[Rapport] Projet Julia

https://github.com/mathusanm6/ProjectJulia

Installation

1. Téléchargez le ZIP du projet sur GitHub
2. Dans la racine du projet, exécutez

```
julia --project=. -e 'using Pkg; Pkg.add(["JuMP", "HiGHS", "Test",
"JuliaFormatter"])'
```

Lancement du programme

- Pour faire les tests unitaires, exécutez

```
julia --project=. -e "using Pkg; Pkg.test()"
```

- Pour lancer sans faire les tests unitaires, exécutez

```
julia --project=. main.jl
```

Arborescence

```
ProjectJulia/
├── Manifest.toml    # Versions exactes des dépendances utilisées par le package
├── Project.toml     # Dépendances et métadonnées du projet
├── README.md
├── Rapport.pdf
├── main.jl          # Point d'entrée de l'application
├── src/             # Répertoire du code source
│   ├── ProjectJulia.jl  # Fichier du module principal
│   ├── examples.jl      # Implémentations d'exemples
│   ├── optimizer.jl     # Algorithme d'optimisation
│   ├── types.jl         # Définitions des types
│   └── validator.jl     # Fonctionnalité de validation des entrées
└── test/            # Répertoire des tests
    ├── runtests.jl           # Exécuteur principal des tests
    ├── validate_examples.jl  # Tests pour les exemples
    └── validate_optimizer.jl # Tests pour l'optimiseur
```

Nouvelles structures de données

```
mutable struct Port
    top::Bool
    right::Bool
    bottom::Bool
    left::Bool
end

mutable struct Cell
    entry::Port
    exit::Port
end

mutable struct Circuit
    grid::Array{Cell,2}
end

mutable struct Grid
    size::Int
    circuit::Circuit
    row_constraints::Vector{Int}
    column_constraints::Vector{Int}
end
```

Exemple de définition d'une grille valide

```
grid = Grid(2, Circuit(2), [2, 2], [2, 2])

grid.circuit.grid[1, 1] =
    Cell(Port(false, false, true, false), Port(false, true, false, false))
grid.circuit.grid[1, 2] =
    Cell(Port(false, false, false, true), Port(false, false, true, false))

grid.circuit.grid[2, 1] =
    Cell(Port(false, true, false, false), Port(true, false, false, false))
grid.circuit.grid[2, 2] =
    Cell(Port(true, false, false, false), Port(false, false, false, true))

# == Affichage == #
       2    2
     +---++---+
     |   ||   |
  2  | **||** |
     | * || * |
     +---++---+
     | * || * |
  2  | **||** |
     |   ||   |
     +---++---+
```

Variables de décision

- **entry_top[i, j]** : Variable binaire indiquant si la cellule à la position (i,j) possède une entrée par le haut.
- **entry_right[i, j]** : Variable binaire indiquant si la cellule à la position (i,j) possède une entrée par la droite.
- **entry_bottom[i, j]** : Variable binaire indiquant si la cellule à la position (i,j) possède une entrée par le bas.
- **entry_left[i, j]** : Variable binaire indiquant si la cellule à la position (i,j) possède une entrée par la gauche.

- **exit_top[i, j]** : Variable binaire indiquant si la cellule à la position (i,j) possède une sortie par le haut.
- **exit_right[i, j]** : Variable binaire indiquant si la cellule à la position (i,j) possède une sortie par la droite.
- **exit_bottom[i, j]** : Variable binaire indiquant si la cellule à la position (i,j) possède une sortie par le bas.
- **exit_left[i, j]** : Variable binaire indiquant si la cellule à la position (i,j) possède une sortie par la gauche.

Contraintes

1. **Contraintes de lignes** :
    - Chaque ligne doit contenir exactement le nombre spécifié de cellules non-vides (selon les contraintes de ligne).
2. **Contraintes de colonnes** :
    - Chaque colonne doit contenir exactement le nombre spécifié de cellules non-vides (selon les contraintes de colonne).
3. **Contraintes par cellule** :
    - Une cellule peut avoir au maximum une direction d'entrée.
    - Une cellule peut avoir au maximum une direction de sortie.
    - Une cellule ne peut pas avoir une entrée et une sortie dans la même direction.
    - Une cellule doit avoir un nombre égal de directions d'entrée et de sortie (0 ou 1).
4. **Contraintes de connectivité** :
    - Une sortie vers le haut de la cellule (i,j) doit correspondre à une entrée par le bas de la cellule (i-1,j).
    - Une sortie vers la droite de la cellule (i,j) doit correspondre à une entrée par la gauche de la cellule (i,j+1).
    - Une sortie vers le bas de la cellule (i,j) doit correspondre à une entrée par le haut de la cellule (i+1,j).
    - Une sortie vers la gauche de la cellule (i,j) doit correspondre à une entrée par la droite de la cellule (i,j-1).
    - Les cellules au bord de la grille ne peuvent pas avoir de sorties vers l'extérieur.
5. **Contraintes anti-boucles :**
    - Des contraintes supplémentaires sont ajoutées pour éliminer les sous-tours détectés.

Fonction objective

La fonction objective n’est pas importante car il s'agit principalement d'un problème de faisabilité plutôt que d'optimisation.

Méthode efficace pour valider les solutions proposées par l’optimiseur

Afin d’automatiser la validation des solutions obtenues à la fin d’exécution de l’optimiseur, j’ai défini des fonctions spécialisées dans le fichier `validator.jl` . La plus importante de ces fonctions est `check_valid_grid` qui est utilisée à plusieurs reprises lors des tests unitaires.

Un choix critique

Dans un premier temps, après avoir determiné les sous-tours, j’ai relancé l’optimisation avec de nouvelles contraintes interdisant toutes ces sous-tours, mais cela s’est révélé trop contraignant pour aboutir à une solution. Pour y remédier, j’ai opté pour un ajout progressif de contraintes, avec une nouvelle optimisation à chaque étape. La solution est atteinte lorsqu’il ne reste plus qu’un seul tour.