Le principe MDL au service de l'automatisation de tâches uniques d'abstraction et de raisonnement (ARC) à partir de peu d'exemples

Sébastien Ferré

Le principe MDL au service de l'automatisation de tâches uniques d'abstraction et de raisonnement (ARC) à partir de peu d'exemples

Sébastien Ferré*

*Univ Rennes, CNRS, IRISA
Campus de Beaulieu, 35042 Rennes, France
ferre@irisa.fr

Résumé. Le challenge ARC (*Abstraction and Reasoning Corpus*) a été proposé pour pousser la recherche en IA vers plus de capacité de généralisation plutôt que vers toujours plus de performance. C'est une collection de tâches uniques où il s'agit d'apprendre à générer une grille colorée en fonction d'une autre, et ce à partir de quelques exemples seulement. En contraste avec les programmes transformatifs proposés par les approches existantes, nous proposons des modèles centrés-objets analogues aux programmes naturels produits par des humains. Le principe MDL (*Minimum Description Length*) est exploité pour une recherche efficace dans le vaste espace des modèles. Nous obtenons des résultats encourageants avec une classe de modèles simples: des tâches variées sont résolues et les modèles appris sont proches des programmes naturels.

1 Introduction

L'Intelligence artificielle (IA) a fait des progrès spectaculaires sur certaines tâches, parfois au-delà des performances humaines : ex., reconnaissance d'images, jeux de plateau, traitement automatique de la langue (Devlin et al., 2019). Cependant, l'IA manque encore de la flexibilité de l'intelligence humaine pour s'adapter à de nouvelles tâches à partir de peu d'exemples. Pour pousser la recherche en IA dans ce sens, Chollet (2019) a introduit une mesure de l'intelligence qui valorise l'*efficacité d'acquisition* d'aptitudes plutôt que la *performance* dans ces aptitudes. Autrement dit, la quantité de connaissances a priori et d'expérience qu'un agent nécessite pour atteindre un niveau correct dans toute une famille de tâches (ex., les jeux de plateaux) compte plus que la performance atteinte dans n'importe quelle tâche particulière (ex., les échecs).

Chollet a aussi introduit le *challenge* ARC (*Abstraction and Reasoning Corpus*), une forme de test psychométrique visant à mesurer et comparer l'intelligence des machines comme des humains. ARC est une collection de tâches qui consistent à apprendre à générer une grille colorée à partir d'une autre grille colorée, et ce à partir de quelques exemples seulement. C'est un *challenge* très difficile : le gagnant d'une compétition Kaggle [1] n'a pu résoudre que 20% des tâches (avec beaucoup de codage en dur et une recherche brute) alors que les humains peuvent résoudre plus de 80% des tâches (Johnson et al., 2021).

1. https://www.kaggle.com/c/abstraction-and-reasoning-challenge

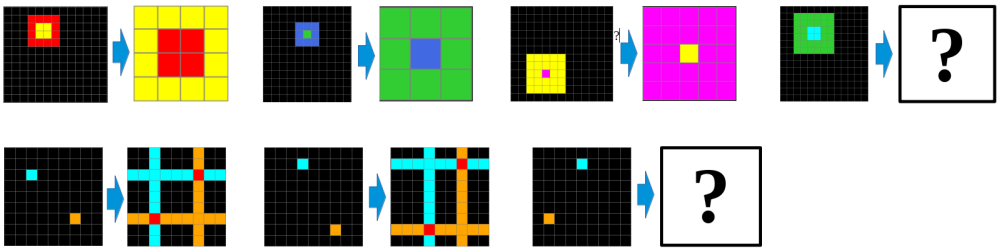

FIG. 1 – *Tâches d'entrainement* `b94a9452` *(en haut) et* `23581191` *(en bas).*

Les approches existantes (Fischer et al., 2020; Alford et al., 2021) font de la *synthèse de programme* où un programme est une composition de transformations primitives opérant sur des grilles, et où l'apprentissage se fait par recherche dans le vaste espace des programmes. En revanche, les études psychologiques ont montré que les *programmes naturels* produits par les humains sont centrés-objets et déclaratifs plutôt que procéduraux (Johnson et al., 2021; Acquaviva et al., 2021). Quand on demande aux humains de verbaliser des instructions permettant de résoudre une tâche, ils commencent par décrire la grille d'entrée en terme d'objets puis comment générer la grille de sortie à partir des ces objets.

Nous apportons deux contributions par rapport aux approches existantes :

1. des *modèles de grilles* qui permettent à la fois d'analyser et de générer des grilles en terme d'objets et de calculs sur ces objets ;
2. une recherche efficace parmi les modèles de grilles avec le *principe MDL (Minimum Description Length)* (Rissanen, 1978).

Le principe MDL vient de la théorie de l'information et dit que le modèle qui décrit le mieux des données est celui qui les compresse le plus (Rissanen, 1978; Grünwald et Roos, 2019), appliqué par exemple en fouille de données (Vreeken et al., 2011; Faas et Leeuwen, 2020).

Nous présentons des résultats encourageants avec des modèles de grilles simples et loin de couvrir toutes les connaissances a priori supposées connues dans ARC (Chollet, 2019). Des modèles corrects sont trouvés pour 61 tâches variées. La plupart de ces modèles sont similaires aux programmes naturels produits par les humains. Un dépôt GitHub [2] fournit le code source et un rapport technique contenant des détails techniques et un historique de versions montrant notre progression dans le challenge ARC.

Le papier est organisé comme suit. La section 2 présente le challenge ARC et la section 3 les approches existantes. La section 4 définit nos modèles centrés-objet et la section 5 explique comment les apprendre avec le principe MDL. La section 6 présente les résultats expérimentaux et les compare avec les approches existantes.

2 ARC – *Abstraction and Reasoning Corpus*

ARC est une collection de tâches [3], où chaque tâche est constituée d'exemples d'apprentissage (en moyenne 3.3) et d'exemples de test (1 parfois 2). Chaque exemple est composé d'une

2. `https://github.com/sebferre/ARC-MDL`
3. Données et interface de test à `https://github.com/fchollet/ARC`

grille d'entrée et d'une grille de sortie. Chaque grille est une matrice (de taille 1x1 à 30x30) remplie d'entiers codant des couleurs (10 couleurs distinctes). Pour une tâche donnée, la taille des grilles peut varier d'un exemple à l'autre et entre l'entrée et la sortie. Chaque tâche est un problème d'apprentissage dont le but est d'apprendre un programme ou modèle qui génère la grille de sortie à partir de la grille d'entrée, et ce à partir de seulement quelques paires de grilles comme exemples. Le critère de succès est que toutes les grilles de sorties test soient identiques aux grilles attendues (à la cellule près). Cependant, trois tentatives par grilles sont permises pour chaque grille test pour compenser les éventuelles ambiguités dans la définition d'une tâche. La figure 1 montre deux tâches ARC (sans les grilles de sortie test attendues). La première sert d'exemple courant dans la suite du papier. Le challenge ARC est composé de 1000 tâches : 400 "tâches d'entrainement" [4] pour le développement d'IA, 400 tâches d'évaluation et 200 tâches secrètes pour des évaluations indépendantes.

3 Approches existantes

ARC est un challenge récent et difficile et peu d'approches ont été publiées jusqu'à présent. Toutes celles dont nous avons connaissance définissent un DSL (*Domain-Specific Language*) de programmes qui transforment une grille en une autre grille, et cherchent un programme qui soit correct sur les exemples d'apprentissage (Fischer et al., 2020; Alford et al., 2021). Leurs différences se trouvent dans les transformations primitives considérées (connaissances a priori) et dans la stratégie de recherche. Il est tentant de définir toujours plus de primitives pour résoudre plus de tâches, comme le gagnant Kaggle l'a fait, mais cela implique une moindre intelligence d'après la mesure de Chollet. Pour guider la recherche dans le vaste espace des programmes, Fischer et al. (2020) utilisent un algorithme évolutionniste grammatical, et Alford et al. (2021) utilisent des réseaux de neurones. Une difficulté est que la grille de sortie est généralement utilisée seulement pour l'évaluation d'un programme candidat, ce qui rend la recherche comme aveugle. Alford et al. (2021) réduisent cette difficulté avec une recherche bidirectionnelle, partant des deux grilles à la fois. Ces approches ont un problème de passage à l'échelle car l'espace de recherche croit de façon exponentielle avec la taille du DSL. Nous comparons leur performance sur ARC dans la section 6.

Johnson et al. (2021) rapportent une étude psychologique sur ARC qui révèle que les humains utilise des représentations mentales à base d'objets. C'est en contraste avec les programmes à base de transformations des approches existantes. De façon intéressante, les tâches qui sont perçues comme les plus difficiles par les humains – celles basées sur l'algèbre de Boole (ex., un ou exclusif entre deux sous-grilles) et les symétries (ex., rotations, miroirs) – sont précisément celles qui sont les mieux traitées par les approches existantes. L'étude fait ressortir deux défis : (1) un large ensemble de primitives semble inévitable, surtout en géométrie ; et (2) une approche centrée-objet entraine le problème d'identifier les objets dans une grille, lesquels peuvent être partiellement visibles du fait d'occlusions entre objets. Une ressource précieuse est LARC (*Language-annotated ARC*) (Acquaviva et al., 2021), collectée par *crowd-sourcing*. Elle fournit pour la plupart des tâches d'entrainement un ou plusieurs *programmes naturels*. Ce sont de courtes descriptions textuelles produites par des participants et ayant permis à d'autres

4. Le terme "entrainement" est trompeur car ces tâches visent à l'entrainement des développeurs d'IA, pas à l'entrainement des systèmes d'IA. Les tâches ARC peuvent être résolues par les humains sans entrainement spécifique.

participants de générer les grilles de sortie attendues sans accès aux exemples d'apprentissage. Ces programmes naturels confirment l'aspect centré-objet des représentations humaines.

Au-delà du challenge ARC, le domaine de la *synthèse de programme* étudie l'apprentissage de programmes à partir de quelques exemples d'entrée-sortie (Lieberman, 2001), par exemple pour remplir automatiquement une colonne d'un tableau à partir des autres colonnes (Gulwani, 2011). Là encore, les programmes sont généralement des compositions de fonctions.

4 Modèles centrés-objet pour les grilles ARC

Nous introduisons des *modèles centrés-objet* pour décrire les grilles ARC en termes d'objets, ayant différentes formes, couleurs, tailles et positions. Ces modèles sont utilisés à la fois pour *analyser* une grille, c'est-à-dire comprendre son contenu selon le modèle et pour *générer* une grille en utilisant le modèle comme patron.

4.1 Des combinaisons de motifs et de fonctions comme modèles

Le principe d'un modèle de grille est de distinguer les éléments invariants des élements variants entre les différentes grilles d'une tâche. Dans la tâche `b94a9452`, toutes les grilles d'entrée contiennent un objet carré mais leurs tailles, couleurs et positions varient. Cela peut s'exprimer par un *motif* **Square**(size: ?, color: ?, pos: ?), où **Square** est appelé un *constructeur* et les points d'interrogation représentent des *inconnues* (similaires aux variables Prolog). Il y a aussi des constructeurs pour les positions en tant que vecteur 2D, **Vec**(i: ?, j: ?), et des valeurs primitives pour les tailles (ex., 3) et les couleurs (ex., blue). Les motifs peuvent être imbriqués, comme dans **Square**(3, ?,**Vec**(?,2)) qui signifie "un carré de taille 3 dont le coin en haut à gauche est sur la colonne d'indice 2", de façon à avoir des modèles aussi spécifiques que nécessaire. Les motifs sans inconnue sont appelés des *descriptions*, par exemple **Square**(3,blue,**Vec**(2,4)).

Les motifs seuls ne permettent pas de faire dépendre la grille de sortie de la grille d'entrée, ce qui est indispensable pour résoudre les tâches ARC. Nous ajoutons donc deux ingrédients aux modèles de grille (en pratique, aux modèles de sortie) : des *références* aux éléments d'une description (en pratique, la description de la grille d'entrée) et des applications de *fonctions* pour effectuer des calculs sur ces éléments. Dans l'exemple, le modèle pour le petit carré dans les grilles de sortie pourrait être $\mathbf{Square}(!small.size, !large.color, !small.pos - !large.pos)$, où par exemple le chemin $!small.size$ est une référence à la taille du petit carré dans la grille d'entrée. Ce modèle dit que "le petit carré en sortie a la taille du petit carré en entrée, la couleur du grand carré et que sa position est la différence entre les positions des deux carrés."

Les figures 2, 3 listent respectivement les constructeurs et fonctions des modèles de grille que nous avons utilisé dans nos expériences. Chaque constructeur/fonction a un type résultat et des arguments typés. Les types des arguments contraignent les constructeurs/fonctions pouvant être utilisés, et les noms des arguments des constructeurs sont utilisés pour référencer les éléments d'un modèle ou d'une description.

Nos modèles de grille décrivent une grille comme ayant une certaine taille, une certaine couleur de fond et une pile de couches (*layers*), où chaque couche contient un objet. Un objet a une forme et une position. Une forme est un point coloré ou est défini par un masque coloré inscrit dans un rectangle. Un masque est soit un bitmap quelconque soit une forme commune

type	constructeurs
Grid	**Grid**(size: *Vector*, color: *Color*, layers: *Object*[])
Object	**PosShape**(pos: *Vector*, shape: *Shape*)
Shape	**Point**(color: *Color*) **Rectangle**(size: *Vector*, color: *Color*, mask: *Mask*)
Vector	**Vec**(i: *Int*, j: *Int*)
Mask	**Bitmap**(bitmap : *Bitmap*) **Full**, **Border**, ...

FIG. 2 – *Constructeurs par type*

Arithmétique : addition et soustraction ; multiplication et division par une petite constante (2..3) ; minimum, maximum et moyenne de deux entiers ; écart entre deux positions ($\|x - y\| + 1$) ; versions vectorisées des fonctions précédentes (e.g., $(i_1, j_1) + (i_2, j_2) = (i_1 + i_2, j_1 + j_2)$) ; projection d'un vecteur sur un axe.
Géométrie : aire d'une forme ; positions extrêmes et médianes d'un objet selon chaque axe (ex., haut et bas, milieu) ; vecteur de translation d'un objet contre un autre ; mise à l'échelle d'un masque ou d'une forme d'un facteur constant ou par rapport à un vecteur taille ; extension d'un masque, forme ou objet à une certaine taille en respectant un motif périodique (ex., en damier) ; tuilage d'un masque ou d'une forme un petit nombre de fois selon les deux axes ; application de symétries aux masques, formes et objets (combinaisons de rotations et de réflexions).
Autres fonctions : recoloration d'une forme ou d'un objet ; opérations logiques sur les masques bitmaps.

FIG. 3 – *Fonctions par catégorie*

telle qu'un rectangle plein ou un contour de rectangle (*border*). Les positions et tailles sont des vecteurs 2D d'entiers. Trois types primitifs sont employés : les entiers, les couleurs et les bitmaps. Les fonctions disponibles couvrent pour l'essentiel des opérations arithmétiques sur les entiers et les vecteurs représentant des positions, des tailles et des déplacements ; et des notions géométriques telles que mesures (ex., aire), translations, symétries, mises à l'échelle et motifs périodiques (ex., tuilage). Les inconnues sont ici limitées aux types primitifs et aux vecteurs et ne peuvent pas apparaitre comme argument de fonction. Les références et les fonctions ne sont utiles que dans les modèles de sortie.

La figure 4 montre un modèle correct pour la tâche `b94a9452`. Ce modèle peut se lire : *"Trouve deux rectangles pleins empilés sur un fond noir dans la grille d'entrée. Puis génère une grille de sortie dont la taille est celle de l'objet de dessous (lay[1]) et dont la couleur de fond est la couleur de l'objet de dessus (lay[0]). Enfin, ajoute un rectangle plein dont la taille est celle de l'objet de dessus, dont la couleur est celle de l'objet de dessous, et dont la position est la différence entre celles des deux objets."*

$M^i =$ **Grid**(?, black, [
PosShape(?, **Rectangle**(?, ?, **Full**)),
PosShape(?, **Rectangle**(?, ?, **Full**))])
$M^o =$ **Grid**(!lay[1].shape.size, !lay[0].shape.color, [
PosShape(!lay[0].pos - !lay[1].pos, **Rectangle**(!lay[0].size, !lay[1].color, **Full**))])

FIG. 4 – *Un modèle correct pour la tâche* `b94a9452`.

4.2 Analyse et génération de grilles selon un modèle

Nous introduisons deux opérations qui doivent être définies pour tout modèle M de grille : l'*analyse* d'une grille g en une description π et la *génération* d'une description π et donc d'une grille g. Ces opérations sont analogues à l'analyse syntaxique et à la génération de phrases à partir d'une grammaire, où les arbres syntaxiques correspondent aux descriptions π.

Dans les deux opérations, les références présentes dans le modèle M sont d'abord résolues en utilisant une description π comme *environnement* ε, c-à-d. comme contexte d'évaluation. Concrètement, chaque référence est un chemin dans ε et est remplacée par la sous-description au bout de ce chemin. Les éventuelles fonctions s'appliquant à ces références sont alors évaluées. Il en résulte un modèle réduit M' seulement constitué de motifs.

Analyse. L'analyse d'une grille g consiste à remplacer les inconnues du modèle réduit M' par des descriptions correspondant au contenu de la grille. Il n'est pas nécessaire que tout le contenu de la grille soit décrit, ce qui autorise des modèles partiels. Une grille est analysée du dessus vers le dessous pour prendre en compte les superpositions d'objets. L'analyse d'un objet est contextuelle, elle dépend de ce qui reste à couvrir dans la grille après l'analyse des couches supérieures. Pour des raisons d'efficacité, chaque grille est pré-traitée pour en extraire une collection de parties unicolores et les objets sont analysés comme des unions de ces parties. Comme l'analyse des grilles peut devenir combinatoire, nous majorons le nombre de descriptions produites par l'analyse et nous les ordonnons d'après les mesures de longueurs de description définies en section 5. À titre d'exemple, l'analyse de la première grille d'entrée de la tâche `b94a9452` avec le modèle M^i de la figure 4 trouve la meilleure description suivante : π^i = **Grid**(**Vec**(12,13), black, [**PosShape**(**Vec**(2,4), **Rectangle**(**Vec**(2,2), yellow, **Full**)), **PosShape**(**Vec**(1,3), **Rectangle**(**Vec**(4,4), red, **Full**))]).

Génération. La génération d'une grille consiste à remplacer les éventuelles inconnues restantes dans le modèle réduit M' par des decriptions aléatoires du bon type, de façon à obtenir une description de grille, qui peut ensuite être convertie en grille concrète. Par exemple, le modèle de sortie M^o de la figure 4 appliqué avec la description π^i ci-dessus de la première grille d'entrée génère la description suivante : π^o = **Grid**(**Vec**(4,4), yellow, [**PosShape**(**Vec**(1,1), **Rectangle**(**Vec**(2,2), red, **Full**))]). Cette description est bien conforme à la grille de sortie attendue.

Un point important est que ces deux opérations retournent des ensembles et non des descriptions uniques. En effet, il existe souvent plusieurs façons d'analyser une grille selon un modèle, par exemple si le modèle mentionne un seul objet alors que la grille en contient plusieurs. Il existe aussi plusieurs grilles pouvant être générées par un modèle dès lors qu'il contient des inconnues. De plus amples détails sur ces opérations sont disponibles dans le rapport technique (voir dépôt GitHub).

4.3 Prédire, décrire et créer des grilles avec des modèles de tâches

Un *modèle de tâche* $M = (M^i, M^o)$ est composé d'un modèle de grille d'entrée M^i et d'un modèle de grille de sortie M^o. Nous démontrons la versatilité de tels modèles en montrant qu'ils peuvent être employés selon trois modes : *prédire* la grille de sortie à partir de la grille d'entrée, *décrire* une paire de grilles de façon conjointe, ou *créer* une nouvelle paire de grilles pour la tâche. Nous utilisons ci-dessous la notation $\pi \in parse(M, \varepsilon, g)$ pour dire que π est une des analyses de la grille g d'après le modèle M et l'environnement ε ; et la notation $\pi, g \in generate(M, \varepsilon)$ pour dire que π est une des descriptions générées par le modèle M avec l'environnement ε, et que g est la grille décrite par π.

Le mode *prédire* est employé quand un modèle a déjà été appris, dans la phase d'évaluation sur les exemples test. Il consiste d'abord à analyser la grille d'entrée avec l'environnement vide nil pour en obtenir une description π^i, puis à générer la grille de sortie en utilisant cette description de la grille d'entrée comme environnement.

$$predict(M, g^i) = \{g^o \mid \pi^i \in parse(M^i, nil, g^i), \pi^o, g^o \in generate(M^o, \pi^i)\}$$

Le mode *décrire* est employé dans la phase d'apprentissage du modèle (voir la section 5). Il permet d'obtenir une description jointe d'une paire de grilles. Il consiste en l'analyse successive de la grille d'entrée et de la grille de sortie. Notons que l'analyse de la grille de sortie dépend du résultat de l'analyse de la grille d'entrée, d'où le terme de description *jointe*.

$$describe(M, g^i, g^o) = \{(\pi^i, \pi^o) \mid \pi^i \in parse(M^i, nil, g^i), \pi^o \in parse(M^o, \pi^i, g^o)\}$$

Le mode *créer* permet de créer un nouvel exemple de la tâche. Il consiste en la génération successive d'une grille d'entrée et d'une grille de sortie, cette dernière étant conditionnée par la première. Ce mode n'est pas employé dans le challenge ARC mais il pourrait contribuer à la mesure de l'intelligence d'un système. En effet, si un agent a vraiment compris une tâche, il devrait être capable de produire de nouveaux exemples.

$$create(M) = \{(g^i, g^o) \mid \pi^i, g^i \in generate(M^i, nil), \pi^o, g^o \in generate(M^o, \pi^i)\}$$

Ces trois modes font apparaitre une différence essentielle entre nos modèles centrés-objet et les programmes transformatifs des approches existantes. Ces derniers sont conçus pour la prédiction (calcul de la sortie en fonction de l'entrée), ils ne fournissent pas une description des grilles. Un nouvel exemple pourrait être créé en générant aléatoirement une grille d'entrée et en lui appliquant le programme mais en général, il ne respecterait pas la plupart des invariants de la tâche (ex., bitmap aléatoire plutôt qu'un carré plein).

5 Apprentissage d'un modèle avec le principe MDL

L'apprentissage avec le principe MDL consiste à chercher le modèle qui compresse le plus les données. Dans le cadre de ARC, les données à compresser sont les exemples d'apprentissage. Il y a donc deux choses à définir : (1) les longueurs de description (abbr. DL) des modèles et des exemples et (2) l'espace de recherche des modèles ainsi que la stratégie d'apprentissage.

5.1 Longueurs de description

Une approche courante de MDL consiste à définir la longueur de description globale comme la somme de deux parties (*two-parts MDL*) : le modèle M et les données D encodées selon le modèle (Grünwald et Roos, 2019).

$$L(M, D) = L(M) + L(D \mid M)$$

Dans notre cas, le modèle est un modèle de tâche composé de deux modèles de grilles ; et les données sont l'ensemble des exemples d'apprentissage. Pour compenser le faible nombre d'exemples et permettre des modèles suffisamment complexes, nous utilisons un *facteur de répétition* $\alpha \geq 1$, comme si chaque exemple était répété α fois.

$$L(M) = L(M^i) + L(M^o)$$
$$L(D \mid M) = \alpha \sum_{(g^i, g^o)} L(g^i, g^o \mid M)$$

La DL d'un exemple est la description jointe la plus compressive d'une paire de grilles.

$$L(g^i, g^o \mid M) = min\{L(\pi^i, g^i \mid M^i, nil) + L(\pi^o, g^o \mid M^o, \pi^i) \mid \pi^i, \pi^o \in describe(M, g^i, g^o)\}$$

Les termes de la forme $L(\pi, g \mid M, \varepsilon)$ dénotent la DL d'une grille g encodée selon un modèle de grille M et un environnement ε, via la description π résultant de l'analyse. On peut décomposer ces termes en se servant de π comme représentation intermédiaire d'une grille.

$$L(\pi, g \mid M, \varepsilon) = L(\pi \mid M, \varepsilon) + L(g \mid \pi)$$

Le terme $L(\pi \mid M, \varepsilon)$ mesure la quantité d'information qui doit être ajoutée au modèle et à l'environnement pour coder la description, typiquement les valeurs des inconnues. Le terme $L(g \mid \pi)$ mesure les différences entre la grille originale et la grille produite par la description. Un modèle correct est obtenu quand $L(\pi^o, g^o \mid M^o, \pi^i) = 0$ pour tous les exemples, c'est-à-dire quand il ne reste plus rien à coder pour les grilles de sorties et donc que les grilles de sorties sont parfaitement prédites.

Trois DL élémentaires doivent donc être définies :

— $L(M)$: la DL d'un modèle de grille ;
— $L(\pi \mid M, \varepsilon)$: la DL d'une description de grille, selon le modèle de grille et l'environnement utilisés pour son analyse ;
— $L(g \mid \pi)$: la DL d'une grille, relativement à une description de grille, c'est-à-dire l'encodage des erreurs et omissions commises par la description.

Nous esquissons ces définitions pour nos modèles de grilles définis en section 4. Nous rappelons que les DL sont généralement dérivées de distributions de probabilités via l'équation $L(x) = -\log P(x)$, qui correspond à un codage optimal (Grünwald et Roos, 2019).

$L(M)$ correspond à l'encodage d'un arbre de syntaxe avec des constructeurs, valeurs, inconnues, références et fonctions comme nœuds. Leur typage pose des contraintes sur les imbrications possibles : ex., le type *Object* a un seul constructeur et pas de fonction. Nous appliquons une distribution uniforme entre constructeurs possibles (resp. entre valeurs ou entre fonctions). Nous appliquons un codage universel pour les entiers pour lesquels il n'y a pas de bornes

connues. Une référence est encodée selon une distribution uniforme parmi tous les éléments de l'environnement qui ont un type compatible. Nous donnons aux inconnues une probabilité plus basse qu'aux constructeurs, et aux références/fonctions une plus haute, afin de favoriser les modèles qui sont plus spécifiques et font dépendre la sortie de l'entrée.

$L(\pi \mid M, \varepsilon)$ correspond à l'encodage des éléments de la description π qui sont inconnus dans le modèle, ou bien qui diverge du modèle. Comme les descriptions forment en fait un sous-ensembles des modèles, les définitions pour $L(M)$ peuvent être réutilisées.

$L(g \mid \pi)$ correspond à encoder quelles cellules de la grille g sont mal spécifiées par la description π. Chaque cellule est codée comme un objet point **PosShape**(**Vec**(i,j),**Point**(c)).

5.2 Espace de recherche et stratégie

L'espace de recherche des modèles est caractérisé par : (1) un modèle initial et (2) un opérateur de *raffinement* qui retourne une liste de raffinements de modèle $M_1 \dots M_n$, à partir d'un modèle M. Un raffinement peut insérer un nouvel élément dans le modèle, remplacer une inconnue par un constructeur (introduisant de nouvelles inconnues pour les arguments du constructeur), ou remplacer un élément par une *expression* (une composition de références, valeurs et fonctions). L'opérateur de raffinement a accès aux données via les descriptions jointes, il peut donc être guidée par elles. Comme dans des approches MDL antérieures en fouille de données (Vreeken et al., 2011), nous adoptons une heuristique gloutonne guidée par la DL des modèles. À chaque étape, en partant du modèle initial, le raffinement réduisant le plus la DL globale $L(M, D)$ est sélectionné. La recherche s'arrête quand aucun raffinement ne permet de réduire la DL. Pour compenser le fait que dans certaines tâches les grilles d'entrée et de sortie ont des tailles très différentes, ce qui peut arrêter précocement la recherche de modèle, nous utilisons comme critère une *DL normalisée* $\hat{L}$ qui donne le même poids aux composantes entrée et sortie de la DL globale $L(M, D)$, avec par convention une valeur de 2 (1+1) pour le modèle initial : $\hat{L}(M, D) = \frac{L(M^i, D^i)}{L(M^i_{init}, D^i)} + \frac{L(M^o, D^o)}{L(M^o_{init}, D^o)}$.

Notre modèle initial utilise le modèle de grille vide **Grid**(?, ?,[]) pour l'entrée et la sortie. Les raffinements disponibles sont :

- l'insertion d'un nouvel objet, à n'importe quelle position dans la pile de couches – pris parmi **PosShape**(?,**Point**(?)), **PosShape**(?,**Rectangle**(?, ?, ?)), **PosShape**(?, !$shape$) ou !$object$ – où $shape$/$object$ est une référence à une forme/objet de l'entrée ;
- le remplacement d'une inconnue au chemin p par un constructeur (resp. une valeur) quand, pour chaque exemple, il existe une description π telle que $\pi.p$ matche ce constructeur (resp. égale cette valeur) ;
- le remplacement d'un élément du modèle au chemin p par une expression e du même type quand, pour chaque exemple, il existe une description π telle que $\pi.p = e$.

La figure 5 montre la trace d'apprentissage de la tâche `b94a9452`, montrant à chaque étape le raffinement le plus compressif. Cette trace révèle comment l'agent apprenant résoud la tâche : *"Il y a un rectangle $lay[1]$ dans l'entrée (1) et un rectangle $lay[0]$ dans la sortie (2). La grille de sortie est de la taille de $lay[1]$ en entrée (3). Il y a un autre rectangle $lay[0]$ dans l'entrée, au-dessus de $lay[1]$ (4). On peut utiliser sa couleur pour le fond de la sortie (5), et sa taille pour $lay[0]$ en sortie (6). La couleur de $lay[0]$ en sortie est la couleur de $lay[1]$ en entrée (7) et sa position est égale à la différence entre les positions des deux rectangles en entrée (9). Tous les rectangles sont pleins (8,10,11) et le fond de l'entrée est noir (12)."*

étape	grille	raffinement	$\hat{L}$
0		(modèle initial)	2.000
1	in	$lay[1] \leftarrow \mathbf{PosShape}(?, \mathbf{Rectangle}(?, ?, ?))$	1.212
2	out	$lay[0] \leftarrow \mathbf{PosShape}(?, \mathbf{Rectangle}(?, ?, ?))$	0.748
3	out	$size \leftarrow !lay[1].shape.size$	0.604
4	in	$lay[0] \leftarrow \mathbf{PosShape}(?, \mathbf{Rectangle}(?, ?, ?))$	0.531
5	out	$color \leftarrow !lay[0].shape.color$	0.445
6	out	$lay[0].shape.size \leftarrow !lay[0].shape.size$	0.367
7	out	$lay[0].shape.color \leftarrow !lay[1].shape.color$	0.310
8	out	$lay[0].shape.mask \leftarrow \mathbf{Full}$	0.277
9	out	$lay[0].shape.pos \leftarrow !lay[0].shape.pos - !lay[1].shape.pos$	0.201
10	in	$lay[0].shape.mask \leftarrow \mathbf{Full}$	0.196
11	in	$lay[1].shape.mask \leftarrow \mathbf{Full}$	0.191
12	in	$color \leftarrow black$	0.187

FIG. 5 – *Trace d'apprentissage de la tâche* `b94a9452`

6 Évaluation

Nous avons évalué notre approche sur les 800 tâches publiques ARC. Les quelques paramètres ont été réglés sur la base des tâches d'entrainement. Pour assurer un partage du temps de calcul entre l'analyse des grilles et la recherche dans l'espace des modèles, nous définissons quelques limites. Le nombre de descriptions produites par l'analyse d'une grille est limité à 64 et seules les 3 plus compressives sont retenues pour le calcul des raffinements. À chaque étape, au plus 10.000 expressions sont considérées et seuls les 20 raffinements qui sont estimés réduire le plus la DL globale sont évalués. Le taux de répétition α est fixé à 10. Les tâches sont traitées indépendamment les unes des autres, sans apprentissage de l'une à l'autre. Les résultats sont donnés pour un temps d'apprentissage par tâche limité à 30s, l'augmenter augmente le temps de calcul sans résoudre davantage de tâches.

Les logs d'apprentissage/prédiction et les images des tâches d'entrainement résolues sont disponibles sur le dépôt GitHub (voir la version 2.5 pour les résultats de ce papier).

Taux de succès. Notre approche résoud 52/400 (13%) tâches d'entrainement et 9/400 (2.25%) tâches d'évaluation. Cela suggère que les tâches d'évaluations sont significativement plus difficiles que les tâches d'entrainement, ce qui a également été observé par d'autres auteurs (Fischer et al., 2020). Bien que trois prédictions soit permises par exemple test, la première prédiction est correcte dans 41 des 52 tâches d'entrainement résolues. Cela montre que les modèles appris sont précis dans leur compréhension des tâches. Pour quelques autres tâches, le système trouve un modèle qui est correct sur les exemples d'apprentissage mais échoue sur au moins un exemple test : 7 tâches d'entrainement et 6 tâches d'évaluation.

Il n'est malheureusement pas aisé de comparer ces résultats à ceux des travaux antérieurs. Le gagnant Kaggle "icecuber" atteint le score impressionnant de 20.6% mais il est mesuré sur 100 tâches secrètes ne faisant pas partie des 800 tâches publiques. Fischer et al. (2020) atteignent un score de 7.68% sur les tâches d'entrainement et de 3% sur les 100 tâches secrètes. Alford et al. (2021) rapportent deux expériences sur de petits sous-ensembles de tâches d'entrainement, choisies pour correspondre à leur DSL. Les taux de succès sont respectivement de

22/36 et 14/18, soit 5.5% et 3.5% des 400 tâches. Sur les tâches d'entrainement, qui offrent à ce jour la meilleure base de comparaison, notre approche est donc celle qui résoud le plus grand nombre de tâches.

Efficacité et complexité des modèles. Le temps moyen d'apprentissage est de 15.1s sur les tâches d'entrainement (avec timeout=30s). Sur les 52 tâches résolues, ce temps moyen chute à 4.9s et le temps médian encore plus bas à 2.1s. Cela montre que quand une solution peut être trouvée, elle est généralement trouvée rapidement. Dans les tâches résolues, le nombre de raffinements va de 5 à 29 (médiane=13). Cela donne une mesure de la complexité des modèles appris. Une telle profondeur dans l'espace de recherche ne pourrait jamais être atteinte dans une recherche par force brute (le gagnant Kaggle se limite à une composition de 4 fonctions). Le principe MDL joue ici un rôle crucial dans le guidage de la recherche.

Modèles appris. Les modèles appris pour les tâches résolues sont très divers malgré la simplicité de leur classe. Ils expriment des transformations diverses : ex., déplacer un objet, échanger deux couleurs, prolonger des lignes, mettre un objet derrière un autre, ordonner des objets du plus grand au plus petit, enlever du bruit. Notons qu'aucune de ces transformations n'est une primitive dans notre classe de modèles, ils sont appris en terme d'objets, d'arithmétique de base, de géométrie simple et de principe MDL.

Nous avons comparé nos modèles appris aux programmes naturels de LARC (Acquaviva et al., 2021). De façon remarquable, pour une majorité de nos modèles, il existe un programme naturel qui peut être considéré comme une reformulation de notre modèle, c'est-à-dire mettant en jeu les mêmes objets et les mêmes opérations. Par exemple, le programme naturel pour la tâche courante `b94a9452` est : *"[L'entrée a] une forme carrée avec un petit carré centré à l'intérieur du grand carré sur un fond noir. Les deux carrés sont de différentes couleurs. Faire une grille de sortie qui est de la même taille que le grand carré. La taille et la position du petit carré intérieur devrait être les mêmes que dans la grille d'entrée. Les couleurs des deux carrés sont échangées."* Pour les autres modèles, certaines notions utilisées par les programmes naturels manquent à notre classe de modèles mais sont compensées par d'autres éléments de nos modèles : ex., des relations topologiques tels que "à côté de" ou "au dessus" (compensées par les trois tentatives), la couleur majoritaire (compensée par le principe MDL sélectionnant le plus gros objet). Cependant, dans la plupart des cas, les mêmes objets sont identifiés.

Ces observations démontrent que nos modèles centrés-objet s'alignent bien avec les programmes naturels produits par des humains, contrairement aux approches basées sur la composition de transformations. Un exemple de programme appris par Fischer et al. (2020) sur la tâche `23b5c85d` est `strip_black; split_colors; sort_Area; top; crop`, qui est une séquences de transformations grille-à-grille, sans mention explicite d'objets.

7 Conclusion et perspectives

Nous avons montré la capacité de notre approche à automatiser des tâches ARC variées et à produire des modèles similaires aux programmes naturels. Des résultats encourageants sont obtenus pour un challenge reconnu comme particulièrement difficile, et ce malgré des modèles simples, loin de couvrir les connaissances a priori supposées dans ARC. Nous croyons que l'extensibilité de notre approche permettra des progrès continus dans le futur. Au-delà de l'ajout de constructeurs et de fonctions, relativement aisé, des verrous scientifiques sont les traitements conditionnels et les itérations sur des collections d'objets. Nous souhaitons

également transposer notre approche à d'autres contextes comme celui du traitement de chaines dans les feuilles de calcul (Gulwani, 2011).

Références

Acquaviva, S., Y. Pu, M. Nye, C. Wong, M. H. Tessler, et J. Tenenbaum (2021). LARC : Language annotated Abstraction and Reasoning Corpus. In *Annual Meeting of the Cognitive Science Society*, Volume 43.

Alford, S., A. Gandhi, A. Rangamani, A. Banburski, T. Wang, S. Dandekar, J. Chin, T. Poggio, et P. Chin (2021). Neural-guided, bidirectional program search for abstraction and reasoning. In *Int. Conf. Complex Networks and Their Applications*, pp. 657–668. Springer.

Chollet, F. (2019). On the measure of intelligence. *arXiv preprint arXiv :1911.01547*.

Devlin, J., M. Chang, K. Lee, et K. Toutanova (2019). BERT : pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, et T. Solorio (Eds.), *Conf. North American Chapter of the Association for Computational Linguistics : Human Language Technologies, NAACL-HLT*, pp. 4171–4186. Assoc. Computational Linguistics.

Faas, M. et M. v. Leeuwen (2020). Vouw : geometric pattern mining using the MDL principle. In *Int. Symp. Intelligent Data Analysis*, pp. 158–170. Springer.

Fischer, R., M. Jakobs, S. Mücke, et K. Morik (2020). Solving Abstract Reasoning Tasks with Grammatical Evolution. In *LWDA*, CEUR-WS 2738, pp. 6–10.

Grünwald, P. et T. Roos (2019). Minimum description length revisited. *arXiv preprint arXiv :1908.08484*.

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In *Symp. Principles of Programming Languages*, pp. 317–330. ACM.

Johnson, A., W. K. Vong, B. M. Lake, et T. M. Gureckis (2021). Fast and flexible : Human program induction in abstract reasoning tasks. *arXiv preprint arXiv :2103.05823*.

Lieberman, H. (2001). *Your Wish is My Command*. The Morgan Kaufmann series in interactive technologies. Morgan Kaufmann / Elsevier.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica 14*(5), 465–471.

Vreeken, J., M. Van Leeuwen, et A. Siebes (2011). Krimp : mining itemsets that compress. *Data Mining and Knowledge Discovery 23*(1), 169–214.

Summary

The ARC (Abstraction and Reasoning Corpus) challenge has been proposed to push AI research towards more generalization capability rather than ever more performance. It is a collection of unique tasks about generating colored grids, specified by a few examples only. We propose object-centered models analogous to the natural programs produced by humans. The MDL (Minimum Description Length) principle is exploited for an efficient search in the vast model space. We obtain encouraging results with a class of of simple models: various tasks are solved and the learned models are close to natural programs.