_{gst} sont redondantes pour les générateurs lents.

Cette redondance n'est justifiée que par l'utilisation de la relaxation lagrangienne et la dualisation des contraintes de non-anticipativité, car elle permet de borner les sous-problèmes produits par la décomposition lagrangienne. En l'absence d'une telle relaxation il convient alors de retirer de la modélisation les variables u_{gst} et v_{gst} associées aux générateurs lents. Puisque nous avons gardé la même approche qu'Anthony Papavasiliou pour la décomposition lagrangienne, toutes les variables ont été gardées.

3.1.3 Démarrage des générateurs et contraintes temporelles

Les contraintes de type (3.35) et (3.36) couplent des variables liées à des périodes de temps adjacentes. Par exemple, les contraintes de type (3.35) sont définies $\forall t \in T$. Or les valeurs des variables w_{gt} ne sont pas connues pour $t = 0$. Nous n'avons pas fait l'hypothèse que les générateurs en question sont éteints en $t = 0$, et simplement retiré la première contrainte. Les contraintes sont donc définies $\forall t \in T \backslash \{0\}$, ce qui simplifie le problème et allège légèrement les temps d'exécution.

4 Tâche 2

Développer une heuristique qui trouve une solution entière admissible

Arrondir les variables binaires aux entiers les plus proches ne permet pas de trouver une solution admissible du primal. Il est nécessaire de trouver une heuristique menant à une solution admissible le plus proche possible de la solution entière optimale. L'heuristique dive-and-fix ne permet pas de trouver une telle solution car elle fixe des variables u_{gst} de manière très disparate dans le temps (t), ce qui amène rapidement à la non-satisfaisabilité de certaines contraintes liées au démarrage des générateurs tels que les contraintes (3.32). Ces contraintes sont gênantes car elles couplent un grand nombre de variables appartenant à différentes périodes de temps. La solution que nous proposons est un algorithme itératif répétant les trois étapes suivantes :

— **Résolution de la relaxation linéaire du problème** : la relaxation linéaire est celle formulée dans la partie 3 du rapport.

— **Minimisation du nombre de contraintes violées via un algorithme génétique** : L'algorithme génétique est implémenté en Cython (Behnel et al., 2011) afin d'améliorer la vitesse d'exécution. La partie sensible du code est optimisée en pur C et parallélisée, ce qui fait que l'exécution est instantanée et que le goulot d'étranglement de l'algorithme est la résolution de la relaxation linéaire et dépend donc uniquement du solveur et du temps pris par PuLP pour effectuer son pré-traitement du problème. Il convient donc d'installer Cython pour pouvoir bénéficier de cet algorithme. Dans le cas contraire, l'exécution de notre programme peut encore se faire mais de manière limitée.

— **Fixation des variables suivant des règles spécifiques au problème** : Ces règles sont déterminées de manière purement empiriques, peuvent encore être améliorées ou choisie plus intelligemment.

Nous avons modestement nommé cette heuristique "evolve-and-fix" dans notre code source.

4.1 Résolution de la relaxation linéaire

La solution obtenue par le solveur est infaisable pour le problème d'origine, et requiert d'être arrondie de manière heuristique. Les valeurs des variables sont alors extraites du problème PuLP et injectée dans un *CyProblem* (voir *src/genetic.pyx*) qui représente le problème d'origine dans la partie C du code. La représentation du problème du côté Python est quant à elle assurée par la classe *SUCLpProblem* (voir *src/utils.py*).

4.2 Algorithme génétique

L'algorithme génétique proposé a une forme très générale à l'exception qu'il génère des solutions infaisables et cherche à se rapprocher de la zone admissible en diminuant les valeurs des contraintes non satisfaites. L'implémentation se trouve dans le fichier *src/genetic.pyx*. Décrivons chacune des composantes de l'algorithme.

4.2.1 Représentation binaire des individus

Chaque bit correspond à l'une des variables binaires du problème d'origine. Le nombre de bits est donc égal au nombre de contraintes d'intégralité de l'instance du problème. Les variables faisant partie de la représentation binaire sont toutes les variables u_{gst} et w_{gt}.

4.2.2 Initialisation de la population

Les contraintes d'intégralité du problème sont généralement non-satisfaites après résolution de la relaxation linéaire. Les variables concernées étant de valeurs fractionnaires, nous avons exploité ce problème et généré aléatoirement des membres de la population sur base de leurs valeurs. Chaque individu de la population est créé de telle sorte que chaque variable x_i de la solution associée à cet individu est arrondie suivant cette simple règle :

$$x_i = \begin{cases} 1 & \text{si } x_i > \zeta \text{ , } \zeta \sim U(0,1) \\ 0 & sinon \end{cases} \tag{4.1}$$

La taille de la population est un paramètre que nous fixons à 100 par défaut.

4.2.3 Fonction d'adaptation

Deux différentes fonctions d'adaptation ont été expérimentées.

— *Somme pondérée des valeurs des contraintes* : la difficulté est d'associer à chaque type de contraintes un coefficient. Étant donné qu'il y a près d'une dizaine de types de contraintes en jeu lors de l'exécution de l'algorithme génétique (et une vingtaine dans le problème d'origine), déterminer les paramètres idéaux peut prendre beaucoup de temps et l'introduction de ces paramètres risquent de rendre l'heuristique trop spécifique au problème.

— *Nombre de contraintes violées* : cette heuristique fonctionne mieux en pratique et **ne requiert aucun ajustement de paramètres**.

Les contraintes intervenant dans le calcul de la fonction d'adaptation sont les suivantes : (3.25), (3.26), (3.29), (3.30), (3.31), (3.32), (3.35), (3.36), (3.37). En effet toutes les autres contraintes voient leur valeur inchangée durant l'exécution de l'algorithme génétique.

4.2.4 Tournois et sélection des parents

Deux sous-ensembles (généralement de taille 2) sont créés à partir de la population et l'individu avec la meilleure fonction d'adaptation est gardé. Les deux sous-ensembles sont disjoints. Un tournoi est effectué pour chacun des deux sous-ensembles afin d'obtenir deux parents.

4.2.5 Opérateur d'enjambement

L'opérateur implémente un enjambement uniforme qui, pour chaque variable sur laquelle porte une contrainte d'intégralité, choisit aléatoirement de quel parent récupérer la valeur de cette variable et la fixer dans la solution du fils. Plus formellement, P le nombre de contraintes d'intégralité et ξ_i un entier $\in \{0,1\}$ $\forall i \in \{1,\ldots,P\}$, le fils c peut alors être décrit ainsi :

$$c_i = \begin{cases} a_i & \text{if } \xi_i = 0 \\ b_i & \text{if } \xi_i = 1 \end{cases} \tag{4.2}$$

où c_i est le ième bit de c (la valeur de la ième variable de c sur laquelle porte une contrainte d'intégralité), a_i et b_i sont les ièmes bits des parents a et b, respectivement.

4.2.6 Opérateur de mutation

L'opérateur de mutation inverse aléatoirement certains bits dans la représentation binaire de l'individu fils. Le taux de mutation est un paramètre que nous fixons à 2 par défaut.

Algorithm 1 Algorithme génétique pour le problème SUC

```
1 : procedure GENETIC_ALGORITHM
2 :     t ← 0
3 :     Créer une population Pop aléatoirement sur base des variables fractionnaires
4 :     Soit s* l'individu tel que f(s*) ≥ f(s)∀s ∈ Pop
5 :     while (t < t_max) do
6 :         Créer un sous-ensembles aléatoires P_1, P_2 ⊆ Pop
7 :         a = tournament(P_1)
8 :         b = tournament(P_2)
9 :         c = crossover(a, b)
10 :        mutate(c)
11 :        if c est identique à un autre individu then
12 :            Détruire d'individu c
13 :        else
14 :            Détruire l'individu w le moins adapté et le remplacer par c
15 :        if f(c) > f(s*) then
16 :            s* = c
17 :        t ← t + 1
```

4.3 Fixation de variables

Si l'algorithme génétique ne renvoit pas de solution admissible, l'étape suivante consiste à sacrifier une partie de l'objectif en augmentant les coûts, créant une marge et permettant à l'algorithme génétique de satisfaire plus de contraintes à l'itération suivante.

— Si une contrainte de type (3.26) est non satisfaite, alors la variable p_{gst} impliquée est fixée à P^+_{gs}. Ceci permet de créer de la marge pour changer les quantités produites par les autres générateurs tout en satisfaisant les mêmes demandes d'énergie.

— Si une contrainte de type (3.25) est non satisfaite, alors la variable u_{gst} impliquée est fixée à 1. Ceci permet de forcer la production lors de la résolution de la relaxation linéaire de l'itération suivante. Ceci rejoint l'idée du point présent et est nécessaire pour la concrétiser car une variable u_{gst} à 0 force la production p_{gst} à 0.

— Si une contrainte de type (3.36) est non satisfaite, alors la variable v_{gst} impliquée est fixée à 1 avec une probabilité $(u_{gst} - u_{gs,t-1} - v_{gst})/2$. L'ajout du caractère aléatoire à la fixation permet simplement de limiter le nombre de variables fixées et d'éviter de violer une des contraintes temporelles (3.29), (3.30), (3.31) ou (3.32) trop rapidement. Il est clair que ce choix est arbitraire et peut être abandonné au profit d'une approche plus "dive-and-fix" en fixant uniquement la variable v_{gst} la moins fractionnaire. Cependant, la fixation d'une seule variable par itération rend l'exécution du programme significativement plus longue et rend l'utilisation d'une heuristique injustifiée lorsque la résolution du problème d'origine via le solveur est plus rapide.

— Si une contrainte de type (3.35) est non satisfaite, alors la variable z_{gt} impliquée est fixée à 1 avec une probabilité $(w_{gt} - w_{g,t-1} - z_{gt})/2$.

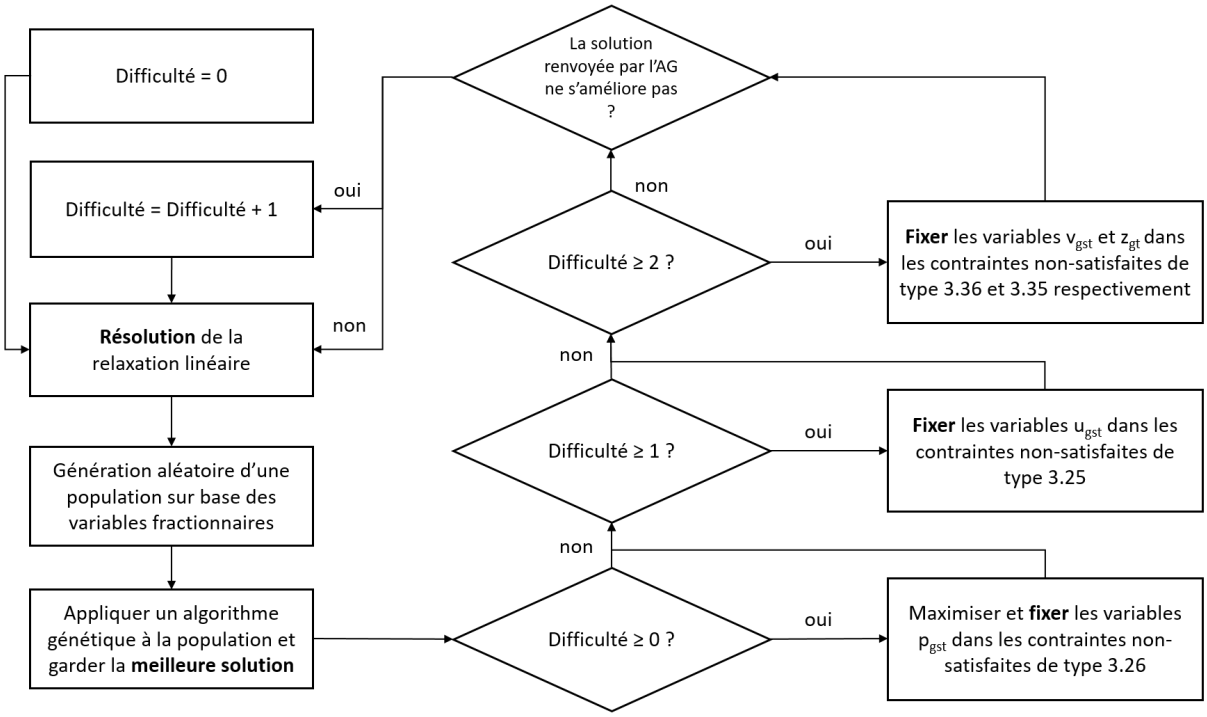
— Si une contrainte de type (3.29), (3.30), (3.31) ou (3.32) est non satisfaite et aucune contrainte de type (3.26), (3.25), (3.36) ou (3.35) est violée, alors l'heuristique échoue.

— Enfin, les variables les plus susceptibles de violer les contraintes de type (3.26), (3.25), (3.36) ou (3.35) sont fixées en dernier, ce qui est illustré clairement dans la section suivante.

4.4 Vue générale de l'algorithme

Il est important de noter que l'heuristique échoue sur des instances difficiles lorsqu'appliquée directement sur une solution obtenue par relaxation linéaire. Nous imaginons que le cheminement de la solution optimale de la relaxation vers la zone admissible du primal n'est pas trivial et nécessite des améliorations au niveau de la fixation de variables. **L'algorithme proposé ne garantit donc aucune solution faisable et peut échouer**. Cependant, une solution primale est trouvée dans tous les cas lorsque l'heuristique est appliquée sur une solution obtenue avec l'algorithme du sous-gradient (voir partie 3) et une séquence ergodique (voir partie 4), et que l'algorithme converge effectivement.

Le graphique suivant reprend l'ensemble du fonctionnement d'*evolve-and-fix*, tout en montrant comment la difficulté introduite par la fixation des différents types de variables est pris en compte.



Vue générale de l'algorithme proposé "evolve-and-fix"

5 Tâche 3

Résoudre une relaxation Lagrangienne de la Formulation SUC

5.1 Décomposition lagrangienne

Le choix des contraintes relâchées est le même que celui effectué par A. Papavasiliou : seules les contraintes de non-anticipitavité ont été dualisées. Les deux raisons principales sont que d'une part elles compliquent fortement le problème à résoudre et que d'autre part, elles sont les seules contraintes à coupler les différents scénarios entre eux. Sans elles, *il est désormais possible de décomposer le problème suivant ses différents scénarios*.

Enfin, la relaxation de ses contraintes seules est suffisante pour diminuer significativement le temps d'exécution du solveur. En particulier, la somme des temps d'exécution sur les différents sous-problèmes créés est fortement inférieure au temps d'exécution sur le problème primal, et ceci est d'autant plus valable pour les instances de grande taille.

Le dual lagrangien est donc obtenu en reprenant la fonction objectif du primal et en y dualisant les contraintes de non-anticipativité (3.37) et (3.38) :

$$\begin{aligned}\mathcal{L} &= \sum_{g\in G}\sum_{s\in S}\sum_{t\in T}\pi_s(K_s u_{gst} + S_g v_{gst} + C_g p_{gst}) + \sum_{g\in G_s}\sum_{s\in S}\sum_{t\in T}\pi_s(\mu_{gst}(u_{gst} - w_{gt}) + \nu_{gst}(v_{gst} - z_{gt}))\\ &= \sum_{s\in S}\Big(\sum_{g\in G}\sum_{t\in T}(K_s u_{gst} + S_g v_{gst} + C_g p_{gst}) + \sum_{g\in G_s}\sum_{t\in T}\pi_s(\mu_{gst}u_{gst} + \nu_{gst}v_{gst})\Big)\\ &- \sum_{g\in G_s}\sum_{s\in S}\sum_{t\in T}\pi_s(\mu_{gst}w_{gt} + \nu_{gst}z_{gt})\end{aligned}$$

Nous observons qu'en réarrangeant les termes de la fonction il est possible de l'exprimer sous la forme d'une somme de plusieurs objectifs, dont un est indépendant du scénario et chacun des autres est assigné à un scénario s. La décomposition lagrangienne se fait alors comme suit :

— **Les sous-problèmes $\mathbf{P1_s}$** concernent la plannification des générateurs rapides. Le sous-problème $P1_s$ requiert de plannifier les générateurs rapides dans le cadre du scénario s. La fonction objectif est donnée par (Papavasiliou and Oren, 2013) :

$$\min \sum_{g\in G}\sum_{t\in T}(K_s u_{gst} + S_g v_{gst} + C_g p_{gst}) + \sum_{g\in G_s}\sum_{t\in T}\pi_s(\mu_{gst}u_{gst} + \nu_{gst}v_{gst}) \tag{5.1}$$

— **Le sous-problème P2** concerne la plannification des générateurs lents et ne prend donc pas en considération les différents scénarios. La valeur de son objectif est quant à elle donnée par :

$$\min -\sum_{g\in G_s}\sum_{s\in S}\sum_{t\in T}\pi_s(\mu_{gst}w_{gt} + \nu_{gst}z_{gt}) \tag{5.2}$$

La valeur de l'objectif du dual lagrangien est donc exprimé sous la forme d'une somme des objectifs des différents sous-problèmes issus de cette décomposition. La tâche consiste à présent à optimiser le dual lagrangien afin de rapprocher la solution duale de la zone admissible du primal.

5.2 Optimisation du dual lagrangien

5.2.1 Choix de l'algorithme

Comme expliqué par A. Papavasiliou dans une présentation de 2016 (Papavasiliou and Aravena, 2016), il est possible que certains sous-problèmes $P1_s$ prennent considérablement plus de temps que le autres pour être résolus, *jusqu'à 75 fois le temps pris par le sous-problème le plus rapide*. Ceci crée des goulots d'étranglement lors de l'optimisation du dual, qui peuvent cependant être résolus par l'utilisation d'un algorithme d'optimisation asynchrone. En particulier, l'*algorithme de descente par blocs* a été présenté et permet de mettre à jour un sous-ensemble des multiplicateurs lagrangiens en calculant une partie du sous-gradient à la fois. Ce fragment est le sous-ensemble de composantes associées à un seul scénario. L'algorithme se distingue donc des algorithmes de descente par coordonnée car plusieurs variables duales sont mis à jour à la fois.

Nous avons eu la chance de ne pas observer de tels ralentissements pour les instances fournies et les temps de résolution des différents sous-problèmes étaient fort similaires. En conséquence, nous avons fait le choix d'utiliser l'*algorithme du sous-gradient* car il est plus simple à implémenter que l'algorithme de descente par blocs. Le sous-gradient d^k associés aux multiplicateurs lagrangiens à l'itération k est donné par :

$$d^k_{\mu_{gst}} = \pi_s(w^k_{gt} - u^k_{gst}) \ \ \forall g \in G_s, s \in S, t \in T \tag{5.3}$$

$$d^k_{\nu_{gst}} = \pi_s(z^k_{gt} - v^k_{gst}) \ \ \forall g \in G_s, s \in S, t \in T \tag{5.4}$$

où d^k_μ est le partie du sous-gradient associée aux contraintes (3.37) et d^k_ν est celle associée aux contraintes (3.38). Il est important de noter que **les multiplicateurs lagrangiens sont mis à jour à l'aide de l'équation décrite dans cette même présentation et non celle présentée dans la thèse** (Papavasiliou, 2012) (il s'agit selon nous d'une simple erreur de signe). Les multiplicateurs lagrangiens sont donc calculés d'itération en itération de la façon suivante :

$$\mu^0_{gst} = 0 \ \ \forall g \in G_s, s \in S, t \in T \tag{5.5}$$

$$\nu^0_{gst} = 0 \ \ \forall g \in G_s, s \in S, t \in T \tag{5.6}$$

$$\mu^{k+1}_{gst} = \mu^k_{gst} - \alpha_k d^k_{\mu_{gst}} \ \ \forall g \in G_s, s \in S, t \in T \tag{5.7}$$

$$\nu^{k+1}_{gst} = \nu^k_{gst} - \alpha_k d^k_{\nu_{gst}} \ \ \forall g \in G_s, s \in S, t \in T \tag{5.8}$$

L'étape suivante est donc de trouver un pas α_k assurant une convergence de l'algorithme.

5.2.2 Calcul du pas

Un pas de déplacement fortement utilisé en pratique, décrit par Fisher et Held notamment, repris par Papavasiliou, est décrit ainsi :

$$\alpha^k = \frac{\lambda(\hat{L} - L^k)}{\sum_{g \in G_s} \sum_{s \in S} \sum_{t \in T} (\pi^2_s \, (u^k_{gst} - w^k_{gt})^2 + \pi^2_s \, (v^k_{gst} - z^k_{gt})^2)} \tag{5.9}$$

où λ est un paramètre constant, $\hat{L}$ est une borne supérieure sur la solution optimale du primal et L^k est la somme des objectifs des sous-problèmes à l'itération k. Il reste donc le problème de trouver une borne supérieure raisonnable sur la valeur de la solution optimale, et que l'on puisse calculer rapidement. Il reste encore à calculer une borne supérieure, qui peut être trouvée notamment avec une solution primale faisable. Le problème est que trouver une solution faisable constitue déjà une difficulté. Puisque l'obtention d'une solution primale faisable concerne la tâche 4 du projet, nous avons décidé de **nous affranchir de ce problème dans le cadre de la tâche 3**. Nous n'avons donc pas pris en considération

l'idée de Papavasiliou consistant en l'obtention d'une solution faisable par résolution du problème de répartition économique (*economic dispatch*). Deux autres raisons à ce choix est qu'une telle solution ne peut être obtenue que vers les dernières itérations du sous-gradient (nous laissant donc sans borne supérieure durant la majorité du temps d'exécution), et que la résolution des différents sous-problèmes de répartition économique (un par scénario) ralentit le sous-gradient, malgré le fait que les différents sous-problèmes peuvent être résolus en parallèle. Pour cette tâche nous avons opté pour une alternative plus simple inspirée de (Zhuang and Galiana, 1988). La méthode a été montrée comme efficace sur le problème de *unit commitment* et calcule le pas de déplacement uniquement sur base de deux paramètres de contrôle et du numéro de l'itération courante. Dans l'article en question le pas est mis à jour de la façon suivante :

$$\alpha^k = \frac{1}{\alpha + \beta k} \ , \ \alpha, \beta > 0 \tag{5.10}$$

Dans l'implémentation de notre sous-gradient, nous avons plutôt utilisé la formule suivante :

$$\alpha_k = \alpha_0 \rho^k \tag{5.11}$$

où α_0 et ρ doivent être choisis suffisamment grands afin d'assurer la convergence de l'algorithme.

6 Tâche 4

Développer une heuristique basée sur la solution obtenue par la relaxation de la tâche précédente

Contrairement à ce qui a été suggéré dans l'énoncé et malgré l'utilité que peuvent avoir les multiplicateurs lagrangiens dans la recherche d'une solution primale admissible, nous avons choisi d'exploiter exclusivement l'information contenue dans l'historique des solutions primales non-admissibles obtenues avec la méthode du sous-gradient. En effet, beaucoup d'auteurs dont (Feltenmark and Kiwiel, 2000) et (Zhuang and Galiana, 1988) suggèrent de rapprocher d'avantage la solution courante de la zone admissible du primal en calculant une combinaison convexe des différentes solutions obtenues pour le primal.

6.1 Séquences ergodiques

Nous définissons une séquence ergodique comme étant une série $\{\overline{x}^k\}$ où l'élement $\overline{x}^k$ est calculé ainsi (Aldenvik and Schierscher, 2015) :

$$\overline{x}^k = \sum_{s=0}^{t-1} \mu_s^t x^s \ , \ \sum_{s=0}^{t-1} \mu_s^t = 1 \ , \ \mu_s^t \leq 0 \ , \ s = 0, \ldots, t-1 \tag{6.1}$$

où x^s est une solution primale infaisable obtenue à l'itération s de l'algorithme du sous-gradient et $\mu_s^t x^s$ est un coefficient de la combinaison convexe $\overline{x}^k$. La nouvelle solution primale obtenue à l'itération t est donc bien une combinaison convexe des solutions déjà obtenues.

Andrea Simonetto et Hadi Jamali-Rad (Simonetto and Jamali-Rad, 2016) fournissent une preuve de convergence forte de cette série. En revanche, la combinaison des variables binaires de notre problème ont de grandes chances de **générer des valeurs fractionnaires** pour celles-ci. Il a été montré que ce nombre de valeurs fractionnaires est fortement réduit lorsque le nombre d'unités dans le problème est significativement supérieur au nombre de périodes de temps. Cependant ceci n'est pas notre cas et il est nécessaire de recourir à une méthode de recherche locale afin de trouver une solution respectant les contraintes d'intégralité. La convergence forte n'est pas assurée mais cependant dans le cas d'un problème entier (binaire) mixte (ce qui est le cas), la série converge vers un point de l'enveloppe convexe de la zone admissible. Beaucoup d'auteurs interprètent les valeurs fractionnaires comme des probabilités d'occurence : u_{gst} peut par exemple être vu comme la probabilité que le générateur g soit actif lors de la période t dans le scénario s. Comme méthode de recherche locale, nous avons simplement utilisé **l'heuristique implémentée dans le cadre de la tâche 2** car nous l'avons justement spécialement conçue pour les solutions contenant des valeurs fractionnaires.

6.2 Bornes supérieures

L'algorithme tel qu'implémenté dans la tâche 3 a été modifié afin de calculer une borne supérieure sur base de solution primales admissibles. Cependant, au début de la méthode du sous-gradient, les solutions duales trouvées génèrent des solutions primales correspondantes fort éloignée de la zone admissible : il y a donc de plus grandes choses de voir l'heuristique échouer. Nous introduisons un paramètre

n_{ar} correspondant au nombre d'itérations du sous-gradient avant de commencer à utiliser l'heuristique pour reconstruire une solution du primal. Lorsqu'une première solution admissible est trouvée, la borne supérieure est abaissée à la valeur de l'objectif trouvé et le pas du sous-gradient est calculé sur base des bornes supérieures et inférieures. Tant qu'aucune solution admissible n'est disponible, le pas est calculé via la suite géométrique décrite à la fin de la tâche 3.

Calcul du pas sur base d'une borne supérieure

Une façon naïve de calculer une borne supérieure rapidement est de fixer les variables $u_{gst}, v_{gst}, p_{gst}$ $\forall g \in G, s \in S, t \in T$ à leurs valeurs maximales respectives. Il est assez clair que la solution correspondante devient infaisable et que seules les contraintes (3.25), (3.33), (3.34), (3.39), (3.40) sont garanties d'être satisfaites. Par exemple, certaines contraintes (3.36) cessent d'être satisfaites car il n'est pas possible d'allumer un générateur deux fois d'affilée. Cependant, la valeur de l'objectif constitue une borne supérieure car toute solution faisable nécessite d'abaisser la valeur d'une des variables $u_{gst}, v_{gst}, p_{gst}$ $\forall g \in G, s \in S, t \in T$ afin de satisfaire les contraintes restantes. En revanche cette borne est très mauvaise et **peut atteindre une valeur considérablement plus grande que l'objectif de la solution optimale**. Cette borne WUB est calculée ainsi :

$$WUB = \sum_{g \in G} \sum_{s \in S} \sum_{t \in T} \pi_s (K_g + S_g + C_g P^+_{gs}) \tag{6.2}$$

Cette borne supérieure a été utilisée lors de l'implémentation du sous-gradient de la tâche 3 et a été abandonnée au profit de la suite géométrique et des objectifs des solutions primales trouvées heuristiquement.

6.3 Vue générale de la méthode

Algorithm 2 Méthode du sous-gradient pour le problème SUC

procedure SOLVE_WITH_SUBGRADIENT
  $k \leftarrow 0$
  $UB \leftarrow WUB$, $LB \leftarrow -\infty$
  $\mu_{gst} \leftarrow 0$, $nu_{gst} \leftarrow 0$ $\forall g \in G, s \in S, t \in T$
  while $\frac{(UB-LB)}{UB} \leq \epsilon$ **do**
    Résoudre les sous-problèmes $P2$ et $P1_s$ $\forall s \in S$
    $L_k \leftarrow$ somme des objectifs des différents sous-problèmes
    if $L_k = LB$ **then**
      $\lambda \leftarrow \lambda/2$
    if $L_k > LB$ **then**
      $LB \leftarrow L_k$
    if $k > n_{ar}$ **then**
      Somme ergodique $\overline{x}^k$ de toutes les solutions primales infaisables obtenues
      Obtenir une solution primale admissible de manière heuristique depuis $\overline{x}^k$
    if $\hat{L} < UB$ **then**
      $UB \leftarrow \hat{L}$
    if $k > n_{ar}$ et une solution admissible a été trouvée **then**
      $\alpha_k = \frac{\lambda(\hat{L} - L^k)}{\sum_{g \in G_s} \sum_{s \in S} \sum_{t \in T} \left(\pi_s^2 \, (u^k_{gst} - w^k_{gt})^2 + \pi_s^2 \, (v^k_{gst} - z^k_{gt})^2\right)}$
    else
      $\alpha_k = \alpha_0 \, \rho^k$
    Mise à jour des multiplicateurs μ_{gst}, ν_{gst} $\forall g \in G, s \in S, t \in T$
    $k \leftarrow k + 1$

7 Tâche 5

Comparer les meilleures solutions obtenues (optimum et temps de résolution) avec les heuristiques, par rapport à celle trouvées avec la relaxation choisie

7.1 Convergence d'evolve-and-fix (tâche 2)

Comme expliqué dans la partie portant sur la tâche 2 du projet, l'heuristique proposée fonctionne principalement sur des instances de très petite taille et nécessite de fixer les variables de manière plus stratégique afin d'éviter de violer les contraintes liées aux démarrages des générateurs (de type 3.32 entre autres).

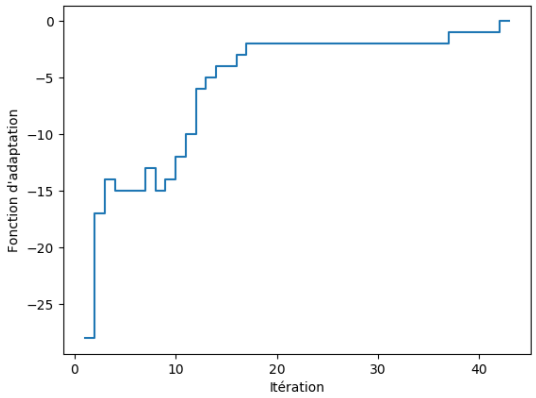


FIGURE 7.1 – Évolution de la fonction d'adaptation calculée par evolve-and-fix sur l'instance *inst-10-6-5-0.txt* - Méthode d'arrondi de la solution obtenue par la relaxation linéaire : Pour cette l'instance, le nombre de contraintes non satisfaites tombe rapidement à 0, permettant à l'heuristique de renvoyer une solution admissible.

Cependant, l'heuristique a l'avantage de ne posséder aucun paramètre si ce n'est ceux de l'algorithme génétique, qui quant à eux peuvent être laissés à leurs valeurs par défaut. En effet, les résultats obtenus sont fort robustes à l'ajustement des paramètres. Les valeurs par défaut sont les suivantes :

- Taille de la population : 100
- Nombre d'individus lors d'un tournoi : 2
- Taux de mutation : 2
- Nombre maximum d'itérations : 100000

7.2 Convergence du sous-gradient (tâches 3 et 4)

Il est important de noter qu'evolve-and-fix trouve une solution plus aisément lorsqu'appliqué à une solution proche de la zone admissible, en particuler la somme ergodique d'une (de préférence longue) série de solutions primales non-faisables.

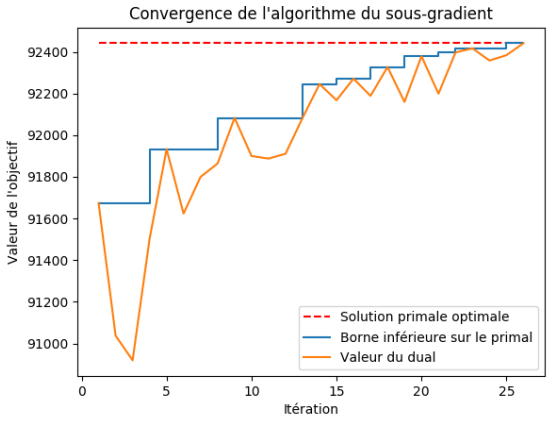


FIGURE 7.2 – Convergence du sous-gradient pour l'instance *inst-10-6-5-0.txt* : Pour cette instance, la solution optimale du primal a été retrouvée.

Sur les plus petites instances fournies, le temps requis pour trouver la solution optimale du problème d'origine (sans relaxation ni heuristique) est largement inférieur à celui pris par le sous-gradient pour converger et trouver la solution optimale. Ce phénomène s'estompt rapidement lorsque la taille du problème augmente (que ce soit par son nombre de scénarios, son nombre de générateurs ou son nombre de périodes de temps). Il y a donc intérêt à utiliser la décomposition lagrangienne lorsque le problème gagne en difficulté. De plus, les temps que nous avons mesuré ne tiennent pas compte de la nature parallèle de la décomposition.

En effet nous avons fait le choix de **ne pas simuler la distribution de l'algorithme du sous-gradient** car PuLP ne gère pas le parallélisme et est sujet à de nombreux bugs lorsqu'il est confronté au multiprocessing sur une seule machine. Le gain potentiel de vitesse qu'il reste possible d'obtenir est considérable et est d'au plus le nombre de scénarios du problème.

7.3 Résultats

Les deux tableaux qui suivent reprennent les solutions trouvées avec les différentes méthodes, ainsi que les temps d'exécution. *RL* fait référence à la relaxation linéaire : la valeur de l'objectif correspondant n'est donc pas celui du problème d'origine mais de la relaxation elle-même et est donc inférieure à la solution optimale. Les solutions optimales des problèmes d'origine n'ont pas été reprises dans les résultats car, bien que très rapides à calculer pour les plus petites instances, elles peuvent prendre un temps démentiel sur certaines des instances utilisées.

λRL fait référence à la relaxation lagrangienne : la valeur de l'objectif est donc celui de la solution primale obtenue de manière heuristique sur base de la dernière itération de la méthode du sous-gradient. La colonne *dual* donne à titre indicatif la dernière valeur obtenue pour le dual lagrangien. Notons qu'il ne s'agit pas de la borne inférieure du primal et donc pas de la meilleure valeur pour le dual. Comme il n'était malheureusement pas possible d'obtenir les résultats de toutes les instances dans un temps raisonnable avec l'approche de la décomposition lagrangienne, les résultats des instances non-traitées sont remplacés par le symbole "-". **Les valeurs sont indiquées en gras lorsque les solutions trouvées de manière heuristique sont optimales pour le primal**.

Les valeurs des paramètres utilisés dans l'algorithme de sous-gradient sont reprises à titre indicatif dans le tableau ci-dessous. Pour spécifier ces paramètres en ligne de commande, veuillez vous référer au début du rapport.

	λ	ϵ	α	ρ	nar
inst-10-6-5-x.txt	0.01	0.01	2000	0.96	25
inst-10-6-10-x.txt	0.01	0.01	5000	0.96	40

	RL		Méthode d'arrondi			λRL			
	Objectif	Temps(s)	Objectif	Temps(s)	Admissible	Objectif	Temps(s)	Dual	Saut
inst-10-6-5-0.txt	91174.75	0.33	94480.54	31.53	-	**92441.36**	78.55	**92441.36**	0%
inst-10-6-5-1.txt	90652.18	0.30	94580.37	30.45	oui	94751.17	111.31	91886.78	3.02%
inst-10-6-5-2.txt	89810.63	0.28	94891.98	5.42	-	-	-	-	-
inst-10-6-5-3.txt	91470.78	0.34	103855.17	30.70	-	**92685.26**	56.85	**92685.26**	0%
inst-10-6-5-4.txt	91547.35	0.34	101224.54	29.72	-	**92908.99**	28.64	**92908.99**	0%
inst-10-6-10-0.txt	90751.45	0.76	93219.45	17.31	oui	**91499.47**	27.36	**91499.47**	0%
inst-10-6-10-1.txt	89549.23	0.68	98493.06	48.37	-	**90476.24**	174.04	**90476.24**	0%
inst-10-6-10-2.txt	90286.37	0.66	95190.96	44.08	oui	**90934.95**	164.12	**90934.95**	0%
inst-10-6-10-3.txt	91200.85	0.53	95188.59	18.26	oui	**92549.49**	62.70	**92549.49**	0%
inst-10-6-10-4.txt	90726.53	0.67	94696.54	49.17	-	94693.54	192.91	92131.74	2.71%
inst-10-12-5-0.txt	236718.61	0.69	255901.57	37.68	-	-	-	-	-
inst-10-12-5-1.txt	234103.99	0.74	263560.52	27.64	-	-	-	-	-
inst-10-12-5-2.txt	243033.56	0.82	271783.36	42.32	-	-	-	-	-
inst-10-12-5-3.txt	235307.27	0.79	263751.59	38.17	-	-	-	-	-
inst-10-12-5-4.txt	232639.77	0.85	255517.90	23.88	-	-	-	-	-
inst-10-12-10-0.txt	239023.59	1.31	258728.97	70.95	-	-	-	-	-
inst-10-12-10-1.txt	234931.20	1.69	273970.32	96.84	-	-	-	-	-
inst-10-12-10-2.txt	235774.17	1.49	252770.10	90.19	-	-	-	-	-
inst-10-12-10-3.txt	241277.17	1.43	284073.02	80.20	-	-	-	-	-
inst-10-12-10-4.txt	235551.55	1.84	249891.31	93.06	-	-	-	-	-

	RL		Méthode d'arrondi			λRL			
	Objectif	Temps(s)	Objectif	Temps(s)	Admissible	Objectif	Temps(s)	Dual	Saut
inst-10-24-5-0.txt	478886.40	1.54	544541.99	72.62	-	-	-	-	-
inst-10-24-5-1.txt	475638.19	1.44	557913.85	111.23	-	-	-	-	-
inst-10-24-5-2.txt	469943.06	1.38	526371.57	95.51	-	-	-	-	-
inst-10-24-5-3.txt	473632.74	1.63	506284.01	101.82	-	-	-	-	-
inst-10-24-5-4.txt	476237.22	1.60	510716.24	79.76	-	-	-	-	-
inst-10-24-10-0.txt	490417.15	3.41	576754.35	164.86	-	-	-	-	-
inst-10-24-10-1.txt	477188.42	3.27	536734.65	183.04	-	-	-	-	-
inst-10-24-10-2.txt	475276.39	4.07	554725.74	227.55	-	-	-	-	-
inst-10-24-10-3.txt	482149.53	4.24	545631.74	236.90	-	-	-	-	-
inst-10-24-10-4.txt	475704.60	3.99	539894.73	239.95	-	-	-	-	-

References

1. Aldenvik, P. and Schierscher, M. (2015). Recovery of primal solutions from dual subgradient methods for mixed binary linear programming ; a branch-and-bound approach. Technical report.

2. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., and Smith, K. (2011). Cython : The best of both worlds. *Computing in Science Engineering*, 13(2) :31 –39.

3. Feltenmark, S. and Kiwiel, K. C. (2000). Dual applications of proximal bundle methods, including lagrangian relaxation of nonconvex problems. *SIAM Journal on Optimization*, 10(3) :697–721.

4. Mitchell, S., OSullivan, M., and Dunning, I. (2011). Pulp : a linear programming toolkit for python. the university of auckland, auckland, new zealand.

5. Papavasiliou, A. (2012). Coupling renewable energy supply with deferrable demand. Technical report.

6. Papavasiliou, A. and Aravena, I. (2016). An asynchronous distributed subgradient algorithm for solving stochastic unit commitment.

7. Papavasiliou, A. and Oren, S. S. (2013). A comparative study of stochastic unit commitment and security-constrained unit commitment using high performance computing. In *2013 European Control Conference (ECC)*, pages 2507–2512.

8. Simonetto, A. and Jamali-Rad, H. (2016). Primal recovery from consensus-based dual decomposition for distributed convex optimization. *Journal of Optimization Theory and Applications*, 168(1) :172–197.

9. Zhuang, F. and Galiana, F. D. (1988). Towards a more rigorous and practical unit commitment by lagrangian relaxation. *IEEE Transactions on Power Systems*, 3(2) :763–773.