Construction de tables horaires ferroviaires et mesure de leurs robustesses

Lucas Tabary — MP*, Reims, lu.tabary@gmail.com

10 juin 2021

Résumé

Le réseau ferroviaire est aujourd'hui une option à privilégier afin de réduire les émissions carbonées dans le domaine des transports. Son organisation est encore largement manuelle [1]. Des recherches se concentrent déjà sur l'élaboration de méthodes informatiques permettant d'obtenir des tables horaires efficaces et fiables.

En se limitant à certains choix de modélisation, cette étude propose plusieurs approches pour construire de telles tables horaires. On détaille et compare ensuite plusieurs méthodes pour mesurer la fiabilité *(robustesse)* de ces réseaux.

1 Description du problème

Tout d'abord, indiquons les choix de modélisation faits dans la mise en place de ce problème :

- les trains étudiés parcourent, tous dans le même sens, une unique ligne rejoignant un nombre fini de gares ;
- chaque train a pour objectif de rejoindre deux gares situées sur cette ligne [1], et son voyage ne peut démarrer qu'à partir d'une certaine heure de départ minimale. Les trains peuvent faire escale ou non aux différentes gares intermédiaires ;
- les trains ne peuvent pas se dépasser et doivent toujours maintenir (sauf en gare) une distance de sécurité entre eux.

1.1 Formalisation

Par la suite, on ne fera pas la distinction entre les variables mathématiques du problème et les variables informatiques utilisées pour les représenter. Étant donnés un nombre `s` de gares et `t` de trains, on désigne par $s_0, \dots, s_{\mathtt{s}-1}$ les gares de la ligne (ainsi ordonnées) et $t_0, \dots, t_{\mathtt{t}-1}$ les trains en service. On appelle instance du TTP (pour *Train Timetabling Problem*) la donnée de :

- un tableau à `t` éléments, noté `types` dont les éléments sont les vitesses maximales de chaque train ;
- un tableau à `s` éléments, eux-mêmes des tableaux de `t` éléments indiquant le temps minimal à passer dans chaque gare pour chaque train, noté `gares` ;
- un tableau `trajets`, à `t` éléments, précisant les gares de départ et d'arrivée de chaque train ;
- un tableau `lignes` précisant pour chacun des `s` − 1 trajets (indexés par leur gare de départ) sa longueur et la distance minimale de sécurité à y respecter ;
- d'autres données relatives au calcul du coût d'une solution.

Une solution au TTP est alors la donnée de deux tableaux :

- un tableau `trajets` précisant les gares de départ et d'arrivée de chaque train (éventuellement différent de celui de la donnée du problème, selon que le train a été partiellement déprogrammé ou non) ;
- un tableau `horaires` à `s` − 1 éléments, qui sont des tableaux de `t` éléments indiquant pour chaque trajet, les temps de départ et d'arrivée

1. L'itinéraire des trains et le nombre de trains mis à disposition, dépendant du nombre d'usagers de chaque gare, de leur affluence, *etc.* est ici fixé à l'avance. La constitution d'une bonne disposition constitue un problème séparé, appelé *problème du routage*, que certains auteurs traitent parfois simultanément.

de chaque train. On le représente donc sous la forme d'une matrice.

Ces deux structures constituent les types enregistrement `probleme` et `solution` définis dans le fichier `TTP.ml` dont le contenu est présenté en annexe A.

Une solution à un TTP est donc un objet informatique de type `solution`, vérifiant que :

— aucun train ne roule plus vite que sa vitesse maximale autorisée ;
— aucun train ne part plus tôt que sa date de départ minimale ;
— sur chaque trajet entre deux gares, le temps qui sépare le trajet de deux trains est supérieur à la marge imposée.

La fonction `est_solution` présente dans le même fichier `TTP.ml`, dont la signature est donnée ci-dessous, vérifie toutes ces contraintes pour une solution donnée.

```
val est_solution : probleme -> solution
    -> bool
```

1.2 Représentation usuelle

La visualisation d'une solution à une instance du TTP peut se faire par un schéma comme celui de la figure 1. Cette représentation est un diagramme espace-temps, dans laquelle chaque train est représenté par une couleur distincte, et où la lecture des horaires de départ et d'arrivée d'un train au cours de son trajet se lisent en suivant de gauche à droite la ligne brisée. L'axe temporel n'est pas gradué car l'unité de temps choisie lors de la modélisation est arbitraire.

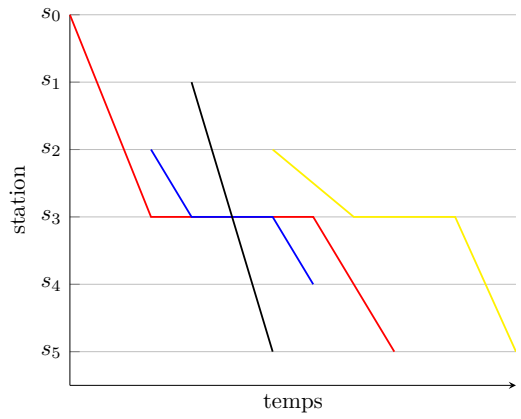

FIGURE 1 – Représentation usuelle pour un TTP constitué de 6 gares et 4 trains

2 Méthodes de résolution

Pour une instance `p` du TTP, on note $\mathcal{S}_\mathtt{p}$ l'ensemble des solutions valides associées. On définit alors une fonction de coût `cout p` ou $c_\mathtt{p}$ (décrite dans `TTP.ml`) sur $\mathcal{S}_\mathtt{p}$. Celle-ci mesure notamment le temps total passé par chacun des trains sur le réseau, et pénalise les solutions dans lesquelles des trains ont été partiellement ou totalement déprogrammés. On cherche alors une solution $\mathtt{x}^*$ optimale, c'est-à-dire telle que :

$$c_\mathtt{p}(\mathtt{x}^*) = \min_{\mathtt{x}\in\mathcal{S}_\mathtt{p}} c_\mathtt{p}(\mathtt{x}).$$

2.1 Optimisation linéaire

La recherche d'une solution minimale peut se faire en introduisant un autre formalisme pour ce problème, dans lequel une solution peut être vue comme un vecteur d'un espace de dimension le nombre de paramètres du problème. Toutes les inégalités qui caractérisent une solution pour un certain TTP sont linéaires. Considérons par exemple l'expression :

$$\forall t, \forall s,\ v_{\max,t}(\mathrm{arr}_{t,s} - \mathrm{dep}_{t,s}) \geq d_s \qquad (1)$$

où $v_{\max,t}$ est la vitesse maximale du t-ème train, $\mathrm{arr}_{t,s}$ et $\mathrm{dep}_{t,s}$ sont les temps de départ et d'arrivée du t-ème train sur le trajet au départ de la s-ème gare et d_s est la distance à parcourir sur ce trajet. Cette expression traduit le fait qu'aucun train ne peut dépasser sa vitesse maximale sur son trajet. De ces inégalités, il découle une expression de la forme :

$$\exists A, b,\ \mathcal{S}_\mathtt{p} = \{\, x \mid Ax \leq b \,\},$$

où l'inégalité est à comprendre terme à terme. Géométriquement, $\mathcal{S}_\mathtt{p}$ peut ainsi être interprété comme un simplexe convexe. On peut de plus faire en sorte que la fonction de coût soit une forme linéaire sur l'espace considéré. Dès lors, il est possible de mettre en place des méthodes de résolution comme celle de l'algorithme du simplexe, exploitant le fait qu'une solution minimale se situe sur un sommet du simplexe d'étude, afin d'obtenir une solution optimale en temps fini.

Depuis les travaux de A. CAPRARA et son équipe, on sait que la recherche d'une solution optimale au TTP, avec des choix de modélisation proches de ceux de cette étude, est un problème NP-complet [4]. Ainsi, quand B. EROL met en place différentes versions de l'algorithme du simplexe [5], celles-ci sont basées sur des méthodes dites de relaxation, qui permettent d'obtenir en un temps acceptable des « bonnes » solutions, non nécessairement optimales.

Certains outils permettent une mise en place efficace de ces différentes méthodes, comme la GLPK *(GNU Linear Programming Kit)* ; mais la présente étude ne couvre pas les détails de son utilisation.

2.2 Algorithme génétique

Les algorithmes génétiques permettent souvent d'obtenir de bons résultats pour une grande diversité de problèmes. Plusieurs équipes comme celle de D. Arenas proposent des approches fructueuses au TTP par cette méthode [3], mais il décrit, comme d'autres, relativement peu la démarche exacte pour mener cette résolution.

Rappelons le concept de cette méta-heuristique : il s'agit de faire évoluer un ensemble de solutions (appelé *population*) selon des principes inspirés de l'évolution génétique pour obtenir, après un certains nombres de générations, de « meilleures » solutions. Ces solutions doivent être caractérisées par certains éléments, appelés *gènes* de la solution, qui seront modifiés au cours des différentes étapes de l'algorithme.

Le type `solution` est inadapté à la mise en place d'un tel algorithme. On ne peut distinguer aisément de gènes pour cet objet : modifier les horaires d'une solution, les croiser avec ceux d'une autre, donne généralement un résultat qui n'est pas une solution valide. Pour pallier cette contrainte, on introduit le type `consignes`. La figure 2 juxtapose ces deux représentations. La consigne de la figure indique par exemple que, pour le trajet au départ de s_2, on a :

$$t_3 > t_2 > t_0 > t_1,$$

où $(>)$ traduit un ordre de priorité. Remarquons donc que chaque ligne de cette matrice doit correspondre à une permutation de $[\![0, \mathtt{t}-1]\!]$. Ces relations sont à la base d'une fonction `appliquer_consigne` qui détermine une solution à une instance de TTP à partir d'une consigne, de signature :

```
val appliquer_consigne : probleme ->
    consignes -> solution
```

On remarque alors que le type `consignes` est bien plus facile à manipuler dans le cadre d'un algorithme génétique. On procédera ainsi de la manière suivante, à chaque génération :

1. on reproduit les consignes deux à deux (2.2.2) ;
2. certaines consignes subissent des mutations (2.2.3) ;
3. on associe à chaque consigne sa solution (`appliquer_consigne`, 2.2.1), puis on sélectionne uniquement les consignes dont les solutions ont les meilleurs coûts.

L'implantation de toutes ces méthodes est faite dans le fichier `genetique.ml` (annexe B).

s_0	$(0,2)$	$(*,*)$	$(*,*)$	$(*,*)$
s_1	$(2,4)$	$(8,9)$	$(*,*)$	$(*,*)$
s_2	$(4,6)$	$(9,10)$	$(6,10)$	$(14,17)$
s_3	$(18,20)$	$(10,11)$	$(14,16)$	$(21,23)$
s_4	$(20,22)$	$(11,2)$	$(*,*)$	$(23,25)$

s_0	1	3	0	2
s_1	3	1	2	0
s_2	3	2	0	1
s_3	0	1	2	3
s_4	0	2	1	3

Figure 2 – Table horaire du trajet de la figure 1 (au dessus), table de priorité d'une consigne (en dessous)

2.2.1 Principe de `appliquer_consigne`

On construit successivement les parcours des trains au départ de la première gare, puis de la deuxième, *etc.* Sur chaque trajet, on ajoute le parcours de chaque train dans l'ordre de la priorité ainsi définie par la consigne. Dans l'exemple de la figure 2, sur le trajet au départ de s_2, on place d'abord le train t_3, puis t_2, et ainsi de suite. La figure 3 représente une partie du déroulement de cet algorithme.

On appelle espace de parcours toute zone (délimitée sur la figure par des lignes bleues hachurées) dans laquelle un train s'y ajoutant ne violera aucune contrainte de distance minimale avec un autre train. La liste de tous les espaces de parcours est modifiée à chaque ajout d'un nouveau train sur le parcours. Chaque train est placé dans le premier espace de parcours possible. Si aucun n'est possible, il est déprogrammé à partir de ce trajet.

Par construction, la solution retournée par cet algorithme est nécessairement une solution valide. Faire suffisamment varier les différentes priorités sur chaque trajet est essentiel pour obtenir un grand nombre de solutions, parmi lesquelles l'algorithme génétique pourra faire sa recherche.

2.2.2 Reproduction de consignes

La reproduction dans un algorithme génétique est le plus souvent effectuée à l'aide de *crossover operators*, c'est-à-dire un découpage puis une recombinaison des caractéristiques de deux parents pour former deux descendants. La figure 4 donne un exemple adapté ici à notre problème.

On choisit donc ici, de manière aléatoire pour chaque reproduction, un numéro de trajet tel que les

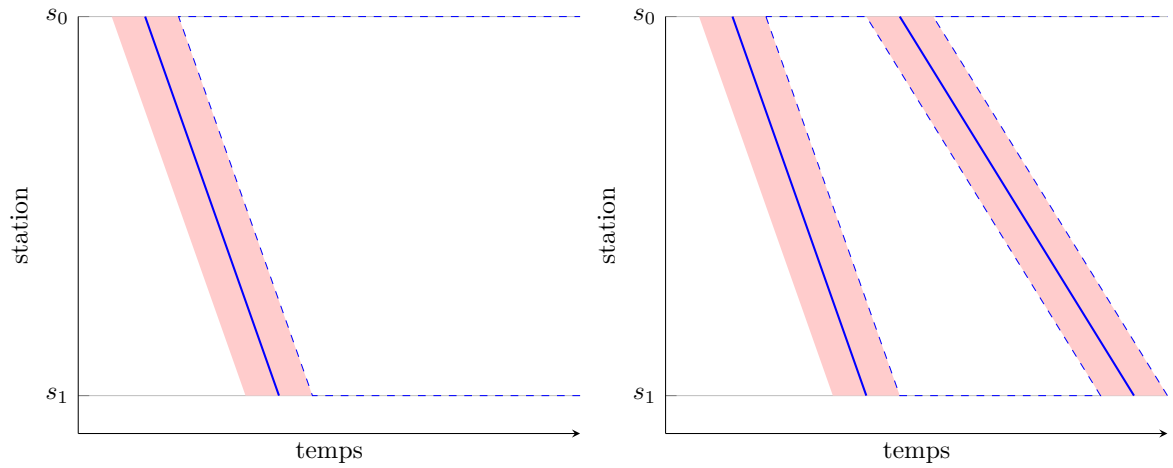

FIGURE 3 – à gauche : le premier train est placé et délimite l'espace des parcours possibles ; à droite : le second train est placé selon ses contraintes et délimite à nouveau l'espace des parcours.

priorités des trajets qui le précèdent chez le premier parent seront attribuées au premier descendant, la seconde moitié étant complétée par l'autre parent ; et le second descendant est constitué avec le complémentaire.

Cette méthode introduit une certaine diversité génétique, mais elle n'est pas suffisante : aucune diversité dans les priorités n'est introduite sur chaque trajet autrement que par la diversité de la population initiale.

s_0	1	3	0	4	2
s_1	3	1	4	0	2
s_2	3	2	0	1	4
s_3	0	1	2	3	4
s_4	0	2	4	3	1
s_5	0	2	4	3	1

s_0	2	1	3	4	0
s_1	0	2	4	1	3
s_2	2	1	0	3	4
s_3	1	0	3	2	4
s_4	2	1	4	3	0
s_5	3	2	4	0	1

s_0	1	3	0	4	2
s_1	3	1	4	0	2
s_2	2	1	0	3	4
s_3	1	0	3	2	4
s_4	2	1	4	3	0
s_5	3	2	4	0	1

s_0	2	1	3	4	0
s_1	0	2	4	1	3
s_2	3	2	0	1	4
s_3	0	1	2	3	4
s_4	0	2	4	3	1
s_5	0	2	4	3	1

FIGURE 4 – Exemple de reproduction de deux consignes parentes (au-dessus)

2.2.3 Mutation d'une consigne

Pour assurer une diversité dans les priorités d'un même trajet à travers la population, l'opération de mutation effectue un mélange sur les éléments d'une même ligne, sur une certaine part de la population — à noter que la probabilité de muter est en pratique plus élevée que dans d'autres implantations d'algorithmes génétiques. Un exemple est donné en figure 5. Plutôt que de remplacer une permutation par une autre totalement distincte, on préfère effectuer un nombre borné (par un paramètre) de transpositions, afin de ne pas trop s'éloigner de solutions qui seraient déjà intéressantes.

s_0	1	3	0	4	2
s_2	3	1	4	0	2
s_3	3	2	0	1	4
s_3	0	1	2	3	4
s_4	0	2	4	3	1
s_5	0	2	4	3	1

$\longmapsto$

s_0	1	3	0	4	2
s_2	3	1	4	0	2
s_3	2	1	0	3	4
s_3	0	1	2	3	4
s_4	0	2	4	3	1
s_5	0	2	4	3	1

FIGURE 5 – Exemple de mutation d'une consigne (deux transpositions appliquées)

3 Mesures de la robustesse

On sait qu'en situation réelle, l'organisation des parcours des trains sur un réseau est très sensible : par exemple, des retards de l'ordre de la minute à Paris peuvent avoir des conséquences sur des trains en gare de Lyon. Partant de constats similaires, des études plus récentes s'intéressent à la mesure de la fiabilité des tables horaires ferroviaires [5, 2]. On souhaite donc mesurer, pour une solution à une instance du TTP donnée, l'impact qu'ont des retards d'une durée raisonnable sur la qualité (c'est-à-dire, ici, le coût) de la solution, une fois adaptée à ce retard.

Disposer d'une telle mesure peut par exemple servir à comparer différents algorithmes de construction de

tables horaires comme ceux exposés précédemment, pour déterminer quelles méthodes ou quelles modifications d'un même algorithme fourniraient des solutions plus robustes en moyenne.

Dans la suite, on présente plusieurs méthodes de calcul de cette robustesse :

— La mesure par déviation est une transcription de la description de la robustesse qu'on a pu faire précédemment ; elles est détaillée dans la partie 3.1. Cette méthode est coûteuse mais fiable par définition.
— Le dénombrement des points critiques est une méthode motivée par le travail de A. Anderson et son équipe [2], détaillée en 3.2. Cette méthode est moins coûteuse, et on cherche à justifier sa pertinence en 3.3.

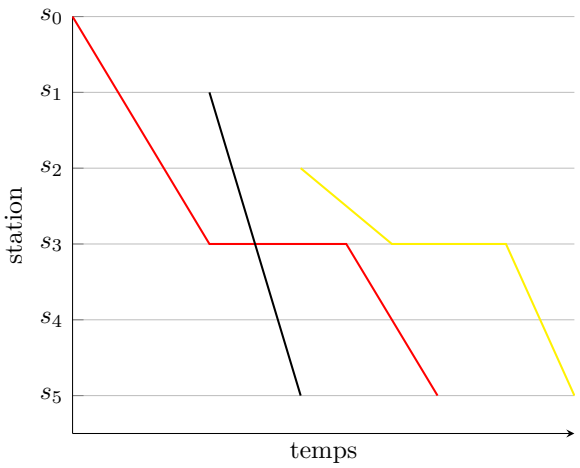

Figure 6 – Table horaire corrigée après retard de t_0

3.1 Mesure par déviation

Soit une solution $\mathtt{x} \in S_\mathtt{p}$ et $n \in \mathbb{N}^*$. On construit un ensemble de `solutions` perturbées associés à $\mathtt{x}$, noté $W_\mathtt{x}$, de cardinal n. Ces solutions perturbées ne sont donc plus des solutions valides. On note alors χ une fonction dite *correctrice*, qui associe à tout élément de $W_\mathtt{x}$ une nouvelle solution valide, dite solution corrigée. On note $k(\mathtt{s},\mathtt{t})$ la complexité moyenne de calcul de $\chi(x)$ pour $x \in W_\mathtt{x}$. Alors, on appelle n-fragilité de $\mathtt{x}$ pour cette ensemble de perturbation la quantité :

$$F_n(\mathtt{x}) := \frac{1}{n} \sum_{x \in W_\mathtt{x}} (c_\mathtt{p}(\mathtt{x}) - c_\mathtt{p}(\chi(x)))^2,$$

qui s'interprète donc comme une « variance » du coût des solutions sur l'ensemble des perturbations considérées. Plusieurs méthodes de correction sont possibles. Ici on procède de la manière suivante :

1. on applique le retard à un train choisi aléatoirement ;
2. on répercute le retard du train sur ses trajets ultérieurs. Si le train a du temps d'arrêt en gare en excès ou bien ne roule pas à vitesse maximale sur un trajet, on utilise ce temps pour rattraper le retard ;
3. si la nouvelle disposition du train entre en conflit avec un trajet déjà existant, on déprogramme le train le moins prioritaire des deux à partir de ce trajet.

En reprenant la table de la figure 1 et en ajoutant un retard au train t_0 en rouge pour son arrivée en gare s_3, on obtient, après modification par l'algorithme ici décrit, la table représentée par la figure 6.

L'implantation de cette méthode de calcul est présentée dans le fichier `robustesse.ml` (annexe C). La complexité temporelle au pire de cette implantation est en $O(n \cdot \mathtt{s} \cdot \mathtt{t} \cdot k(\mathtt{s},\mathtt{t}))$. En pratique, la valeur de n doit être grande ; de plus, des méthodes de correction plus sophistiquées peuvent entraîner des valeurs de $k(\mathtt{s},\mathtt{t})$ plus élevées. La complexité attendue pour cette méthode est donc significative.

3.2 Dénombrement des points critiques

Comme défini par A. Anderson, un point critique correspond à un départ de deux trains d'une gare dans un intervalle de temps suffisamment proche. Par exemple, toujours sur la figure 1, le départ des trains t_0 et t_2, respectivement rouges et bleus, de la gare s_2 se fait dans un intervalle de temps réduit.

Une telle proximité entraîne une dépendance entre ces trains, et les fluctuations de l'emploi du temps d'un train peuvent avoir des conséquences sur l'autre. On cherche à quantifier cette intuition par l'étude présentée en 3.3.

L'implantation de cette méthode est présentée en annexe C. Sa complexité temporelle au pire est en $O\big(\mathtt{st}^2 \log \mathtt{t}\big)$, ce qui reste intéressant.

3.3 Comparaison des mesure

Pour comparer ces mesures, on génère par l'algorithme génétique présenté en 2.2 un grand nombre de solutions pour des instances du TTP construites arbitrairement. Pour chacune de ces solutions, on calcule par les méthodes décrites précédemment leur 100-fragilité et leur nombre de points critiques. Le code de

génération des instances du TTP et celui de compilation des résultats sont fournis en annexes E et C, respectivement.

Les résultats sont présentés en figure 7.

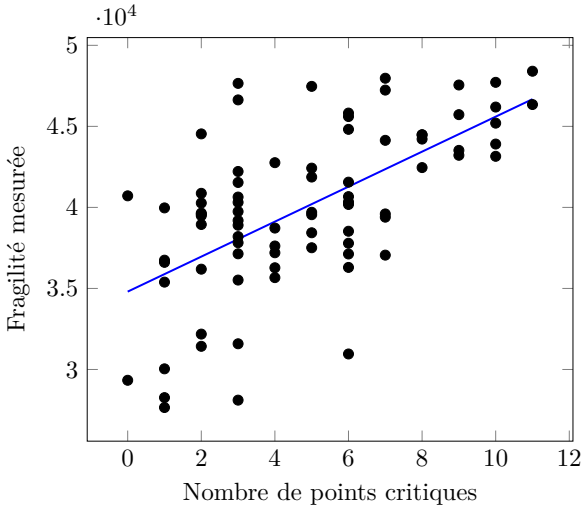

FIGURE 7 – Recherche d'une corrélation entre les deux mesures ($r = 0,62$)

Conclusion et perspectives

Les méthodes de construction de tables horaires ferroviaires proposées dans cette étude fonctionnent dans le cadre de modélisation ici fixé. Même s'il n'a ici pas été fait d'étude des performances de ces algorithmes, le fait qu'ils soient aussi utilisés pour la construction de tables dans le cadre d'une modélisation plus précise renforce leur crédibilité.

Les conclusions à tirer de la comparaison des mesures de robustesse faites en section 3 ne sont pas immédiates. Le coefficient de corrélation entre ces mesures est relativement faible, et même s'il suggère que cette piste n'est pas dénuée de sens, il reste insuffisant pour affirmer que la mesure des points critiques pourrait remplacer le calcul de la n-fragilité dans le cadre d'une étude de comparaison entre différentes méthodes de résolution du TTP, par exemple.

Une approche pour améliorer ce résultat serait de proposer d'autres paramètres peu coûteux et mesurés aussi simplement que les points critiques, afin d'étudier une corrélation entre la n-fragilité et *plusieurs* indicateurs.

Au long de cette étude, on a choisi de s'intéresser au cas d'une seule ligne de train. A. CAPRARA justifie ce choix et montre qu'on peut déjà en tirer suffisamment de conclusions si l'on désirait généraliser ces méthodes à des cas plus complexes (réseau maillé). Toutefois, pour la robustesse, aucune généralisation n'est *a priori* envisageable.

Finalement, on pourrait envisager de compléter le cadre de modélisation de ce problème, en ajoutant des caractéristiques dont on sait qu'elles influencent la nature des solutions : voies ferrées doubles ou à deux sens, dépassements de trains, solutions cycliques, *etc.*

Références

[1] Comment fixe-t-on les horaires de trains en France ? *Sciences et Avenir*, Feb. 2012.

[2] E. V. Andersson, A. Peterson, and J. Törnquist Krasemann. Quantifying railway timetable robustness in critical points. *Journal of Rail Transport Planning & Management*, 3(3):95–110, 2013. Robust Rescheduling and Capacity Use.

[3] D. Arenas, R. Chevrier, S. Hanafi, and J. Rodriguez. Solving the Train Timetabling Problem, a mathematical model and a genetic algorithm solution approach. In *6th International Conference on Railway Operations Modelling and Analysis (RailTokyo2015)*, Tokyo, Japan, Mar. 2015.

[4] A. Caprara, M. Fischetti, and P. Toth. Modeling and solving the train timetabling problem. *Operations Research*, 50(5):851–861, 2002.

[5] B. Erol. *Models for the Train Timetabling Problem.* Theses, Institut für Mathematik der Technischen Universität Berlin, Aug. 2009.

Annexes

A Définition du problème — `TTP.ml`

```
(* Représentation d'un TTP *
 * s est le nombre de gares, t le nombre de trains. c_arret et c_marche sont
 * les coups à l'arrêt et en marche des trains (ne dépendent donc pas des trains)
 * trajets indique la gare de départ et d'arrivée de chaque train. Le dernier trajet
 * est donc le numéro de la dernière station, moins 1. *)
type probleme = {
  nom: string;
  s: int; t: int; c_arret: int; c_marche: int;
  k: int; (* coefficient pour calcul du coût *)
  priorites: int array;
  tmax: int; (* durée maximale d'une solution *)
  departs_initiaux: int array; (* temps minimaux de départ des trains *)
  trajets: (int * int) array; (* deb, fin *)
  gares: int array array; (* temps_mini pour chaque train dans chaque gare *)
  lignes: (int * int) array; (* distance minimale entre deux trains, longueur *)
  types: int array (* vitesse maximale *)
}


(* Représentation d'une solution au TTP *
 * trajets indique pour chaque train les premières et dernières gares du trajet.
 * Si celles-ci sont égales cela signifie que le train n'est pas programmé.
 * Les valeurs de horaires.(i) "situées" avant la gare initiale ou après la
 * gare finale sont arbitraires. *)
type solution = {
  nom: string;
  trajets: (int * int) array;
  horaires: (int * int) array array
}


(* Vérifie qu'un problème est sémantiquement correct *)
let est_probleme {s; t; trajets; gares; lignes; c_marche; c_arret} =
  Array.length trajets = t && Array.length gares = s &&
  Array.length lignes = s - 1 && c_marche >= 0 && c_arret >= 0 &&
  Array.length gares = s && Array.length gares.(0) = t


(* Vérifie qu'une solution à un TTP donné est sémantiquement correcte *)
let est_solution {s; t; trajets; departs_initiaux; gares; lignes; types; tmax}
    {trajets=trajets'; horaires} =
  (* vérifie que les trajets ne se font pas plus loin que prévu initialement *)
  let est_trajet_reel (deb, fin) (deb', fin') =
    deb = deb' && deb' <= fin' && fin' <= fin
  in
  (* vérifie que chaque train ne roule pas plus vite que possible *)
  let est_trajet_bonne_vitesse s' (deb, fin) vmax (dep, arr) =
    let longueur = snd lignes.(s') in
    (not (deb <= s' && s' < fin )) || (longueur <= (vmax * (arr - dep)) && arr <
        tmax)
  in
  (* vérifie qu'aucun train n'est trop proche d'un autre (nécessite un tri
     préalable) *)
  let est_parcours_ligne_correct (marge, longueur) portion =
```

```
52      let aux (valid, dep, arr) (dep', arr') =
53        (valid && (dep' - dep >= marge) && (arr' - arr >= marge), dep', arr')
54      and fst (a, b, c) = a in
55      (* On aimerait pouvoir utiliser - infini à la place de -marge ci-dessous, car
            cela correspond à une
56       * inégalité toujours vérifiée, mais min_int ne convient pas non plus (donne
            résultat aberrant) *)
57      fst (List.fold_left aux (true, -marge, -marge) portion)
58    in
59    let est_trajet_partie_route n_trajet n_train =
60      let (deb, fin) = trajets'.(n_train) in deb <= n_trajet && n_trajet < fin
61    in
62    let est_partie_bonne_heure n_train =
63      let (deb, fin) = trajets.(n_train) in
64      (deb = fin) || fst horaires.(deb).(n_train) >= departs_initiaux.(n_train)
65    in
66    let b = ref true in
67    (* partie des vérifications qui se font sur une liste non triée *)
68    Array.length horaires = s - 1 && Array.length horaires.(0) = t &&
69    Array.length trajets = t &&
70    Array.for_all2 est_trajet_reel trajets trajets' && (
71    for n_train = 0 to t - 1 do
72      b := !b && est_partie_bonne_heure n_train
73    done;
74    for n_station = 0 to s - 2 do
75      b := !b && Arraymod.for_all3 (est_trajet_bonne_vitesse n_station) trajets' types
            horaires.(n_station)
76    done; !b) &&
77    (* partie des vérifications qui se font sur une liste triée *)
78    (let nh = Array.mapi (fun i x -> Arraymod.liste_filtre (est_trajet_partie_route i)
          x) horaires in
79    Array.for_all2 est_parcours_ligne_correct lignes nh)
80
81
82  (* Détermine le coût d'une solution à un TTP donné, en la supposant correcte *
83   * La manière dont le calcul est mené est peu naturelle car la structure de donnée
84   * est ici inadaptée: il faudrait ici avoir 'horaires' définie comme la transposée
85   * de la définition actuelle. *)
86  let cout {trajets; priorites; c_arret; c_marche; k} {trajets=trajets'; horaires} =
87    let cout_trajet n_train (deb, fin) =
88      let t_marche = ref 0 in (* Peu élégant mais plus rapide *)
89      for i = deb to fin - 1 do
90        t_marche := !t_marche + (fun (x, y) -> y - x) horaires.(i).(n_train)
91      done;
92      let t_tot = (snd horaires.(fin - 1).(n_train)) - (fst horaires.(deb).(n_train))
            in
93      c_marche * !t_marche + c_arret * (t_tot - !t_marche)
94    in
95    let penalite_trajet_incomplet (deb, fin) (deb', fin') p =
96      p * (fin' - fin) / (fin - deb)
97    in
98    (* On pénalise pour chaque train non programmé, en fonction de leur priorité *)
99    Arraymod.sum (Arraymod.map3 penalite_trajet_incomplet trajets trajets' priorites)
100   (* On détermine le temps d'activité de chaque train, en marche et à l'arrêt *)
101   + k * Arraymod.sum (Array.mapi cout_trajet trajets)
```

B Algorithme génétique — `genetique.ml`

```
1  open TTP
2
3  (* Initialisation du module aléatoire *)
4  let _ = Random.self_init ()
5
6
7  (* D'une consigne on peut déduire une unique table horaire, à l'aide de
       appliquer_consigne
8   * priorites = [|2; 4; 5; ..|] si 2 est prioritaire devant 4, prioritaire devant 5,
       etc.
9   * vmax.(i).(t) est la vitesse maximale de t sur le trajet i (entre la gare i et
       i+1) *)
10 type consignes = {
11   nom: string; (* nom de la solution *)
12   priorites: int array array;
13   vmax: int array array;
14   mutable cout: int; (* cout de la solution associé *)
15   mutable est_solution_calculee: bool
16 }
17
18
19 (* Ensemble de consignes qui évolue à chaque étape de l'algorithme génétique *)
20 type population = {
21   mutable n: int;
22   nmax: int;
23   pop: consignes array
24 }
25
26
27 (* Consigne par défaut, remplit les cases de population non utilisées *)
28 let defaut = {
29   nom = "defaut";
30   priorites = [|[|0|]|];
31   vmax = [|[|0|]|];
32   cout = max_int;
33   est_solution_calculee = true
34 }
35
36
37 (* Construit de manière déterministe une solution à partir d'un TTP et d'une consigne
38  * comme détaillée dans le rapport, la solution obtenue est par construction une
39  * solution valide, que l'on n'a pas besoin de tester. *)
40 let appliquer_consigne {s; t; trajets; gares; departs_initiaux; lignes; tmax} {nom;
       priorites; vmax} =
41   let trajets' = Array.copy trajets and departs_min = Array.copy departs_initiaux in
42   let horaires = Array.make_matrix (s - 1) t (0, 0) in
43   (* ajouter_train ajoute un train sur le trajet au départ de la gare n_trajet et
       renvoie
44    * la liste des espaces de parcours encore disponibles après ajout éventuel du
       train *)
45   let rec ajouter_train n_trajet l n_train =
46     let (deb, fin) = trajets'.(n_train) and (marge, dist) = lignes.(n_trajet) in
         match l with
47       (* on n'essaie pas d'ajouter des trains hors de leur période d'activité *)
48     | l when not (deb <= n_trajet && n_trajet < fin) -> l
49     | [] -> let (deb, _) = trajets'.(n_train) in
50       trajets'.(n_train) <- (deb, n_trajet); []
51     | (depmin, depmax, arrmin, arrmax) :: q ->
52         let dep = max depmin departs_min.(n_train) in
```

```
53          let arr = dep + dist / vmax.(n_trajet).(n_train) in
54          if dep <= depmax && arrmin <= arr && arr <= arrmax
55            then begin
56              let t_min = gares.(n_trajet + 1).(n_train) in
57              horaires.(n_trajet).(n_train) <- (dep, arr);
58              departs_min.(n_train) <- arr + t_min;
59              if dep - marge <= depmin || arr - marge <= arrmin (* plus de place à
                    gauche *)
60                then (dep + marge, depmax, arr + marge, arrmax) :: q
61                else if dep + marge >= depmax || arr + marge >= arrmax (* plus de
                       place à droite *)
62                  then (depmin, dep - marge, arrmin, arr - marge) :: q
63                  else (depmin, dep - marge, arrmin, arr - marge) ::
64                       (dep + marge, depmax, arr + marge, arrmax) :: q
65            end
66            else (depmin, depmax, arrmin, arrmax) :: ajouter_train n_trajet q n_train
67    in
68    for s' = 0 to s - 2 do
69      (* ajoute successivement chaque train sur le trajet au départ de la gare s' *)
70      ignore (Array.fold_left (ajouter_train s') [(0, tmax, 0, tmax)] priorites.(s'))
71    done; {nom; trajets=trajets'; horaires}
72
73
74  (* Étant donnée une consigne, fournit une autre consigne, qui a subi une
75   * mutation des priorités pour au plus chaque trajet. *)
76  let une_mutation mut_par_ligne {nom; priorites; vmax} =
77    let priorites' = Arraymod.copym priorites and vmax' = Arraymod.copym vmax in
78    let permutation a =
79      let n = Array.length a in
80      let i = Random.int n and j = Random.int n in
81      let t = a.(i) in
82      a.(i) <- a.(j); a.(j) <- t
83    in
84    let choix = Random.int (Array.length priorites) in
85    for i = 0 to Random.int mut_par_ligne do
86      permutation priorites'.(choix)
87    done;
88    {nom = nom ^ "m"; priorites = priorites'; vmax = vmax';
89     cout = max_int; est_solution_calculee = false}
90
91
92  let une_descendance {nom=nom1; priorites=priorites1; vmax=vmax1}
93                      {nom=nom2; priorites=priorites2; vmax=vmax2} =
94    let n = Array.length priorites1 in
95    let coupure = Random.int n in
96    let priorites1' = Array.init n (fun i -> Array.copy (if i < coupure
97      then priorites1.(i) else priorites2.(i)))
98    and priorites2' = Array.init n (fun i -> Array.copy (if i >= coupure
99      then priorites1.(i) else priorites2.(i)))
100   and vmax1' = Array.init n (fun i -> Array.copy (if i < coupure
101     then vmax1.(i) else vmax2.(i)))
102   and vmax2' = Array.init n (fun i -> Array.copy (if i >= coupure
103     then vmax1.(i) else vmax2.(i)))
104   in
105   ({nom = nom1 ^ "g"; priorites = priorites1'; vmax = vmax1'; cout = max_int;
          est_solution_calculee = false},
106    {nom = nom2 ^ "g"; priorites = priorites2'; vmax = vmax2'; cout = max_int;
          est_solution_calculee = false})
107
```

```
108
109 (* Mélange de Knuth, renvoie une permutation au hasard à n éléments *)
110 let melange_p n =
111   let a = Array.init n (fun i -> i) in
112   for i = n - 1 downto 1 do
113     let j = Random.int (i + 1) in
114     let t = a.(i) in
115     a.(i) <- a.(j); a.(j) <- t
116   done; a
117
118
119 (* former des paires d'éléments tous distincts entre 0 et n - 1 *)
120 let former_paires n =
121   let a = melange_p n in
122   let rec aux i =
123     if i < n - 1
124       then (a.(i), a.(i + 1)) :: aux (i + 2)
125       else []
126   in aux 0
127
128
129 let generer_solutions_initiales (pop_init, rho, mut) {nom; s; t; priorites; types} =
130   let p = Arraymod.indices_tries priorites in
131   let priorites' = Array.init (s - 1) (fun i -> Array.copy p)
132   and vmax = Array.init (s - 1) (fun i -> Array.copy types) in
133   let pop_init_copie = int_of_float (rho *. (float_of_int pop_init))
134   and pop = Array.make (4 * pop_init) defaut
135   and copie = {
136     nom = nom ^ "-g";
137     priorites = priorites';
138     vmax;
139     cout = max_int;
140     est_solution_calculee = false
141   } in
142   for i = 0 to pop_init_copie - 1 do
143     pop.(i) <- copie
144   done;
145   for i = pop_init_copie to pop_init - 1 do
146     pop.(i) <- une_mutation mut copie
147   done;
148   {n = pop_init; nmax = 4 * pop_init; pop}
149
150
151 let cout_consignes probleme consignes =
152   if consignes.est_solution_calculee
153     then consignes.cout
154     else begin
155       let c = cout probleme (appliquer_consigne probleme consignes) in
156       consignes.est_solution_calculee <- true;
157       consignes.cout <- c; c
158     end
159
160
161 let compare_consignes probleme c1 c2 =
162   let cc1 = cout_consignes probleme c1 and cc2 = cout_consignes probleme c2 in
163   compare cc1 cc2
164
165
166 (* Sélectionne, après tri, les rang_max meilleurs individus, et remplace le reste
```

```
 * par des solutions par défaut *)
let selection rang_max probleme population =
  Array.sort (compare_consignes probleme) population.pop;
  population.n <- min population.n rang_max;
  for i = population.n to population.nmax - 1 do
    population.pop.(i) <- defaut
  done


(* avec une certaine probabilité, modifie les solutions aléatoirement *)
let mutation (p_mut, mut_par_ligne) population =
  let nv = ref 0 and nn = population.n in
  for i = 0 to nn - 1 do
    if Random.float 1. <= p_mut then begin
      population.pop.(nn + !nv) <- une_mutation mut_par_ligne population.pop.(i);
      incr nv
    end
  done; population.n <- nn + !nv


(* Ajoute de nouveaux individus en croisant les traits des individus présents *)
let reproduction population =
  let nn = population.n in
  let rec ajouter_descendance = function
    | [] -> ()
    | (p, m) :: r ->
        let papa = population.pop.(p) and mama = population.pop.(m) in
        let (alice, bob) = une_descendance papa mama in
        population.pop.(nn) <- alice;
        population.pop.(nn + 1) <- bob;
        population.n <- nn + 2
  in ajouter_descendance (former_paires nn)

(* Fait évoluer un ensemble de solutions selon le concept d'algorithme génétique.
 * Les paramètres sont:
 * nb_gen: nombre de générations (itérations)
 * pop_init: population initiale
 * rho: part de la population initiale qui utilise la méthode de copie
 * p_mut: probabilité pour un individu de muter pour en donner un autre
 * mut_par_ligne: nombre de transpositions maximal pouvant s'appliquer à
 *                une liste de priorité pendant une mutation *)
let genetique (nb_gen, pop_init, rho, p_mut, mut_par_ligne) probleme =
  let p = generer_solutions_initiales (pop_init, rho, 3 * mut_par_ligne) probleme in
  for i = 1 to nb_gen do begin
    reproduction p;
    mutation (p_mut, mut_par_ligne) p;
    selection pop_init probleme p
  end done; p
```

C Mesures de robustesse — `robustesse.ml`

```
open TTP

let _ = Random.self_init ()


(* Associe à toute solution une n-fragilité
```

```
 * tc est le temps caractéristique des perturbations *)
let fragilite n (probleme, {trajets; horaires}) =
  (* plus grande des marges du problème: *)
  let tc = fst (Array.fold_left max (min_int, min_int) probleme.lignes) in
  let c = cout probleme {nom=""; trajets; horaires} in
  let t = probleme.t in
  let f = ref 0 in
  for i = 1 to n do
    let n_train_retard = Random.int t in
    let (deb, fin) = trajets.(n_train_retard) in
    if fin > deb then begin
      let n_trajet_retarde = deb + (Random.int (fin - deb)) in
      let vmax = probleme.types.(n_train_retard) in
      let trajets' = Array.copy trajets
      and horaires' = Arraymod.copym horaires in

      let (depi, arri) = horaires.(n_trajet_retarde).(n_train_retard) in
      horaires'.(n_trajet_retarde).(n_train_retard) <- (depi, arri + tc);

      let rec ajouter_retard n_trajet = if n_trajet < fin then begin
        let (marge, dist) = probleme.lignes.(n_trajet_retarde) in
        let (_, arrp) = horaires'.(n_trajet - 1).(n_train_retard)
        and (dep, arr) = horaires.(n_trajet).(n_train_retard)
        and tmin = probleme.gares.(n_trajet).(n_train_retard) in
        if dep - arrp < tmin then begin (* le retard n'est pas amorti par le temps
            d'arrêt en gare *)
          let dep' = arr + tmin in
          (* on essaie de rattraper le retard dans le cas où le train n'était pas
           * à vitesse maximale: *)
          let arr' = if vmax * (arr - dep') >= dist then arr else dep' + dist / vmax
              in

          let est_train_retarde_prioritaire n_autre_train (depo, arro) =
            if (n_autre_train <> n_train_retard) && (abs (depo - dep') < marge ||
                abs (arro - arr') < marge) then
              if probleme.priorites.(n_autre_train) <
                  probleme.priorites.(n_train_retard) then
                let (debo, fino) = trajets'.(n_autre_train) in
                trajets'.(n_autre_train) <- (debo, n_trajet); true
              else false
            else true
          in
          if Arraymod.for_alli est_train_retarde_prioritaire horaires.(n_trajet)
              then begin
            horaires'.(n_trajet).(n_train_retard) <- (dep', arr');
            ajouter_retard (n_trajet + 1)
          end
        end
      end in ajouter_retard (n_trajet_retarde + 1);
      let c' = cout probleme {nom=""; trajets=trajets'; horaires=horaires'} in
      f := !f + abs (c - c')
    end
  done; (float_of_int !f) /. (float_of_int n)


(* Associe à chaque solution son nombre de points critiques
 * tc est l'intervalle de temps dans lequel on cherche des paires
 * de points dits critiques *)
let npc (probleme, {trajets; horaires}) =
```

```
61    (* plus grande des marges du problème: *)
62    let tc = fst (Array.fold_left max (min_int, min_int) probleme.lignes) in
63    let est_trajet_partie_route n_trajet n_train =
64      let (deb, fin) = trajets.(n_train) in deb <= n_trajet && n_trajet < fin
65    in
66    let horaires_triees = Array.mapi (fun i x -> Arraymod.liste_filtre
          (est_trajet_partie_route i) x) horaires in
67    let rec compter_npc = function
68      | [] | [_] -> 0
69      | (dep1, arr1) :: (dep2, arr2) :: q -> (if dep2 - dep1 < tc then 1 else 0)
70        + (compter_npc ((dep2, arr2) :: q))
71    in
72    Array.fold_left (+) 0 (Array.map compter_npc horaires_triees)
73
74
75  (* Détermine deux tableaux (np-fragilité et npc)  pour un tableau de
76   * solutions (jointes avec leurs problèmes), les enregistre à
77   * l'emplacement 'fichier', puis effectue une régression linéaire
78   * pour obtenir le coefficient de corrélation. *)
79  let construire_relation fichier np a_solutions =
80    let a_fragilite = Array.map (fragilite np) a_solutions
81    and a_npc = Array.map (fun a -> float_of_int (npc a)) a_solutions in
82    let n = Array.length a_npc in
83    let oc = open_out fichier in
84    for i = 0 to n - 1 do
85      Printf.fprintf oc "%f␣%f\n" a_npc.(i) a_fragilite.(i)
86    done; close_out oc;
87    (* Calculs usuels pour une régression linéaire: *)
88    let covfn = Arraymod.cov a_fragilite a_npc
89    and varf = Arraymod.var a_fragilite
90    and varn = Arraymod.var a_npc
91    and moyf = Arraymod.moy a_fragilite
92    and moyn = Arraymod.moy a_npc
93    in
94    let rr = (covfn *. covfn) /. (varf *. varn) in
95    let a = covfn /. varn in
96    let b = moyf -. a *. moyn in
97    print_string ("r^2␣=␣" ^ (string_of_float rr) ^ ",␣a␣=␣" ^
98                  (string_of_float a) ^ ",␣b␣=␣" ^ (string_of_float b));
99    (rr, a, b)
```

D Extension du module `Array` — `TTP.ml`

```
1   open Array
2
3   (* Extension du module Array
4    * La structure des fonctions est inspirée du code source
5    * originale des fonctions qui leur ressemblent (map2 pour
6    * map3 par exemple). *)
7
8   let map3 f a b c =
9     let la = length a in
10    let lb = length b in
11    let lc = length c in
12    if la <> lb || lb <> lc then
13      invalid_arg "Arraymod.map3:␣arrays␣must␣have␣the␣same␣length"
14    else begin
```

```
15      if la = 0 then [||] else begin
16        let r = make la (f (unsafe_get a 0) (unsafe_get b 0) (unsafe_get c 0)) in
17        for i = 1 to la - 1 do
18          unsafe_set r i (f (unsafe_get a i) (unsafe_get b i) (unsafe_get c i))
19        done;
20      r
21      end
22      end
23
24  let for_alli p l =
25    let n = length l in
26    let rec loop i =
27      if i = n then true
28      else if p i (unsafe_get l i) then loop (succ i)
29      else false in
30    loop 0
31
32  let for_all3 p l1 l2 l3 =
33    let n1 = length l1
34    and n2 = length l2
35    and n3 = length l3 in
36    if n1 <> n2 || n2 <> n3 then invalid_arg "Arraymod.for_all3"
37    else let rec loop i =
38      if i = n1 then true
39    else if p (unsafe_get l1 i) (unsafe_get l2 i) (unsafe_get l3 i) then loop (succ i)
40      else false in
41    loop 0
42
43  let sum = fold_left (+) 0
44  let sum' = fold_left (+.) 0.
45
46  (* moyenne d'un tableau de flottants *)
47  let moy a =
48    let n = length a in
49    (sum' a) /. (float_of_int n)
50
51  (* covariance de deux tableaux de flottants *)
52  let cov a b =
53    let aa = moy a and bb = moy b in
54    let ab = Array.map2 (fun x y -> (x -. aa) *. (y -. bb)) a b in
55    moy ab
56
57  let var a = cov a a
58
59  (* renvoie la liste des indices associés au plus grandes valeurs,
60   * dans l'ordre décroissant *)
61  let indices_tries a =
62    let a' = mapi (fun i e -> (e, i)) a in
63    let compare' x y = - (compare x y) in
64    sort compare' a'; map (fun (e, i) -> i) a'
65
66  (* copie "en profondeur" un tableau de tableaux *)
67  let copym (a: 'a array array) =
68    Array.init (Array.length a) (fun i -> Array.copy a.(i))
69
70  (* conserve dans une liste les éléments d'un tableau
71   * dont les indices vérifient un prédicat *)
72  let liste_filtre predicat a =
73    let n = Array.length a in
```

```
74    let rec aux i = if i < n
75      then if predicat i
76        then a.(i) :: aux (i + 1)
77        else aux (i + 1)
78      else []
79    in List.sort compare (aux 0) (* la liste renvoyée est triée *)
```

E Génération des problèmes et exécution des tests — `test.ml`

```
1  #load "TTP.cma"
2  open TTP
3
4  (* Enregistre une solution au format .dat pour être utilisée ensuite
5   * par solution.tex pour en obtenir une représentation graphique *)
6  let solution_vers_dat {nom; trajets; horaires} =
7    let oc = open_out ("./résultats/" ^ nom ^ ".dat") in
8    let valeur_formatee choix n_trajet n_train (dep, arr) =
9      let deb, fin = trajets.(n_train) in
10     if deb <= n_trajet && n_trajet < fin then
11       string_of_int (choix (dep, arr))
12     else
13       "nan"
14   in
15   let chaine_par_station n_trajet trajet =
16     let n = string_of_int n_trajet and n' = string_of_int (n_trajet + 1) in
17     let u = n  ^ "␣" ^ (String.concat "␣" (Array.to_list (Array.mapi
         (valeur_formatee fst n_trajet) trajet)))
18     and v = n' ^ "␣" ^ (String.concat "␣" (Array.to_list (Array.mapi
         (valeur_formatee snd n_trajet) trajet)))
19     in
20     Printf.fprintf oc "%s\n%s\n" u v
21   in
22   Array.iteri chaine_par_station horaires; close_out oc
23
24
25
26 (*************************************************************)
27 (* Définition des problèmes d'étude, et de solutions d'exemple *)
28 (*************************************************************)
29
30 let solution_test = {
31   nom = "solution_test";
32   horaires = [| [|(0, 2); (1, 4); (3, 5)|];
33                 [|(3, 5); (7, 9); (6, 8)|] |];
34   trajets = [|(0, 2); (0, 2); (0, 2)|]
35 }
36
37 let p1 = { (* aucune interaction entre les trains *)
38   nom = "problème_1";
39   s = 5; t = 4;
40   c_arret = 10; c_marche = 50;
41   k = 100; tmax = 80;
42   priorites = [|500; 200; 300; 100|];
43   departs_initiaux = [|0; 20; 0; 30|];
44   trajets = [|(1, 4); (0, 4); (1, 3); (0, 2)|];
45   gares = [|[|0; 0; 0; 0|];
46             [|0; 0; 0; 10|];
```

```
47              [|10; 20; 10; 0|];
48              [|0; 0; 0; 0|];
49              [|0; 0; 0; 0|]|];
50    lignes = [|(3, 50); (5, 20); (5, 25); (3, 40)|];
51    types = [|5; 5; 5; 5|];
52  }
53
54
55  let rac n = int_of_float (sqrt (float_of_int n))
56  (* Loi normale simulée avec la méthode de Box-Muller *)
57  let loi_normale esp ec =
58      esp +. ec *. sqrt (-. log (Random.float 1.)) *. (cos (2. *. Float.pi *.
            (Random.float 1.)))
59  let loi_normale_discrete min max esp ec =
60    let v = int_of_float (loi_normale (float_of_int esp) ec) in
61    if v < min then min
62    else if v > max then max
63    else v
64
65  (* La méthode de génération qui suit est entièrement empirique *)
66  let generer_probleme nom s t tmax =
67    let priorites = Array.init t (fun i -> 100 * (1 + i + (Random.int (1 + rac i)))) in
68    let departs_initiaux = Array.init t (fun i -> loi_normale_discrete 0 tmax (tmax /
          t) ((float_of_int tmax) /. 10.)) in
69    let trajets = Array.init t (fun i ->
70      let deb = Random.int (s - 1) / 2 in let fin = (s - 1) / 2 + (Random.int ((s - 1)
          / 2))
71      in (deb, fin)) in
72    let gares = Array.init s (fun n_trajet ->
73      Array.init t (fun n_train -> let (deb, fin) = trajets.(n_train) in
74        if deb <= n_trajet && n_trajet < fin
75          then if Random.float 1. > 0.2 then t + (Random.int t) else 0
76          else 0
77        )) in
78    let lignes = Array.init (s - 1) (fun i -> (loi_normale_discrete 6 20 10 1.,
          loi_normale_discrete 20 80 40 10.)) in
79    let types = Array.init t (fun i -> loi_normale_discrete 3 6 5 0.5) in
80    {nom; s; t; priorites; departs_initiaux; trajets; gares; lignes; types;
81     c_arret = 10; c_marche = 50; k = 100; tmax}
82
83
84  let gen1 = Genetique.genetique (100, 50, 0.3, 0.1, 3)
85
86  let correlation n m =
87    let p0 = generer_probleme "correlation" 15 10 100 in
88    let sol0 = gen1 p0 in
89    let a = Array.make n (p0, Genetique.appliquer_consigne p0 sol0.pop.(0)) in
90    for i = 1 to n - 1 do
91      let p = generer_probleme "correlation" 15 10 100 in
92      let sol = gen1 p in
93      a.(i) <- (p, Genetique.appliquer_consigne p sol.pop.(0))
94    done;
95    Robustesse.construire_relation "./résultats/correlation.dat" m a;;
96
97  correlation 80 100
```