QuickFill : travailler plus pour générer moins en synthèse de programmes

Vanessa Fokou*, Peggy Cellier**
Maurice Tchuente*, Alexandre Termier**

*Université de Yaoundé I, Faculté des Sciences, Département d'Informatique
BP 812 Yaoundé Cameroun
IRD, Sorbonne Université, UMMISCO, F-93143, Bondy, France
vanfokou@gmail.com, maurice.tchuente@gmail.com
**Univ Rennes, Inria, INSA Rennes, CNRS, IRISA - UMR 6074, France
prenom.nom@irisa.fr

Résumé. Afin de faciliter l'exécution de tâches répétitives, des approches de synthèse de programmes ont été développées. Elles consistent à inférer automatiquement des programmes qui satisfont l'intention d'un utilisateur. L'approche la plus connue est *FlashFill* qui est intégrée au tableur Excel. Dans *FlashFill* l'intention de l'utilisateur est représentée par des exemples, i.e. des couples $(entrée, sortie)$. *FlashFill* explore un très grand espace de recherche des programmes et peut donc nécessiter un temps d'exécution important et inférer beaucoup de programmes dont certains fonctionnent sur les exemples donnés mais ne capturent pas l'intention de l'utilisateur. Dans cet article nous présentons *Quickfill*, une approche qui réduit l'espace des programmes à explorer en enrichissant les spécifications fournies par l'utilisateur. Nous montrons qu'avec cette approche, il est souvent possible de donner moins d'exemples qu'avec l'algorithme *FlashFill* pour une plus grande proportion de programmes corrects.

1 Introduction

Le pré-traitement des données est l'une des tâches les plus chronophages, et sans doute les moins intéressantes, d'un processus d'analyse de données. Cette tâche demande souvent de faire des transformations non triviales de formats, qui sont trop complexes pour être effectuées par les outils existants, mais trop simples pour mériter l'attention soutenue d'un analyste (e.g., extraire les initiales du nom complet d'une entité afin d'obtenir son acronyme, cf figure 1). En pratique, la solution classique pour effectuer les tâches de pré-traitement de données est l'écriture de petits scripts dans des langages de programmation généralistes (e.g., Python). La difficulté est que de nombreux utilisateurs ne maîtrisent pas les bases de la programmation, et n'ont donc pas la possibilité d'écrire de tels scripts.

Pour pallier ce problème, une solution très prometteuse est l'utilisation de techniques de "synthèse de programmes" (*program synthesis* en anglais) (Gulwani et al., 2017). La synthèse de programmes consiste à inférer automatiquement des programmes qui satisfont l'intention

de l'utilisateur. Elle a été utilisée dans plusieurs applications comme la réparation de code (Nguyen et al., 2013), la modélisation probabiliste (Nori et al., 2015), la suggestion de code (Perelman et al., 2012) ou encore le traitement de données (Gulwani, 2016). En synthèse de programmes trois dimensions sont considérées (Gulwani, 2010) : le type de spécification permettant de modéliser l'intention de l'utilisateur, l'espace des programmes et le mode d'exploration de l'espace des programmes. Le type de spécification le plus simple est un ensemble de couples $(entrée, sortie)$ fournis par l'utilisateur. À partir de ces couples, les approches de synthèse de programmes infèrent un ou plusieurs programme(s) P tel que $P(entrée) = sortie$ pour tous les couples $(entrée, sortie)$ fournis en paramètre. Ainsi, grâce à un travail modéré de l'utilisateur (fournir quelques exemples de couples), il est possible d'obtenir un programme capable de faire la transformation voulue sur une grande quantité de données. L'exemple le plus connu en synthèse de programmes est *FlashFill* (Gulwani, 2011), qui est intégré à toutes les versions récentes du tableur Excel. *FlashFill* considère des lignes dans le tableur, et apprend des programmes prenant en entrée le contenu d'une ou plusieurs colonnes de la ligne, et dont le résultat est stocké dans une autre colonne. Un exemple d'utilisation de *FlashFill* est montré à la figure 1. À partir de l'exemple ("International Business Machine","IBM") *FlashFill* a inféré automatiquement un programme qu'il a ensuite appliqué à l'entrée "Extraction Gestion Connaissances" pour générer la sortie "EGC".

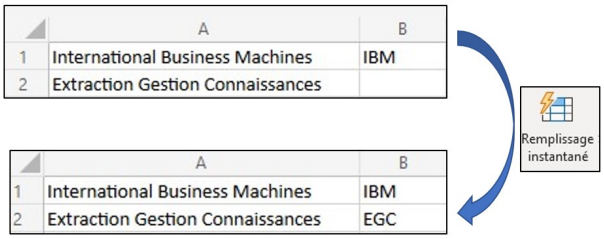

FIG. 1 – *Exemple d'utilisation de FlashFill : la cellule $B2$ a été automatiquement remplie en appliquant à la cellule $A2$ le programme inféré à partir de l'exemple des cellules $(A1, B1)$.*

Si dans cet exemple *FlashFill* arrive à inférer un programme qui semble correct à partir d'un unique exemple $(entrée, sortie)$, des cas plus complexes peuvent nécessiter de fournir deux voire trois exemples afin de réduire davantage l'espace des programmes et de ne garder que les programmes qui sont vraiment pertinents par rapport à l'intention de l'utilisateur.

L'algorithme original *FlashFill*, tel que présenté dans Gulwani (2011), doit explorer un espace de recherche des programmes très important, et peut donc nécessiter un temps d'exécution important (plusieurs minutes voire heures). De plus, beaucoup de programmes peuvent être inférés dont certains fonctionnent sur les exemples donnés mais ne capturent pas l'intention de l'utilisateur, et donneront des résultats faux sur d'autres entrées. En pratique, l'algorithme proposé dans les produits *Microsoft* est épaulé par de nombreuses heuristiques l'aidant à naviguer dans l'espace de recherche, mais ayant demandé un important travail de mise au point.

Dans cet article nous présentons une approche, *Quickfill*, qui réduit l'espace des programmes à explorer par *FlashFill* en enrichissant les spécifications fournies par l'utilisateur. En effet, pour chaque exemple, l'utilisateur est invité à donner quelques éléments supplémentaires sur le *matching* entre les sous-parties de l'entrée et de la sortie dans l'exemple. Cela permet de

réduire de manière significative l'espace de recherche de *FlashFill*, et d'atteindre plus rapidement des programmes corrects. Nous montrons qu'avec cette approche, il est souvent possible de donner moins d'exemples qu'avec l'algorithme *FlashFill* original, et la proportion de programmes corrects obtenue est plus grande dans la majorité des cas.

2 Méthode *Quickfill*

Nous commençons par présenter les bases du fonctionnement de l'approche *FlashFill*, puis nous présentons notre adaptation *QuickFill* qui réalise un compromis différent (interactions plus importantes avec l'utilisateur mais un espace de recherche réduit).

FlashFill. L'approche *FlashFill*, présentée en détail dans (Gulwani, 2011), prend en entrée un ensemble d'exemples de type (σ, s) où $\sigma = (\sigma_1, ..., \sigma_n)$ est un tuple dont chaque élément est une chaîne de caractères, et où s est une chaîne de caractères. En pratique, dans un tableur comme Excel, un exemple est une ligne du tableur, σ est un ensemble de cellules de la ligne contenant des données, et s est le résultat d'une transformation de chaîne de caractères sur les cellules σ. Par exemple $((A1 : "Jean", B1 : "Dupont"), C1 : "Jean\ Dupont")$ permet d'inférer qu'il faut concatener le contenu des colonnes A et B et ranger le résultat dans la colonne C. Pour simplifier, dans la suite les noms des colonnes seront omis : σ concernera toujours les n premières colonnes, et s sera stocké en $(n+1)$ème colonne.

À partir de l'ensemble d'exemples à lui fourni, *FlashFill* apprend des programmes de transformation de chaînes de caractères capables de transformer pour chaque exemple l'entrée σ en la sortie s. Pour cela, *FlashFill* dispose d'un ensemble de primitives simples pour la manipulation de chaînes de caractères, et de règles de composition. L'approche est donc avant tout un algorithme pour explorer aussi efficacement que possible le vaste espace de programmes généré par ce langage de manipulation de chaînes de caractères.

QuickFill. Notre proposition *QuickFill* part du même problème initial que *FlashFill* : à partir d'un ensemble d'exemples, il faut apprendre un ensemble des programmes pouvant transformer l'entrée σ en la sortie s. Le but est ici de réduire la taille de l'espace de recherche à explorer (gain en temps d'exécution), mais aussi d'apprendre plus rapidement et mieux : c'est à dire avoir besoin de moins d'exemples que l'approche *FlashFill* originale pour apprendre des programmes "corrects". Ici "corrects" signifie que les programmes capturent bien l'intention de l'utilisateur. En effet beaucoup de programmes trouvés par *FlashFill* fonctionnent sur les exemples fournis mais ne généralisent pas correctement. Ils ne donnent donc pas les résultats attendus sur d'autres entrées.

Pour cela, *QuickFill* demande à l'utilisateur, pour chaque exemple $(\sigma = (\sigma_1, ..., \sigma_n), s)$, d'identifier dans s l'ensemble de sous-chaînes $\{s_1, ..., s_k\}$ impactées par l'entrée σ, et pour chaque sous-chaîne s_i, d'identifier l'élément de l'entrée lui étant associée : au moins une chaîne σ_j, voire une sous-chaîne σ_j^{sub} de σ_j. Les sous-chaînes de s n'étant pas impactées par l'entrée s sont des constantes. Ces informations permettent de guider plus finement l'exploration de l'espace de recherche en comparaison avec l'algorithme *FlashFill*. Cela évite la génération de nombreux programmes trop spécifiques, et permet d'accélérer le temps de calcul, comme nous le montrerons en Section 4.

Il est important de noter que dans cet article, nous nous basons sur l'algorithme *FlashFill* tel que défini dans (Gulwani, 2011), et pas sur la version qui peut être testée dans Excel 365 (par exemple) : l'algorithme de cette dernière contient des heuristiques non publiées (entre autres

améliorations). Par ailleurs, notre but étant de tester l'intérêt d'ajouter certaines interactions avec l'utilisateur et pas de reproduire *FlashFill* dans toute sa généralité, nous ne considérons pas le constructeur $Loop$ du langage de manipulation de chaînes de caractères (qui permet de gérer des chaînes pouvant être découpées en un nombre arbitraire de tokens). Nous ne considérons pas non plus le constructeur $Switch$, qui permet de gérer des conditions dans les programmes (ce qui signifie que les exemples se partitionnent en plusieurs "types", chaque type étant résolu par un programme de transformation de chaînes différent). Cette version simplifiée de *FlashFill* est appelée *FlashFill (simplified)* dans la suite.

3 Interface utilisateur de *Quickfill*

La figure 2 présente l'interface de l'outil. Nous avons la possibilité de charger nos fichiers à partir de "File" puis d'exécuter *QuickFill* ou *FlashFill* (simplified).

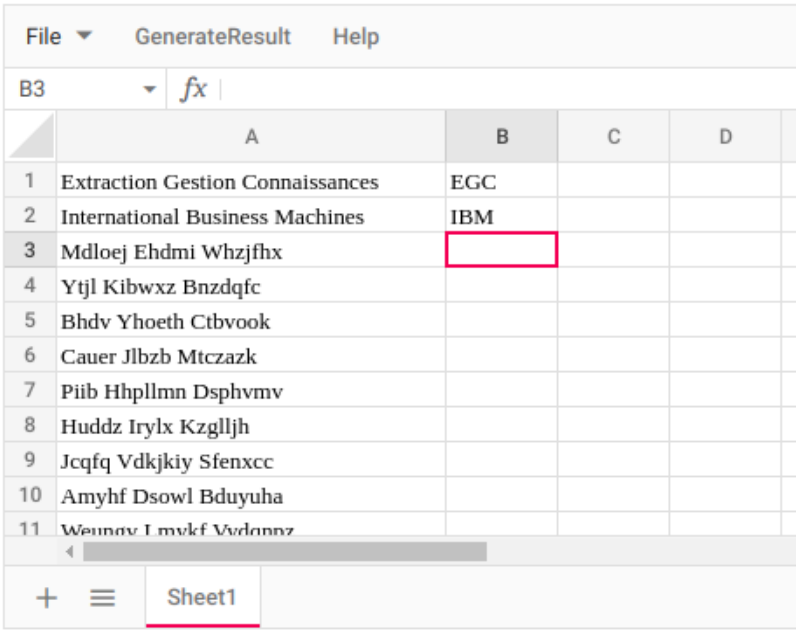

FIG. 2 – *Interface utilisateur.*

La figure 3 montre une exécution de *FlashFill*. Pour une exécution de *FlashFill*, on a à gauche, une fenêtre présentant des informations concernant l'exécution de *FlashFill* (temps d'exécution, nombre de programmes et espace mémoire utilisé). À droite nous avons une feuille de calcul où la colonne B est remplie automatiquement par l'un des programmes inférés à partir de l'exemple d'entraînement (cellules $(A1, B1)$). En bas de la feuille de calcul, la liste des programmes inférés à partir de l'exemple d'entraînement est affichée ainsi que le nombre de programmes.

Pour une exécution de *QuickFill*, il y a une étape supplémentaire à faire, celle du *matching* entre les sous-parties de la sortie et de l'entrée dans l'exemple. La figure 4 présente cette étape. Dans ce cas il s'agit d'utiliser l'exemple (" $Extraction\ Gestion\ Connaissances$ ", " EGC ") pour inférer les programmes permettant d'obtenir un sigle à partir de sa définition. Le *matching* se matérialise par les blocs, par exemples " E " $=>$ " $Extraction$ " pour dire que la sous-partie " E " provient de la sous-partie " $Extraction$ " de l'entrée. Les résultats d'exécution s'affichent comme le montre la figure 3 dans le cas de *FlashFill*.

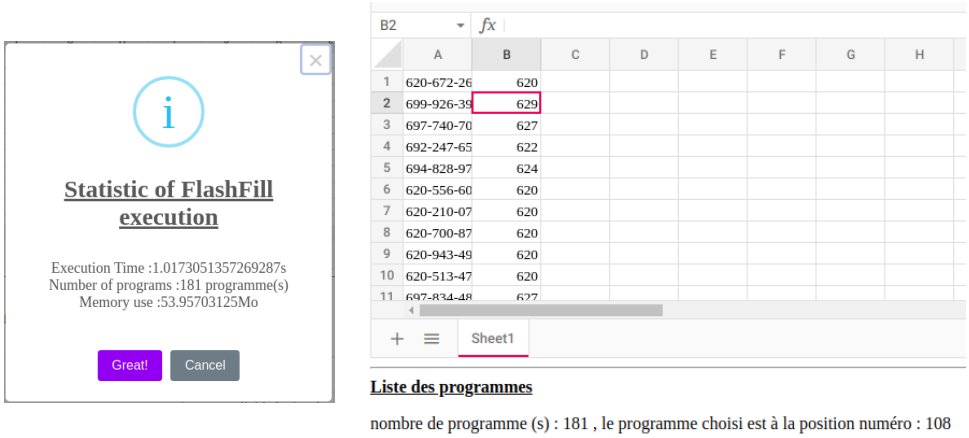

FIG. 3 – *L'exécution de FlashFill génère 181 programmes. Parmi eux un programme est sélectionné arbitrairement puis appliqué sur les valeurs de la colonne A pour compléter automatiquement la colonne B.*

FIG. 4 – *Interface de matching entre les sous-parties de la sortie et de l'entrée.*

4 Expériences

Afin de montrer l'intérêt de *Quickfill*, nous avons mené une série d'expériences dans l'outil présenté précédemment. Nous avons ainsi pu mesurer le gain obtenu par rapport à *FlashFill*. Les expérimentations ont été menées sur une machine Core i3 2.1 GHz dotée d'une mémoire RAM de 8GO. Les méthodes *Quickfill* et *FlashFill* (simplified) ont été implémentées en Python. Le code est disponible sur github [1].

Jeux de données. Les 16 jeux de données utilisés [2] ont été obtenus en adaptant les cas traités dans l'article de *FlashFill*. La génération de ces jeux de données s'est faite via un générateur aléatoire de mots à partir d'expressions régulières [3]. Chaque jeu de données contient 30 éléments dont les premiers (couples $(entrée, sortie)$) représentent les exemples d'entraînement (i.e., à partir desquels les programmes sont appris) et le reste représente les exemples de tests (i.e., les entrées sur lesquels les programmes générés sont testés). Les jeux de données couvrent plusieurs types de tâches comme l'extraction des prénoms à partir des adresses mails, l'extraction des jours dans les dates ou encore l'extraction des initiales d'un sigle.

Évaluation du nombre de programmes. On cherche à comparer le nombre total de programmes produits par *FlashFill* par rapport au nombre de programmes produits par *QuickFill*. Les programmes sont obtenus pour chaque jeu de données et pour un nombre d'exemples allant de 1 à 4. La figure 5 permet de visualiser les différences entre le nombre total de programmes générés par *QuickFill* (*QT*) et par *FlashFill* (*FT*) sur les 16 jeux de données, en ne donnant qu'un seul exemple. Cela permet aussi de comparer le nombre de programmes corrects générés par *QuickFill* (*QC*) et par *FlashFill* (*FC*). Notons l'échelle logarithmique en Y (nombre de programmes). On observe que *QuickFill* produit entre deux et trois ordres de grandeur de programmes en moins que *FlashFill* et que dans plus de la moitié des jeux de données, tous les programmes produits par *QuickFill* sont corrects. Ceci confirme expérimentalement la capacité de *QuickFill* à réduire significativement le nombre de programmes inférés par rapport à *FlashFill*.

Évaluation du taux de programmes corrects. La figure 6 permet de visualiser la variation des taux de programmes corrects dans *FlashFill* et *QuickFill* par jeu de données en fonction du nombre d'exemples donnés. Cette figure se limite aux jeux de données $\{B1, B2, B6, B7\}$ par manque de place ; des résultats similaires sont observés sur les autres jeux de données. Cette expérience montre que pour un nombre d'exemples donné, *QuickFill* a toujours un taux de programmes corrects supérieur ou égal à celui de *FlashFill*. En pratique, cela signifie qu'il faut souvent moins d'exemples à *QuickFill* pour découvrir les programmes capturant l'intention de l'utilisateur. Ceci valide notre hypothèse de départ, qui était que demander un peu plus de travail à l'utilisateur sur chaque exemple lui permettrait souvent d'avoir moins d'exemples à renseigner.

5 Conclusion

La synthèse de programmes est une solution de plus en plus utilisée pour décharger l'utilisateur de tâches fastidieuses comme certaines transformations de données. La difficulté de ce

1. `https://github.com/vanes11/InteractQuickFill`
2. `https://github.com/vanes11/InteractQuickFill/tree/master/Benchmarks`
3. `https://onlinerandomtools.com/generate-random-data-from-regexp`

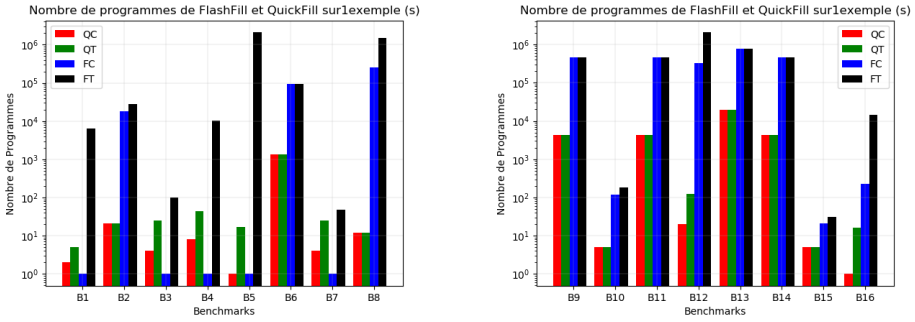

FIG. 5 – *À partir d'un exemple, nombre total de programmes générés par QuickFill (QT), nombre de programmes corrects générés par QuickFill (QC), nombre total de programmes générés par FlashFill (FT), nombre de programmes corrects générés par FlashFill (FC).*

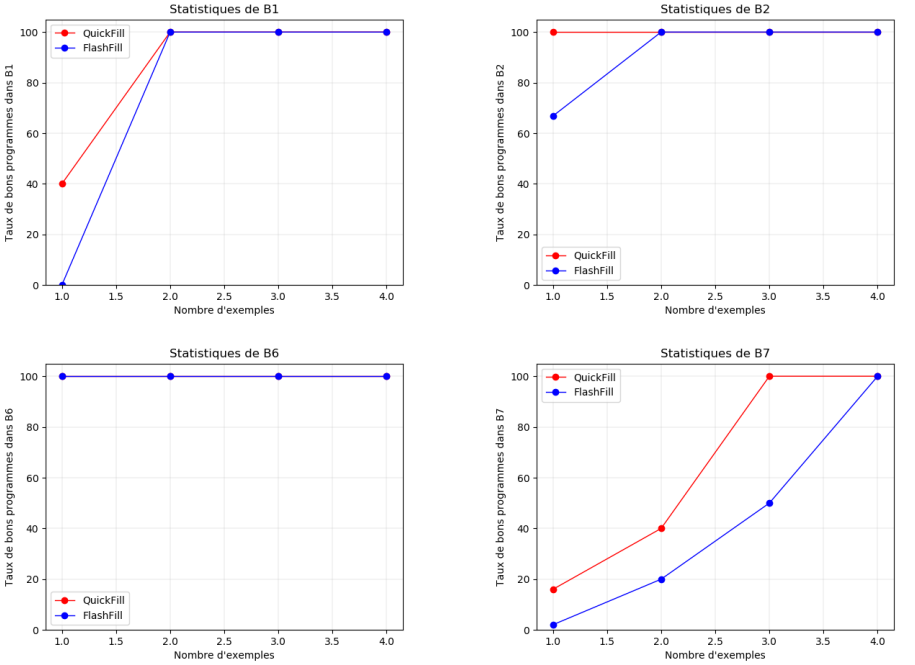

FIG. 6 – *Variation du taux de programmes corrects en fonction du nombre d'exemples dans les benchmarks B1, B2, B6, B7.*

domaine est de naviguer dans un espace de recherche immense, tout en limitant au maximum les interactions avec l'utilisateur (en général il est juste autorisé de lui demander quelques exemples). Dans ce papier de démonstration, nous avons présenté l'approche *QuickFill*, qui demande à l'utilisateur des interactions plus soutenues sur les exemples qu'il fournit, mais en contrepartie peut arriver plus vite à des programmes correspondant à son intention : à la fois en moins de temps de calcul, et via moins d'exemples. Ce compromis nous semble intéressant, et la démonstration montre un premier prototype simple d'interface utilisateur.

Une amélioration non triviale serait de lancer l'exploration de l'espace de recherche dès le premier *mapping* sous-chaîne de sortie/sous-chaîne d'entrée fourni par l'utilisateur, afin d'essayer de prédire les autres *mappings* que va fournir l'utilisateur. Si cette prédiction est juste alors on aura réduit une partie du temps demandé à l'utilisateur pour notre approche.

Références

Gulwani, S. (2010). Dimensions in program synthesis. In *Proc. of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pp. 13–24.

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices 46*(1), 317–330.

Gulwani, S. (2016). Programming by examples. *Dependable Software Systems Engineering 45*(137), 3–15.

Gulwani, S., O. Polozov, R. Singh, et al. (2017). Program synthesis. *Foundations and Trends® in Programming Languages 4*(1-2), 1–119.

Nguyen, H. D. T., D. Qi, A. Roychoudhury, et S. Chandra (2013). Semfix : Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pp. 772–781. IEEE.

Nori, A. V., S. Ozair, S. K. Rajamani, et D. Vijaykeerthy (2015). Efficient synthesis of probabilistic programs. *ACM SIGPLAN Notices 50*(6), 208–217.

Perelman, D., S. Gulwani, T. Ball, et D. Grossman (2012). Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 275–286.

Summary

In order to facilitate the execution of repetitive tasks, program synthesis approaches have been developed. Program synthesis consists in automatically inferring programs that satisfy a user's intention. *FlashFill*, in Excel, is the most famous. In *FlashFill* the user's intention is represented by examples, i.e. pairs $(input, output)$. *FlashFill* explores a very large search space of programs and thus may require a large runtime and infer many programs, some of which work on the given examples but do not capture the user's intent. In this paper we present *Quickfill*, an approach that reduces the space of programs to infer by enriching the user specifications. We show that with this approach, it is often possible to give fewer examples than with the original *FlashFill* algorithm, and that the proportion of correct programs obtained is greater in most cases.