DISPOSITIF DIDACTIQUE DÉBRANCHÉ POUR APPRENDRE À RÉSOUDRE DES PROBLÈMES ALGORITHMIQUES NUMÉRIQUES ET À LES PROGRAMMER

Christian BLANVILLAIN[1]
Haute École Pédagogique du canton de Vaud, Suisse
Département des sciences de l'éducation, Université de Patras, Grèce

Résumé. Présentation d'un dispositif didactique débranché pour apprendre à programmer dans un langage très simple, avec peu d'instructions et proche de l'assembleur. Le code est représenté par des instructions en bois que l'on assemble. L'élève trace les états de la mémoire à la main tout en lisant son code pas à pas, en s'aidant d'un papier et d'un crayon, ce qui lui permet de se constituer rapidement une représentation mentale de la machine notionnelle associée. L'alternance de moments de programmation en binômes et de moments de réflexion en grand groupes permet de développer simultanément les compétences en conception d'algorithmes et l'aptitude à analyser les stratégies cognitives mobilisées dans l'acte de penser les algorithmes. Le témoignage des élèves nous montre que le dispositif leur a appris non seulement à programmer, mais aussi à raisonner.

Mots-clés. Algorithmes numériques, Programmation débranchée, Stratégies cognitives.

Abstract. This paper introduces an unplugged didactic device designed to learn to program in a very simple language, with few instructions and close to assembly language. The code is represented by wooden instructions that are assembled. The student traces the states of memory by hand while reading their code step by step, using paper and a pencil, which allows them to quickly construct a mental representation of the associated notional machine. The alternation of moments of pair programming and moments of reflection in large groups allows to simultaneously develop algorithm design skills and the ability to analyze the cognitive strategies used in algorithmic thinking. Student testimonials show that the device has taught them not only to program, but also to reason.

Keywords. Numerical algorithms, Unplugged programming, Cognitive strategies.

Resumen. Este artículo presenta un dispositivo didáctico desconectado diseñado para aprender a programar en un lenguaje muy simple, con pocas instrucciones y cercano al lenguaje ensamblador. El código se representado con instrucciones de madera que se ensamblan. El estudiante rastrea los estados de la memoria a mano mientras lee su código paso a paso, utilizando papel y lápiz, lo que le permite construir rápidamente una representación mental de la máquina nocional asociada. La alternancia de momentos de programación en parejas y momentos de reflexión en grupos grandes permite el desarrollo simultáneo de habilidades de diseño de algoritmos y la capacidad de analizar las estrategias cognitivas utilizadas en el pensamiento algorítmico. Los testimonios de los estudiantes muestran que el dispositivo no sólo les ha enseñado a programar, sino también a razonar.

Palabras clave. Algoritmos numéricos, Programación desenchufada, Estrategias cognitivas.

[1] blanvillain@gmail.com

Introduction

Pour acquérir un regard critique et éclairé sur les technologies numériques, pour éviter de devenir dépendants d'une technologie opaque, pour comprendre et savoir mieux interagir avec les intelligences artificielles, nous considérons essentiel l'éducation des élèves au numérique et à la science informatique. Cependant, la croissance exponentielle des indicateurs du changement climatique questionne la durabilité écologique de l'introduction de l'éducation numérique dans les écoles du canton de Vaud en Suisse. Cette inquiétude est renforcée par la décision de collaborer avec les géants du numérique, connus pour leur modèle propriétaire et leur écosystème fermé. Leur système nécessite des mises à jour fréquentes, ce qui a pour effet de rendre rapidement le matériel obsolète. Comment l'école peut-elle limiter son impact environnemental tout en formant les élèves à l'informatique et tout particulièrement aux concepts fondamentaux de la programmation, alors qu'elle se repose entièrement sur le choix d'un matériel rarement évolutif, difficilement réparable, et sur le choix d'applications et de logiciels non libres ? Un premier pas serait de promouvoir des options open source, capables de redonner vie à des équipements d'occasion, adéquats pour les besoins éducatifs et l'enseignement de l'informatique. Ces alternatives permettraient de soutenir les valeurs du permacomputing tout en éduquant les élèves à ces solutions viables pour une économie durable et respectueuse de l'environnement. Retarder le moment où l'on introduit les écrans à l'école tout en permettant aux élèves d'apprendre à programmer, permettrait également de rendre l'éducation numérique un peu plus écologiquement responsable. Mais peut-on apprendre à programmer sans ordinateurs ? C'est la réponse que nous allons apporter dans cet article.

En tant qu'enseignant-informaticiens, nous avons observé dans nos classes certains élèves modifier leur code sans réfléchir, jusqu'à ce que "ça marche", en enchaînant les exercices sans véritablement chercher à comprendre la logique de leurs programmes. La réponse instantanée offerte par la machine les incite à modifier le code et à le tester à nouveau avant même d'avoir fait l'effort de se poser la question de ce qui se passe (Chevalier, 2022, voir résultats pp. 86-89). Utilisé de cette manière, l'outil informatique ne favorise pas l'apprentissage. Ce qui est paradoxal étant donné qu'un ordinateur capable d'interpréter le code pour nous aider à identifier les erreurs est un outil d'apprentissage autonome formidable. Peut-être qu'en début de formation, un dispositif didactique débranché serait plus approprié pour ces élèves qui modifient leur code aléatoirement sans chercher à comprendre leur travail ? Cela les aiderait à se construire une représentation mentale de leur code en le lisant et en l'interprétant dans leur tête.

Nous formulons l'hypothèse qu'un tel dispositif, combiné à une approche axée sur le développement des stratégies cognitives, permettrait aux élèves débutants de développer leur pensée informatique plus efficacement qu'avec une approche traditionnelle basée sur l'usage d'un écran. Nous considérons que l'enseignement de la pensée informatique en classe est une opportunité unique pour enseigner aux élèves comment résoudre des problèmes et développer leur créativité. C'est pourquoi nous avons conçu un dispositif didactique débranché basé sur un langage de programmation proche de l'assembleur et de l'architecture du processeur, dans le but d'expliciter de manière pertinente les stratégies cognitives utiles pour apprendre à penser les algorithmes et résoudre un problème donné. Notre hypothèse est qu'avec ce dispositif et notre approche pédagogique, il est possible d'aider les élèves à développer leur intelligence.

Lorsque nous avons commencé ces travaux en 2018, nous ne savions pas à partir de quel âge nous réussirions à enseigner aux élèves débutants comment penser des algorithmes numériques pour les programmer, tout en leur faisant prendre conscience des processus et stratégies cognitives mobilisés. Ainsi, nous l'avons testé avec des élèves d'âges variés. Pour les 10 – 12

ans, la métacognition s'est avérée être difficile. Même si la plupart ont résolu les problèmes algorithmiques posés, nous n'avons pas réussi à faire émerger du groupe classe des stratégies utiles pour les élèves en difficulté. Chez les 12 – 14 ans, la prise en main du dispositif fut plus aisée, mais nous n'avons hélas pas pu le tester suffisamment longtemps pour savoir si nous arrivions à aider les élèves à développer leurs stratégies de résolution de problème. En revanche, avec les 15 – 17 ans, nous avons observé une excellente adhésion au dispositif et à l'identification de nouvelles stratégies. Des résultats tout aussi positifs ont été obtenus avec les 20 – 24 ans. C'est cette première expérimentation, menée en 2019, que nous avons choisi de vous présenter ici, enrichie des derniers résultats obtenus en 2025 avec des élèves de 17 ans.

1. Méthode

Pour concevoir notre dispositif, nous nous sommes inspirés d'un jeu intitulé "*human resource machine*" à architecture Harvard, créé par Allan Blomquist, Kyle Gabler et Kyle Gray en 2015 (société Tomorrow Corporation). Ce jeu est lui-même inspiré de "*Little Man Computer*", un ordinateur à vocation basé sur une architecture de von Neumann, créé par le Dr Stuart Madnick en 1965. Le "petit homme" représente l'accumulateur qui se déplace au sein du processeur (ou chez nous, entre les lignes de code). Le domaine des problèmes posés est celui des algorithmes numériques. Un flux de nombres entiers arrive en entrée et le "petit homme" doit effectuer les calculs nécessaires pour résoudre le problème numérique proposé, en s'aidant d'une zone mémoire et d'opérations élémentaires telles que l'addition et la soustraction. Lorsque le résultat final est généré, le "petit homme" le dépose sur le flux de sortie. Les instructions que l'on écrit décrivent les actions du "petit homme" dans un langage très proche de l'assembleur. Eliasz (2016, chap. 3.4) donne des exemples d'implémentation des instructions "*if…else…*", "*while…*", "*do...while*" dans le langage assembleur du jeu "*Little Man Computer*".

Nous avons utilisé un processus de recherche orienté conception, qui consiste à procéder par une mise au point du dispositif au travers d'itérations successives durant lesquelles le dispositif est testé en classe. Quatre versions se sont ainsi succédé avant d'aboutir au dispositif qui vous est présenté ici. Par souci de concision, nous présentons le dispositif dans sa forme la plus aboutie sans revenir sur les différences entre les versions ni sur toutes les améliorations conceptuelles et ergonomiques apportées. Nous avons testé ce dispositif sur une durée d'un semestre avec des étudiants débutants en informatique, qui suivaient des cours d'algorithmique et de programmation en langage Python à raison de huit périodes de 45 minutes par semaine conjointement à notre cours optionnel hebdomadaire sur une seule période. Nous évaluons l'efficacité de notre démarche de deux manières : en observant si les étudiants réussissent à résoudre les problèmes posés de semaine en semaine et en les questionnant en fin de formation pour leur demander ce qu'ils ont pensé des activités proposées et des éventuels apports que ces exercices ont eus sur leur maîtrise de l'algorithmique et de la programmation dans le grand cours traditionnel.

2. Expérimentation

2.1. Dispositif didactique

Nous avons repris le langage assembleur du Dr Stuart Madnick et l'avons simplifié (Blanvillain, 2020a). Conceptuellement, l'ensemble du langage peut être réduit à deux groupes d'instructions donnant lieu à des opérations tangibles facilement compréhensibles. Chaque instruction se présente sous la forme d'une plaquette en bois découpée au laser, sur laquelle est gravée une représentation visuelle de l'instruction avec le strict minimum de texte. Avec un langage

composé principalement de deux groupes d'instructions à comprendre, la prise en main du dispositif est particulièrement rapide, d'autant plus que nous commençons par une activité totalement débranchée sans dispositifs, où chaque élève joue le rôle d'un des composants du processeur, mettant ainsi les concepts en acte. Un groupe de quatre instructions courtes servent à recopier des valeurs. Elles ont des flèches qui indiquent le sens de recopiage de la valeur. Elles peuvent soit remplacer la valeur précédente, soit effectuer une opération arithmétique d'addition ou de soustraction avec la valeur existante. Un groupe de trois instructions longues servent à effectuer des sauts dans la séquence d'instruction du code. Ces instructions longues vont par paires de plaquettes identifiées à l'aide d'un motif ayant la même couleur et le même dessin pour les élèves atteints de daltonisme partiel ou total. Dans les photos qui suivent, nous avons utilisé une version non colorée des pièces pour une meilleure lisibilité avec le rendu en niveaux de gris qui nous est imposé. Une des plaquettes indique le saut proprement dit et l'autre plaquette indique la ligne où l'on doit sauter dans le code. Les sauts peuvent être inconditionnels ou bien soumis à une condition, si la valeur de l'accumulateur est nulle ou bien si la valeur de l'accumulateur est négative. Ce sont ces instructions qui sont utilisées pour fabriquer des tests et des boucles. Les couples de plaquettes sont uniques et comptent comme une seule instruction lorsqu'on évalue la longueur du code écrit.

Le programme se commence et se termine par un demi-cercle ce qui est surtout utile pour que l'enseignant qui navigue d'une table à l'autre puisse identifier rapidement où se trouve le code en cours d'écriture sur la table. L'accumulateur est représenté par un disque en plexiglas transparent que l'on déplace par-dessus les instructions représentant le code et sur lequel on peut écrire la valeur en cours de traitement à l'aide d'un feutre effaçable. Lorsque l'élève trace les états de la mémoire à la main tout en lisant son code pas à pas, il sert à identifier le point d'arrêt dans le code.

Une feuille de papier plastifiée sur laquelle l'élève indique l'état de chaque mémoire, est utilisée pour que l'élève interprète son code pas à pas et identifie d'éventuelles erreurs dans son programme. À la fin de l'interprétation manuelle de son code, en comparant les sorties effectives avec les sorties attendues fournies en exemple sur les fiches d'exercices, l'élève peut s'assurer que tout fonctionne correctement pour ces valeurs d'exemple ou prendre conscience de la ligne de code qui pose problème et, grâce à la trace produite, identifier plus facilement la correction à apporter à son programme. L'élaboration manuelle de l'historique des différents états de la mémoire a une très grande valeur pédagogique. C'est un artefact didactique puissant pour aider les élèves à se constituer une image mentale de leur code et de la machine notionnelle du dispositif, ce qui leur permet assez rapidement d'interpréter le code qu'ils écrivent uniquement dans leur tête.

Le dispositif ne propose que trois mémoires nommées *X*, *Y*, *Z*, représentées par des petits rectangles en bois. Les exercices proposés ont été choisis pour pouvoir être résolus avec uniquement ces trois mémoires, ce qui apporte une contrainte supplémentaire sur l'espace des solutions possibles. Les données arrivent par un flux entrant identifié par un point d'interrogation et les résultats des calculs sont envoyés sur un flux sortant identifié par un point d'exclamation. Les entrées et sorties sont aussi des petits rectangles en bois. Pour construire des compteurs ou bien simplement pour initialiser le calcul d'une somme, nous avons également besoin de deux constantes : zéro et un. Tous ces rectangles en bois (mémoires, entrées/sorties, constantes) sont appelés "paramètres" et s'insèrent dans des espaces prévus à cet effet dans la partie gauche des pièces courtes. Nous les avons découpés dans un bois un peu plus épais que les instructions pour faciliter leur manipulation.

Les élèves disposent d'un cahier d'exercices numérotés en binaire qui expliquent ce qu'il faut faire. Chaque exercice est accompagné d'un ensemble de valeurs d'entrée et de valeurs de sortie représentant un jeu de test. Souvent, un challenge indique quel est le nombre minimal d'instructions à utiliser pour écrire le code solution. Ces challenges sont surtout utiles pour amener les élèves les plus rapides à faire un raisonnement plus poussé d'optimisation de leur code.

Vous pouvez voir en figure 1 et 2 deux photos du dispositif didactique, avec l'énoncé à gauche, le code solution au centre et la feuille utilisée pour tracer à la main l'évolution des états des mémoires à droite. Ce ne sont pas des exemples triviaux. Ils se situent vers la fin de la formation. Nous les avons choisis, car ils utilisent toutes les pièces et instructions existantes, ce qui vous permet d'apprécier la simplicité du langage. Nous conseillons au lecteur qui trouverait ces exemples trop complexes de se référer au livret 1 du manuel enseignant où chaque instruction est introduite de manière progressive au fil des exercices.

Les deux livrets pour l'enseignant ainsi que le cahier d'exercices sont partagés sous licence Creative Commons (CC BY-NC-SA). Les plans des pièces du dispositif sont également distribués librement et sont téléchargeables sur le site https://human-processor.xyz, ce qui fait que vous pouvez les découper vous-même dans du papier ou dans du bois si vous avez accès à une découpeuse laser. Vous pouvez aussi faire acheter par votre école une machine à découper le carton dont le prix est équivalent à celui d'une tablette et qui permet ensuite d'offrir un ordinateur en carton à tous les élèves. C'est une solution particulièrement intéressante pour les écoles soucieuses de l'impact environnemental du numérique ou pour les pays émergents. Enfin, si le temps vous manque, vous pouvez aussi tout simplement commander la découpe des pièces sur un site en ligne.

Nous avons envisagé de constituer une version du jeu pour personnes malvoyantes. Les pièces peuvent être gravées en relief ce qui permet d'écrire du braille. Nous n'avons pas cherché à rencontrer des élèves malvoyants pour tester cette version du dispositif. Mais c'est une idée que nous conservons pour plus tard.

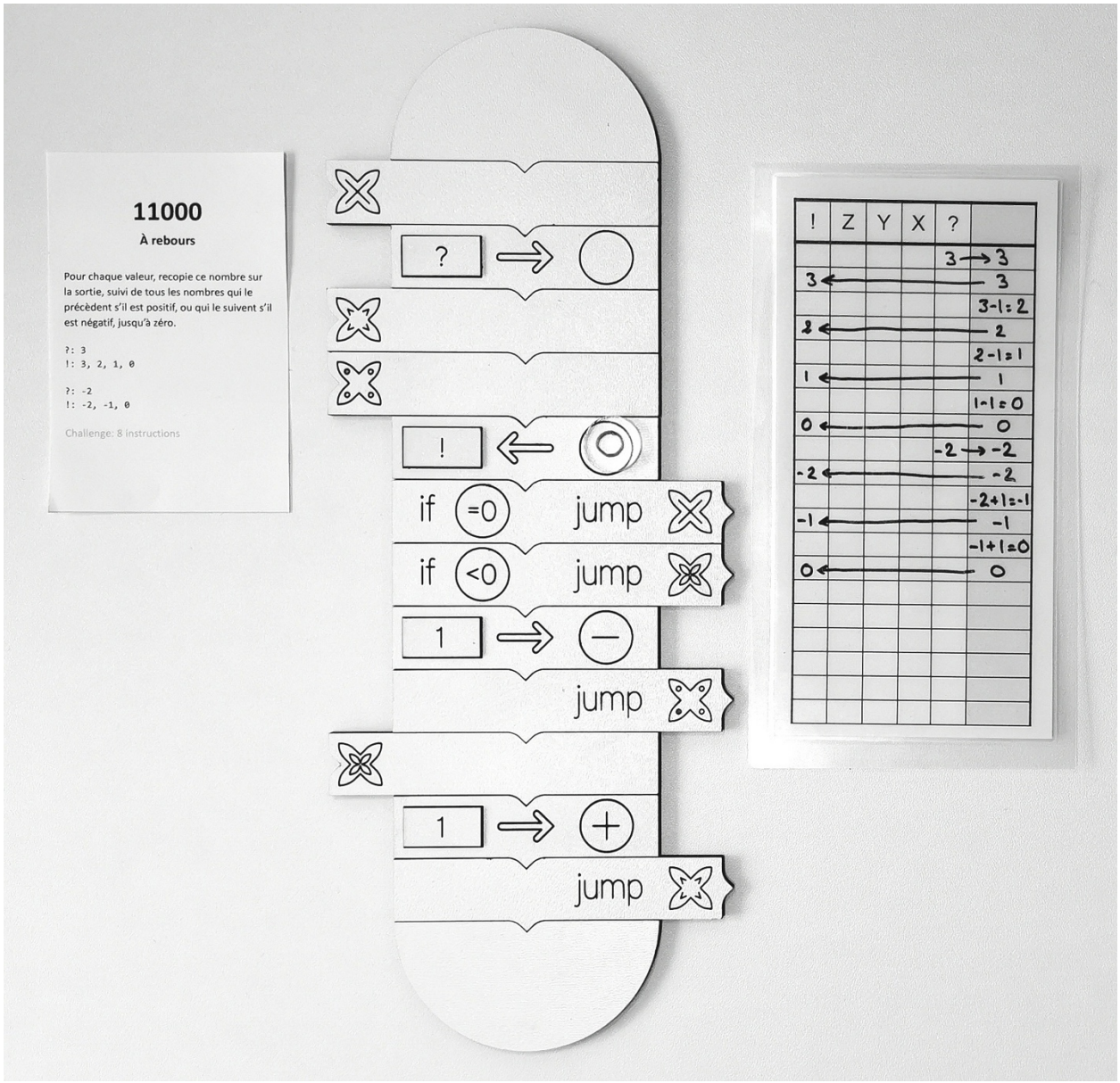


Figure 1 : *Compte à rebours.*

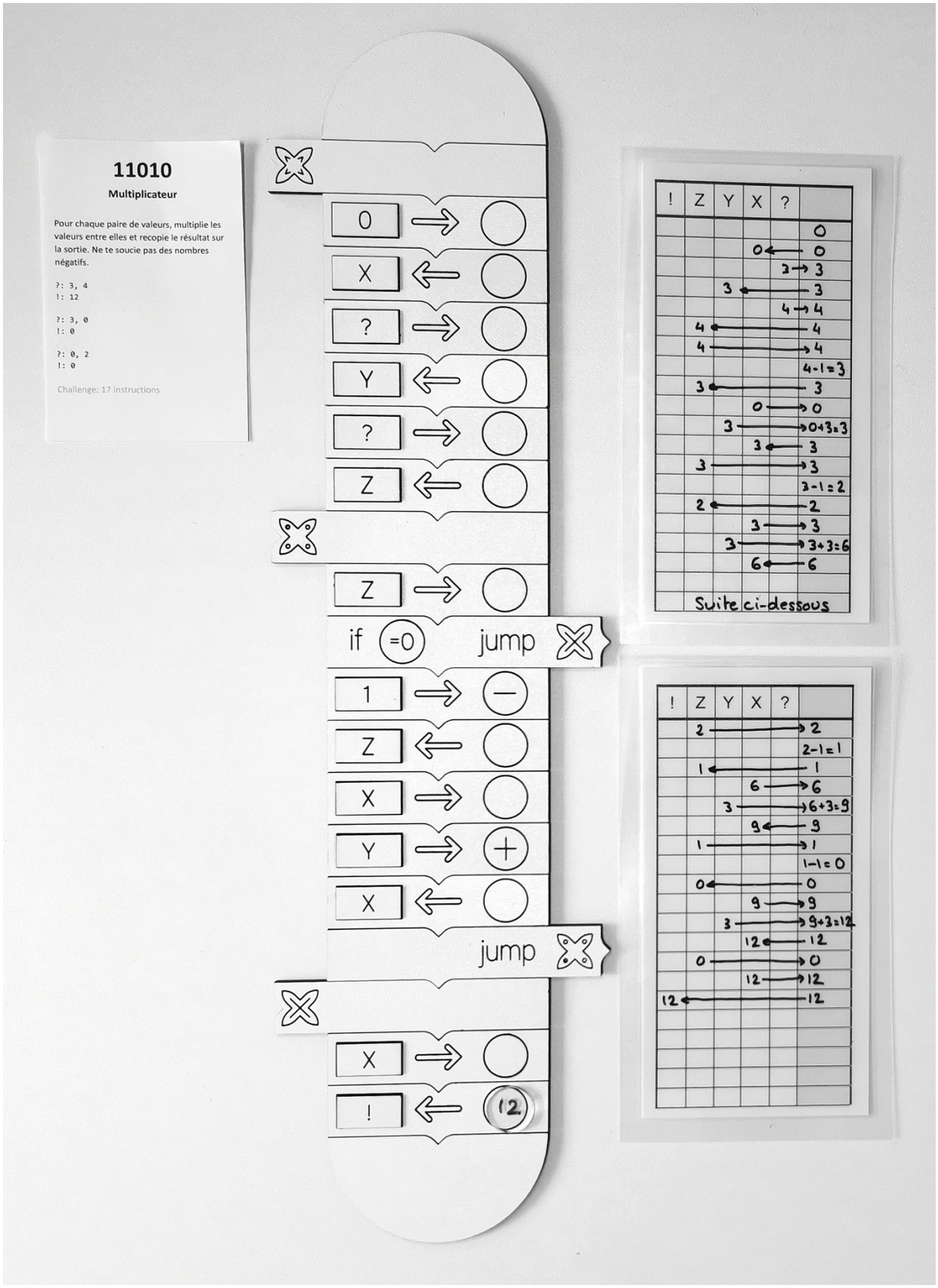


Figure 2 : *Multiplicateur.*

Dans chaque figure, on voit le jeton rond en plexiglas qui indique le point d'arrêt dans le code en cours d'interprétation. L'historique des différents états de la mémoire est rempli et l'on peut voir la trace d'exécution pour les valeurs du jeu de test. Les symboles sur la ligne du haut représentent respectivement la sortie (point d'exclamation), les trois mémoires (*X, Y, Z*), l'entrée (point d'interrogation). La dernière colonne est utilisée pour noter les calculs et les différentes valeurs de l'accumulateur. Les flèches indiquent dans quel sens les valeurs sont recopiées. Les instructions longues qui vont par paires identifiables grâce aux dessins de "fleurs" sont décalées vers la gauche ou vers la droite pour mieux voir les sauts dans le flux d'instruction.

Le jeu de pièces proposé correspond aux pièces minimales requises pour réaliser tous les exercices du manuel et tient parfaitement dans une petite boite en plastique, alors que les premières versions du dispositif avaient beaucoup plus de pièces et d'instructions différentes pour permettre de faire des exercices plus complexes. Une fois ouverte, la boite de rangement des pièces nécessitait un espace huit fois plus important sur les tables de classe que la boite actuelle, en plus de nécessiter beaucoup plus de temps de tri des pièces en fin de séance. Ces remarques peuvent paraître anodines, mais font vraiment une grosse différence dans une utilisation régulière en salle de classe.

Comme nous n'utilisons comme mécanisme de base que le déplacement de valeurs, la machine notionnelle sous-jacente au dispositif est triviale : toutes les opérations effectuées sont visibles, y compris les sauts dans le code, ce qui rend très simple le modèle mental que doit se construire l'élève pour interpréter son code dans sa tête en visualisant les recopiages de valeurs ou les sauts dans la liste des instructions. L'exercice de multiplication en figure 2 est le vingt-septième et dernier exercice proposé. Nous pourrions faire des exercices plus complexes, mais le code risque de devenir assez rapidement difficile à lire. Nous considérons que lorsque l'élève est capable de coder la multiplication en assembleur à l'aide d'additions et de soustractions, il est temps de passer à un langage de plus niveau.

2.2. Résultats

Nous avons testé ce dispositif à l'ESIG (École Supérieure d'Informatique de Gestion) de Genève, durant un semestre, lors d'un cours optionnel d'une période par semaine ayant lieu durant la pause de midi. Le public est constitué d'élèves débutants ayant entre 20 et 24 ans. Environ 80% des élèves arrêtent l'école avant la fin de la 2^{e} année par ce que c'est trop difficile. C'est donc un public qui a vraiment besoin de soutien stratégique pour les aider à développer leur aptitude à résoudre des problèmes algorithmiques. Nous avons commencé avec une quinzaine d'élèves et le groupe s'est réduit en cours de semestre pour finir avec six élèves. Nous avons alterné les moments de manipulation du dispositif en groupes de deux ou trois élèves pour que le socioconstructivisme puisse opérer, et les moments de réflexion collective avec toute la classe sur les manières de résoudre les problèmes donnés et les stratégies utilisées par les élèves. Nous avons filmé toutes nos interventions, mais nous ne les avons pas mesurées et comptabilisées pour évaluer le niveau de soutien prodigué par l'enseignant, car nous considérons que cela fait partie du rôle normal de l'enseignant que d'aider les élèves : nous procéderions de même avec un environnement de programmation sur écran. Dans l'ensemble, les élèves ont réussi à résoudre les exercices, souvent seuls, mais parfois avec un peu d'aide. Étant donné que les élèves arrivaient à résoudre les problèmes proposés, nous pouvons dire qu'ils ont appris à penser les algorithmes par eux même, à les programmer, à corriger les erreurs dans leur code et donc à développer leur pensée informatique.

Nous profitions des moments de mise en commun pour expliciter certaines stratégies cognitives utiles pour la résolution des problèmes proposés. Ces stratégies sont suffisamment génériques pour pouvoir s'appliquer à toute situation de résolution de problème. Nous espérons ainsi que

des transferts puissent avoir lieu dans d'autres disciplines, mais ces éventuels transferts ne sont pas l'objet de nos recherches. Voici les stratégies qui ont émergées de nos discussions :

Avant de se mettre au travail, pour apprendre à se concentrer et se mettre en condition :

Je suis capable de le faire. Qu'est-ce que je dois faire ? Est-ce que j'ai bien toutes les données ? C'est OK de ne pas y arriver du premier coup ou de faire des erreurs : garde confiance en toi. Si c'était trop facile, tu n'apprendrais rien.

Je me concentre pour réussir. Est-ce que tu as besoin qu'on te résume ce qu'il faut faire ? Focalise ton attention sur la tâche que tu es en train d'accomplir. Tu penseras au reste plus tard. Il y a un temps pour apprendre et un temps pour s'amuser.

Je résiste à donner la première réponse sans vérifier. Est-ce que tu as pris le temps avant de donner ta réponse ? Vérifie bien l'exactitude de ton idée. Parfois, la vérité se cache ailleurs. Une approche critique et réfléchie permet d'éviter les pièges.

Je pense avant d'agir. Comment ne pas répondre au hasard ? Demande-toi, si tu fais ça, alors qu'est-ce qui se passe ? Tu exploreras très rapidement beaucoup plus de possibilités en raisonnant dans ta tête qu'en essayant pour de vrai.

Pendant la recherche d'une solution, pour apprendre à résoudre des problèmes :

Je fais des liens avec ce que je connais. Qu'est-ce qui est pareil ? Qu'est-ce qui est différent ? Cherche des similitudes avec ce que tu sais déjà et fais une hypothèse "c'est comme si...". Observe les différences et essaye de voir si ton intuition est juste.

Je me construis une image mentale du problème. Qu'est-ce qui est utile ? Qu'est-ce qui ne l'est pas ? Qu'est-ce que je garde dans ma tête ? Représente-toi visuellement ce qu'il faut faire avant de commencer, ça va t'aider à mieux réfléchir.

Je manipule mon image mentale pour résoudre le problème. Utilise l'image mentale construite pour chercher une solution. Déplace les choses avec tes yeux pour vérifier tes idées.

Je cherche à simplifier le problème. Découpe le problème en plus petits problèmes ou bien résous un problème semblable, mais plus simple. Qu'est-ce qui est constant ? Qu'est-ce qui est variable ? Essaye de généraliser ta solution.

Au moment où l'on est bloqué, pour apprendre à être créatif :

J'explore une autre piste. Lorsqu'une piste de réflexion ne mène à rien, fais marche arrière et envisage d'autres alternatives partout où tu as dû faire des choix.

Je change de chemin. Tu es bloqué et tu as listé toutes les possibilités et leurs alternatives ? Cherche alors à changer d'approche, de stratégie ou à faire totalement autrement.

Je trouve de nouvelles idées. Ferme les yeux et évacue toutes les pensées parasites en faisant taire la petite voix dans ta tête. Des idées vont émerger d'elles-mêmes.

J'utilise l'intelligence du groupe. Vraiment bloqué ? Cherche de l'aide et lorsque tu as trouvé, regarde si tu peux à ton tour aider les autres à trouver par eux-mêmes. Demande-toi si les autres vont comprendre ce que tu as fait ou ce que tu dis.

Lorsque le travail est terminé, dans une étape d'analyse réflexive pour enrichir les apprentissages :

Quand la tempête est passée, j'apprends à baisser les tours. Lorsque je ne trouve pas, que je suis bloqué, je ne dois pas stresser ni paniquer. Je dois juste m'habituer à accepter le doute et l'incertitude. C'est la condition pour laisser émerger de nouvelles idées.

Je réfléchis aux chemins pris pour apprendre. Quelles sont les étapes que tu as suivies ? Juste après avoir résolu un problème, pose-toi la question de ce qui t'a permis de résoudre le problème pour t'en souvenir plus tard.

Je réfléchis plus pour travailler moins et apprendre mieux. Qu'est-ce que tu as vu ou entendu dans ta tête ? Comment as-tu su que c'était correct ? Dirige ton attention sur ta manière d'apprendre, pour apprendre mieux et avec moins d'efforts.

Je réfléchis aux applications futures. Quel principe général peux-tu dégager de cette expérience ? Comment l'appliquer ailleurs ou autrement ? Imagine comment réutiliser ce

que tu apprends pour pouvoir faire des liens avec des situations futures.

Le développement des stratégies cognitives en classe a très bien fonctionné. Voici un extrait du témoignage des élèves à la fin de la formation. Vous pouvez retrouver l'intégralité de leurs retours et de nos discussions dans la vidéo publiée sur https://human-processor.xyz/info.

> Apprendre à réfléchir de manière différente. C'est plus ludique, plus visuel que de travailler sur écran. Ça développe une autre capacité. Dans notre tête ça se fait plus rapidement qu'avec du code. Notre cerveau il réfléchit et il assemble le truc.
>
> L'exercice de la multiplication par 40 m'a bien fait comprendre comment décomposer un problème en petits problèmes. C'est ce qui m'a le plus aidé durant les épreuves.
>
> Ce que j'ai le plus appris c'est à chaque fois que j'ai réussi à faire quelque chose, de faire le chemin inverse. De réfléchir comment je suis arrivé au résultat. Ça m'a permis de bien comprendre et de mieux structurer les choses : de savoir pourquoi on fait ça.
>
> J'ai compris comment ça marche un ordinateur en soi : c'est pas magique, il y a rien de magique, en fait c'est logique. J'ai compris ça au tout début. Tout ce qui était binaire j'avais pas compris en cours. J'ai trouvé l'algorithme de la multiplication extraordinaire.
>
> Prendre plus le temps de réfléchir avant de se mettre à écrire le code. Maintenant je prends le temps de lire plusieurs fois une épreuve avant de la commencer et de temporiser alors qu'avant c'était pas le cas, je me précipitais à coder et après je me rendais compte que j'avais perdu du temps. Maintenant je réfléchis à ce que je vais faire, ça crée une structure dans ma tête avant de commencer.
>
> Les stratégies cognitives m'ont bien aidé durant les épreuves. Je les ai essayées. J'ai fait des pauses durant l'examen. Je suis sorti pour aller aux toilettes même si j'avais pas besoin, juste pour penser à autre chose et c'est comme ça que j'ai trouvé des idées et des solutions aux problèmes qu'on avait.

À la lecture de ces témoignages, il semble que l'utilisation hebdomadaire du dispositif dans notre groupe d'élèves a contribué à un développement efficace de leur pensée informatique, comparativement au cours obligatoire d'algorithmique et de programmation. On peut aussi constater que des transferts au niveau des stratégies ont eu lieu. Cependant, il convient de noter que le caractère facultatif de notre cours pourrait impliquer une sélection naturelle d'élèves déjà motivés et prédisposés à l'informatique. Bien que les six élèves assidus aient obtenu des moyennes supérieures en algorithmique, cette observation pourrait refléter davantage leur motivation intrinsèque qu'une efficacité intrinsèque du dispositif. Cette hypothèse mériterait d'être explorée à travers une expérimentation dans un cadre obligatoire, pour confirmer si les avantages observés s'étendent à un public plus large.

3. Discussion

C'est en résolvant les problèmes que l'élève va exercer sa pensée informatique. C'est en traçant les états de la mémoire qu'il va se construire et renforcer sa représentation mentale de la machine notionnelle du dispositif didactique et du langage associé. C'est en repérant systématiquement ses erreurs lors de l'interprétation pas à pas de son code qu'il va peu à peu prendre conscience qu'il est capable de manipuler mentalement les états de la mémoire et de prévoir avec exactitude ce que va faire son code, juste en le lisant. Lorsqu'il atteint ce niveau d'aptitude, nous considérons que l'élève a acquis une des compétences clés d'un programmeur. Il n'est plus dans une stratégie minimaliste d'essais-erreurs, mais réellement dans une dynamique de lecture, prédiction, réflexion et correction du code, sans nécessairement devoir l'exécuter. Vu la simplicité du dispositif et de la machine notionnelle sous-jacente, cette maîtrise arrive assez vite au fil des exercices. En développant cette compétence, l'élève diminue sa charge cognitive

intrinsèque liée au travail de compréhension de son code et de ses erreurs. Les ressources intellectuelles ainsi libérées peuvent être allouées à la réflexion sur l'algorithme à concevoir. Il pourra mettre également plus d'attention sur ses méthodes de résolutions et sur les stratégies cognitives qu'il utilise. Comme durant les phases réflexives communes nous l'invitons systématiquement à revenir sur comment il procède pour résoudre les problèmes posés, il va développer un habitus métacognitif c'est-à-dire une conscience réfléchie de ses propres processus cognitifs et développer en situation la capacité de réguler, de contrôler et d'adapter sa propre pensée. Nous espérons que cette prise de conscience de sa manière de résoudre les problèmes lui sera utile dans d'autres circonstances.

Un reproche que les enseignants-informaticiens nous ont fait est d'avoir choisi de commencer à expliquer les bases de la programmation en utilisant un langage machine proche d'un assembleur très primitif, au lieu d'utiliser un langage de haut niveau, plus élégant et mieux structuré. Notre réponse à cette critique est la suivante. Ce n'est pas parce que le langage assembleur n'a pas été inclus dans le programme scolaire qu'il n'est pas adapté pour comprendre les rudiments d'architecture et de programmation. Les premiers ordinateurs étaient plus faciles à fabriquer et à comprendre que les ordinateurs actuels. Les premiers langages de programmation étaient plus proches du hardware et de la logique de fonctionnement des machines que les langages modernes de haut niveau. Si l'enseignant considère important que les élèves comprennent réellement le fonctionnement d'un ordinateur et les concepts fondamentaux qui constituent son architecture et qui déterminent les bases des premiers langages de programmation, alors notre approche épistémologique revenant aux origines de l'informatique, à l'époque où tout était encore relativement simple, sera adaptée. Par exemple, pour illustrer l'algorithme de la multiplication et le concept d'accumulateur, nous apportons une machine à calculer mécanique en classe (un arithmomètre d'Odhner). En revenant aux sources du savoir, nous pouvons plus facilement comprendre les mécanismes qui ont inspiré la conception des premiers ordinateurs et des premiers langages. Commencer par un langage de bas niveau permet plus tard de se représenter dans ce langage les instructions d'un langage de plus haut niveau. Prenons par exemple la ligne de code "*x = x + 1*" qui, dans notre langage primitif proche du matériel, se traduit en trois lignes de code "*lire x; ajouter 1; écrire x*". Lorsque l'on sait déjà programmer, il est clairement plus agréable d'écrire une seule ligne de code pour incrémenter une valeur. Cependant, pour bien comprendre cette ligne de code, un débutant devra la décomposer dans sa tête en trois actions qui correspondent à ce qui se passe réellement dans notre langage minimaliste. Le débutant n'aura donc pas besoin de faire un effort d'abstraction pour se représenter mentalement ce qui se passe. C'est certes plus long à écrire, mais plus simple à comprendre, car tout est explicite, rien n'est magique.

Un autre reproche que les enseignants-informaticiens utilisant notre dispositif didactique nous ont fait est de réintroduire l'usage de sauts dans le code alors que depuis les années 70, il est déconseillé de les utiliser *en production* au profit des structures de contrôles. L'argument avancé est qu'en présentant ce concept nous risquons d'induire de mauvaises pratiques chez les débutants. Notre contre-argument est que les élèves ne pourront de toute façon pas conserver l'habitude de l'utiliser puisqu'il n'y a pas de "*goto*" en Python. L'intérêt pédagogique du "*goto*" est de rendre le code lisible de manière séquentielle sans aucun saut mystérieux dans le flux d'exécution du code : on ne saute pas d'un endroit à l'autre du code sans comprendre pourquoi, ce qui peut porter à confusion dans un langage de haut niveau pour le débutant. Par exemple dans un "*if…then…else…*" on saute la partie "*else*" lorsque l'on a terminé la partie "*then*" alors qu'aucun "*goto*" n'est écrit dans le code. Dans notre langage il faut faire un "*jump*" explicite ce qui est plus simple à comprendre. Lorsque le concept de "*if…then…else…*" sera introduit plus tard, l'enseignant pourra faire référence au code assembleur correspondant pour démystifier le

comportement non explicite du langage. Idem pour les boucles.

Lors d'une expérience avec deux groupes d'élèves de 10 – 12 ans travaillant avec la version sur tablette du jeu "*human resource machine*" (celle qui nous a servi d'inspiration pour concevoir ce dispositif), nous avons observé certains élèves qui adoptent dès le début une stratégie de recherche dépourvue de réflexion pour finir les exercices au plus vite sans chercher à comprendre. C'est comme si le problème était terminé lorsque l'ordinateur disait que c'était terminé, au lieu d'être terminé uniquement lorsqu'ils avaient vraiment compris pourquoi leur code était considéré comme correct par la machine. Pour ces élèves, les objectifs d'apprentissage sont possiblement masqués par l'aspect ludique de l'environnement de programmation : on joue et le principal intérêt est de gagner et de tout finir avant les autres élèves de la classe. Après une séance avec l'application sur tablette, nous avons refait les mêmes exercices d'introduction à l'aide de notre dispositif débranché. Nous avons pu observer que les élèves les plus rapides sur tablette n'avaient pas appris ce qui était supposé être nécessaire de comprendre pour résoudre les exercices ! Une des élèves les plus lentes sur tablette, qui était donc allée moins loin dans l'avancement des exercices que les autres, avait en fait parfaitement compris les concepts et a pu refaire les exercices débranchés très rapidement en les expliquant aux autres. La version numérique sur tablette est donc adaptée pour les élèves posés et réfléchis, mais n'aide pas les élèves pressés ou paresseux, qui se retrouvent ensuite en difficulté.

Dans une autre recherche (Blanvillain & Baumberger, 2022), nous avons observé comment les élèves de 10 – 12 ans et de 20 – 24 ans qui ne parviennent pas bien à apprendre avec notre dispositif didactique débranché travaillent, en les comparant avec d'autres élèves du même âge, pas très forts, mais qui parviennent cependant à apprendre. Nous avons cherché à comprendre ce qui pourrait les aider à mieux concevoir les solutions algorithmiques et à les programmer. De nos observations, nous avons déduit que c'est l'absence ou le refus de s'entraîner, déjà visibles dans les vidéos lors des tout premiers exercices du tutoriel, qui pénalise ces élèves. Par leur manque d'implication initiale, ils ne se constituent pas de représentation mentale de la machine notionnelle du dispositif. Ce qui rend leur détection particulièrement difficile pour l'enseignant, c'est que certains donnent l'impression de travailler de manière studieuse, alors qu'en fait ils ne mobilisent pas leur attention sur ce qu'ils font ou sur ce qu'il faut comprendre et apprendre. Or, si ces élèves ne sont pas détectés de manière précoce, lorsque les exercices deviennent plus complexes ils ne peuvent plus suivre. Il est possible de les repérer en étant attentif au fait qu'ils restent longtemps dans la manipulation du dispositif didactique, faisant mine de travailler, sans alterner avec des moments de réflexion ou des moments de discussion avec les autres élèves pour partager leurs découvertes. Une autre manière de les identifier est d'aller questionner chaque élève sur ce qu'il a compris qu'il fallait faire, ce qu'il avait déjà réussi à faire et ce qu'il est en train de faire. Lorsque l'enseignant soupçonne être en présence d'un élève qui ne parvient pas à aborder les problèmes simples du tutoriel, nous lui conseillons d'expliciter une stratégie métacognitive présentant un processus possible pour apprendre à penser les algorithmes (Blanvillain, 2020b, 2021). Cette expérimentation s'est arrêtée à la formulation de la stratégie métacognitive, avant de pouvoir déterminer si ce serait suffisant pour aider ces élèves en difficulté.

Si nos exercices et notre jeu de pièces permet pour le moment de ne couvrir qu'un seul semestre en débranché dans le cursus de formation des élèves au numérique, elle pourra peut-être inspirer d'autres chercheurs qui souhaiteraient étendre le dispositif proposé et couvrir plus de concepts de programmation en ajoutant des pièces, ou même partir sur la conception d'un autre langage de programmation débranché plus ambitieux.

En 2025, au gymnase de Chamblandes avec une classe d'élèves à profil scientifique, nous avons

réalisé une expérience un peu différente. Le premier tiers de l'année, nous avons travaillé les stratégies cognitives pour apprendre à apprendre à l'aide du ebook "Hack ton cerveau" https://www.hack-ton-cerveau.xyz/ et un portfolio réflexif d'apprentissage hebdomadaire, dans lequel les élèves devaient expérimenter ces stratégies dans les cours de leur choix et consigner le résultat de leur vécu, quel qu'il soit (positif, neutre ou négatif). L'objectif étant de les habituer à faire de l'introspection pour développer leurs aptitudes à la métacognition. Puis, le deuxième tiers de l'année, nous avons travaillé l'algorithmique et les stratégies de résolution de problème à l'aide du dispositif. Lors de l'explicitation de leurs processus de réflexion durant de la recherche de solutions, trois élèves ont fait des propositions riches et très accessibles qui nous ont permis de réaliser une synthèse (présentée en annexe).

4. Conclusion

Coder une solution à un problème algorithmique nécessite habituellement l'usage d'une machine qui se charge de vérifier si ce à quoi nous avons pensé "marche" ou pas… En éliminant la machine de l'équation, c'est l'élève qui, en interprétant son code pas à pas, va jouer le rôle du démonstrateur. Ce travail supplémentaire accélère le développement de sa pensée informatique.

Ce dispositif s'inscrit idéalement après une initiation à la programmation pour entraîner les élèves à visualiser mentalement les déplacements d'un robot et avant l'apprentissage de la programmation textuelle. Les élèves débutants en programmation avec lesquels nous l'avons testé étaient âgés de 20 – 24 ans. Chez des élèves trop jeunes, il sera probablement difficile de faire émerger les stratégies cognitives du groupe classe. Chez des élèves qui ont déjà acquis des bases de la programmation, les exercices seront peut-être trop simples. Nous supposons que l'âge optimal se situerait entre 14 et 16 ans, ce qui correspond à la fin du secondaire I en Suisse ou la fin du collège en France et le début du secondaire II en Suisse ou l'entrée au lycée en France.

Les objectifs du dispositif sont triples :

1) Enseigner les protoconcepts de la programmation textuelle, tels que la notion de mémoire pour introduire plus tard la notion de variable et de type de données, la notion de branchement conditionnel pour introduire ensuite la notion d'expressions booléennes, de bloc de tests et de boucles. Nous avons réduit le nombre d'instructions pour simplifier le langage (et le nombre de pièces du dispositif) ce qui le rend très facile à apprendre : on ne peut que recopier une valeur (avec ou sans addition, dans un sens ou dans l'autre) ou bien sauter à un autre endroit du code (avec ou sans condition). La machine notionnelle que doivent se construire les élèves pour résoudre les problèmes proposés est très proche de ce qui se passe réellement au sein du processeur. De plus, le dispositif didactique débranché permet de rendre plus facilement visibles les réflexions des élèves pour l'enseignant que lorsqu'ils travaillent sur écran. Cette particularité va permettre à l'enseignant d'apporter des aides ciblées pendant la phase de conception de l'algorithme à ceux qui en auraient besoin.

2) Entraîner les élèves à la lecture et à l'interprétation des instructions de leur propre programme et les habituer ainsi à effectuer mentalement ces opérations. La feuille de papier pour tracer l'historique des différents états de la mémoire aide les élèves à se représenter les données pour chaque étape et chaque ligne de leur code et ainsi réussir à trouver les éventuelles erreurs. L'idée de départ de ce dispositif didactique débranché est de mettre l'élève à la place du processeur, pour qu'il apprenne à raisonner comme lui et développe sa pensée informatique. Nous avons vu des élèves pressés de résoudre les exercices sans chercher à les comprendre. Nous souhaitons éviter qu'ils ne se complaisent dans une stratégie de recherche de solution "à tâtons" en exploitant à outrance la puissance de l'ordinateur par des clics irréfléchis sur le bouton exécuter. Il est pédagogiquement judicieux de ralentir ces élèves dans la phase de découverte des

bases de la programmation afin qu'ils apprennent à se construire une représentation mentale du code et à apprendre à prédire ce que leur code va faire. Notre dispositif didactique débranché les oblige à penser à ce qu'ils sont en train de coder et permet de donner un rythme lent d'apprentissage aux élèves en leur proposant de jouer le rôle de l'ordinateur et de vérifier eux-mêmes leur propre code (d'où son nom "*human processor!*"). Le temps d'entraînement à réaliser ces exercices mentaux que nous leur proposons devrait favoriser le développement rapide de leur pensée informatique. Ce n'est qu'après avoir développé cette aptitude que nous conseillons de passer sur écran de manière à pouvoir résoudre des problèmes plus complexes tout en jouissant des facilités offertes par un environnement numérique interactif avec accès à internet et aux intelligences artificielles capables de générer du code.

3) Apprendre à penser les algorithmes grâce à l'explicitation de stratégies cognitives pour concevoir des solutions innovantes à des problèmes algorithmiques nouveaux. Le travail demandé à l'élève consiste en un acte intellectuel complexe pour découvrir une solution à un problème posé. Pour aider les élèves les moins habitués à effectuer cet exercice mental, nous concluons chaque séance par une phase réflexive durant laquelle nous tentons de mutualiser les stratégies cognitives mobilisées par les élèves ayant réussi à résoudre le problème. En favorisant la prise de conscience des stratégies utilisées, nous les aidons à apprendre à penser les algorithmes, ce qui permet de développer un état d'esprit dynamique et de promouvoir une vision de l'intelligence dans laquelle les élèves comprennent qu'avoir du "talent" est quelque chose qui se développe à l'aide de travail et de bonnes stratégies. C'est l'objet de toute notre ingénierie didactique : développer la métacognition et la faculté de penser les algorithmes en faisant prendre conscience des stratégies de résolution de problèmes.

Concrètement, après avoir fait réaliser l'activité aux élèves, l'enseignant les questionne en leur demandant de présenter leur démarche et d'exprimer leurs stratégies de résolution. Énoncées en situation, ces stratégies prennent tout leur sens. Pour aider les élèves à se remémorer cet instant et ce qui s'est passé juste avant le "moment eurêka", nous utilisons des questions inspirées des techniques d'entretien d'explicitations, sans influer ou induire les réponses attendues, juste en aidant l'élève dans la remémoration de sa pensée en actes. Ces pauses réflexives ne sont pas réservées qu'au partage par les élèves, c'est aussi un moment approprié pour que l'enseignant propose de nouvelles approches ou pour qu'il puisse expliciter comment faire pour penser les algorithmes en s'aidant d'une stratégie métacognitive (Blanvillain, 2020b, 2021).

Notre approche crée les conditions pour que tous les élèves apprennent à développer leur aptitude à résoudre des problèmes algorithmiques. Certes, le côté spartiate de l'environnement didactique risque de rebuter une partie de ces élèves qui préfèrent jouer sur tablette sans trop réfléchir plutôt que de résoudre des problèmes algorithmiques. Mais justement, ici on ne joue pas : on apprend à penser. Notre approche centrée sur l'enseignement explicite de comment faire pour résoudre des problèmes algorithmiques, met tous les élèves dans une situation où ils peuvent apprendre à réfléchir et à être créatifs, et pour ceux qui jouent le jeu, développer leur sentiment d'efficacité personnel, leur faire éprouver la satisfaction de réussir et, nous l'espérons, susciter ainsi une motivation intrinsèquement liée au plaisir de coder. Il est également important de souligner le rôle crucial de l'affect comme activateur de la motivation à apprendre dans ce processus de développement de la faculté de résoudre des problèmes.

L'analyse réflexive des chemins qui ont été pris pour trouver les solutions entraîne les élèves à pratiquer de la métacognition et leur fait prendre conscience de leurs stratégies. Nous avons pu observer au travers du témoignage des élèves, que les stratégies conscientisées en situation, c'est-à-dire qui ont effectivement permis à certains élèves de se débloquer, ont non seulement été comprises, mais aussi intégrées et même transférées dans un autre cours. Nous pensons qu'une

généralisation de cette pratique éducative introspective à tous les niveaux et dans tous les domaines d'étude serait bénéfique pour le développement de la capacité d'apprendre et de la compétence à résoudre des problèmes. Le fait de partager régulièrement les approches stratégiques en réalisant des exercices de prise de conscience de ces stratégies dans d'autres disciplines, face à diverses situations et problèmes, devrait favoriser les transferts interdisciplinaires et rendre les élèves plus posés, plus réfléchis et plus intelligents. Pour permettre aux élèves de s'adapter et de réussir dans un mode en constante évolution où les savoirs et les compétences deviennent rapidement obsolètes, n'est-ce pas dans le développement des capacités cognitives et métacognitives que le rôle le plus noble de l'école se manifeste ?

Références bibliographiques

Blanvillain, C. (2020a). *Human Processor!* Communication présentée à Ludovia#CH, Yverdon-les-Bains, Suisse. http://hdl.handle.net/20.500.12162/5376

Blanvillain, C. (2020b). *Apprendre à penser les algorithmes. Concepts et exemple.* BIAA, No 117, pp. 7-44. http://biaa.eu/-upload/biaano117.htm

Blanvillain, C. (2021). *Apprendre à penser les algorithmes.* Communication présentée à APIMU EIAH, Fribourg, Suisse. http://hdl.handle.net/20.500.12162/5377

Blanvillain, C. et Baumberger, B. (2022). *Détection précoce des élèves rencontrant des difficultés d'apprentissage de l'algorithmique.* Poster présenté à Didapro - DidaSTIC Le Mans, France. http://hdl.handle.net/20.500.12162/5971

Blomquist, A. Gabler, K. Gray, K. (2015). *Human Ressource Machine.* Consulté le 1er octobre 2022 sur https://tomorrowcorporation.com/about

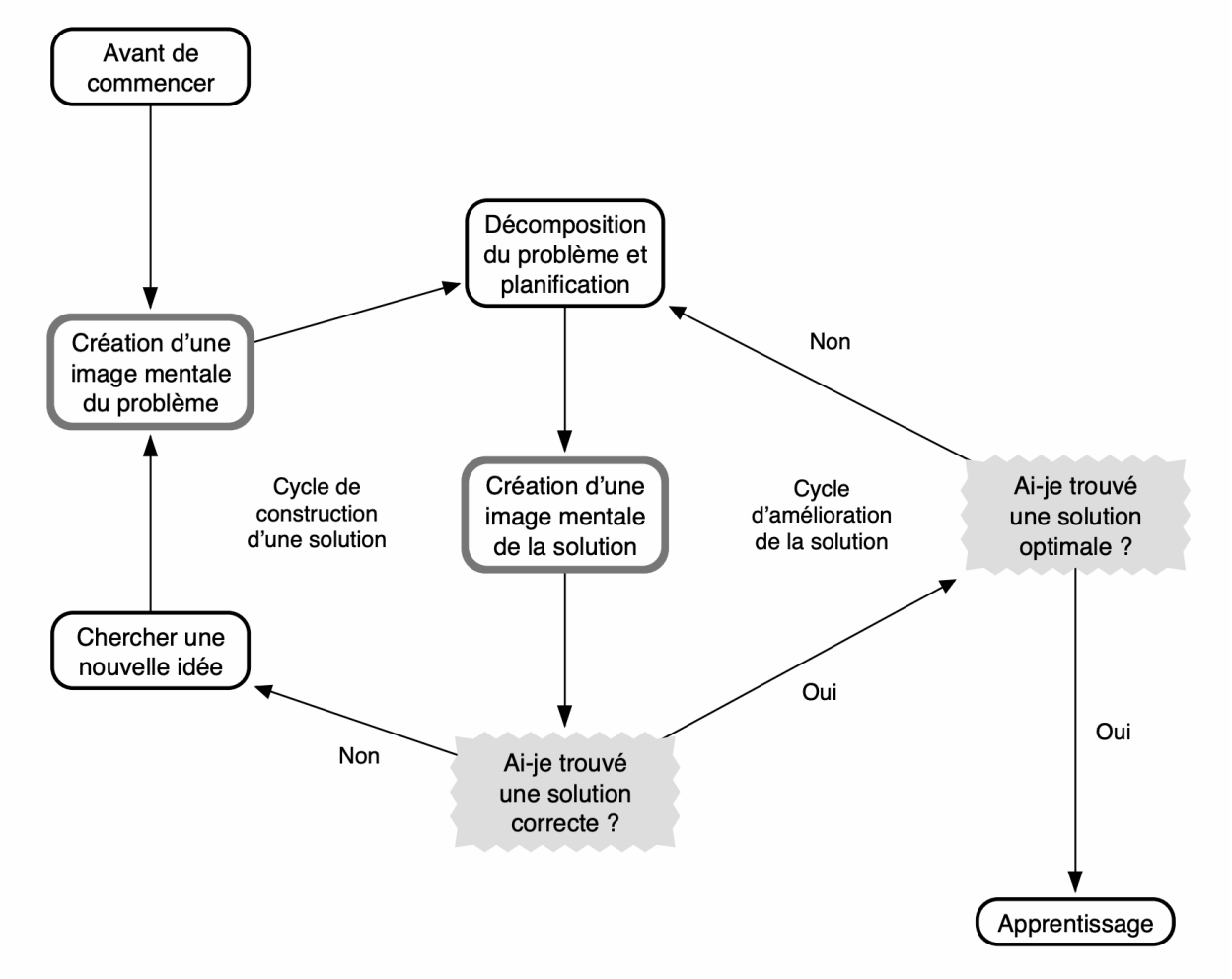
Chevalier, M. (2022). *Mediating Computational Thinking Through Educational Robotics In Primary School.* [Médiation de la pensée computationnelle grâce à la robotique éducative à l'école primaire.]. Thèse de doctorat, EPFL. Infoscience. http://dx.doi.org/10.5075/epfl-thesis-7575

Eliasz, A. (2016). *Little Man Computer Programming: for the perplexed from the ground up*. [Programmation informatique Petit Homme : pour les perplexes, de la base au sommet]. The Internet Technical Bookshop.

Madnick, S. (1965). *Little Man Computer.* Consulté le 1er octobre 2022 sur https://fr.m.wikipedia.org/wiki/Little_man_computer

Annexe : Processus pour concevoir un algorithme

Les phrases en italique correspondent aux propositions des trois élèves et de l'enseignant-auteur (Christian).



1) Avant de commencer

- **Je prends confiance en moi** : accepter l'erreur comme partie de l'apprentissage / maintenir une attitude positive.
- **Je me concentre pour réussir** : se focaliser sur la tâche / éliminer les distractions / s'immerger dans le problème.

Se mettre dans un bon état mental est essentiel. Alyssa insiste sur l'importance de la préparation mentale. Samantha recommande d'éliminer les distractions et de se concentrer pleinement sur le problème. Selon Frédéric, il faut accepter que l'erreur fasse partie de l'apprentissage. L'école est un lieu pour apprendre, pas pour être parfait du premier coup.

2) Création d'une image mentale du problème

- **Je lis méthodiquement** : lire attentivement l'énoncé / identifier les données de départ / repérer les objectifs à atteindre.
- **J'allège ma charge cognitive** : relire plusieurs fois / identifier les éléments essentiels / éliminer les informations superflues.

- **Je me fais une image mentale du problème** : visualiser les éléments clés / comprendre leurs relations / saisir la structure globale.

Frédéric propose de commencer par identifier les données de départ et les objectifs. Alyssa suggère de lire plusieurs fois l'énoncé pour repérer les éléments importants. Samantha recommande de reformuler le problème avec ses propres mots. Alyssa conseille de visualiser mentalement le problème et ses composantes.

3) Décomposition du problème et planification

- **Je m'approprie le problème** : reformuler avec mes propres mots / vérifier ma compréhension.
- **Je garde mon calme** : rester confiant et serein / ignorer temporairement les défis trop complexes / chercher d'abord une solution partielle, mais simple.
- **Je découpe en sous-problèmes** : identifier les composantes principales / créer des parties gérables / organiser les éléments logiquement.
- **Je planifie ma démarche** : évaluer les ressources disponibles / imaginer leur utilisation / formuler des hypothèses de travail.

Christian insiste sur l'importance de ne pas se précipiter. Il faut prendre le temps d'analyser différentes approches possibles avant de choisir une piste prometteuse. Pour Frédéric, si le problème semble trop difficile, il ne faut pas paniquer. Mieux vaut chercher d'abord une solution simple qui fonctionne.

Samantha propose de diviser le problème complexe en parties plus simples. Alyssa recommande d'analyser chaque composant séparément, puis de les réassembler mentalement. Elle suggère aussi de faire un plan en analysant les ressources disponibles.

4) Création d'une image mentale de la solution

- **J'analyse chaque sous-problème** : examiner les éléments séparément / visualiser leur signification / comprendre leur rôle.
- **Je fais des liens avec ce que je connais** : connecter avec des expériences antérieures / similitudes avec des problèmes connus / utiliser les apprentissages récents du cours.
- **Je manipule les éléments dans ma tête** : tester mentalement différentes approches jusqu’à résoudre chaque sous-problème / commencer par ne traiter que les cas simples.

Christian explique qu'après avoir compris et décomposé le problème, il faut imaginer une solution possible pour chaque partie. Frédéric rappelle l'importance de faire des liens avec les connaissances déjà acquises en classe. Alyssa et Samantha suggèrent de tester mentalement différentes approches pour chaque sous-problème. Cette étape permet d'essayer rapidement plusieurs solutions avant de les réaliser avec le dispositif.

5) Ai-je trouvé une solution correcte ?

- **Je reconstruis l'ensemble** : assembler mentalement les sous-problèmes / former un algorithme cohérent / vérifier les connexions / créer une vision globale.
- **Je valide par étapes dans ma tête** : simuler mentalement la solution / tester chaque sous-problème / vérifier régulièrement l'énoncé.
- **Je vérifie la justesse de mes idées avec les pièces** : tester dans la réalité / vérifier les résultats avec la bande en plastique / traiter à ce moment-là les cas limites.
- **Je corrige et j'améliore** : comparer les résultats attendus et obtenus / identifier les dysfonctionnements pour les cas limites / apporter les corrections nécessaires.

Frédéric recommande de simuler mentalement la solution avant de la tester concrètement.

Samantha propose d'utiliser des outils pratiques comme la bande plastique. Alyssa insiste sur l'importance de vérifier les résultats et de corriger les erreurs. Samantha rappelle qu'il ne faut pas paniquer si la solution ne fonctionne pas. Elle suggère de relire le problème pour identifier les éléments oubliés. Alyssa recommande de vérifier particulièrement tous les points de l'énoncé (et donc les cas limites).

6) En cas de blocage, chercher une nouvelle idée

- **Je persévère méthodiquement** : prendre du recul si nécessaire / garder confiance / relire méticuleusement l'énoncé.
- **Je pense créatif** : explorer des alternatives / changer de point de vue en abordant le problème sous différents angles / essayer de nouvelles approches / réfléchir différemment pour chercher des solutions innovantes / chercher des analogies.
- **Pause stratégique** : prendre du recul / laisser l'esprit vagabonder pour favoriser l'émergence d'idées originales.
- **Je demande de l'aide** : consulter d'autres élèves / discuter avec des pairs / solliciter l'enseignant ou d'autres ressources.

Alyssa et Frédéric encouragent à ne pas se décourager et à explorer de nouvelles pistes. Samantha propose de changer de perspective et de faire des analogies. Christian suggère de faire une pause pour laisser l'esprit vagabonder. Il recommande aussi de demander de l'aide aux autres élèves ou à l'enseignant si nécessaire.

7) Ai-je trouvé une solution optimale ?

- **Une fois la solution trouvée l'optimiser** : optimiser les performances / chercher les opérations inutiles / chercher une autre approche pour réduire la complexité / identifier la meilleure solution.
- **Je réfléchis à d'autres cas d'usage possibles** : se demander s'il est possible de généraliser la solution pour traiter d'autres situations.

Alyssa conseille d'identifier la meilleure solution tout en gardant les autres possibilités en tête. Christian suggère d'optimiser le code en réduisant sa complexité ou les ressources utilisées. Il recommande aussi de réfléchir à une éventuelle généralisation de la solution pour d'autres contextes.

8) Apprentissage : qu'ai-je appris d'utile ?

- **Je prends du recul** : analyser le processus de résolution global / identifier les points d'amélioration / mémoriser la démarche.

Christian souligne l'importance de la métacognition : réfléchir aux stratégies qui ont bien fonctionné et analyser le processus de résolution global. Cette réflexion permet de mémoriser la démarche pour de futures situations similaires et de s'améliorer.