ÉCOLE POLYTECHNIQUE CONCOURS D'ADMISSION 2026

Filière MPI

Analyse de Documents Scientifiques en Informatique

(120 minutes de préparation et 40 minutes d'interrogation)

Vous trouverez ci-après un dossier comprenant un ou plusieurs articles portant sur un problème informatique, ainsi qu'éventuellement des documents annexes donnant des éléments théoriques sous-jacents à ce problème.

Vous disposez pour analyser ces documents de deux heures de préparation, à l'issue desquelles vous serez interrogé·e pendant 40 minutes. Lors de l'interrogation, il vous est demandé de présenter tout d'abord à l'examinateur un exposé de synthèse de 15 minutes, qui sera suivi d'un entretien de 25 minutes portant sur le contenu scientifique du dossier et sur votre culture générale.

Il vous est rappelé que, conformément à la notice du concours, cette épreuve a pour but de vous permettre de montrer votre esprit critique, vos facultés à mobiliser vos connaissances, à déterminer les problèmes posés, et à examiner les solutions apportées, votre aptitude à collecter, analyser, synthétiser et communiquer l'information, et la qualité de votre expression orale.

Pendant votre préparation, en cas de difficulté, et notamment de problème technique avec la tablette, n'hésitez surtout pas à demander de l'aide aux appariteurs.

DOSSIER N° 02

SUJET POSTÉ EN 2026

L'algorithme A* ou comment calculer rapidement un chemin optimal

Dans le précédent article de la rubrique I.A. [1], j'ai décrit en détail l'algorithme de Dijkstra qui permet de calculer relativement efficacement un chemin optimal dans un graphe. J'ai aussi expliqué comment appliquer cet algorithme au problème du calcul du chemin optimal d'une unité dans un jeu de stratégie temps réel (STR). Nous avons vu en conclusion de l'article que, malgré une implémentation soignée, les temps de calcul théoriques restaient beaucoup trop élevés pour une utilisation en temps réel sur un terrain de jeu étendu et en présence de nombreuses unités.

Une modification subtile et astucieuse de l'algorithme de Dijkstra permet cependant de réduire de façon importante son coût d'exécution. L'algorithme modifié porte le nom d'algorithme A*. Il est massivement utilisé dans les jeux modernes, à tel point que certains spécialistes considèrent que le problème de base du *pathfinding* est réglé (on utilise A*), seules les subtilités (réglages de l'algorithme, choix des structures de données, etc.) restant à traiter pour implémenter un jeu.

Le but du présent article est de décrire l'algorithme A* en illustrant ses performances et en montrant comment les gains de temps qu'il induit (par rapport à l'algorithme de Dijkstra) permettent d'envisager des applications plus ambitieuses. Attention, l'article considère comme acquis les développements de [1] : je vous conseille donc vivement de relire l'article en question !

Les jeux vidéo de stratégie temps réel modernes ont des exigences très élevées en termes de rendu graphique et d'intelligence artificielle (I.A.). Le joueur attend un jeu en 3D construit à partir de milliers de polygones, pour le plaisir de l'œil, mais il veut aussi un adversaire d'un bon niveau, pour le plaisir de l'esprit. La brique de base de l'I.A., le pathfinding, c'est-à-dire le calcul des chemins suivis par les unités, doit donc être rapide et donner de bons résultats. L'algorithme de Dijkstra, présenté dans Linux Mag 53, garantit des résultats exacts, mais est en général trop lent pour fonctionner dans des conditions stressantes (un grand nombre d'unités, un terrain de jeu très grand, etc.). Cependant, des modifications minimes mais très subtiles permettent de conserver l'exactitude des chemins obtenus tout en accélérant notablement leur calcul. L'algorithme obtenu porte le nom d'algorithme A*.

Pourquoi l'algorithme de Dijkstra est-il lent ?

Un exemple de lenteur

Même si l'algorithme de Dijkstra fonctionne correctement, il donne parfois des résultats décevants. Bien entendu, Dijkstra a démontré mathématiquement l'exactitude de l'algorithme : les chemins obtenus sont toujours optimaux. Par contre, l'algorithme met parfois beaucoup de temps à trouver un chemin évident. Pour le voir, considérons un exemple très simple, celui d'une carte vide dans laquelle chaque case possède un coût de franchissement identique (égal à 1, par exemple). Dans ce cas, le chemin optimal est bien entendu la ligne droite qui relie le point de départ au point d'arrivée, comme le montre l'exemple de la figure 1 (le chemin est donné par la flèche noire).

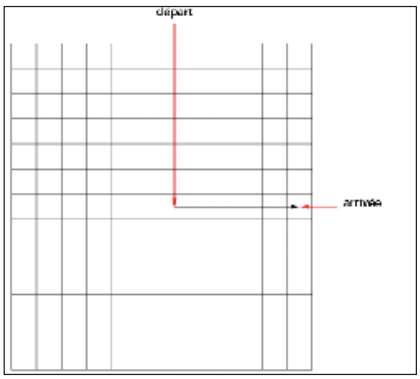


Figure 1

Le problème est que pour arriver à ce chemin optimal, l'algorithme de Dijkstra utilise beaucoup de ressources, comme l'illustre la figure 2. On constate en effet que presque toutes les cases de la carte ont été explorées (elles sont presque toutes dans la liste Fermés), alors qu'intuitivement, il est inutile d'étudier les cases qui éloignent du but !

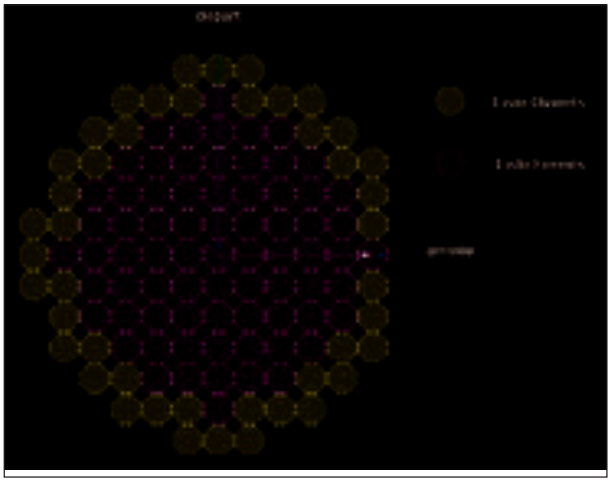


Figure 2

Le comportement de l'algorithme de Dijkstra est pourtant complètement correct. En effet, la carte est traduite en graphe et l'algorithme ne sait pas que certaines cases peuvent le rapprocher du but alors que d'autres l'en éloignent. En fait, l'exploration se fait selon deux critères :

1. Le prochain nœud exploré est celui qui possède le coût cumulé le plus faible dans la liste Ouverts : ce coût cumulé est en quelque sorte la distance entre le point de départ et la case courante. On explore donc la carte en s'éloignant progressivement du point de départ, sans tenir compte de la distance au but.

2. Pour placer de nouveaux nœuds dans la liste Ouverts, l'algorithme étudie les voisins directs du nœud exploré. Encore une fois, la distance au but n'est pas prise en compte. De même, la structure de graphe très particulière de la carte n'est pas exploitée.

Tout ceci fait que l'algorithme de Dijkstra peut être lent, car il explore un grand nombre de nœuds du graphe. Or, l'analyse algorithmique évoquée dans [1] montre que le coût du calcul croît rapidement avec le nombre de cases explorées.

Comment aider l'algorithme ?

On constate que le principal problème de l'algorithme de Dijkstra vient du fait qu'il ne tient pas compte de la distance au but pour choisir le prochain nœud à étudier. En fait, cela est parfaitement normal, car calculer la véritable distance au but est justement ce que fait l'algorithme (indirectement). En effet, si on connaît la distance entre tout point de la carte et la case d'arrivée souhaitée, autrement dit, si on connaît pour tout point de la carte le coût du chemin optimal le reliant à la case d'arrivée, le calcul d'un tel chemin optimal est très simple. Il suffit d'appliquer l'algorithme suivant :

1. point courant = point de départ

2. chercher dans les voisins du point courant le plus proche du point d'arrivée noté pp

3. point courant = pp

4. si le point courant n'est pas le point d'arrivée, retourner en 2

Cet algorithme est extrêmement efficace puisque le temps de calcul est proportionnel à la longueur du chemin. En fait, on ne peut pas faire mieux, mais comme il faut faire tourner un autre algorithme pour obtenir les distances entre tous les points de la carte et le point d'arrivée souhaité, cela n'a pas beaucoup d'intérêt. Notons que l'algorithme de Dijkstra calcule en fait la distance entre le point de départ et (presque) tous les points de la carte (c'est le coût cumulé), ce qui permet de déterminer le chemin optimal avec un algorithme similaire à celui qui vient d'être présenté : la principale différence est qu'on construit le chemin à l'envers, en partant du point d'arrivée et en remontant par les points qui sont les plus proches du point de départ.

On doit donc oublier l'idée de modifier l'algorithme de Dijkstra pour qu'il tienne compte de la véritable distance entre un nœud et le but. Cependant, en utilisant notre connaissance de la structure de la carte, il est possible de donner une estimation de cette distance. Pour la carte vide proposée au-dessus, il suffit par exemple de calculer la distance (au sens classique du terme, c'est-à-dire la distance à vol d'oiseau, plus techniquement, la distance euclidienne) entre le point courant et le but. Bien entendu, la mise en place d'obstacles ou de zones avec des coûts de déplacement distincts rend l'estimation proposée assez fausse, mais ce n'est pas très important (nous verrons dans la suite du texte quelle propriété théorique doit être satisfaite par l'estimation). L'idée n'est pas d'appliquer l'algorithme basique proposé au-dessus, mais d'aider l'algorithme de Dijkstra en lui permettant d'étudier en priorité les nœuds qui le rapprochent du but.

L'algorithme A*

Énoncé

L'algorithme A* [2] est construit sur l'algorithme de Dijkstra associé à une fonction qui estime la distance entre un nœud quelconque du graphe et le nœud cible (par distance, on entend bien sûr le coût du chemin optimal reliant le nœud concerné et le nœud cible). L'idée fondamentale est de modifier le classement dans la liste Ouverts afin de favoriser les nœuds les plus proches du but (au sens de l'estimation).

On suppose donc donnée une fonction `h` qui à un nœud `n` associe `h(n)`, une estimation du coût du chemin optimal

EXTRAIT PROGRESSIF

de n au nœud d'arrivée. Cette fonction s'appelle l'heuristique associée à l'algorithme. On calcule alors un chemin optimal de la façon suivante :

1. créer deux listes vides Ouverts et Fermés

2. insérer dans la liste Ouverts le quadruplet formé du nœud de départ, de la valeur h(départ), d'une valeur arbitraire (null par exemple) comme nœud parent et de la valeur 0 comme coût cumulé, soit donc (départ,h(départ),null,0). Le deuxième élément du quadruplet est une estimation du coût du chemin optimal partant du point de départ et se terminant au nœud arrivée en passant par le premier élément du quadruplet. Cette estimation est obtenue comme la somme du coût cumulé (coût du chemin optimal entre le nœud de départ et le nœud courant) et de l'estimation du coût du chemin optimal reliant le nœud courant au nœud cible

3. tant que la liste Ouverts n'est pas vide :

   a. chercher dans Ouverts le quadruplet (N,f,p,c) dont le coût estimé (f) est minimal (parmi les quadruplets contenus dans la liste)

   b. ajouter le quadruplet à la liste Fermés (on peut se contenter du triplet (N,p,c))

   c. si N est le nœud d'arrivée, quitter la boucle et passer à l'étape 4

   d. supprimer le quadruplet de la liste Ouverts

   e. pour chaque successeur S de N (c'est-à-dire chaque nœud S pour lequel il existe une arête de N à S dans le graphe) :

      i. calculer c' comme la somme de c et du coût associé à l'arête qui relie N à S

      ii. si S est dans Fermés :

         1. si le coût cumulé de S dans Fermés est supérieur à c', supprimer S de Fermés et ajouter le quadruplet (S,c'+h(S),N,c') à Ouverts

         2. sinon passer au successeur suivant

      iii. si S est dans Ouverts avec un coût cumulé supérieur à c', mettre à jour le coût cumulé, le nœud parent (qui est maintenant N) et le coût estimé (qui devient c'+h(S))

      iv. sinon ajouter à Ouverts le quadruplet (S,c',N)

4. si le nœud d'arrivée n'est pas dans Fermés, terminer l'algorithme sur un échec : il n'existe pas de chemin depuis le nœud de départ vers le nœud d'arrivée

5. sinon, reconstruire le chemin en suivant l'information parent dans la liste Fermés

Comparaison avec l'algorithme de Dijkstra

A première vue, les deux algorithmes sont très proches. Il est donc important d'insister sur les deux différences fondamentales :

1. Le classement des nœuds dans la liste Ouverts n'est pas réalisé selon le même critère. En effet, dans l'algorithme de Dijkstra, le prochain nœud exploré est le plus proche du point de départ, ce qui explique le comportement décevant de l'algorithme dans certains cas. Par contre, A* choisit d'explorer le nœud par lequel passe le chemin le plus court menant du point de départ au point d'arrivée. Bien entendu, le coût exact du chemin n'est pas connu, mais grâce à l'heuristique, on possède une bonne estimation de celui-ci. A* favorise donc les nœuds qui mènent au but.

2. Dans l'algorithme de Dijkstra, un nœud placé dans Fermés n'est jamais réexaminé. L'introduction d'un ordre différent pour l'examen des nœuds fait que A* étudie parfois plusieurs fois un même nœud. En effet, il arrive qu'on explore un nœud sans connaître le chemin optimal qui y mène depuis le nœud de départ car les nœuds qui le constituent sont mal "notés" par l'heuristique. De ce fait, le coût cumulé du nœud est faux lors de la première exploration. On doit donc stocker dans Fermés le coût cumulé de chaque nœud afin de pouvoir éventuellement réexaminer les nœuds.

L'implémentation de A* ne pose pas de problème particulier par rapport à celle de l'algorithme de Dijkstra. Les algorithmes sont tellement proches que la traduction d'une implémentation de celui de Dijkstra en A* ne prend que quelques minutes. Les structures de données utilisées sont pratiquement les mêmes et les remarques présentées dans [1] restent valables. Il est donc judicieux d'implémenter Ouverts sous forme d'un tas combiné à des annotations dans la carte. Fermés s'implémente aussi très efficacement et facilement sous forme d'annotation dans la carte. La seule véritable différence avec les solutions présentées dans [1] est que la liste Fermés doit maintenant contenir le coût cumulé de chaque nœud parcouru. L'occupation mémoire de l'algorithme A* est donc potentiellement plus importante que celle de l'algorithme de Dijkstra. Notons qu'il faut aussi pouvoir supprimer un nœud de la liste Fermés, ce qui peut être coûteux dans certaines implémentations (mais pas dans celle proposée dans [1]).

Exemples de mise en œuvre

Pour bien comprendre le fonctionnement de A*, étudions son comportement sur l'exemple de la carte vide proposée dans la section précédente. Pour cette carte, nous choisissons comme heuristique la distance euclidienne entre le centre d'une case et le centre de la case cible. Celle-ci est calculée simplement grâce à la formule suivante : $\sqrt{(x-a)^2+(y-b)^2}$. Dans cette formule, x et y sont les coordonnées du centre de la case considérée, alors que a et b sont celles de la case cible. Dans les deux algorithmes, on commence par insérer le point de départ dans la liste Ouverts. Le coût cumulé est nul et donc, pour A*, le classement se fait à partir de la distance entre le point de départ et le point d'arrivée (5 dans notre exemple). Comme dans les deux algorithmes, la liste Ouverts ne contient qu'un élément, la différence de critère de classement ne

POST-PRO

change rien et on commence donc par étudier les voisins de la case de départ.

Les différences entre les algorithmes apparaissent dès l'étude des voisins. En effet, comme l'illustre la figure 3, les coûts estimés changent radicalement la donne. Sur cette figure, une case contient un nombre en bleu qui est le coût cumulé pour passer optimalement de la case de départ à celle-ci, ainsi qu'un nombre en rouge qui indique le coût estimé pour aller optimalement de la case au but. Pour l'algorithme de Dijkstra, les quatre voisines directes (situées horizontalement et verticalement) de la case de départ sont équivalentes (coût cumulé 1) et l'algorithme étudiera donc une d'entre elles plus ou moins au hasard. Au contraire, A* range les cases selon la somme du coût cumulé et de l'estimation du coût pour aller au but. Les valeurs approchées indiquées sur la figure permettent de voir que la prochaine case étudiée est celle qui est située juste à droite de la case de départ car son coût total est de 5, alors que les autres voisines non diagonales ont des coûts plus élevés (environ 6,10 et 7), de même d'ailleurs que les voisines situées en diagonales (coûts approximatifs 5,5 et 7,5).

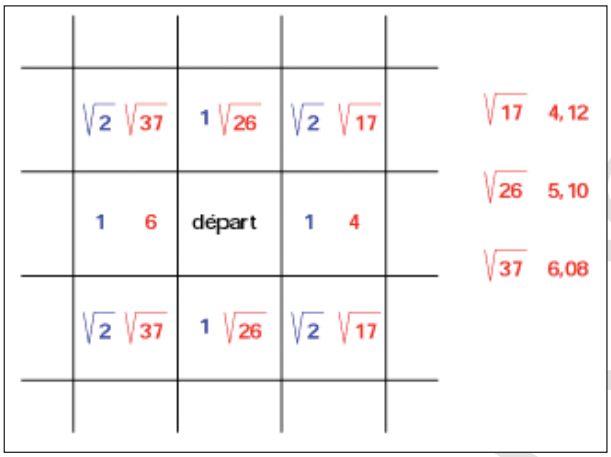


Figure 3

Les différences s'accentuent dès la deuxième case étudiée. Soyons en effet généreux avec Dijkstra et considérons que l'algorithme choisit lui aussi la case située juste à droite de la case de départ. La figure 4 donne les coûts cumulés et les coûts estimés obtenus. Quand on étudie les coûts cumulés, on constate que l'algorithme de Dijkstra va nécessairement étudier les trois autres cases de coût cumulé 1 avant de passer à des cases proches du but.

En effet, les trois nouvelles cases explorées ont toutes un coût cumulé supérieur à 1, ce qui est normal car on s'éloigne du point de départ. Au contraire, on constate que pour A* les choses se présentent sous un bien meilleur jour. En effet, la case située directement à droite de la nouvelle case possède un coût estimé de 5 (2+3), ce qui en fait la nouvelle case prioritaire dans la liste Ouverts. De ce fait, la troisième case étudiée par A* le rapproche encore du but.

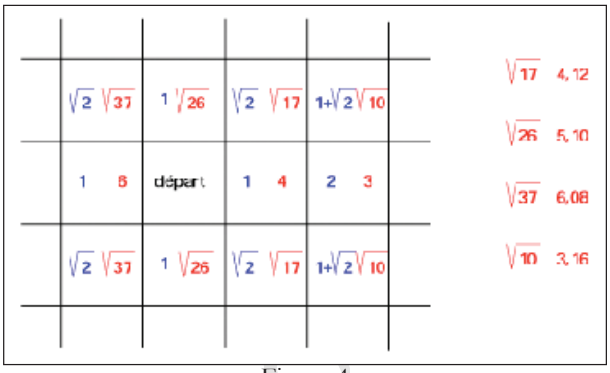


Figure 4

Globalement, la tendance d'A* à se rapprocher du but en priorité donne un résultat très satisfaisant illustré par la figure 5. La comparaison avec la figure 2 est très parlante : l'algorithme de Dijkstra a étudié 71 nœuds alors que A* n'en a étudié que 6 !

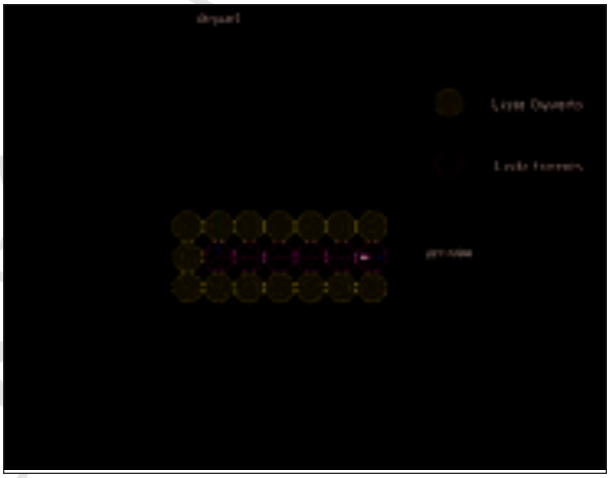


Figure 5

Bien entendu, j'ai choisi un exemple qui est particulièrement favorable à A*, mais en général, il est toujours plus efficace que Dijkstra. En fait, on peut démontrer que si l'heuristique vérifie une propriété théorique que nous verrons dans la section suivante, A* est un algorithme optimal, au sens où tout nœud exploré par A* doit être exploré par tout algorithme capable de trouver un chemin optimal (à condition que l'algorithme n'utilise pas d'autres informations que le graphe et l'heuristique).

La marge de manœuvre réside donc dans le choix de l'heuristique et dans l'implémentation proprement dite. Cette dernière doit être particulièrement soignée pour éviter tout surcoût par rapport à l'algorithme de Dijkstra. Dans certaines situations, les deux algorithmes explorent presque le même nombre de nœuds.

Considérons par exemple la figure 6 qui correspond à une carte de coût uniforme dans laquelle on a ajouté un obstacle infranchissable. On constate que l'algorithme de Dijkstra explore toutes les cases de la carte pour obtenir le chemin optimal.

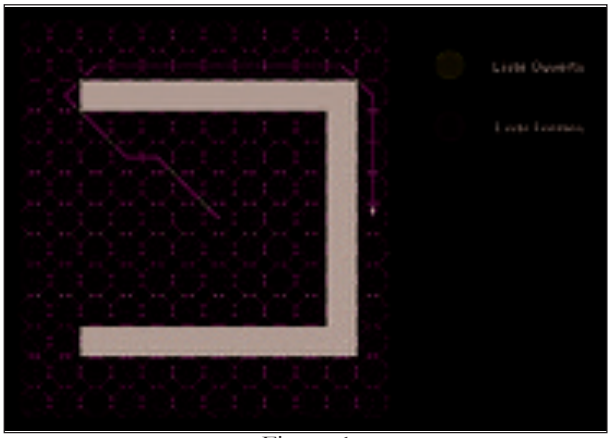


Figure 6

La figure 7 montre que dans une telle situation, A* ne gagne presque rien car il évite d'explorer seulement 10 cases sur 130. Il suffit donc que l'implémentation de A* soit seulement 10% plus lente que celle de l'algorithme de Dijkstra pour que ce dernier prenne l'avantage. Nous remarquons au passage que les chemins trouvés par les deux algorithmes sont différents, même s'ils sont touts deux optimaux. Ce comportement est assez fréquent, car l'heuristique, en modifiant l'ordre d'exploration, change aussi la structure de parents qui est utilisée pour construire le chemin optimal.

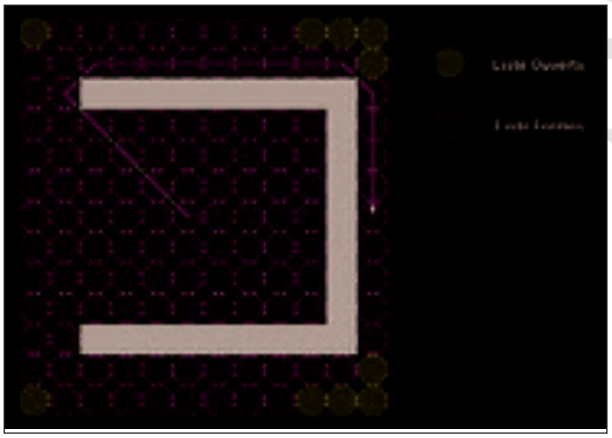


Figure 7

Il est important de noter que A* n'est pas un algorithme magique. Il est fréquent que les développeurs se plaignent de l'algorithme de Dijkstra et demandent comment passer à A*.

Pourtant, le premier réflexe devrait être de vérifier l'implémentation de Dijkstra, car l'utilisation de structures de données inadaptées (en particulier une liste classique pour la liste Ouverts) rend l'algorithme très lent. Or, cette lenteur se retrouvera sur A*...

Une implémentation ?

Il existe de nombreuses implémentations de l'algorithme A* disponibles sur Internet. Voici quelques pistes pour le lecteur curieux :

- Charité bien ordonnée : victime du syndrome du "non inventé ici", j'ai réalisé ma propre implémentation pédagogique en Java, sous licence GPL, disponible à l'URL **http://apiacoa.org/software/pathfinding**. L'intérêt de cette implémentation est de permettre une comparaison simple entre diverses stratégies, en faisant varier les heuristiques, les implémentations des listes Ouverts et Fermés, le modèle de coût de la carte, etc.

- Le moteur libre (GPL) Stratagus (**http://www.nongnu.org/stratagus/index.html**), dont j'ai déjà parlé dans [1], contient une implémentation en C de A* réalisé par votre serviteur, ainsi qu'une autre implémentation d'une version hiérarchique de A*, encore plus efficace, œuvre d'un certain Latimerius.

- Boson (**http://boson.eu.org**), lui aussi évoqué dans [1], contient une implémentation libre (GPL) en C++ d'un A* simple (les structures de données ne sont pas optimales).

- Heroic BattleField (**http://hbf.tuxfamily.org**), toujours évoqué dans [1], propose aussi une implémentation simple d'A* en C sous licence GPL.

On trouvera d'autres liens dans la section référence en fin d'article.

Du bon choix de l'heuristique

L'admissibilité

Pour que A* fonctionne correctement, il faut que l'heuristique qui estime le coût du chemin optimal d'un nœud quelconque au but soit admissible. En pratique, cela signifie que l'heuristique doit sous-estimer le coût du chemin optimal.

Dans ce cas, on montre que les chemins obtenus par A* sont bien optimaux. Au contraire, si l'heuristique n'est pas admissible et donc qu'elle surestime le coût de certains chemins, A* peut obtenir un chemin non optimal.

L'heuristique que nous avons utilisée dans la section précédente est admissible. En effet, elle calcule le coût comme la distance euclidienne entre les deux cases considérées.

Or, le coût du passage d'une case à une de ses voisines est égal à la distance (euclidienne) entre les centres des cases concernées (et ceci en tout point de la carte).

Donc, l'heuristique donne le coût exact pour le passage d'une case à une de ses huit voisines.

Quand le but est plus éloigné, l'heuristique sous-estime le coût. En effet, la distance euclidienne correspond à la distance "à vol d'oiseau", alors que le chemin optimal passe par les centres des cases et est donc en général plus long, comme l'illustre la figure 8.

De plus, l'introduction d'obstacles ne peut qu'allonger les chemins optimaux, alors que l'heuristique n'est pas modifiée. Elle reste donc admissible.

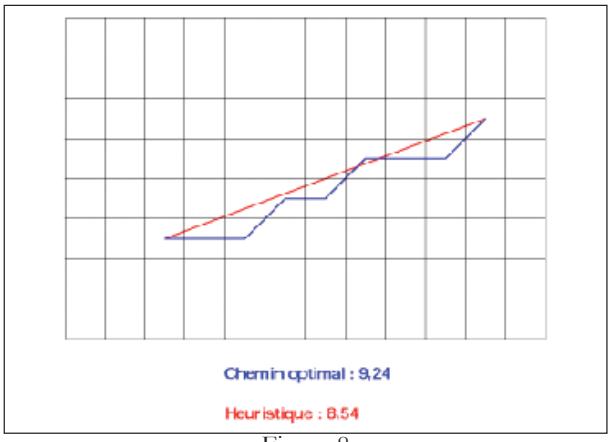


Figure 8

Notons qu'on peut améliorer l'heuristique en tenant compte du fait que les chemins passent par les centres des cases. On obtient la formule suivante : 1,4 * Min(|x-a|,|y-b|) + ||x-a|-|y-b||. Dans celle-ci, les barres verticales désignent la valeur absolue, alors que 1,4 est une approximation de la racine carrée de 2 (on peut d'ailleurs prendre une valeur plus précise). Le premier terme de la formule correspond en fait à la distance parcourue en diagonale, alors que le second correspond à la distance franchie horizontalement ou verticalement. Comme cette heuristique est plus précise que la distance euclidienne, elle améliore encore les performances de l'algorithme.

Sous-estimer...

En fait, les performances de A* dépendent fortement de la qualité de l'heuristique. Quand celle-ci donne une très bonne estimation du coût (par le bas, c'est-à-dire en sous-estimant), A* fonctionne très bien : non seulement il produit un chemin optimal, mais en plus il le fait vite, c'est-à-dire en étudiant peu de nœuds. Plus l'heuristique s'éloigne du coût optimal (toujours par le bas), plus l'algorithme devient lent. Le chemin calculé reste optimal, mais A* étudie beaucoup plus de nœuds. Le cas extrême est atteint quand l'heuristique est uniformément nulle (le coût estimé est alors toujours 0), car A* devient alors identique à l'algorithme de Dijkstra.

Nous avons déjà observé la dégradation de performance dans la section précédente. En effet, l'introduction d'obstacles dans la carte dégrade la qualité de l'heuristique qui sous-estime parfois fortement le coût du chemin optimal. Par exemple, le coût réel du chemin optimal entre le point de départ et le point d'arrivée sur la figure 7 est d'environ 21,5, alors que l'heuristique de la distance euclidienne estime ce coût à 5 !

Pour mieux comprendre l'effet de la sous-estimation, dégradons volontairement notre heuristique, par exemple en utilisant comme estimation la distance euclidienne divisée par 2. L'effet mathématique est bien entendu de changer le classement des nœuds dans la liste Ouverts : l'importance relative du coût cumulé augmente avec la diminution du coût calculé par l'heuristique. On se rapproche ainsi du classement utilisé par Dijkstra : A* explore donc plus facilement des nœuds qui l'éloignent du but. La figure 9 illustre les résultats de cette exploration moins focalisée. Quand on compare avec les figures 2 et 5, on constate qu'on obtient une configuration intermédiaire : A* a étudié 23 cases, contre 71 pour Dijkstra et 6 pour A* avec une heuristique meilleure.

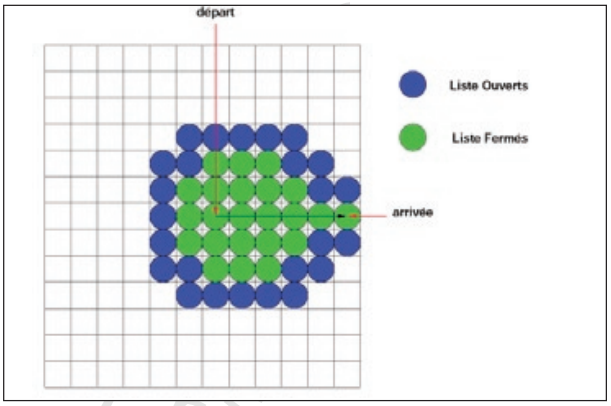


Figure 9

Il est donc important dans la pratique que l'heuristique sous-estime le moins possible les coûts. Pour ce faire, on doit être particulièrement attentif au modèle de mouvement choisi. En effet, dans [1], comme dans le début du présent article, nous avons choisi de prendre en compte la distance parcourue pour passer du centre d'une case au centre d'une de ses voisines afin de calculer le coût du déplacement. Or, dans certains jeux, la distance n'est pas prise en compte. Ainsi, un mouvement en diagonale est-il aussi coûteux qu'un autre horizontal ou vertical. Dans d'autres jeux, au contraire, les mouvements diagonaux sont interdits. Dans les deux cas, l'heuristique de la distance euclidienne n'est pas adaptée. En effet, elle surestime les coûts dans le cas où la distance n'est pas prise en compte (ce qui est très gênant, comme nous allons le voir dans la section suivante), et sous-estime les coûts quand les mouvements diagonaux sont interdits, comme l'illustre la figure 10 (le chemin bleu correspond aux déplacements diagonaux "rapides", alors que le chemin vert correspond aux mouvements diagonaux interdits).

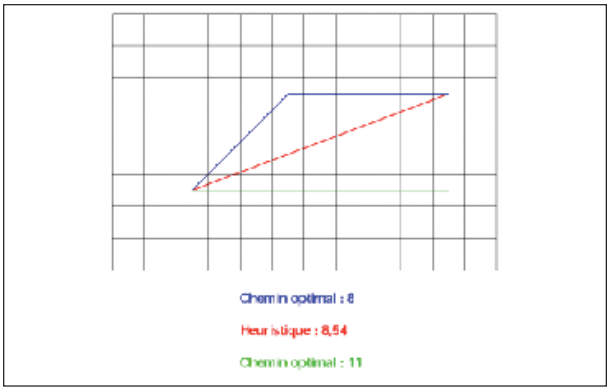


Figure 10

SUJET PROPOSÉ

Quand on interdit les déplacements diagonaux, la bonne heuristique consiste à calculer la somme de |x-a|+|y-b| (il s'agit de la distance Manhattan). On constate que la distance obtenue correspond au nombre de colonnes et de lignes qui séparent la case de départ de la case d'arrivée (en comptant la ligne et la colonne de la case d'arrivée). Bien entendu, cette heuristique n'est valable que si le coût de franchissement d'une case est de 1 au minimum.

Quand on autorise les déplacements diagonaux sans les pénaliser, c'est-à-dire sans tenir compte de la distance réellement parcourue, on peut utiliser comme heuristique la formule suivante : max(|x-a|,|y-b|). Ceci correspond à compter le nombre de colonnes et de lignes qui séparent la case de départ de la case d'arrivée, puis à conserver le plus grand des deux nombres.

Ou surestimer ?

Nous venons donc de voir que la sous-estimation du coût d'un chemin ne change pas la correction de A* qui continue dans une telle situation à calculer un chemin optimal. Le seul effet pratique est le ralentissement de l'algorithme. La situation se complique en présence de surestimation. En effet, on ne peut plus garantir dans ce cas que l'algorithme A* produit des résultats corrects. Dans certaines situations, les chemins obtenus ne sont pas optimaux. Mais en payant ce prix (parfois exorbitant), on peut accélérer les calculs !

Étudions un exemple simple pour lequel l'algorithme ne fonctionne pas. Dans une carte de coût uniforme 1, on ajoute une route de coût 0,5. On conserve le modèle de voisinage classique dans lequel on tient compte de la distance parcourue pour calculer le coût du passage d'une case à une de ses voisines. L'heuristique adaptée est donc celle basée sur la distance euclidienne. Cependant, celle-ci n'est pas admissible. En effet, un chemin sur la route ne coûte que 0,5 fois la distance parcourue. Donc, une estimation du coût par la simple distance surestime celui-ci. La figure 11 donne le chemin optimal obtenu en utilisant l'heuristique euclidienne simple. Ce chemin a un coût de 7.

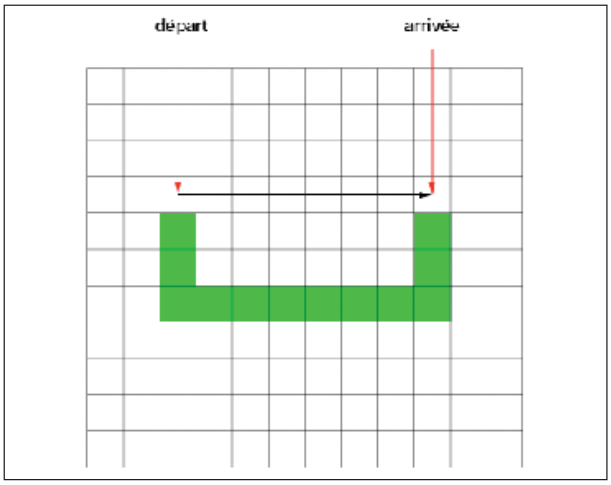


Figure 11

La figure 12 représente le chemin optimal qu'on obtient avec l'algorithme de Dijkstra ou encore avec A* et une heuristique admissible (par exemple 0,5 fois la distance euclidienne). Ce chemin a un coût de 6,4. En termes d'efficacité du calcul, l'heuristique "fausse" est particulièrement agréable car elle ne considère que 8 cases. L'heuristique admissible provoque l'analyse de 44 cases, alors que l'algorithme de Dijkstra étudie presque toute la carte (en fait 100 cases sur 132).

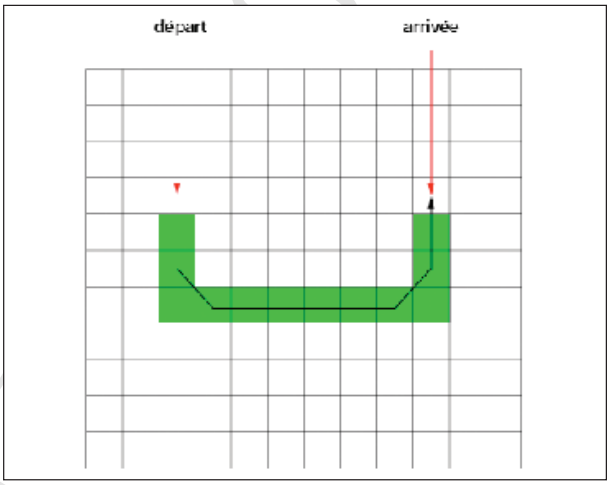


Figure 12

Il est assez difficile de prédire l'effet d'une surestimation du coût d'un chemin optimal. Sur l'exemple proposé, le chemin obtenu est totalement différent de celui cherché. On pourrait d'ailleurs faire en sorte que les chemins soient encore plus éloignés, en jouant avec les coûts de déplacement et la forme du tracé. Cependant, on constate que les coûts des deux chemins sont très proches. Or, en pratique, le tracé du chemin n'est pas toujours important, c'est avant tout le temps qu'on met à atteindre le but qui compte. Si l'utilisation d'une heuristique non admissible permet d'accélérer les calculs, on peut imaginer que l'unité va commencer son déplacement plus tôt que dans le cas d'un calcul lent et donc finalement arriver aussi vite à bon port qu'avec un chemin véritablement optimal. Le problème est que rien ne permet d'affirmer que l'algorithme ne va pas proposer un chemin totalement délirant quand il utilise une heuristique non admissible. C'est pourquoi leur utilisation doit être réservée à des cas bien maîtrisés.

Comment faire en pratique ?

La mise au point d'une heuristique efficace est un problème assez délicat. Je suggère donc de commencer par une heuristique simple, adaptée au modèle de déplacement choisi, comme la distance euclidienne ou la distance Manhattan. On peut ensuite utiliser diverses astuces pour améliorer l'heuristique et donc les performances de A* :

- Dans certaines situations, la prise en compte de caractéristiques particulières de la carte est un plus indéniable. Un exemple assez parlant est celui d'une carte divisée en deux zones, avec un seul point de passage entre

celles-ci. On pense par exemple à une île reliée au continent par un pont, comme sur la figure 13. Si on tente de calculer un chemin optimal entre le continent et l'île, la distance euclidienne sous-estime énormément le coût de ce dernier, car elle fonctionne à vol d'oiseau et utilise donc un chemin (en rouge sur la carte) qu'une unité ne pourra pas emprunter. Le vrai chemin optimal passe en fait nécessairement par l'unique point de passage (chemin vert sur la carte). Or, il est tout à fait possible de construire une heuristique qui tient compte de cette situation. L'idée est de déterminer dans quelle zone sont situés les points d'arrivée et de départ. S'ils sont dans une même zone, l'heuristique reste la distance euclidienne. Par contre, s'ils sont situés dans des zones différentes, l'heuristique évalue le coût comme la somme des distances au point de passage obligé entre les deux zones.

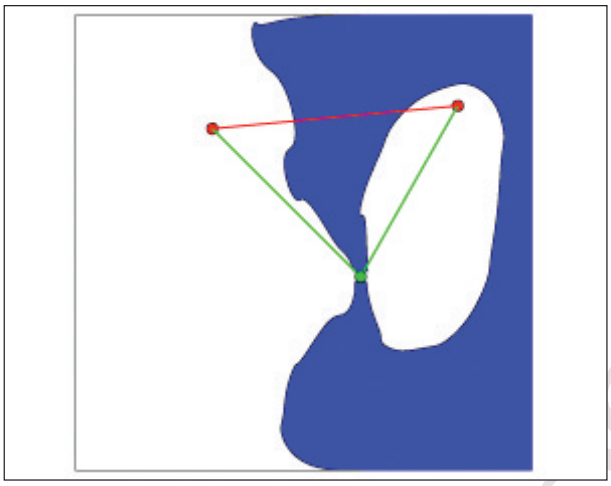


Figure 13

On peut montrer que cette heuristique améliore grandement les performances de A* qui étudie beaucoup moins de cases tout en trouvant toujours le chemin optimal. L'idée se généralise sans difficulté au cas de plusieurs zones, comme par exemple les pièces d'une maison. Cependant, il faut rester attentif au temps de calcul de l'heuristique. En effet, si le terrain étudié comporte de nombreuses zones et donc de nombreux points de passage, l'évaluation de l'heuristique devient un problème complexe, en fait c'est un calcul de chemin optimal entre les zones ! On peut donc perdre en temps de calcul de l'heuristique ce qu'on gagne en nombre de cases étudiées.

L'analyse du terrain de jeu est un domaine en plein essor (cf [3]) qui vise entre autres à produire automatiquement des aides à l'algorithme de pathfinding (et plus généralement à l'IA du jeu), en procédant par exemple à un découpage du terrain en zones qui permettent d'obtenir une heuristique efficace.

Dans d'autres situations, il est intéressant de jouer avec l'admissibilité pour obtenir à la fois un algorithme plus efficace et des effets maîtrisés sur les chemins obtenus. Pour cela, il faut bien comprendre l'effet d'une modification de l'heuristique. Nous savons qu'une heuristique admissible attire A* vers les cases qui le rapprochent du but, ce qui lui permet de l'atteindre plus rapidement. Si nous augmentons le coût estimé du chemin qui relie une case au but (sans changer l'estimation pour les autres cases), nous repoussons le moment où cette case sera explorée par l'algorithme. Au contraire, une réduction du coût favorise une exploration rapide de la case. Si on souhaite, par exemple, favoriser les lignes droites (car un chemin dans lequel une unité change en permanence de direction n'est pas très réaliste), on peut augmenter légèrement le coût estimé d'un chemin qui effectue un tournant (grâce au nœud parent conservé dans la liste Ouverts, on sait d'où le chemin optimal provient et on peut donc calculer l'angle de rotation entre la direction précédente et la direction induite par le passage par un voisin donné). La perte d'admissibilité induite par cette pénalisation a en général des effets complètement négligeables, à condition bien sûr de lui donner une très faible amplitude. On peut poursuivre dans cette voie en mettant en œuvre des adaptations de A* qui produisent des chemins plus réalistes, mais les solutions obtenues sont assez évoluées (cf [5]).

Utilisations avancées

La traduction du terrain en un graphe permet des applications évoluées qui dépassent le modèle que j'ai décrit jusqu'à présent. Il n'est pas question ici de faire une liste exhaustive des possibilités ouvertes par cette traduction (associée au gain algorithmique induit par A*), mais simplement de développer deux thèmes intéressants parmi tant d'autres.

Prise en compte de l'altitude

Le modèle de terrain que j'ai choisi est extrêmement simple et assez peu réaliste. On peut améliorer notablement ce point en prenant en compte l'altitude en plus de la nature de chaque case. Le coût du passage d'une case à une autre est ainsi obtenu comme une combinaison de deux facteurs. Comme dans le modèle simple, on tient compte de la difficulté de progression dans la case liée à la nature de celle-ci, mais on ajoute à ce coût une autre pénalité correspondant au changement d'altitude. Le point intéressant de cette pénalité est qu'elle n'est pas symétrique. En effet, passer de la case A à la case B ne coûte pas la même chose que passer de la case B à la case A. Pour s'en convaincre, il suffit d'observer les coureurs cyclistes dans une étape de montagne ! La vitesse de progression dans la montée est bien évidemment très faible par rapport à celle atteinte dans la descente. Pour plus de réalisme, les coûts peuvent d'ailleurs dépendre de la nature de l'unité considérée : un char est relativement lent, mais aussi assez peu sensible à la nature du terrain, alors qu'une voiture de sport est très rapide sur une piste et ne peut pas emprunter certains types de terrain.

L'algorithme de Dijkstra et A* s'adaptent parfaitement à ce nouveau modèle car ils travaillent sur des graphes orientés pour lesquels le coût de passage d'un nœud à un

SUPINFO

autre est porté par l'arête. Le modèle choisi introduit donc 16 arêtes orientées entre chaque nœud au lieu de 8 arêtes non orientées. Du point de vue de l'implémentation, on ne change rien, excepté bien entendu le calcul du coût de passage d'une case à une autre.

Déplacements coordonnés

Dans de nombreux jeux RTS, les unités de base ne sont pas amphibies et doivent donc utiliser des transports de troupes pour traverser une zone de mer.

Or, il est rare que l'algorithme de pathfinding soit programmé pour réaliser cette opération automatiquement, même dans les cas simples. Pourtant, cela ne pose aucun problème théorique.

Considérons en effet le cas d'un fantassin situé sur le continent A qui souhaite se rendre sur le continent B et d'un transport de troupes situé quelque part en mer entre les deux continents. Le problème se décompose en trois phases :

1. déplacer le fantassin et le transport de sorte qu'ils se rencontrent sur les "plages" du continent A ;

2. déplacer le transport contenant le fantassin jusqu'à une "plage" du continent B ;

3. déplacer le fantassin sur le continent B vers son but.

Pour traiter ce problème comme une exploration de graphe, il suffit de considérer que chaque nœud correspond aux positions des deux unités sur le terrain. Une arête du graphe modélise le déplacement simultané des deux unités, ce qui signifie quelles seront nombreuses !

En effet, si on autorise les mouvements diagonaux, on a 9 mouvements possibles par unité (les 8 mouvements réels auxquels on ajoute l'immobilité) et donc 80 mouvements possibles par tour (soit autant d'arêtes), comme par exemple le mouvement simultané suivant : le fantassin se déplace d'une case vers le nord, alors que le transport se déplace d'une case vers le sud-ouest. On compte 80 mouvements au lieu de 81 car on interdit la combinaison dans laquelle les deux unités restent immobiles.

De même, les nœuds sont très nombreux, en théorie N^2 si N désigne le nombre de cases dans le terrain de jeu (en fait moins car certaines combinaisons sont interdites, en particulier quand le fantassin est seul en mer ou quand le transport se retrouve sur le continent). On comprend donc que l'algorithme de Dijkstra risque d'être très long à s'exécuter.

A* permet d'obtenir des résultats, mais il faut envisager pour cela une implémentation spécifique pour les listes Ouverts et Fermés car l'annotation du graphe n'est plus possible en raison de sa taille.

Il est aussi important de mettre en œuvre une heuristique efficace, par exemple en demandant dans la première phase aux deux unités de se rapprocher : ce comportement peut être obtenu en utilisant une technique proche de celle proposée dans la section précédente, quand on introduisait un point intermédiaire sur le pont entre l'île et le continent. On peut ainsi calculer le point `a` d'intersection entre la ligne qui joint les deux unités et la plage du continent.

Le coût estimé est alors la distance entre l'unité la plus éloignée du point `a` et ce dernier, à laquelle on ajoute la distance entre `a` et la cible du fantassin sur le continent `B`, comme l'illustre la figure 14 (à condition que la vitesse de déplacement des deux unités soit la même).

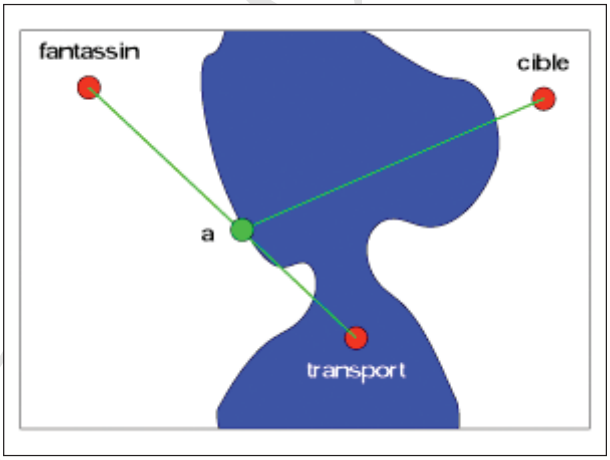


Figure 14

Grâce à la traduction en graphe, on obtient un algorithme qui calcule un chemin optimal mettant en jeu deux unités qui coordonnent leurs actions. L'apport d'A* est ici crucial car la taille du graphe interdit son exploration complète.

On peut même utiliser ici une heuristique non admissible car le plus important est d'obtenir un chemin, même s'il n'est pas optimal.

Notons pour conclure sur cet exemple que les grands graphes s'explorent aussi avec d'autres algorithmes, en particulier IDA* (cf [4]), moins efficaces en termes de nœuds explorés, mais très économes en mémoire.

Conclusion

L'algorithme A* présenté dans cet article améliore radicalement l'algorithme de Dijkstra et permet le calcul de chemins optimaux dans des graphes beaucoup plus volumineux.

Bien implémenté, c'est-à-dire associé à des structures de données efficaces et une heuristique adaptée au problème, A* apporte donc une solution satisfaisante au problème du pathfinding, à tel point que certains participants à la table ronde sur l'IA dans les jeux vidéo animée par Steven Woodcock [6] ont pu affirmer que les questions de base concernant le calcul de chemins optimaux avaient trouvé une réponse.

Cependant, de nombreuses questions restent ouvertes, comme par exemple :

- Comment construire une heuristique efficace ? Les

réponses sont à chercher dans l'analyse de terrain (cf [3]) et dans les approches hiérarchiques (cf [7]).

Comment travailler sur des cartes ou des graphes gigantesques ? Les réponses sont aussi à chercher dans les approches hiérarchiques [7] ou dans les méthodes multi-résolutions (comme celle utilisée par Stratagus).

Comment prendre en compte l'aspect dynamique du terrain ? Des solutions existent comme l'algorithme D* (cf [8]), mais elles sont assez coûteuses en termes de stockage et les développeurs préfèrent en général des solutions très simples, comme par exemple un découpage du chemin optimal en tronçon, avec un recalcul systématique (i.e., après chaque mouvement) du tronçon le plus proche.

Comment résoudre les problèmes de collisions entre unités, ou plus généralement les problèmes de mouvements coordonnées (les formations par exemple) ? Diverses solutions ont été utilisées, comme par exemple celles décrites dans [9] et [10], qui consistent à calculer un chemin optimal classique, puis à l'exécuter astucieusement grâce à des primitives de modification de la formation associées à une gestion des priorités entre les unités pour éviter les inter-blocages. Comme nous l'avons vu plus haut, il est aussi possible de formuler le problème comme une exploration de graphe, mais cette technique pose des problèmes spécifiques. D'autres solutions, inspirées de l'intelligence collective, ont aussi été proposées pour réaliser des mouvements coordonnées (cf [11]).

Le domaine du pathfinding reste donc en perpétuelle évolution, poussé, comme toute l'industrie du jeu vidéo, par les exigences croissantes des joueurs, en particulier en matière de réalisme.

Les jeux vidéo libres ont encore du mal à concurrencer les jeux commerciaux, en particulier en termes de rendu graphique. Pourtant, l'expérience ludique est un tout et l'IA en général, et le pathfinding en particulier, sont souvent des éléments en retrait dans les jeux commerciaux.

Les jeux libres pourraient donc profiter de la richesse de la recherche dans ce domaine pour marcher sur les plates-bandes de leurs concurrents à gros budget.

Fabrice Rossi
Equipe AXIS
INRIA
Fabrice.Rossi@apiacoa.org
http://apiacoa.org

Post Scriptum

Une petite erreur s'est malencontreusement glissée dans l'article précédent [1] : les figures 2 et 10 ont subrepticement échangé leurs places !

Références

Certaines références en ligne proviennent de l'excellent site http://www.gamasutra.com. Ce site consacré à l'industrie du jeu vidéo est une mine d'informations, mais son accès demande une inscription (gratuite).

[1] Fabrice Rossi - Recherche des chemins optimaux pour les jeux vidéo, GNU/Linux Magazine France, No 53, Septembre 2003.

[2] Stout, W. Bryan. - Smart Moves: Intelligent Pathfinding, October/November 1996. Disponible sur gamasutra.com à l'URL http://www.gamasutra.com/features/19970801/pathfinding.htm

[3] Dave C. Pottinger - Ensemble Studios Terrain Analysis in Realtime Strategy. Disponible à l'URL http://www.ensemblestudios.com/news/devnews/terrain1.shtml

[4] Mark Brockington - Pawn Captures Wyvern: How Computer Chess Can Improve Your Pathfinding, June 2000. Disponible sur gamasutra.com à l'URL http://www.gamasutra.com/features/20000626/brockington_01.htm

[5] Marco Pinter - Toward More Realistic Pathfinding, mars 2001. Disponible sur gamasutra.com à l'URL http://www.gamasutra.com/features/20010314/pinter_01.htm

[6] Steven Woodcock - AI Roundtable Moderator's Report 2003, Game Developer's Conference, March 6-8. Disponible à l'URL : http://www.gameai.com/cgdc03notes.htm

[7] Robert Holte, M.B. Perez, Robert Zimmer, and Alan MacDonald - Hierarchical A*: Searching Abstraction Hierarchies Efficiently, Technical report tr-95-18. Disponible à l'URL : http://www.csi.uottawa.ca/~holte/Publications/tr-95-18.ps

[8] A. Stentz - The Focussed D* Algorithm for Real-Time Replanning, , Proceedings of IJCAI-95, August 1995. Disponible à l'URL : http://www.frc.ri.cmu.edu/~axs/doc/ijcai95.ps

[9] Pottinger, Dave C. - Coordinated Unit Movement, January 1999. Disponible sur gamasutra.com à l'URL http://www.gamasutra.com/features/19990122/movement_01.htm

[10] Pottinger, Dave C. - Implementing Coordinated Movement, February 1999. Disponible sur gamasutra.com à l'URL http://www.gamasutra.com/features/19990129/implementing_01.htm

[11] Le site de Craig Reynolds (URL : http://www.red3d.com/cwr/boids/) est une référence en matière de simulation de troupeaux.

[12] Le site de Amit Patel (URL : http://www-cs-students.stanford.edu/~amitp/gameprog.html) est une mine d'informations sur le pathfinding et les jeux vidéo (on y trouve entre autre une implémentation intéressante en C++ de A*, ainsi que des liens vers d'autres implémentations).

[13] Le site de Steven Woodcok (URL : http://www.gameai.com/) est une autre mine d'information sur l'IA dans les jeux vidéo.

Recherche des chemins optimaux pour les jeux vidéo

Introduction

Dans un précédent article, *L'intelligence artificielle : des fourmis dans votre Linux* [1], j'ai évoqué l'intelligence collective des insectes sociaux et montré comment des règles très simples permettaient à une colonie de fourmis de récolter efficacement de la nourriture. J'ai mentionné brièvement le problème du calcul d'un chemin optimal d'une position à une autre. Le présent article revient sur ce problème crucial en se focalisant sur une application qui consomme le temps libre de beaucoup d'entre nous : les jeux vidéo.

Dans les jeux de stratégie temps réel (STR), le calcul du chemin optimal que doit suivre une unité pour obéir aux ordres du joueur (ou de l'ordinateur) absorbe une part assez importante du temps de développement et d'exécution du programme. D'après Steven Woodcock [2], le modérateur de la table ronde sur l'intelligence artificielle (IA) dans les jeux vidéo de la Game Developer's Conference (de mars 2003), 50% du temps de développement de l'intelligence artificielle d'un jeu est consacré à l'algorithme de calcul de chemins optimaux.

Deux raisons expliquent ce fait. La première est que le calcul de chemin optimal est un mécanisme d'intelligence augmentée, au sens où c'est (presque) le seul mécanisme d'IA du programme qui est utilisé par le joueur humain. La moindre erreur dans l'algorithme est considérée comme une "trahison" par le joueur. De plus, le piratage à grande échelle des jeux non libres fait que les éditeurs commerciaux misent de plus en plus sur le jeu *online* (car il est plus facile de contrôler les numéros de licence avec un serveur externe) dans lequel les joueurs s'affrontent entre eux. Dans ce contexte, un bogue dans le calcul de chemin optimal est extrêmement pénalisant pour le joueur qui en est victime. Il peut lui faire perdre la partie !

> Les jeux vidéo constituent un des domaines dans lesquels des utilisateurs très exigeants sont confrontés à des mécanismes d'intelligence artificielle (IA) ambitieux. Les communiqués de presse des éditeurs de jeux vantent régulièrement les mérites de leur nouvelle "IA révolutionnaire qui donne une nouvelle dimension à l'expérience ludique". Par-delà le discours marketing, on constate que les jeux font des progrès : les comportements des ennemis et des alliés sont plus réalistes, l'IA devient capable de nous surprendre, le challenge augmente au fil des générations de jeux. Le but de cet article est de présenter un des problèmes fondamentaux que l'IA des jeux vidéo doit résoudre, le calcul de chemin (pathfinding). L'IA utilise bien entendu ces calculs pour diriger ses troupes, mais aussi pour réaliser bien d'autres tâches de planification basées sur l'analyse du terrain de jeu. En ce sens, le pathfinding est la brique de base de l'IA des jeux vidéo.

La deuxième raison est liée à l'aspect temps réel des jeux. Pour qu'un jeu tourne de façon satisfaisante, il faut qu'il produise au moins 25 images par seconde. De plus, il est fréquent de jouer avec une centaine d'unités par joueur. Le moteur doit donc être capable de calculer le mouvement d'au moins 200 unités simultanément, et ceci 25 fois par seconde. En théorie, on doit donc utiliser moins de 0.2 ms pour calculer un chemin optimal et ce sur une machine "raisonnable" (par exemple un Pentium III 800 Mhz qui semble devenir peu à peu, quand il est associé à une carte graphique efficace, le strict minimum pour pouvoir jouer aux jeux "modernes"). Bien entendu, en réalité on dispose de beaucoup moins de temps car le programme doit réaliser le rendu graphique (de plus en plus exigeant) du jeu.

Les programmeurs responsables de l'IA ont donc une importante responsabilité : ils doivent produire un algorithme à la fois exact et très efficace, ce qui explique le temps qu'ils y consacrent. Comme nous allons le voir dans cet article, les algorithmes permettant le calcul d'un chemin optimal sont assez complexes, tant dans leur formulation que dans les structures de données qu'ils nécessitent pour

X 2026 – ADS informatique filière MPI – dossier 02 – page 12

fonctionner correctement. Le but de l'article est avant tout de décrire précisément le problème (dans une version simplifiée pour l'instant) et sa solution efficace la plus simple (l'algorithme de Dijkstra), en insistant sur les détails d'implémentation qui permettent d'accélérer de façon très importante les calculs. Dans un prochain article, je décrirai des variantes plus rapides (mais plus complexes) de l'algorithme et des modèles de terrain plus riches, le tout illustré par des moteurs de jeu libres.

Une première modélisation du problème

Principes généraux des jeux STR

Les jeux de Stratégie Temps Réel sont basés sur des principes simples et universels :

- Le jeu est en temps réel, c'est-à-dire qu'il n'y a pas de pause entre les mouvements et autres actions qui ponctuent le déroulement d'une partie ;
- Les joueurs s'affrontent sur un terrain de jeu limité géographiquement ;
- Chaque joueur (humain ou ordinateur) contrôle des unités, l'équivalent des pièces des échecs, en leur donnant des ordres simples comme "rends-toi ici", "attaque cette unité", etc. ;
- Les unités possèdent des caractéristiques qui les différencient, comme la vitesse de déplacement, la résistance aux dégâts, la puissance et la portée des armes, etc. ;

Dans certains jeux STR, les joueurs doivent récolter des ressources (du bois, de l'or, du pétrole, etc.) qui leur permettent de construire des bâtiments formant une base d'opération. Certains bâtiments autorisent la production de nouvelles unités ou encore l'amélioration des unités déjà existantes. D'autres bâtiments ont un rôle défensif (essentiellement des fortifications).

Il existe quelques projets intéressants de jeux STR libres dont voici une très rapide sélection :

- L'un des plus anciens est Stratagus (**http://www.nongnu.org/stratagus/index.html**, écrit en C sous licence GPL) anciennement appelé Freecraft. Ce moteur est inspiré du design de Warcraft II (modèle simple avec des cases) et peut même fonctionner avec les graphismes de ce dernier (à condition de posséder légalement une copie de ce jeu commercial). J'ai implémenté la deuxième version du *pathfinder* de Stratagus qui a été depuis remplacée par un algorithme encore plus efficace. Une anecdote intéressante est que le projet a dû changer de nom en catastrophe en juin 2003 (et ceci après des années d'existence) suite à des menaces de Blizzard, l'éditeur de la série des Warcraft (cf [3]).

- Un projet plus récent, Boson (**http://boson.eu.org**, écrit en C++ sous licence GPL) propose un moteur plus moderne, avec un rendu OpenGL et un *gameplay* inspiré de Command and Conquer. Le projet est assez avancé, mais excepté un pathfinder basé sur A* (comme celui que j'avais implémenté dans Freecraft), il ne propose aucune IA et est donc réservé à l'affrontement entre joueurs humains.

- Un autre projet récent, Heroic BattleField (**http://hbf.tuxfamily.org**, écrit en C sous licence GPL) propose lui aussi une interface 3D moderne (basée sur OpenGL) avec un gameplay inspiré de Warcraft III.

Même quand un jeu ne permet que l'affrontement entre des joueurs humains, un minimum d'intelligence artificielle est nécessaire. En effet, le terrain de jeu comporte des obstacles, des constructions, d'autres unités, etc. qui rendent délicat le mouvement d'une unité. Quand on donne l'ordre à une unité de se rendre en un point précis, la ligne droite est très rarement le meilleur chemin. Il faut donc au minimum qu'une IA détermine le chemin optimal (le plus rapide en général) pour simplifier la tâche du joueur (qui peut alors se focaliser sur la gestion de haut niveau, beaucoup plus passionnante). D'autres mécanismes sont gérés par l'IA, comme par exemple la poursuite d'une unité par une autre, l'attaque automatique des unités ennemies passant à proximité, etc.

Le terrain de jeu

Les ancêtres des jeux STR modernes, comme Dune II et Warcraft (I et II), étaient basés sur un terrain discret de type damier. Plus précisément, le jeu se déroulait sur un rectangle constitué de cases. Chaque case contenait un décor pour une représentation graphique agréable et pouvait accueillir une seule unité. Le mouvement d'une unité se faisait de case en case, sans position intermédiaire (sauf pour l'animation du mouvement).

L'évolution des jeux s'est faite dans la direction d'un plus grand réalisme, avec l'abandon apparent des cases. Dans des jeux comme Starcraft ou Age of Empire, les unités peuvent se positionner librement sur le terrain, sans avoir à respecter un quadrillage. Pourtant, du point de vue technique, le terrain est toujours représenté par un damier (cf la description du moteur de Age of Empires par le directeur technique de Ensemble Studios [4]). Diverses astuces sont utilisées pour masquer au joueur l'existence de ce terrain, mais il peut parfois apparaître quand même (cf [5] pour l'exemple d'un jeu très récent, Warcraft III).

Dans cet article, je me contente donc de considérer un terrain discret à l'ancienne car il permet de se focaliser sur la partie la plus importante du calcul du chemin optimal, à savoir celle qui travaille sur les cases. L'adaptation à un modèle de type Starcraft est relativement facile et fera l'objet d'un prochain article.

Le mouvement d'une unité

Dans les jeux STR les plus simples, une unité se déplace de case en case, de sa case courante vers l'une de ses huit voisines. On peut au choix tenir compte ou non de la

SUJET POSSIBLE

distance parcourue pour modéliser le déplacement. En effet, un déplacement en diagonale doit en théorie prendre plus de temps qu'un déplacement vertical ou horizontal, car la distance entre les centres de deux cases en diagonale est égale à 1.4 fois (en fait la racine carrée de 2) la distance entre deux cases situées sur un même axe horizontal ou vertical, comme l'illustre la figure 1.

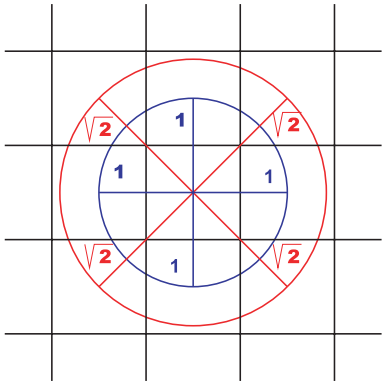


Figure 1 : Une case et ses huit voisines

Un chemin est simplement une suite de cases contiguës. Le coût d'un chemin est naturellement le temps que met l'unité à le parcourir. Déterminer le chemin optimal entre une case de départ et une case d'arrivée consiste donc simplement en la recherche de la suite de cases contiguës reliant la case de départ à la case d'arrivée qui minimise le temps de parcours pour l'unité sélectionnée.

Déterminer ce chemin est difficile si le terrain comporte deux éléments qui contribuent au réalisme :

1. des obstacles : excepté dans ces cas très particuliers, certaines cases du terrain sont infranchissables, parce qu'elles contiennent des éléments naturels (une forêt très dense, une rivière large, un précipice, etc.) ou artificiels (un bâtiment, une fortification, une autre unité, etc.) qui bloquent le passage ;

2. des modificateurs de vitesse : la présence d'une route facilite la progression des unités, alors qu'une forte pente réduit leur vitesse, etc.

On est amené à associer au passage d'une case à une autre un coût de franchissement qui tient compte de la distance parcourue et de la nature des cases de départ et d'arrivée. Dans certaines situations, le coût de franchissement est infini car la case d'arrivée n'est pas libre. Le coût d'un chemin est alors la somme des coûts de franchissement associés à la suite des cases qui le constitue.

Pour fixer les idées, considérons le terrain très simple de la figure 2. Sur celle-ci, les cases noires sont des obstacles infranchissables, alors que les cases vertes constituent une route. Le coût de franchissement d'une case de la route est de une unité de temps, alors que les cases blanches ont un coût de franchissement de 2. Quand on passe d'une case à une autre, on parcourt la moitié du chemin dans la première case et l'autre moitié dans la seconde, ce qui conduit à une combinaison des coûts.

On constate par exemple que le chemin (en bleu) qui relie le point de départ au point d'arrivée en suivant la route passe par 21 cases vertes et une case blanche (une demi-case au départ et une autre demi-case à l'arrivée). On a donc un coût total de 23 (21+2) car aucun mouvement ne se fait en diagonale. Il existe cependant un autre chemin, qui n'utilise pas la route et contourne l'obstacle (en violet). Ce chemin parcoure 5 cases en diagonales, pour un coût de 5 x 2 x 1.4, et 4 cases sans passer par les diagonales, pour un coût de 4 x 2, soit un total de 22 ! Le chemin violet est donc ici plus rapide que le chemin qui utilise la route. Mais en fait, on peut améliorer ce dernier en coupant les virages (les modifications en rouge), ce qui permet de gagner deux cases, au détriment de l'utilisation de deux diagonales. On passe alors à 17 cases de route parcourues verticalement ou horizontalement (coût 17), deux cases parcourues en diagonale (coût 2 x 1.4) et une case blanche (coût 2). Le coût total est donc de 21.8 ! Peut être est-il possible de faire mieux...

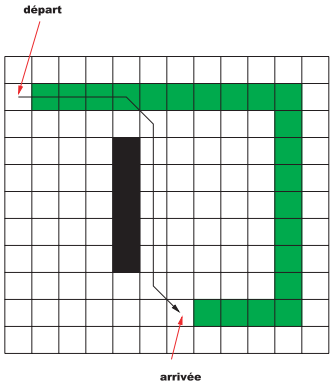


Figure 2 : Un exemple de terrain

Un algorithme au nom imprononçable

Un modèle par graphe

Comme le montre l'exemple de la figure 2, l'introduction de coût de déplacement rend le calcul d'un chemin optimal difficile même pour un humain. Fort heureusement, le problème peut être résolu relativement efficacement en utilisant des algorithmes adaptés. Je pourrais les décrire dans le cadre présenté dans la section précédente, mais je préfère m'en abstraire un peu. En effet, le problème du chemin optimal sur le terrain d'un jeu n'est qu'une instance du problème plus général qui consiste à trouver un chemin optimal dans un graphe. L'intérêt de cette vision plus abstraite est qu'elle permet de passer du modèle basique du mouvement dans une grille à des modèles beaucoup plus évolués (cf par exemple [6]) sans avoir à remettre en question les principes généraux des algorithmes. De plus, elle permet aussi de faire le lien avec d'autres problématiques de l'IA comme les jeux classiques de type Othello ou échecs (cf [7]) que j'aborderai dans de futurs articles.

POSE MAI 2026

Un graphe est une structure de données constituées de nœuds et d'arêtes. Les nœuds sont reliés entre eux par les arêtes. La figure 3 donne un exemple de graphe dans lequel les arêtes sont représentées par des flèches bleues. On remarque un point important : les arêtes sont orientées, elles relient un nœud à un autre, dans cet ordre.

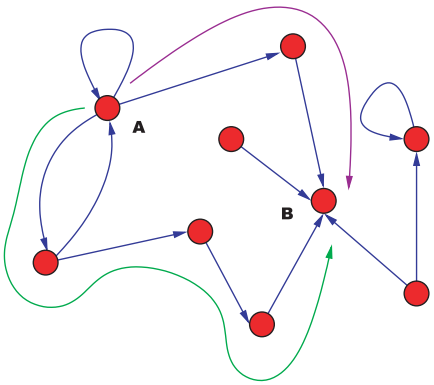


Figure 3 : Un exemple de graphe

La notion de chemin proposée dans un terrain quadrillé s'applique sans difficulté à un graphe. L'idée est de passer d'un nœud du graphe à un autre, en suivant les arêtes dans le bon sens. La figure 3 donne par exemple deux chemins depuis le nœud A vers le nœud B (en vert et en violet). Dans un graphe, le coût d'un chemin est mesuré par la somme des coûts associés aux arêtes utilisées pour progresser d'un nœud à un autre. Si on suppose que toutes les arêtes ont un coût de 1, le chemin vert qui utilise 4 arêtes est donc plus coûteux que le chemin violet (deux arêtes). Cependant, il n'y aucune raison d'avoir des arêtes avec un coût uniforme...

Le lien avec les terrains

Le modèle graphe est extrêmement général. Pour le comprendre, il suffit de considérer les nœuds comme des états du monde (par exemple la position d'une unité) et les arêtes comme des actions élémentaires (par exemple passer d'une case à une autre). Le principe fondamental est qu'une arête modélise l'effet d'une action sur le monde et se traduit donc par le passage d'un nœud à un autre. Pour réaliser une action complexe, on combine des actions élémentaires, ce qui revient à parcourir un chemin dans le graphe. Pour bien voir le lien avec la recherche d'un chemin dans un terrain, il suffit de considérer que chaque case franchissable de celui-ci peut être représentée par un nœud d'un graphe. Si deux cases sont voisines dans le terrain, le graphe contient deux arêtes (une dans chaque sens) entre les nœuds correspondants du graphe. Dans le modèle simple que nous étudions ici, on peut remplacer les deux arêtes orientées par une seule arête non orientée. Cependant, dans des modèles plus complexes, ce n'est pas le cas (il suffit de penser à l'exemple de la descente opposée à la montée qui montre qu'on peut passer rapidement du nœud A au nœud B sans pour autant pouvoir passer rapidement de B à A).

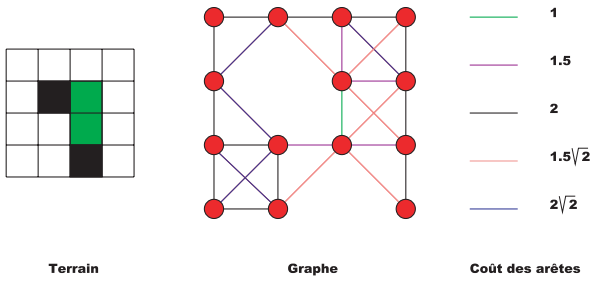


Figure 4 : Un exemple de traduction d'un terrain en graphe

La figure 4 illustre l'interprétation d'un terrain en un graphe. Le terrain d'origine comporte des cases infranchissables (en noir), des cases dont le coût de franchissement est 2 (en blanc) et d'autres dont le coût de franchissement est 1 (en vert). Le graphe obtenu compte moins de sommets que la carte puisqu'il ne contient pas l'équivalent des cases inaccessibles. La figure indique le coût associé à chaque arête. Ce coût tient compte de la distance parcourue dans chaque case, ce qui explique les valeurs comme 1.5 fois la racine de 2 (ce qui correspond à parcourir une demi-diagonale dans une case de coût 1 et une autre demi-diagonale dans une case de coût 2).

Un point important à noter est que malgré la traduction du problème de chemin optimal dans un terrain en un problème de parcours d'un graphe, il n'est pas utile en général de construire une représentation informatique du graphe. En effet, la traduction du terrain en graphe est tellement simple qu'elle peut avantageusement être faite en temps réel pendant le déroulement des algorithmes qui croient travailler sur un graphe.

Notons pour finir que la traduction en graphe permet d'étudier des problèmes plus complexes que le simple chemin optimal avec les mêmes algorithmes. Dans le cadre des jeux STR, on peut s'intéresser par exemple au mouvement d'un groupe d'unités en formation. Une arête du graphe ne correspond plus simplement au mouvement d'une unité mais à celui simultané de toutes les unités du groupe. De même, un mouvement représente la position de la formation complète.

L'algorithme de Dijkstra

L'algorithme de base pour calculer un chemin optimal entre deux nœuds d'un graphe est celui inventé par Dijkstra en 1959 [8]. Il s'applique à des graphes orientés ou non, à condition que le coût associé à chaque arête soit positif (ce qui est le cas dans nos problèmes). L'énoncé de l'algorithme est assez simple :

1. Créer deux listes vides *Ouverts* et *Fermés* ;

2. Insérer dans la liste *Ouverts* le triplet formé du nœud de départ, de la valeur 0 (comme coût cumulé) et d'une valeur arbitraire (*null* par exemple) comme nœud parent ;

SOUFFLOT

3. tant que la liste *Ouverts* n'est pas vide :

a. chercher dans *Ouverts* le triplet (N,c,p) dont le coût cumulé (c) est minimal (parmi les triplets contenus dans la liste) ;

b. ajouter le triplet à la liste *Fermés* (on peut se contenter du couple (N,p)) ;

c. si N est le nœud d'arrivée, quitter la boucle et passer à l'étape 4 ;

d. supprimer le triplet de la liste *Ouverts* ;

e. pour chaque successeur S de N (c'est-à-dire chaque nœud S pour lequel il existe une arête de N à S dans le graphe) :

i. si S est dans *Fermés* passer au successeur suivant ;

ii. calculer c' comme la somme de c et du coût associé à l'arête qui relie N à S ;

iii. si S est dans *Ouverts* avec un coût cumulé supérieur à c', mettre à jour le coût cumulé et le nœud parent (qui est maintenant N) ;

iv. sinon ajouter à *Ouverts* le triplet (S,c',N) ;

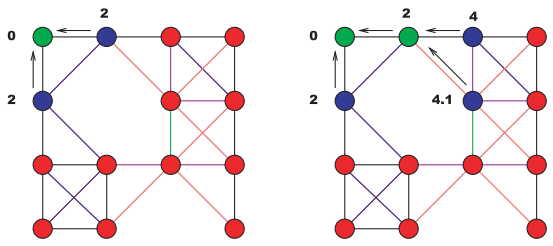
4. si le nœud d'arrivée n'est pas dans *Fermés*, terminer l'algorithme sur un échec : il n'existe pas de chemin depuis le nœud de départ vers le nœud d'arrivée ;

5. sinon, reconstruire le chemin en suivant l'information parent dans la liste *Fermés* ;

Notons que le coût cumulé associé à un nœud N dans la liste *Ouverts* est le coût du chemin optimal menant du nœud de départ à N.

Exemple de mise en œuvre

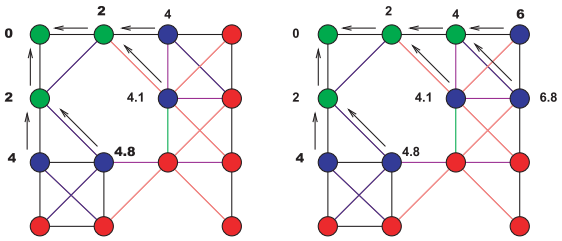
Pour bien comprendre l'algorithme de Dijkstra, nous allons l'exécuter "à la main" sur l'exemple de la figure 4, pour trouver un chemin optimal depuis le coin supérieur gauche de la carte vers le coin inférieur droit. Nous marquons en bleu les nœuds de la liste *Ouverts* et en vert les nœuds de la liste *Fermés*. On initialise l'algorithme en plaçant le nœud en haut à gauche dans *Ouverts*. Comme c'est le seul de la liste, il est placé directement dans *Fermés* dès la première itération de la boucle principale. On place ensuite dans *Ouverts* les deux voisins du premier nœud, ce qui nous donne la situation de la figure 5 (côté gauche). Sur cette figure, le chiffre indiqué à côté d'un nœud est son coût cumulé, alors que la flèche désigne son parent.



Premier tour Deuxième tour

Figure 5

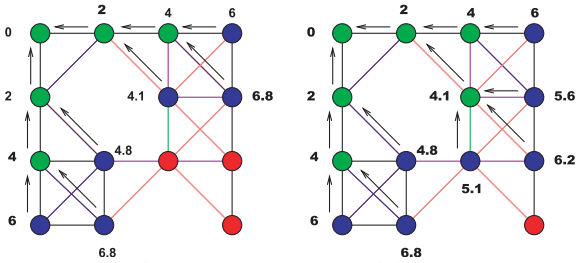
Le deuxième tour de la boucle principale choisit un des deux nœuds de façon indifférente, puisqu'ils ont le même coût cumulé. Nous prenons ici le nœud du haut. Pour simplifier, nous approchons la racine de 2 par 1.4, ce qui nous donne le côté droit de la figure 5. On remarque que certains voisins du nœud n'ont pas été placés dans *Ouverts*. C'est le cas du nœud de départ qui est déjà dans *Fermé*. Quant à l'autre nœud de *Ouverts*, son coût cumulé est inférieur à celui qu'on obtient en passant par le nœud courant et il n'est pas modifié. Le prochain tour sélectionne nécessairement le nœud de coût cumulé 2, ce qui entraîne l'ajout de deux nœuds dans la liste *Ouverts* (côté gauche de la figure 6). On se retrouve avec deux nœuds de coût cumulés 4 dans la liste *Ouverts*, ce qui peut produire par exemple l'étude du troisième nœud de la ligne supérieure du graphe et conduire à la situation représentée par le côté droit de la figure 6.



Troisième tour Quatrième tour

Figure 6

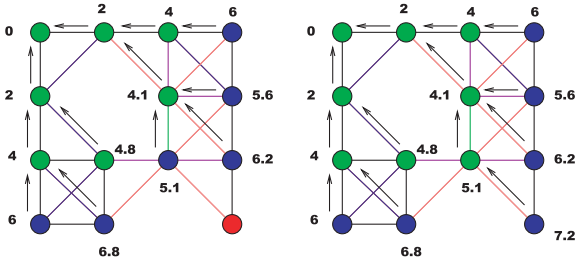
Le développement des nœuds se poursuit sans problème particulier pour le cinquième tour de la boucle. Au sixième tour, on assiste à une modification d'un nœud inséré dans la liste *Ouverts*. En effet, le nœud de coût cumulé 6.8, situé sur la deuxième ligne du graphe est maintenant accessible depuis le nœud de coût 4.1 qui est extrait de *Ouverts* lors du sixième tour. Il passe ainsi à un coût cumulé de 5.6, ce qui modifie aussi son nœud parent.



Cinquième tour Sixième tour

Figure 7

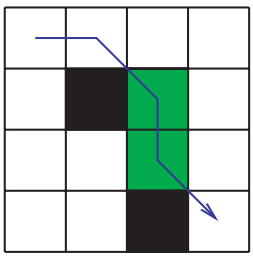
La septième étape se contente de faire passer le nœud de coût cumulé 4.8 de la liste *Ouverts* à la liste *Fermés*. La huitième étape fait de même avec le nœud de coût cumulé 5.1.



Septième tour Huitième tour

Figure 8

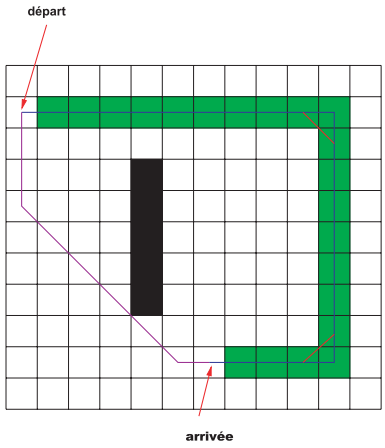
Les étapes suivantes de l'algorithme ne sont pas très intéressantes car elles se contentent de passer les derniers nœuds dans la liste *Fermés*. Aucune mise à jour des coûts n'a lieu et l'algorithme se termine par le traitement du nœud d'arrivée, de coût cumulé 7.2. On peut alors déterminer le chemin optimal, donné sur la figure 9 (traduit sur la grille d'origine).



Le chemin optimal obtenu - Figure 9

Application à une carte

L'application à une carte ne pose pas de problème particulier et nous permet de calculer le chemin optimal dans le cas du terrain de la figure 2. Le chemin obtenu est donné par la figure 10.



Le chemin optimal obtenu - Figure 10

Le chemin obtenu ne fait pas partie des chemins testés dans la présentation de la carte étudiée. On constate cependant que son coût est bien inférieur à ceux des autres chemins. En effet, il comprend 3 cases de route en franchissement horizontal (coût 3), 6 cases normales en franchissement vertical (coût 12), une demi-case normale en franchissement horizontal (coût 1), une case normale et demi en franchissement diagonal (coût 1.5 x 2 x 1.4 = 4.2) et une demi-case de route en franchissement diagonal (coût 1 x 0.5 x 1.4 = 0.7), soit un coût total de 20.9, contre 21.8 pour notre meilleur essai !

Implémentation efficace

Version élémentaire

Les performances de l'algorithme de Dijkstra dépendent énormément de la façon dont les structures de données utilisées sont implémentées. L'algorithme est en effet basé sur trois composants logiciels :

1. le graphe lui-même ;
2. la liste *Ouverts* ;
3. la liste *Fermés* ;

Dans notre cas, un terrain représenté par une grille, l'implémentation du graphe ne pose pas de problème particulier. On peut par exemple proposer un tableau de cellules, chaque cellule correspondant à une case du terrain et contenant les informations intéressantes sur celle-ci (comme le coût de franchissement). De plus, les arêtes du graphe se représentent aussi très facilement de façon implicite : on sait en effet que toute case possède huit voisines, sauf les cases du bord (repérées par un simple test sur la position de la cellule) ainsi que celles dont une voisine est infranchissable (encore un simple test sur les informations contenues dans la cellule concernée). Il faut cependant garder à l'esprit que dans des situations plus complexes, la représentation du graphe peut être délicate.

L'implémentation des deux listes est beaucoup plus complexe. Dans un premier temps, le programmeur a en général le réflexe logique d'utiliser une structure de données standard dans son environnement de travail. Il est donc normal de travailler en C++ avec un conteneur de la STL (par exemple une `deque` ou une `list`) ou en Java avec une `List`. On obtient ainsi une version élémentaire de l'algorithme de Dijkstra. Cette version souffre d'un problème de temps de calcul élevé. On peut démontrer en effet que dans le cas le moins favorable, l'algorithme peut prendre un temps proportionnel à kn^2. Dans cette formule, n désigne le nombre de nœuds dans le graphe (c'est-à-dire le nombre de cases dans notre terrain de jeu) et k le nombre maximal de successeurs pour un nœud quelconque dans la graphe (c'est-à-dire 8 dans le cas d'un terrain classique). Pour fixer les idées, il faut garder à l'esprit qu'un jeu comme Warcraft II (qui fonctionnait à l'époque sur un 486) pouvait utiliser des cartes contenant 128x128 cases (soit n=16384). Avec l'implémentation élémentaire, on obtient un temps de calcul proportionnel à 2 milliards, ce qui est loin d'être raisonnable !

Réflexions sur la liste Fermés

Rappelons les opérations qui doivent pouvoir être réalisées sur la liste *Fermés* :

- On doit pouvoir ajouter un couple (nœud, parent) à la liste ;
- On doit pouvoir tester si un nœud est présent dans la liste ;
- On doit pouvoir suivre les parents dans la liste pour créer le chemin.

Si on dispose d'une mémoire suffisante, une technique très simple et très efficace pour implémenter la liste *Fermés* consiste à annoter le graphe. Si on se reporte en effet à l'exemple de mise en œuvre de l'algorithme, on remarque que le dessin du graphe contient toutes les informations correspondant à la liste *Fermés* : la couleur d'un nœud indique s'il appartient ou non à la liste, et une flèche donne la direction du parent.

SOUFFLOT

Dans le cas d'une carte, il est très simple d'annoter le terrain. Il suffit en effet de prévoir dans chaque cellule un emplacement adapté : un drapeau pour indiquer si la cellule appartient à la liste *Fermés* et une simple indication de direction pour obtenir le parent. En effet, la structure régulière de la carte fait que le parent d'un nœud est obligatoirement une cellule voisine. Comme il n'y a que 8 voisines, il suffit d'un entier compris entre 0 et 7 pour indiquer celle-ci. Avec un codage binaire, le parent prend donc 3 bits, alors que l'appartenance à *Fermés* en consomme 1. Nous avons donc une consommation de mémoire additionnelle de 4 bits par case, ce qui est relativement raisonnable.

En termes de temps de calcul, cette implémentation de la liste *Fermés* produit un gain très important par rapport à l'implémentation de base. En effet, les listes classiques ne permettent pas un accès rapide à un élément, ce qui signifie que le test de présence d'un nœud dans *Fermés* prend dans le cas le pire un temps proportionnel à n. Avec l'implémentation proposée, les trois opérations qui doivent être réalisées sur la liste *Fermés* sont toutes en temps constant, c'est-à-dire que le temps d'exécution d'une quelconque des opérations ne dépend pas de la taille de la carte. On peut obtenir des temps d'exécution comparables en utilisant une table de hachage (un `HashMap` en Java et un `hash_map` en C++), mais à condition de fabriquer une bonne fonction de hachage (ce qui n'est pas facile). De plus, même si les coûts théoriques sont les mêmes, l'implémentation d'une table de hachage est nettement plus complexe que la solution simple que nous proposons ici, et en pratique, cette alternative est plutôt moins efficace.

Réflexions sur la liste Ouverts

Rappelons les opérations qui doivent pouvoir être réalisées sur la liste *Ouverts* :

- On doit pouvoir ajouter un triplet (nœud, coût cumulé, parent) à la liste ;
- On doit pouvoir déterminer l'élément le plus petit de la liste (au sens du coût cumulé) et le supprimer ;
- On doit pouvoir déterminer si un nœud est présent dans la liste ;
- On doit pouvoir mettre à jour la liste si le coût cumulé d'un nœud est modifié.

La différence fondamentale avec la liste *Fermés* est donc l'aspect trié de certaines opérations. Il est alors tentant d'utiliser une structure triée comme le `TreeSet` de Java ou le `set` du C++. Ces classes proposent en effet un conteneur dont les éléments sont toujours triés, ce qui signifie que les opérations qui nous intéressent se font très rapidement (le temps de calcul est proportionnel au logarithme de n).

On peut cependant proposer une structure spécifique qui permet d'améliorer les temps de calcul. L'idée est de construire la liste *Ouverts* comme un tas. Cette structure de données (décrite dans la section suivante) est aussi appelée *file avec priorité*. La principale différence avec un `TreeSet` est que le contenu du tas n'est pas complètement trié. En fait, il est optimisé pour permettre un accès instantané à l'élément le plus petit, tout en permettant une insertion rapide d'un nouvel élément et la suppression rapide de l'élément le plus petit. En ce sens, le tas fournit exactement les fonctionnalités attendues pour la liste *Ouverts*. Le seul défaut du tas est qu'il n'est pas prévu pour les opérations de recherche d'un élément ou de mise à jour du coût d'un élément. Si on implémente ces opérations de façon naïve, on obtient une assez mauvaise solution. C'est pourquoi on adopte la démarche suivante :

- En s'inspirant de la solution pour la liste *Fermés*, on annote le graphe avec deux informations : la présence d'un nœud dans la liste *Ouverts* et le coût cumulé de ce nœud. Cela représente l'ajout dans chaque cellule du terrain d'un bit pour la première information et d'une valeur plus ou moins encombrante suivant le modèle de coût retenu. Si on choisit un modèle de coût très riche, on peut utiliser un double par case (8 octets !), ce qui est assez énorme. L'utilisation d'un `int` ou d'un `float` semble plus raisonnable !

- Pour savoir si on doit réaliser une mise à jour du tas, on commence par comparer la nouvelle valeur du coût avec celle qui est stockée dans le graphe. De plus, au lieu de faire effectivement la mise à jour, on se contente d'ajouter de nouveau le nœud avec sa nouvelle valeur. Cela modifie légèrement l'algorithme et augmente la taille du tas, mais l'expérience prouve que cette solution est très performante.

L'implémentation proposée rend le temps de calcul de l'algorithme de Dijkstra proportionnel à la grandeur $an(k+2log(a)+2log(n))$ dans le cas le pire. La valeur de a désigne le nombre de fois que chaque nœud doit être ajouté ou modifié dans la liste *Ouverts*. L'expérience montre que sur des terrains simples, a est petit, de l'ordre de 2 (il est de toute manière plus petit que k). Pour fixer les idées, l'exemple d'une carte 128x128 avec $a=2$ conduit maintenant à un coût proportionnel à 1,2 millions, soit une réduction d'un facteur 1700 par rapport à une implémentation naïve. Plus la carte est grande, plus la réduction est forte. Le temps de calcul reste cependant très grand, ce qui explique indirectement pourquoi le pathfinder de Warcraft II était assez mauvais... En théorie, la solution basée sur les conteneurs triés est plus efficace, puisqu'elle affiche un temps de calcul proportionnel à la grandeur $n(k+(2+a)log(n))$ (à condition de l'associer à l'annotation du graphe que nous proposons pour le tas). Cependant, la gestion d'un conteneur trié est assez complexe et la pratique montre que cette solution est plutôt moins efficace que celle basée sur un tas, en particulier quand a est petit, ce qui est le cas sur des terrains simples comme ceux utilisés pour les jeux STR.

Un tas

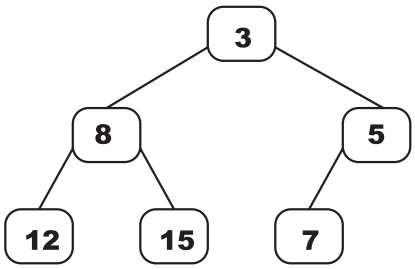
Une structure de *tas* est un codage dans un tableau d'un arbre binaire. Le tas est intéressant car il permet un accès

JOSEPHINE 2026

en temps constant à son élément le plus petit, tout en proposant l'ajout d'un nouvel élément en un temps proportionnel à *log(n)* (si *n* est la taille du tas) et la suppression de l'élément minimal aussi en *log(n)*. Il est donc particulièrement adapté pour l'implémentation de la liste *Ouverts* de l'algorithme de Dijkstra. L'idéal pour comprendre le fonctionnement du tas est de raisonner sur l'arbre binaire correspondant, puis d'étudier sa traduction en tableau. L'arbre du tas possède plusieurs propriétés intéressantes :

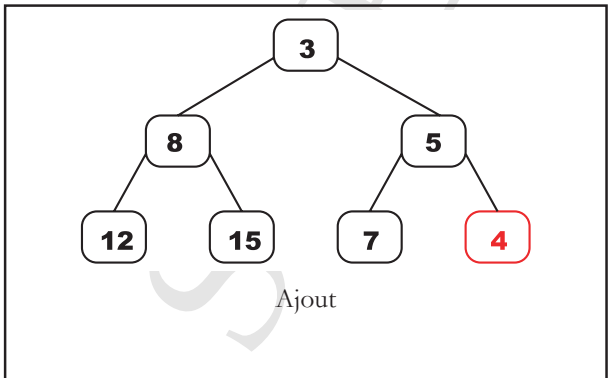
- Il est binaire, c'est-à-dire qu'un nœud possède au plus deux fils ;
- Il est complet, c'est-à-dire que tous les niveaux sont remplis sauf éventuellement le dernier dans lequel tous les nœuds sont tassés vers la gauche ;
- Un nœud est toujours plus petit (au sens du critère de classement choisi) que ses deux fils (et donc la racine de l'arbre contient le plus petit élément du tas).

La figure 11 donne un exemple de tas. On constate qu'excepté le fait qu'un nœud est plus petit que ses fils, il n'y a pas d'autre structure d'ordre dans le tas.

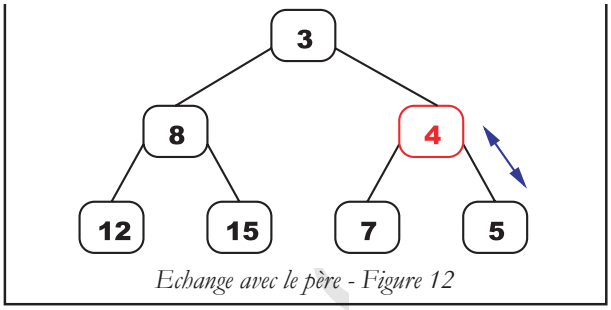


Un tas - Figure 11

Le premier algorithme important lié au tas est celui de l'ajout d'un nouvel élément. Son principe est assez simple : on ajoute le nouvel élément au premier emplacement disponible, c'est-à-dire le plus à gauche possible dans le dernier niveau de l'arbre (cf la partie gauche de la figure 12). Si celui-ci est plein, on crée un nouveau niveau et on place le nouvel élément complètement à gauche dans ce nouveau niveau. Ensuite, on remonte dans l'arbre en partant du nouvel élément. Si celui-ci est plus petit que son père, on réalise l'échange entre les deux et on recommence avec le père. Sinon, on termine l'ajout (cf la partie droite de la figure 12).



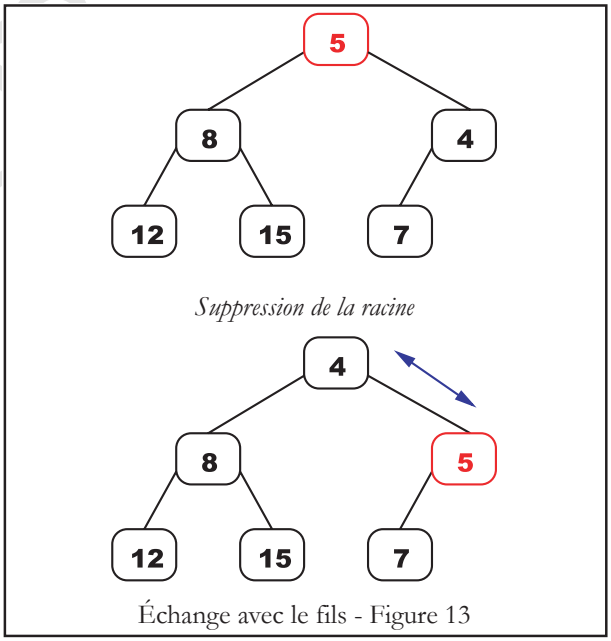
Ajout



Echange avec le père - Figure 12

On montre que l'algorithme d'ajout prend un temps proportionnel à *O(log(n))* quand le tas contient *n* nœuds et qu'il conserve toutes les propriétés du tas. Le deuxième algorithme important est celui de suppression de l'élément minimal du tas.

On procède d'une façon assez similaire à celle de l'ajout. On commence par remplacer le nœud racine par le nœud le plus extrême de l'arbre (situé le plus à droite du dernier niveau), puis on fait descendre ce nœud dans l'arbre tant que la propriété d'ordre (un nœud est plus petit que ses fils) n'est pas vérifiée. Pour ce faire, on échange le nœud courant avec son fils le plus petit, puis on passe au fils échangé et on recommence jusqu'à atteindre une feuille ou une situation où l'ordre est respecté. La figure 13 illustre l'algorithme toujours avec le même tas. On montre que l'algorithme de suppression prend un temps proportionnel à *log(n)* quand le tas contient *n* nœuds et qu'il conserve toutes les propriétés du tas.

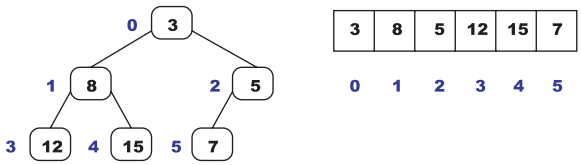


Suppression de la racine

Échange avec le fils - Figure 13

Pour stocker l'arbre dans un tableau, on numérote tout simplement les nœuds en partant de 0, niveau par niveau, et en allant de gauche à droite. La figure 14 illustre cette numérotation et le placement correspondant dans le tableau.

Un tas traduit en tableau - Figure 14



Il est alors très facile de traduire les algorithmes présentés sur l'arbre en algorithme travaillant sur le tableau, à condition de conserver à l'esprit les éléments suivants :

- La racine de l'arbre est dans la case numéro 0 ;
- Pour ajouter un nœud, on le place à la fin du tableau ;
- La fin du tableau contient toujours le nœud le plus extrême de l'arbre ;
- Pour passer d'un nœud de numéro *k* à son père, il suffit de calculer *(k-1)/2* qui est le numéro du père ;
- Les fils d'un nœud de numéro *k* possèdent les numéros 2k+1 et 2k+2.

Conclusion

Malgré la simplicité du modèle de terrain étudié dans l'article, on ne peut que constater la complexité de l'algorithme qui permet de calculer le chemin optimal.

La comparaison avec les algorithmes simplistes mis en œuvre par les fourmis de l'article sur l'intelligence collective [1] est à cet égard édifiante.

Bien entendu, je m'étais bien gardé dans cet article d'aborder les problèmes d'efficacité et de correction. Les chemins découverts par les fourmis ne sont pas optimaux en général et demandent un temps de calcul assez important.

C'est le prix à payer pour des algorithmes simples. Dans un jeu STR, il est impensable d'avoir des chemins faux associés à un programme qui rame (!), ce qui disqualifie (au moins pour le calcul de chemin optimal) les stratégies de type intelligence collective, au profit d'algorithmes classiques mais complexes. Notons que pour des opérations de plus haut niveau, l'intelligence en essaim est un thème de réflexion important pour l'industrie du jeu vidéo (cf [2]).

Malgré les améliorations importantes apportées à l'implémentation de l'algorithme de Dijkstra par la notion de tas et les annotations du graphe, les coûts annoncés restent cependant très importants (1,2 millions d'opérations pour une carte 128x128), alors que notre modèle de terrain est simple et que nous n'avons pas abordé le problème de l'évitement des unités amies ou ennemies.

Il est donc clair que les jeux commerciaux (et libres) n'utilisent pas l'algorithme tel quel, même dans sa version optimale. Nous verrons dans un prochain article comment l'introduction d'une heuristique permet d'améliorer drastiquement l'algorithme de Dijkstra (qui devient alors l'algorithme A*) en lui permettant de focaliser son attention sur les nœuds qui semblent le rapprocher le plus du nœud cible. Nous verrons comment les moteurs de jeu modernes sont construits autour de A* afin de proposer des solutions qui permettent de travailler avec des cartes très étendues, des modèles de terrain plus complexes, etc.

Fabrice Rossi
CEREMADE
Université Paris-IX Dauphine
Fabrice.Rossi@apiacoa.org
http://apiacoa.org

Références

Certaines références en ligne proviennent de l'excellent site http://www.gamasutra.com. Ce site consacré à l'industrie du jeu vidéo est une mine d'informations, mais son accès demande une inscription (gratuite).

[1] Fabrice Rossi - *L'intelligence artificielle : des fourmis dans votre Linux*, GNU/Linux Magazine France, No 51, Juin 2003.

[2] Steven Woodcock - *AI Roundtable Moderator's Report 2003 Game Developer's Conference*, March 6 - 8. Disponible à l'URL : http://www.gameai.com/cgdc03notes.htm

[3] Freecraft se prend un coup de Blizzard !, nouvelle sur Da Linux French Page : http://linuxfr.org/2003/06/21/12976.html

[4] Dave C. Pottinger - *Ensemble Studios Terrain Analysis in Realtime Strategy*. Disponible à l'URL http://www.ensemblestudios.com/news/devnews/terrain1.shtml

[5] Patrick Leclercq - Dossier intelligence artificielle - *Le fonctionnement secret de Warcraft III*, Joystick, No 143, Décembre 2002

[6] Marco Pinter - *Toward More Realistic Pathfinding*, mars 2001. Disponible sur gamasutra.com à l'URL http://www.gamasutra.com/features/20010314/pinter_01.htm

[7] Mark Brockington - *Pawn Captures Wyvern: How Computer Chess Can Improve Your Pathfinding*, juin 2000. Disponible sur gamasutra.com à l'URL http://www.gamasutra.com/features/20000626/brockington_01.htm

[8] E. W. Dijkstra - *A note on two problems in connexion with graphs*. Numerische Mathematik 1:269-271.

ÉCOLE POLYTECHNIQUE CONCOURS D'ADMISSION 2026

Filière MPI

Analyse de Documents Scientifiques en Informatique

(120 minutes de préparation et 40 minutes d'interrogation)

Vous trouverez ci-après un dossier comprenant un ou plusieurs articles portant sur un problème informatique, ainsi qu'éventuellement des documents annexes donnant des éléments théoriques sous-jacents à ce problème.

Vous disposez pour analyser ces documents de deux heures de préparation, à l'issue desquelles vous serez interrogé·e pendant 40 minutes. Lors de l'interrogation, il vous est demandé de présenter tout d'abord à l'examinateur un exposé de synthèse de 15 minutes, qui sera suivi d'un entretien de 25 minutes portant sur le contenu scientifique du dossier et sur votre culture générale.

Il vous est rappelé que, conformément à la notice du concours, cette épreuve a pour but de vous permettre de montrer votre esprit critique, vos facultés à mobiliser vos connaissances, à déterminer les problèmes posés, et à examiner les solutions apportées, votre aptitude à collecter, analyser, synthétiser et communiquer l'information, et la qualité de votre expression orale.

Pendant votre préparation, en cas de difficulté, et notamment de problème technique avec la tablette, n'hésitez surtout pas à demander de l'aide aux appariteurs.

DOSSIER N° 05

SUJET POSTÉ EN 2026



Sur quelques évolutions récentes en cryptologie post-quantique

Franck Leprévost [1]

La cryptologie est aussi ancienne que le besoin de garder ses secrets à l'abri d'une part, et d'avoir accès à ceux de son voisin d'autre part. Le fait que ce dernier soit considéré comme ennemi, ou comme ami joue peu lorsqu'on regarde l'histoire du domaine. On consultera avec profit l'excellent ouvrage [6], et on gardera à l'esprit les rapports pour le Parlement Européen sur le réseau Echelon [2] au tournant des années 2000, l'affaire Snowden, ou encore les écoutes de dirigeants politiques ou de journalistes européens.

Ce domaine scientifique est schématiquement constitué de deux composantes aux objectifs opposés (les spécialistes pardonneront la grande simplification des concepts dont cet article fait preuve tout du long). D'un côté la cryptographie s'attache au développement de méthodes visant à assurer la confidentialité de messages ainsi que leur authenticité ou leur intégrité. De l'autre la cryptanalyse cherche à casser les codes employés afin de découvrir le contenu des messages chiffrés, à impersonnaliser les messages, ou à altérer leur contenu plus ou moins subrepticement.

Ces deux directions ont connu d'importantes évolutions au cours des siècles, mais avec des vitesses variables. L'objectif ici n'est pas de les relater toutes, l'ouvrage [6]

1. Ancien chercheur et professeur en mathématiques pures au CNRS, à l'université Joseph Fourier, au Max Planck-Institut für Mathematik à Bonn, à la Technische Universität Berlin (Allemagne), co-auteur du rapport dit « Echelon » pour le Parlement européen, Franck Leprévost est actuellement professeur d'informatique à l'université du Luxembourg, dont il a été le vice-président de 2005 à 2015. Il y dirige le laboratoire d'algorithmique, cryptologie et sécurité et est l'auteur de tutoriels de mathématiques pour l'informatique.

2. https://fr.wikipedia.org/wiki/Echelon.

DOI:10.48556/SIF.1024.21.185 **1024** – Bulletin de la SIF, numéro 21, avril 2023, p. 185–197

SUJET PROPOSÉ EN 2026

permettant de se faire une idée sur les principales jusqu'aux années 1970 essentiellement, mais de se focaliser sur quelques moments charnières de cette histoire. L'attention est portée sur eux parce qu'ils créent d'une part un avant et un après sans retour puisque la science est incrémentale par essence, et que sa capacité à l'oubli est limitée, et, d'autre part, parce que leur concomitance illustre une accélération de l'état des connaissances.

La cryptographie jusqu'aux années 1970

Jusqu'aux années 1970 — c'est-à-dire sur une période couvrant plus de 3000 ans — le principe utilisé par les cryptosystèmes les plus performants résidait dans l'utilisation d'une clef secrète, servant aussi bien à chiffrer un message qu'à le déchiffrer. C'est essentiellement ce principe qui prévalait par exemple pour la machine Enigma utilisée par les forces allemandes durant la deuxième guerre mondiale. Les clefs secrètes étaient changées fréquemment afin de se prémunir contre les attaques éventuelles du système. Le film *The Imitation Game* [3], donne une idée de cette histoire, qui d'ailleurs rejoint la tragique histoire personnelle du mathématicien Alan Turing (1912 – 1954). La recherche de cryptosystèmes à clef secrète performants s'est poursuivie après la guerre, donnant lieu notamment à la publication du *Data Encryption Standard* (DES) en 1977 par le *National Institute of Science and Technology* (NIST), un organisme que nous retrouverons dans la suite de cet article.

Deux choses sont à noter à cet égard. La première est que, pour la première fois de manière aussi franche, le fonctionnement interne d'un algorithme destiné à devenir un standard était révélé à tous. Cela peut paraître surprenant car le monde du secret est presque ontologiquement allergique à ce type d'approche ouverte. Cependant, en procédant ainsi, le NIST appliquait les recommandations de Kerckhoff (1835 – 1903) selon lesquelles il faut partir du principe que l'ennemi connaît le système (ou le connaîtra), et donc que le secret ne doit résider que dans la clef, et pas dans le maintien dans l'obscurité du cryptosystème utilisé. La seconde est que l'adoubement par le NIST a fait *de facto* du DES le standard des algorithmes de cryptographie à clef secrète à compter de cette date non seulement aux USA, mais quasiment partout sur terre.

Cette domination du DES a duré presque une vingtaine d'années. Toutefois, les recherches se sont démultipliées, donnant lieu à des attaques de plus en plus fortes sur le DES. Le Triple-DES (une variante de DES) a alors été standardisé, ses tailles de clefs plus importantes permettant de résister encore pour un temps à ces attaques. Cependant, le coup d'arrêt est venu en 1999 avec la publication et l'utilisation du DES-Cracker. Cette machine avait été spécifiquement construite pour trouver les clefs secrètes du DES, et parvenait à le faire pour une poignée de dollars (un million de l'époque) et une poignée de jours (deux ou trois en moyenne). Il devint clair que

3. https://fr.wikipedia.org/wiki/Imitation_Game.

X 2026 – ADS informatique filière MPI – dossier 05 – page 03

SUJET X PSI 2026

les jours du DES ou du Triple-DES étaient comptés : le problème n'était plus seulement la taille devenue trop courte des clefs, mais résidait dans la conception même de ces algorithmes. Les avancées technologiques, l'augmentation exponentielle à l'époque de la puissance des ordinateurs (avec un doublement de la capacité des puces tous les 18 mois) faisaient que les deux ou trois jours nécessaires en moyenne pour trouver une seule clef du DES au moment de l'avènement du DES-Cracker seraient fortement réduits dans les années qui viendraient. Maintenir les choses en l'état laissait planer un risque trop grand sur la confidentialité des communications futures. Ouvrons une parenthèse en notant au passage que l'avènement du DES-Cracker a certainement eu des conséquences sur les communications passées : il est en effet probable que des instances gouvernementales aient stocké au fur et à mesure des années des messages, chiffrés par d'autres États avec le DES ou le Triple-DES, en attendant des jours meilleurs. Même si ces messages n'avaient d'impact que pour des périodes courtes, avoir soudain la possibilité de les lire grâce au DES-Cracker, ou à d'autres machines, a certainement été source de confirmation d'informations ou source de surprises... Quoi qu'il en soit, le NIST avait toutes les raisons de lancer alors un appel à candidatures pour le successeur du DES, l'*Advanced Encryption Standard* (AES), avec des critères plus sévères que ceux ayant prévalu pour le DES, et tenant compte des progrès de la science et ceux prévisibles (à l'époque) de la technologie. L'algorithme à clef secrète gagnant de la compétition fut standardisé en 2001.

Premier moment charnière : l'invention de la cryptographie à clef publique

Mais le véritable premier moment charnière de cette histoire ne vient pas de la standardisation du DES ou de l'AES, qui s'inscrivent, d'une certaine manière, dans la lignée de la cryptographie à clef secrète ayant prévalu au cours des millénaires précédents. Il vient de l'invention de la cryptographie à clef publique. Dans un article de 1976 cité plus de 8000 fois depuis sa parution, l'informaticien Diffie et le mathématicien Hellman en posaient la base [4]. L'idée est d'une simplicité biblique. Pourtant il aura fallu des millénaires pour qu'elle émerge. Elle revient à la question suivante : a-t-on nécessairement besoin de la même clef pour chiffrer que pour déchiffrer ? Les cryptosystèmes comme Enigma, DES, AES sont à clef secrète. Le principe de ces algorithmes et machines est que la même clef est utilisée pour ces deux opérations. Elle est connue de l'envoyeur et du destinataire, et c'est tout. Toute autre personne, notamment un attaquant, ne doit pas y avoir accès. Dans le cas contraire, la sécurité des messages est compromise. Maintenant imaginons avec Diffie et Hellman qu'il existe deux clefs par destinataire : une clef publique accessible à tous (publiée par exemple comme le sont les numéros de téléphone dans un annuaire), et une clef

X 2026 – ADS informatique filière MPI – dossier 05 – page 04

SUJET POSE EN 2026

privée connue de cette seule personne. Imaginons encore qu'Alice souhaite communiquer confidentiellement avec Bob (Alice et Bob font partie du jargon standard utilisé en cryptographie), et donc que Bob soit le destinataire auquel Alice envoie un message. Les deux clefs de Bob doivent être conçues de sorte que tout le monde puisse lui envoyer un message chiffré avec sa clef publique (celle qui figure en face du nom de Bob dans l'annuaire), mais que seul Bob puisse déchiffrer le message avec sa clef privée (connue de lui seul). Ce concept novateur, qui suppose évidemment un lien entre clef privée et clef publique, mais qui n'impose plus que ce lien soit l'égalité entre ces clefs (ce concept est clarifié un peu plus loin), est fondamental pour toute la cryptographie qui suivra. Il sera notamment rendu opérationnel dans des travaux de Rivest, Shamir et Adleman, alors au MIT, menant à ce qui s'appellera ensuite l'algorithme RSA d'après les initiales de leurs noms. RSA peut être utilisé aussi bien comme algorithme de chiffrement à clef publique que comme algorithme de signature électronique [10].

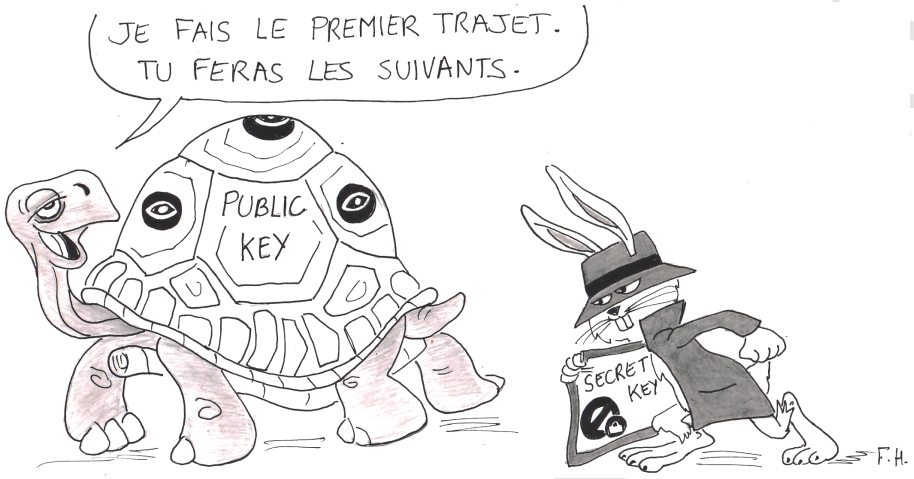
Avant de tirer les conséquences de ces découvertes, mentionnons un fait peu répandu mais significatif de l'environnement dans lequel évoluent ces sujets sensibles. D'une certaine manière, ce qui précède est la partie émergée de l'histoire. Il existe une partie beaucoup plus discrète[4] [14]. En effet, tout porte à croire que les britanniques James Ellis, Cliff Cocks et Malcolm Williamson avaient découvert l'essentiel des principes et algorithmes de la cryptographie à clef publique dès 1975 et donc avant quiconque. La raison d'État a imposé le silence sur près d'un quart de siècle, et ce fait n'a été révélé qu'en 1997. En effet, Ellis, Cocks et Williamson travaillaient pour le *Government Communications Headquarters* (GCHQ) à Cheltenham. Le GCHQ était (et est) le successeur de l'établissement secret de Bletchley Park, où Alan Turing travaillait pendant la guerre sur la machine Enigma. Dans ce milieu, le secret des travaux peut prévaloir (notamment sur le principe de Kerckhoff cité plus haut), que l'on soit en temps de guerre active, de guerre froide, ou autre.

Revenons aux conséquences de l'avènement de la cryptographie à clef publique. On aurait pu penser que ces découvertes mettraient au rebut les cryptosystèmes à clef secrète comme DES, Triple DES ou l'AES. En effet, les cryptosystèmes à clef secrète présentent intrinsèquement beaucoup de problèmes : ils ne permettent pas la signature électronique ; les messages sont compromis lorsqu'une clef est perdue (ou volée) ; ils présentent des difficultés pour l'échange des clefs secrètes sur lesquelles les différentes parties doivent convenir ; ils présentent des difficultés lorsqu'il faut inclure un nouveau participant dans une discussion chiffrée, ou au contraire l'en exclure ; etc. La plupart de ces problèmes disparaissent, ou du moins s'atténuent, avec les algorithmes à clef publique pour des raisons un peu longues à expliquer ici. Pourtant, les cryptosystèmes à clef secrète restent utilisés aujourd'hui pour plusieurs raisons. Pour n'en citer qu'une : la rapidité. Les cryptosystèmes à clef secrète sont

4. https://cryptome.org/ukpk-alt.htm.

X 2026 – ADS informatique filière MPI – dossier 05 – page 05

SUJET CONFIDENTIEL



plus rapides que ceux à clef publique : ils encryptent et décryptent jusqu'à parfois 100 fois plus vite. La situation est donc la suivante : d'une part, les algorithmes de cryptographie à clef secrète sont rapides, mais leur mise en place et leur gestion posent des problèmes importants ; d'autre part, les algorithmes à clef publique résolvent ces problèmes de mise en place et de gestion et font des choses en plus (signature électronique), mais sont lents.

La solution pour sortir de cette alternative est précisément de ne pas en sortir. Il convient en effet de ne pas concevoir les algorithmes de cryptographie à clef secrète et les algorithmes de cryptographie à clef publique comme étant exclusifs l'un de l'autre, mais au contraire de les combiner.

Par exemple, Alice souhaitant communiquer fréquemment avec Bob commencera par récupérer la clef publique de Bob. Puis, Alice construira un message M assez court — pour schématiser encore, imaginez quelque chose comme « *les sanglots longs des violons de l'automne* [5] » — et l'enverra chiffré à Bob, avec la clef publique de Bob bien entendu. De son côté, Bob pourra déchiffrer avec sa clef privée le message M. À ce stade, Alice et Bob connaissent la phrase « *les sanglots longs des violons de l'automne* » : Alice parce qu'elle l'a envoyée, et Bob parce qu'il l'a déchiffrée. Ils sont en outre les seuls à la connaître car toute autre personne ne voit passer qu'un message chiffré entre Alice et Bob. Imaginons que Alice et Bob aient convenu d'utiliser cette phrase connue d'eux seuls comme la clef secrète d'un algorithme à clef secrète, comme AES par exemple, sur lequel ils se sont aussi mis d'accord au préalable, et de l'utiliser pour tous les échanges de la journée. À partir de ce moment et jusqu'à la fin de la journée, aussi bien Alice que Bob n'utilisent plus que l'algorithme à clef secrète AES et la clef secrète commune dont ils disposent (« *les sanglots longs des violons de l'automne* »). Cela leur permet d'échanger beaucoup de messages très vite, grâce à l'AES. Ils ne se servent plus de l'algorithme à

5. https://fr.wikipedia.org/wiki/Chanson_d%27automne.

X 2026 – ADS informatique filière MPI – dossier 05 – page 06

SUJET ROSE

clef publique du tout de la journée, et ce jusqu'au lendemain matin, moment où ils recommenceront leurs échanges pour convenir d'une nouvelle clef secrète commune avec une autre phrase plus ou moins poétique. Ce changement journalier ou même plus fréquent qu'une fois par jour de la clef secrète d'un algorithme comme AES renforce la sécurité de l'ensemble de l'édifice ainsi mis en place. Dans leur article, Diffie-Hellman proposent une autre manière de procéder, dans laquelle Alice et Bob construisent conjointement la clef secrète M (à la différence du cas précédent, où Alice décide unilatéralement de la clef secrète, qu'elle envoie ensuite à Bob). Ce protocole d'échange de clef de Diffie-Hellman joue un rôle très important dans la suite de l'histoire que nous racontons.

Les méthodes ainsi esquissées, combinant algorithme de cryptographie à clef publique (comme par exemple RSA ou le protocole d'échange de clefs de Diffie-Hellman) et algorithme de cryptographie à clef secrète (comme par exemple AES) — ce qui fait au total une clef publique et une clef privée pour le premier algorithme, et une clef secrète pour le second algorithme — sont, pour l'essentiel, celles qui gouvernent les infrastructures à clef publique utilisées dans le monde aujourd'hui (par exemple LuxTrust [6] au Luxembourg, entreprise dont l'auteur de ces lignes a été membre du conseil d'administration pendant pas loin d'une dizaine d'années). La cryptographie à clef publique a connu un essor commercial, industriel, militaire, étatique colossal dans les années qui suivirent son invention. Ces architectures ont intégré de nombreuses applications. Les plus familières au grand public sont certainement celles ayant trait à l'e-commerce, aux déclarations fiscales en ligne, à l'e-banking pour n'en citer que quelques-unes. Les algorithmes de cryptographie à clef publique sont parmi les briques de base les plus importantes de la chaîne de sécurité des communications. Si ces algorithmes ont des failles, alors toute la chaîne de sécurité s'effondre comme un château de cartes.

Des problèmes mathématiques garants de la sécurité des cryptosystèmes à clef publique

Comment mesure-t-on la solidité de ces briques de base ? Le cœur de la chose réside dans le lien entre clef privée et clef publique des algorithmes de cryptologie à clef publique. Comme signalé plus haut, ce lien n'est pas un signe $=$ comme dans le cas de la cryptographie à clef secrète. Le lien est donné par des opérations mathématiques plus compliquées, et par des problèmes mathématiques complexes à résoudre en pratique.

Ces opérations portant sur des objets mathématiques ont, pour l'essentiel, deux propriétés : elles doivent être « faciles » à faire dans un sens, et « difficiles » ou « très difficiles » à faire en sens inverse. Par exemple, il est facile (et donc rapide) de multiplier l'entier $x = 13$ par l'entier $y = 17$ pour obtenir l'entier $z = 221$. Par contre,

6. https://www.luxtrust.com.

X 2026 – ADS informatique filière MPI – dossier 05 – page 07

SUJET POSE EN 2026

si l'on ne connait que la valeur de l'entier z, il est en pratique plus difficile (et donc plus lent) de retrouver les valeurs des entiers x et y qui le composent (il suffit d'essayer de se faire la main sur un autre exemple, tel $z = 437$). Le problème sous-jacent est le problème de la factorisation des entiers. Le protocole d'échange de clef de Diffie-Hellman repose sur d'autres problèmes mathématiques, impliquant des structures plus compliquées que les entiers, au nombre desquelles figurent les courbes elliptiques. Ici n'est pas le lieu de décrire ce que sont les courbes elliptiques. Toutefois, comme elles reviennent un peu plus tard dans cette histoire, il est nécessaire de citer leur nom.

Mais que signifie « facile », « difficile », voire « très difficile » ? Il est possible d'en donner une définition assez précise en termes de classes de complexité de problèmes, ou plutôt d'algorithmes destinés à résoudre ces problèmes. Disons pour simplifier qu'un problème résolu par un algorithme en temps polynomial en la taille des données est facile, qu'un problème dont le meilleur algorithme connu pour le résoudre nécessite un temps exponentiel est très difficile, et qu'un problème pour lequel le meilleur algorithme connu le résolvant nécessite un temps plus que polynomial mais moins qu'exponentiel (on parle alors de temps sous-exponentiel) est difficile. Le problème de la factorisation des entiers illustré précédemment est difficile. Ce problème spécifique est à l'origine de l'algorithme RSA, utilisé des centaines de millions de fois par jour tant pour la cryptographie à clef publique que pour les signatures électroniques. D'autres problèmes mathématiques ont donné naissance à d'autres algorithmes de chiffrement ou de signature électronique. Notamment les problèmes dits du logarithme discret dans des groupes (au sens mathématique du terme) bien choisis sous-tendent le protocole d'échange de clefs de Diffie-Hellman dans ces mêmes groupes. Les algorithmes de cryptage ou de signature électronique construits sur ces problèmes ont fait l'objet, tout comme RSA, d'un effort mondial de standardisation par différentes organisations. Le domaine a gagné en maturité, les standards restant pour l'essentiel stables tant qu'aucune attaque sérieuse n'est découverte. Plus généralement, l'évolution des standards mondiaux s'effectue en adaptant les recommandations au fur et à mesure de la découverte de cas particuliers de faiblesses (auquel cas des paramètres sont modifiés pour ne pas rentrer dans ces cas de faiblesses ponctuelles). Cette évolution se fait aussi en tenant compte de la montée en puissance des capacités de calcul des ordinateurs. En effet, tout comme on peut imaginer qu'un cambrioleur qui aurait à sa disposition l'ensemble des clefs de la création finirait par rentrer chez vous en les essayant toutes les unes après les autres, un attaquant peut essayer d'attaquer un cryptosystème par ce qu'on appelle *force brute* dans le jargon des spécialistes. Il essaye toutes les combinaisons possibles de clefs jusqu'à ce qu'il trouve la bonne. L'attaque par force brute marche toujours. Avec beaucoup de chance, la première tentative peut d'ailleurs être la bonne. Bien entendu, la question cruciale est le temps nécessaire en moyenne pour trouver cette clef. Avec l'augmentation des capacités de calcul mondial, ne rien faire expose à

X 2026 – ADS informatique filière MPI – dossier 05 – page 08

SUPPORT EN LIGNE

des attaques par force brute de plus en plus faciles et de moins en moins onéreuses. Toutefois, l'amélioration des processeurs permet aussi aux cryptosystèmes de remplir leurs missions plus vite, et donc autorisent des tailles de clefs plus grandes pour contrer les attaques par force brute rendues possibles par la puissance croissante des ordinateurs. Par conséquent, les recommandations s'attachent à maintenir un compromis entre l'efficacité du cryptosystème d'un côté, et la faisabilité des attaques de l'autre, en augmentant la taille des clefs périodiquement.

Donc, avec les ordinateurs tels que nous les connaissons actuellement, les algorithmes de cryptographie à clef publique standardisés tels que RSA, ou ceux dérivés des problèmes de logarithme discret dans des groupes bien choisis, dont ceux provenant de courbes elliptiques, continuent d'être utilisés avec des tailles de clefs qui augmentent de temps en temps. La raison de leur pérennité vient de ce que les meilleurs algorithmes pour casser les problèmes mathématiques, sur lesquels reposent leur sécurité, sont en temps sous-exponentiel ou exponentiel. Autrement dit, ces problèmes de factorisation ou de résolution du logarithme discret dans des groupes adaptés sont difficiles ou très difficiles en l'état actuel des connaissances sur les ordinateurs classiques. Un point supplémentaire au passage : de la même manière qu'il n'est guère prudent de mettre tous ses œufs dans le même panier, la communauté cryptographique apprécie d'avoir des algorithmes de cryptage de natures différentes pour réduire l'exposition des utilisateurs aux attaques : une attaque sur un système est en général sans impact sur un autre système dont la sécurité est basée sur un problème mathématique différent.

Deuxième moment charnière : les algorithmes de Shor

Cependant, des efforts très importants sont déployés depuis quelques années pour construire des ordinateurs quantiques (voir plus loin sur l'état actuel de l'art). Si on y arrive, les capacités de calcul de tels ordinateurs seront démultipliées par rapport à celles des ordinateurs actuels. Une ligne de recherche concerne la question de ce que l'on pourrait faire avec de tels ordinateurs. Plus précisément, il s'est agi de construire des algorithmes qui exploitent les caractéristiques quantiques de telles machines, alors même que les machines n'existaient pas (et que l'on ne savait absolument pas si on pourrait en construire un jour). Dans ce contexte, Peter Shor publia à la fin des années 1990 des algorithmes qui résolvent en temps polynomial les problèmes mathématiques sous-jacents aux cryptosystèmes à clef publique usuels — factorisation des entiers, logarithme discret sur les groupes appropriés – sous l'hypothèse d'avoir à disposition un ordinateur quantique performant [13] (voir [8] pour une vulgarisation sur ces algorithmes, voir également [15, 1]). La conséquence est que ces méthodes de Shor rendraient caduques les cryptosystèmes correspondants : RSA, Diffie-Hellman sur le groupe multiplicatif d'un corps fini ou sur le

X 2026 – ADS informatique filière MPI – dossier 05 – page 09

EPREUVE X 2026

groupe des points d'une courbe elliptique sur un corps fini. Notons au passage que les efforts nécessaires pour casser ces cryptosystèmes à l'aide des algorithmes de Shor sur un ordinateur quantique peuvent différer de l'un à l'autre [12]. Les algorithmes de Shor et le fait que des investissements substantiels étaient effectués pour tenter de construire des ordinateurs quantiques ont conduit le NIST à se soucier de la sécurité des communications dans un monde post-quantique.

En 2016, le NIST a publié un appel à candidature [7] pour des cryptosystèmes permettant l'encryption, l'échange de clefs, et les signatures électroniques, qui résisteraient à l'existence d'un ordinateur quantique, autrement dit des systèmes basés sur des problèmes mathématiques qui ne s'exposent notamment pas aux attaques du type de celles de Shor. Le processus de sélection devant mener aux nouveaux standards s'est fait par étapes (comme toujours). Parmi les 69 candidatures reçues, le premier tour d'évaluation (décembre 2017 – janvier 2019) a conduit le NIST à retenir 26 propositions pour le deuxième tour. Celui-ci s'est poursuivi jusqu'au mois de juillet 2020. Il a permis au NIST de sélectionner 7 finalistes et 8 candidats alternatifs pouvant participer au troisième tour. Le 5 juillet 2022 est une date importante dans notre narration : ce jour-là, le NIST publia son rapport d'évaluation sur ces 15 propositions [8]. Ses conclusions furent que l'algorithme à clef publique et l'algorithme d'échange de clefs qui serait standardisé serait CRYSTALS-Kyber, et que les trois algorithmes de signature électronique CRYSTALS-Dilithium, Falcon et SPHINCS+ seraient aussi standardisés. Au-delà de ces décisions, le NIST sélectionna aussi quatre algorithmes d'échange de clefs pour un quatrième tour d'évaluation — BIKE, Classic McEliece, HQC et SIKE — et annonça que ces algorithmes pourraient aussi être pris en considération pour une standardisation future. Pour être complet, notons que ce quatrième tour est aussi ouvert à de nouvelles propositions.

Troisième moment charnière : la fin de SIDH

Comme on l'imagine aisément, comprendre comment fonctionnent ces algorithmes, quels sont leurs avantages respectifs, et comment ils se comparent les uns aux autres nécessite un certain effort, dont nous faisons l'économie ici. Mettons toutefois l'accent sur l'un d'entre eux, SIKE, retenu début juillet 2022 pour le quatrième tour d'évaluation et donc susceptible de faire partie des standards (les autres algorithmes retenus ne font pas l'objet du présent article). Les mathématiques sur lesquelles cet algorithme s'appuie portent sur les courbes elliptiques sur des corps finis. De tels objets ont déjà démontré leur puissance par le passé : elles ont

7. https://csrc.nist.gov/News/2016/Public-Key-Post-Quantum-Cryptographic-Algorithms.

8. https://www.nist.gov/publications/status-report-third-round-nist-post-quantum-cryptography-standardization-process.

X 2026 – ADS informatique filière MPI – dossier 05 – page 10

SUJET POSTPRINT

été utilisées pour prouver la conjecture de Fermat ; elles le sont aujourd'hui pour prouver que des nombres entiers de plusieurs milliers de chiffres sont premiers (pour donner une idée de comparaison, on estime que le nombre de particules dans l'univers observable est de l'ordre d'un nombre ayant au plus 81 chiffres : bien petit à l'égard des milliers de chiffres dont il est question plus haut). Pour ce qui concerne SIKE, il s'agit en l'occurrence, en termes techniques, d'un cryptosystème qui formalise le protocole *Supersingular Isogeny Diffie-Hellman* (SIDH). Le SIDH consiste essentiellement à prendre en compte deux courbes elliptiques et à créer un algorithme à clef publique dont la clef privée est constituée par une application entre ces deux courbes et compatible avec leurs structures algébriques, appelée une isogénie. Ce sujet de recherche, qui s'intègre dans l'ensemble plus vaste de la cryptographie à base de courbes elliptiques, est devenu très actif depuis une bonne quinzaine d'années, donnant lieu à des dizaines de conférences internationales et des centaines d'articles scientifiques.

Quelques jours après la publication du rapport du NIST, trois articles parurent coup sur coup, qui eurent un certain retentissement dans le petit monde de la cryptographie. Le 30 juillet, Castryck et Decru publièrent le premier de la série [3]. De manière indépendante, Maino et Martindale arrivaient à des conclusions comparables le 8 août [9]. Deux jours plus tard, Robert apportait le coup de grâce [11]. Au final, SIDH est cassé en temps polynomial. Il semble donc que le protocole SIDH ait vécu, et qu'il emporte avec lui SIKE. En effet, les auteurs des articles précités ont montré comment casser ce candidat à la standardisation post-quantique sur un ordinateur classique : moins d'une heure de calcul sur un ordinateur portable suffit. Comment en est-on arrivé là ? Dans les trois cas, le point essentiel utilisé par les auteurs est une technique développée par Kani en 1997 pour « coller et séparer » des courbes elliptiques entre elles [7]. Dans un travail commun avec Howe et Poonen [5], nous avions écrit précisément les formules permettant de « recoller » un produit de courbes en une jacobienne de courbe de genre plus élevé. Les auteurs des trois papiers publiés en 2022 utilisent ces considérations très techniques avec beaucoup d'élégance et d'efficacité pour retrouver la clef privée du protocole.

Ouvrons une parenthèse sous la forme d'une méta-remarque et d'une observation menant à une conjecture.

La méta-remarque est que plus un objet mathématique possède de structures algébriques, plus grandes sont les chances de le cerner. Les courbes elliptiques supersingulières, utilisées dans SIDH, offrent des structures plus riches que les autres courbes elliptiques. Ce sont précisément ces structures additionnelles, se traduisant par le S dans SIDH, qui ont été exploitées dans les attaques. SIDH est complètement cassé, et probablement irrécupérable pour les applications cryptographiques, car son design nécessite précisément de telles structures riches.

X 2026 – ADS informatique filière MPI – dossier 05 – page 11

SUJET PROPRIÉTAIRE

L'observation est que l'attaque de Shor sur les cryptosystèmes classiques utilise les propriétés quantiques des ordinateurs quantiques, alors que l'attaque sur le cryptosystème post-quantique SIDH ne s'est aucunement faite en exploitant des aspects quantiques (le paragraphe suivant arrive en soutien de cette observation). La conjecture qui se déduit de cette double observation est que des attaques sur certaines instanciations de cryptosystèmes post-quantiques, tels ceux faisant l'objet ou restant en course pour la standardisation du NIST, exploiteront, à la Shor, les capacités quantiques des ordinateurs quantiques. Dit autrement, si des algorithmes comme ceux de Shor cassent les cryptosystèmes classiques en utilisant des propriétés quantiques, on imagine aisément que des cryptosystèmes post-quantiques souffriront de l'exploitation mathématique des propriétés quantiques. Fin de la parenthèse.

Revenons à la fin de SIDH par les attaques de l'été 2022. Galbraith [9], l'un des meilleurs spécialistes mondiaux de ces sujets, positionne ces attaques dans une perspective historique par une déclaration pertinente à la fin de la page de son blog dédié à ce sujet : « *Je veux insister sur le fait que cette attaque est seulement possible en raison de décennies de recherche sur le calcul d'isogénies sur des variétés abéliennes générales. Ces travaux n'étaient pas motivés en premier lieu par leurs applications en cryptographie, mais par le désir de généralisation en mathématiques pures* ». En effet, aussi bien Kani que Howe, Poonen et moi-même (et certainement plusieurs autres chercheurs cités dans les références des articles de l'été 2022) n'avions aucunement en tête les applications qui ont émergé cet été lorsque nous travaillions sur ces objets il y a plus de vingt ans.

Cependant, c'est avec grand plaisir et même, avouons-le, une pointe de fierté (et aussi, avouons-le également, une pointe de regret, celle de ne pas avoir procédé nous-même aux attaques évoquées ci-dessus) que nous voyons (en tout cas que je vois) des travaux en mathématiques pures (si tant est qu'il y ait des mathématiques impures...) menés dans les années 2000 sans pression, sans nécessité d'applications industrielles à court terme, libres de toute autre motivation que la quête du savoir, porter leurs fruits 20 ans plus tard dans un domaine crucial pour la souveraineté des nations, la protection des citoyens et des industries, et même la préservation de la liberté.

Cette histoire illustre ainsi en premier lieu un phénomène bien connu des chercheurs, et souvent ignoré ou — pire — dénigré par nombre de décideurs en matière de politique de recherche, préoccupés de retombées immédiates. Il existe souvent un continuum de la recherche fondamentale aux applications. Mais il faut le laisser mûrir à son rythme. Cela demande de laisser les chercheurs travailler sereinement sur des défis extrêmement difficiles.

Deuxièmement, gouverner c'est prévoir. L'initiative du NIST concernant la cryptographie post-quantique a été lancée en 2016 alors que les qubits — les fameux *quantum bits*, jouant pour les ordinateurs quantiques le rôle que les bits 0 et 1 jouent

9. https://ellipticnews.wordpress.com/2022/08/12/attacks-on-sidh-sike/.

X 2026 – ADS informatique filière MPI – dossier 05 – page 12

SUJET PROPRIÉTAIRE

pour les ordinateurs classiques — en étaient encore à leurs balbutiements. Aujourd'hui, plus précisément en date du 11 novembre 2022, le record est détenu par la machine Osprey de IBM opérant sur 433 qubits. IBM a déjà annoncé que des travaux étaient en cours pour monter à 1000 qubits, et est en train de revoir à la hausse ses prévisions dans le domaine. Si l'émergence d'ordinateurs quantiques performants se fait encore attendre, si de nombreuses difficultés techniques restent à surmonter, notamment en ce qui concerne la correction des erreurs [2], si les machines quantiques actuelles ne permettent pas de factoriser de grands nombres (un ordinateur quantique est capable de factoriser 15 et d'obtenir 3×5, idem avec $21 = 3 \times 7$, mais guère mieux pour le moment), les progrès des dernières années sont notables et prometteurs. Si des ordinateurs quantiques performants existent un jour, leur impact ne sera pas uniquement de casser des cryptosystèmes, et de forcer à en créer d'autres. Ils permettront aussi d'aborder de très nombreux problèmes que les ordinateurs classiques actuels ne sont pas en mesure de résoudre, et cela dans des domaines variés : prévision météorologique, recherche nucléaire, médecine, science des matériaux, etc.

Enfin, et ce dernier point est moins universellement partagé à mon grand regret, de manière plus générale, la seule recherche qui compte est celle où le seul mot d'ordre est la qualité, et rien d'autre. Dans tout ce qui précède, dans l'histoire que nous avons contée, nous sommes bien éloignés des délires délétères et destructeurs *woke* sur lesquels nous nous sommes exprimés ailleurs [10]. Ces délires du *wokisme* font pourtant actuellement leur lit jusque dans les universités ou revues scientifiques les plus connues, définissent de nouveaux critères retenus dans les politiques nationales ou européennes de financement de la recherche, guident plus généralement ce que les individus, chercheurs ou pas, doivent penser, et culpabilisent ou décrédibilisent les déviants. Laisser aux tenants de l'idéologie *woke* les rênes du pouvoir équivaut à une programmation de la médiocrité, et au déni du réel, scientifique ou non.

Remerciements

L'auteur remercie A. Atashpendar, A. Biryukov, D. Mestel, L. Goubin, J.-Ch. Pomerol et P. Ryan pour leurs éclaircissements sur des aspects relatifs aux performances actuelles des ordinateurs quantiques, pour leurs indications bibliographiques, et pour leur intérêt pour ce travail.

Références

[1] Mirko Amico, Zain H. Saleem, and Muir Kumph. Experimental study of Shor's factoring algorithm using the IBM q experience. *Physical Review A*, 100(1), jul 2019.

10. https://www.causeur.fr/raison-detre-des-universites-occidentales-a-lheure-du-wokisme-225081.

X 2026 – ADS informatique filière MPI – dossier 05 – page 13

SUJET POSÉ EN 2026

[2] Earl T. Campbell, Barbara M. Terhal, and Christophe Vuillot. Roads towards fault-tolerant universal quantum computation. *Nature*, 549(7671) :172–179, sep 2017.

[3] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Paper 2022/975, 2022. https://eprint.iacr.org/2022/975.

[4] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6) :644–654, 1976.

[5] Everett W. Howe, Franck Leprévost, and Bjorn Poonen. Large torsion subgroups of split jacobians of curves of genus two or three. *Forum Mathematicum*, 12 :315–364, 1998.

[6] David Kahn. *The Codebreakers : The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, rev sub edition, December 1996.

[7] Ernst Kani. The number of curves of genus two with elliptic differentials. *Journal für die reine und angewandte Mathematik*, 1997(485) :93–122, 1997.

[8] Franck Leprévost. Peter Shor, prix Nevanlinna 1998. *La Gazette des mathématiciens*, 81 :40–47, 1999.

[9] Luciano Maino and Chloe Martindale. An attack on SIDH with arbitrary starting curve. Cryptology ePrint Archive, Paper 2022/1026, 2022. https://eprint.iacr.org/2022/1026.

[10] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2) :120–126, feb 1978.

[11] Damien Robert. Breaking SIDH in polynomial time. Cryptology ePrint Archive, Paper 2022/1038, 2022. https://eprint.iacr.org/2022/1038.

[12] Martin Roetteler, Michael Naehrig, Krysta M. Svore, and Kristin Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. *Advances in Cryptology, ASIACRYPT 2017*, 559 :241–270, 2017.

[13] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5) :1484–1509, oct 1997.

[14] Simon Singh. *The Code Book : The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography*. Doubleday, USA, 1st edition, 1999.

[15] John A. Smolin, Graeme Smith, and Alexander Vargo. Oversimplifying quantum factoring. *Nature*, 499(7457) :163–165, July 2013.

SUJET POSTFIN 2026

Publié le 30/03/2018)i(interstices Par André Chailloux

L'algorithme quantique de Shor

ALGORITHMES | SÉCURITÉ & VIE PRIVÉE # Quantique

Les avancées aussi bien théoriques que technologiques dans le domaine de la physique quantique nous permettent de contrôler ces phénomènes quantiques et même d'envisager d'utiliser cette technologie dans nos ordinateurs, aussi bien pour sécuriser nos communications que pour créer des supercalculateurs. Comment fonctionnerait un tel ordinateur ? Quel impact aurait-il pour l'informatique et quelles sont les perspectives réalistes de l'informatique quantique ?

L'ordinateur quantique

La physique quantique est considérée comme l'une des théories physiques les plus mystérieuses des temps modernes. Pour comprendre le fonctionnement d'un ordinateur quantique, il faut se placer à l'échelle atomique et subatomique, dans le monde physique des électrons, des photons et autres particules élémentaires. La physique quantique, développée au début du XXe siècle, décrit certains phénomènes fortement contre-intuitifs se déroulant à cette échelle. Par exemple, la physique quantique prévoit que certaines particules puissent être dans une superposition d'états. Concrètement, un électron peut donc être excité ou au repos en même temps, un photon peut être à la fois présent et absent. C'est difficile d'imaginer ce qu'être et ne pas être en même temps veut dire. Schrödinger, l'un des pères de la physique quantique, imagine un chat mort et vivant à la fois pour expliquer ce phénomène. Ce qui n'est certainement pas possible pour les chats l'est par contre pour des particules élémentaires obéissant aux lois de la physique quantique. Par exemple, 30 particules subissant ces effets quantiques peuvent exister dans des milliards d'états différents. Il est alors tentant d'utiliser ce phénomène pour faire de nombreux calculs en même temps grâce à la superposition quantique. Par contre, ces effets quantiques sont extrêmement fragiles. N'importe quelle interaction avec l'environnement extérieur les détériore.

L'idée de faire du calcul quantique apparaît dans les années 1970. On a développé des modèles théoriques qui décrivent très bien ce que pourrait faire un tel ordinateur quantique, quel temps de calcul est nécessaire pour réaliser ces opérations et avec quelles ressources. Dès 1984, les cryptologues Charles H. Bennett et Gilles Brassard proposent le premier protocole d'échange de clés sécurisé grâce à la communication quantique. Ce protocole ne nécessite pas de calcul quantique à proprement parler mais utilise des états quantiques uniquement pour détecter quiconque voudrait espionner des communications. Le point culminant de ces recherches est la découverte en 1994 d'un algorithme quantique de factorisation par Peter Shor, chercheur au MIT. Cet algorithme montre qu'en contrôlant quelques milliers de particules, on serait capable de résoudre le problème de factorisation sur un ordinateur quantique et ainsi de casser une grande partie des systèmes de sécurité informatique utilisés aujourd'hui.

L'algorithme quantique de Shor

L'algorithme de Shor vise à résoudre le problème de factorisation, utilisé dans la majorité de nos systèmes de sécurité informatique. Concrètement, étant donné deux nombres p et q premiers — c'est-à-dire divisibles uniquement par eux-mêmes et par 1 — il est facile de calculer le produit $N = p \times q$ avec des algorithmes rapides de multiplication. Le problème de factorisation est le

SUJET POUR L'EN 2026

problème inverse : étant donné N, retrouver p et q. Ce problème est, aujourd'hui, difficile à résoudre. On est capable avec les algorithmes et les supercalculateurs d'aujourd'hui, de factoriser des nombres ayant environ 250 chiffres. Par contre, dès qu'il y a un trop grand nombre de chiffres, disons au-delà de 500 chiffres, cette factorisation devient impossible…

En 1994, Peter Shor a montré qu'avec un ordinateur quantique, on pourra factoriser des nombres qui auront des milliers voire des millions de chiffres. Shor utilise le lien entre le problème de factorisation et la recherche d'une période dans une fonction. En utilisant la superposition quantique, il montre comment faire résonner des états quantiques autour de la période de la fonction pour la retrouver, et ensuite utiliser ce résultat pour factoriser rapidement. À titre de comparaison, les meilleurs algorithmes actuels pour factoriser un nombre de 600 chiffres ont besoin de millions de millions d'années pour factoriser un tel nombre, temps évidemment inatteignable alors qu'un ordinateur quantique entièrement fonctionnel pourrait effectuer cette opération en quelques minutes.

La découverte de Shor est l'une des motivations principales pour construire un ordinateur quantique. Quelle conséquence cet algorithme a-t-il sur les systèmes de sécurité informatique ?

Faut-il modifier les systèmes de sécurité informatique ? Ou pourquoi être en avance sur son temps n'est plus suffisant.

De nombreux systèmes de sécurité informatique reposent sur des hypothèses calculatoires, c'est-à-dire sur l'incapacité des ordinateurs actuels à résoudre certains problèmes mathématiques. On entend souvent parler d'attaques cryptographiques et de failles de sécurité mais très rarement d'attaques sur ces hypothèses calculatoires proprement dites. En clair, les fondements mathématiques de nos systèmes de sécurité informatique sont relativement solides. Par exemple, l'algorithme cryptographique RSA, utilisé dans de nombreux protocoles sur Internet, repose sur la difficulté de factoriser de grands nombres.

L'arrivée d'un ordinateur quantique diminuerait donc fortement les hypothèses calculatoires sur lesquelles on peut se baser pour garantir la sécurité de nos systèmes informatiques. En particulier, RSA, ainsi que d'autres systèmes basés sur le logarithme discret par exemple, sont inutilisables. En quoi ces attaques nous concernent-elles ? Ne peut-on pas attendre l'arrivée éventuelle d'un ordinateur quantique et modifier nos systèmes de sécurité le cas échéant ?

Pendant longtemps, on considérait qu'être en avance sur les attaques potentielles était suffisant. Par exemple, il est aujourd'hui possible de casser les schémas utilisés en cryptographie il y a 20 ou 30 ans, sans que cela ne pose un grave problème de sécurité. En effet, tous les systèmes obsolètes ont été depuis longtemps remplacés.

Par contre, actuellement, une grande quantité de données sensibles et chiffrées sont accessibles facilement. Quelqu'un pourrait stocker ces données et attendre 20-30 ans des avancées potentielles — comme l'ordinateur quantique — pour déchiffrer ces données. Ceci peut devenir très problématique si ces données sensibles sont par exemple des données médicales, politiques ou d'autres données sensibles qui doivent rester secrètes pendant longtemps.

Heureusement, les ordinateurs quantiques ne sont pas tout-puissants. Il existe en effet des problèmes calculatoires qui restent difficiles même pour un ordinateur quantique. Un effort important est fait pour étudier précisément ces systèmes cryptographiques résistant à l'ordinateur quantique, et la normalisation de tels systèmes est en cours. Notamment, un système d'échange sûr contre les ordinateurs quantiques a été utilisé dans le navigateur Chrome, ce qui montre leur faisabilité pratique.

Le NIST (*National Institute of Standards and Technologies*) vient de lancer une campagne internationale pour standardiser la cryptographie sûre contre les ordinateurs quantiques, ce qui montre le sérieux avec lequel cette menace est considérée.

Des perspectives pour l'ordinateur quantique

On entend beaucoup parler de l'ordinateur quantique. Néanmoins, sa concrétisation technologique semble encore lointaine. En effet, les particules quantiques en superposition sont extrêmement fragiles. N'importe quelle interaction avec l'environnement détériore cette superposition jusqu'à enlever les propriétés quantiques de ces particules. On sait manipuler quelques états quantiques en même temps mais la route est encore longue avant de pouvoir calculer efficacement avec un ordinateur quantique. La seule chose qu'on arrive assez bien à faire est la communication quantique, qui permet de faire de l'échange de clés sécurisé. Mais là encore, l'absence de répéteurs dans les réseaux quantiques limite fortement les distances, et donc l'applicabilité de ces schémas.

Jusqu'à récemment, la construction d'un calculateur quantique relevait principalement du domaine académique mais maintenant, tous les géants de l'informatique, Google, Microsoft, IBM et bien d'autres, se sont lancés dans la course à la construction d'un ordinateur quantique et y investissent des milliards de dollars. IBM a annoncé en novembre 2017 avoir construit un ordinateur quantique de 50 qubits et Google a présenté en mars 2018 l'ordinateur *Bristlecone* capable d'opérer sur 72 qubits. Ces ordinateurs n'ont pas encore pu être testés donc on ne connait pas encore le taux d'erreur ou la durée de vie de ces qubits. En tout cas, l'enjeu principal est de trouver une technologie qui passe bien à l'échelle, aussi bien en termes de nombre de bits quantiques que de durée de vie des états quantiques utilisés.

L'ordinateur quantique capable de faire fonctionner l'algorithme de Shor est donc largement hors de portée. Néanmoins, il y a aussi d'autres applications, notamment pour la simulation de molécules chimiques, pour la conception de nouveaux médicaments, ou pour les algorithmes d'apprentissage par exemple. Avec quelques centaines de bits quantiques stables, on pourrait obtenir des gains importants en termes de puissance calculatoire. En parallèle, D-WAVE, une entreprise canadienne, annonce faire tourner une machine avec plus de 1000 bits quantiques simultanément. Cette machine n'est néanmoins pas un calculateur quantique complet et s'attaque à un problème d'optimisation très particulier via du recuit simulé quantique.

Au final, l'ordinateur quantique sera-t-il l'ordinateur de demain ou restera-t-il dans le domaine de la science-fiction ? Il est difficile de répondre à cette question tant les défis technologiques semblent immenses. Néanmoins, les différentes approches et l'investissement massif mis dans cet objectif sont très encourageants. Un enjeu à court terme est la simulation de certaines molécules soumises aux effets de la physique quantique, avec des applications en chimie pour la confection de médicaments ou de batteries par exemple. Si l'on arrive d'ici 5 à 10 ans à atteindre cet objectif en utilisant un ordinateur quantique rudimentaire, alors ce sera de très bon augure pour l'arrivée future d'un ordinateur quantique plus puissant dont l'existence aura des conséquences encore largement inexplorées.

SUJET POLYTECHNIQUE 2026

Cyptographie post-quantique

La cryptographie post-quantique est une branche de la cryptographie visant à garantir la sécurité de l'information face à un attaquant disposant d'un ordinateur quantique. Cette discipline est distincte de la cryptographie quantique, qui vise à construire des algorithmes cryptographiques utilisant des propriétés physiques, plutôt que mathématiques, pour garantir la sécurité.

En l'effet, les algorithmes quantiques de Shor, de Grover et de Simon étendent les capacités par rapport à un attaquant ne disposant que d'un ordinateur classique. S'il n'existe pas à l'heure actuelle de calculateur quantique représentant une menace concrète sur la sécurité des cryptosystèmes déployés, ces algorithmes permettent conceptuellement de résoudre certains problèmes calculatoires sur lesquels sont fondés plusieurs primitives populaires.

Introduction

Pourquoi développer une cryptographie post-quantique ?

De nombreux systèmes de chiffrement sont, à terme, menacés par les capacités de calcul de l'informatique quantique, domaine encore émergeant, mais où des gains progressifs dans la réduction du taux d'erreur sont constatés. Fin mars 2026, dans un livre blanc [1] dont le titre se traduit par *Sécurisation des cryptomonnaies à courbe elliptique contre les vulnérabilités quantiques : Estimation des ressources et mesures d'atténuation*, des chercheurs de Google Quantum AI, de la fondation Ethereum, et des universités de Stanford et de Californie alertent sur le fait qu'en raison des progrès de l'informatique quantique (nouvelles techniques de correction d'erreurs et d'optimisation des algorithmes quantiques), les futurs ordinateurs quantiques pourraient casser la cryptographie à courbes elliptiques (ECC) avec beaucoup moins de ressources et donc plus tôt qu'on ne le pensait — et peut-être en quelques minutes seulement. Or, la cryptographie sur les courbes elliptiques (ECC) encore utilisée en 2026 par la majorité des blockchains, dont Bitcoin, Ethereum, Solana, Cardano, Polkadot.

L'algorithme de Shor appliqué au problème mathématique qui sécurise aujourd'hui la plupart des blockchains, l'ECDLP-256, peut être exécuté avec moins de 1 200 qubits logiques et 90 M de portes de Toffoli[1] ou 1 450 qubits logiques et 70 M de portes de Toffoli. Cela correspond à moins de 500 000 qubits physiques, soit environ vingt fois moins de qubits que ce que les estimations précédentes estimaient nécessaires. Et l'opération serait exécutable sur une architecture supraconductrice réaliste (un ordinateur quantique doté de 1 200 à 1 450 qubits logiques — soit environ 500 000 qubits physiques dans une architecture supraconductrice). Dans ces conditions, d'après un modèle vérifié par Google via une preuve à divulgation nulle de connaissance, Shor briserait alors l'ECDSA ou l'EdDSA en quelques minutes. En terme de vitesse de calcul, il existe deux types d'ordinateurs quantiques : les plus rapides pourraient alors intercepter des transactions en cours. L'étude montre aussi que certaines fonctions avancées des blockchains modernes présentent des failles, et que les so-

1. La porte de Toffoli est une porte logique réversible et universelle, ce qui signifie que n'importe quel circuit réversible peut être construit à partir de portes de Toffoli. Elle a trois 3 bits en entrée et 3 bits en sortie. Les deux premiers bits de sortie sont égaux aux deux premiers bits d'entrée ; quant au troisième, il est égal au troisième bit d'entrée sauf si les deux premiers bits sont égaux à 1, auquel cas il est inversé.

TOUTE PUBLIQUE FIN 2026

lutions techniques devront être accompagnées de règles publiques pour gérer, par exemple, les cryptomonnaies stockées sans avoir été utilisées, ou oubliées. Elle conclut que toutes les cryptomonnaies encore vulnérables doivent passer « au plus vite » à une sécurité résistante aux futurs ordinateurs quantiques. En effet, presque tous les systèmes de cryptages et de mots de passe (y compris dans la finance traditionnelle et dans une partie de la FinTech) sont concernés, et comme la presque totalité des blockchains repose sur l'ECC (ECDLP-256) pour crypter et protéger les signatures de transactions, sécuriser les portefeuilles de cryptomonnaies et protéger les clés publiques des détenteurs de cryptomonnaies, presque toute la valeur économique du secteur des cryptomonnaies et autres cryptoactifs serait menacée par la mise au point d'un tel ordinateur. La migration vers une cryptographie post-quantique pourrait être couteuse et complexe, mais Google souhaite alerter sans paniquer ; encourageant une transition rapide (horizon 2029) et coordonnée vers la cryptographie post-quantique (PQC, déjà bien étudiée et standardisée). Selon les auteurs du livre blanc, ce moment est une singularité de discontinuité dans l'histoire de la cybersécurité, qu'il faut anticiper dès que possible, pour notamment sécuriser les blockchains.

Notations pour la complexité

La *notation L* est un analogue aux notations de Landau en notation asymptotique. Cette notation a été introduite par Carl Pomerance en 1982 [2] pour comparer différents algorithmes de factorisation et a été généralisée à deux paramètres par Arjen Lenstra et Hendrik Lenstra. Elle est définie comme suit pour $\alpha \in [0,1]$ et $c \in \mathbb{R}_+$:

$$L_n[\alpha, c] = \exp\left((c + o(1))\ln(n)^{\alpha}(\ln\ln(n))^{1-\alpha}\right).$$

On peut remarquer que contrairement à la notation O de Landau, la constante est ici importante puisqu'elle se retrouve dans l'exponentielle, ce qui fait qu'un gain sur la constante peut réduire considérablement les temps de calcul.

La notation L est principalement utilisée en théorie algorithmique des nombres, où elle permet de donner une échelle entre les différents algorithmes exponentiels. En effet, $L_n[0,c] = \ln(n)^{c+o(1)}$ représente les fonctions polynomiales de $\ln(n)$ de degré c, et $L_n[1,c] = n^{c+o(1)}$ les fonctions exponentielles en la taille de l'entrée (ici, la taille est prise comme étant $\log_2 n$, ce qui est le cas pour la représentation binaire des nombres), donc $L_n[1,c]$ représente les fonctions polynomiales de n de degré c. Par exemple :

- $L_n\left[0,c\right] = e^{(c+o(1))\ln\ln n} = (\ln n)^{c+o(1)}$
- $L_n\left[1,\frac{1}{2}\right] = e^{(\frac{1}{2}+o(1))\ln n} = n^{1/2+o(1)}$
- $L_n\left[\frac{1}{2},c\right] = e^{(c+o(1))\sqrt{\ln(n)\ln(\ln(n))}}$

Algorithmes quantiques et cryptographie

En 1995, Peter Shor a publié un algorithme permettant de résoudre le problème du logarithme discret et le problème de la factorisation des entiers. Cet algorithme, utilisant les spécificités d'un calculateur quantique (dont on n'avait alors pas encore construit de prototypes), produit une réponse en temps polynomial. En comparaison, les meilleurs algorithmes classiques connus pour ces problèmes ont une complexité exponentielle en général, et sous-exponentielle mais sur-polynomiale pour les corps finis. En résumé, pour

SUJET PROPOSÉ EN 2026

ces deux problèmes, aucun algorithme classique efficace n'est connu, mais puisqu'un algorithme quantique efficace est disponible, on dit qu'ils appartiennent à la classe de complexité BQP. En conséquence, les constructions cryptographiques dont la sécurité repose sur le problème du logarithme discret (notamment l'échange de clef de Diffie-Hellman et la signature ECDSA), ou sur le problème de la factorisation d'entiers (notamment la signature RSA) seraient vulnérables à un attaquant disposant d'un calculateur quantique suffisamment grand et fiable.

L'algorithme de Grover, quant à lui, améliore de manière faible mais générique l'efficacité des problèmes de recherche. Contrairement à l'algorithme de Shor, il ne dote pas l'attaquant d'un avantage exponentiel, mais seulement quadratique, c'est-à-dire qu'il divise le niveau de sécurité d'un problème de recherche par deux.

Cryptographie symétrique

En 2018, le principal algorithme quantique affectant les cryptosystèmes symétriques est l'algorithme de Grover. Cet algorithme est générique, c'est-à-dire qu'il s'applique en principe à toute primitive indépendamment de sa construction, et permet d'effectuer une recherche parmi n éléments en temps $\mathrm{O}(\sqrt{n})$ à la place de $\mathrm{O}(n)$ dans le cas classique.

Par conséquent, la sécurité des algorithmes n'est affectée que par un facteur 2. Autrement dit, pour atteindre la même sécurité que dans le modèle classique, il suffit de doubler la taille des clés utilisées, ce qui ne pose aucune difficulté conceptuelle, mais peut nécessiter des ajustement techniques. En effet, de nombreux algorithmes symétriques sont conçus et analysés avec une taille de clef fixe.

Dans certains cas, il est possible qu'une construction symétrique soit sujette à des attaques spécialisées par l'algorithme de Simon. Enfin, l'algorithme d'Ambainis permet de gagner un facteur polynomial dans la recherche de collisions.

Cryptographie à clef publique

Pour les problèmes de la factorisation d'un entier n, le meilleur algorithme classique connu est en complexité temporelle heuristique $L_n\left[\frac{1}{3}, \sqrt[3]{\frac{64}{9}}\right]$, ce qui permet d'appuyer la sécurité d'algorithmes tels que la signature de Rabin[2] à 128 bits en utilisant un module n de 3 072 bits. La sécurité d'une signature RSA utilisant un tel module est alors d'au plus 128 bits face à un attaquant classique. Le meilleur algorithme quantique connu possède une complexité quadratique en n ; adapter la sécurité de RSA à ce contexte serait possible, mais requiert des tailles de clef phénoménales [3], de l'ordre du téraoctet. En pratique, au-delà des problèmes de stockage et de transmission de telles clés et de tels messages, l'inconvénient est que toutes les opérations sont ralenties d'autant. De plus, il n'existe pas de preuves que l'algorithme de Shor soit le plus efficace possible : chercher à sauver RSA dans ce contexte est une course perdue d'avance.

De manière générale, les algorithmes à clef publique reposant sur la factorisation peuvent être considérés cassés par un attaquant quantique : RSA, Paillier, Naccache-Stern, Rabin, Benaloh, Blum-Goldwasser, Damgård–Jurik, Goldwasser-Micali, Okamoto-Uchiyama, Schmidt-Samoa.

2. La sécurité des signatures RSA n'est pas équivalente à la difficulté de la factorisation, alors que la sécurité des signatures de Rabin y correspond exactement.

SUJET POSÉ EN 2026

Le problème du logarithme discret peut aussi être résolu par une variante du même algorithme, de sorte que les algorithmes à clef publique dont la sécurité repose sur l'hypothèse de la difficulté du logarithme discret sont également cassés par l'algorithme de Shor. Contrairement au cas classique, où le crible du corps de nombres s'applique dans les corps finis mais pas en général dans le groupe des points rationnels d'une courbe elliptique, l'approche de l'algorithme de Shor ne fait aucune différence.

Ainsi, les algorithmes à clef publique reposant sur la difficulté du calcul d'un logarithme discret peuvent être considérés cassés par un attaquant quantique : Diffie-Hellman, Cramer-Shoup, DSA et ECDSA (aussi ses précurseurs Schnorr et ElGamal, et ses variantes comme EdDSA), SRP, etc.

D'autres problèmes classiquement difficiles reposant sur des problèmes d'arithmétique modulaire sont eux aussi affectés par des variantes de l'algorithme de Shor.

Candidats post-quantiques

Plusieurs constructions cryptographiques ne sont pas affectées par les attaques discutées ci-dessus, et constitueraient une solution facile de substitution. Par exemple, on peut espérer construire l'équivalent de constructions à clef publique en n'utilisant que des primitives symétriques. Mais le risque demeure qu'un algorithme quantique (qui n'a pas encore été découvert) puisse les casser.

Approches basées sur l'hypothèse P ≠ NP

La communauté cryptographique s'est donc tournée vers des approches qu'un attaquant quantique ne peut pas casser. La logique générale est de construire des primitives cryptographiques à partir de problèmes NP-complets, dont on sait qu'un ordinateur quantique (limité à la classe BQP) ne peut les résoudre plus efficacement qu'un ordinateur classique, sous l'hypothèse que P ≠ NP. Les candidats les plus sérieux reposent sur les problèmes suivants :

— Problèmes de vecteurs courts dans les réseaux euclidiens (1996) ;
— Décodage de codes linéaires aléatoires (1978) ;
— Inversion de polynômes multivariés (1995) ;
— Navigation dans les isogénies des courbes elliptiques supersingulières (2006) ;
— Inversion de fonctions de hachage (1978).

Le prix à payer est une perte d'efficacité et des clés plus larges. Pour essayer de gagner sur ces terrains, plusieurs propositions ont été faites qui n'ont qu'heuristiquement la sécurité d'un problème NP. La conséquence est que parfois, ces variantes s'avèrent vulnérables à des attaques classiques (ou quantiques).

En 2017, le NIST a annoncé une compétition visant à établir une standardisation des algorithmes cryptographiques post-quantiques. Ce protocole de standardisation, a fait émerger 4 candidats pour la standardisation en juillet 2022 : un schéma d'encapsulation de clés, CRYSTALS-Kyber, dont la sécurité repose sur la sécurité de problèmes de réseaux euclidiens (structurés), ainsi que trois schémas de signature numérique, CRYSTALS-Dilithium, Falcon et SPHINCS+ , les deux premiers reposant sur des hypothèses de sécurité sur les réseaux euclidiens (structurés) et le dernier étant une construction fondée sur les fonctions de hachage.

SUJET PROPOSÉ EN 2026

Codes correcteurs d'erreur

La cryptographie sur les codes est une autre solution pour concevoir des cryptosystèmes post-quantiques introduite en 1978 par Robert McEliece [4]. Le problème difficile sous-jacent venant de la théorie des codes provient du fait qu'il est difficile, étant donné une famille de codes, de déterminer quel code a été utilisé pour le codage, la clef privée étant la matrice génératrice du code (permettant de corriger l'erreur), et la clef publique étant une version randomisée de cette matrice (pour le cryptosystème de McEliece), permettant de générer des mots du code, sans pouvoir décoder. Ainsi pour chiffrer un message, l'expéditeur rajoute une erreur à son message encodé, ce qui le rend inintelligible pour une personne ne disposant pas de la matrice génératrice.

La principale difficulté de la cryptographie sur les codes correcteurs étant de trouver une famille de codes efficaces et exhibant la difficulté nécessaire. Une famille qui possède ces propriétés sont les codes de Goppa [5].

Le cryptosystème de McEliece est prouvé sûr sous l'hypothèse du décodage de syndrome (syndrome decoding problem ou SDP en anglais), qui est similaire à la recherche d'une solution inhomogène entière courte mais dans le cadre des codes. Autrement dit, étant donné une matrice $\mathbf{H} \in \mathbb{F}_2^{m \times n}$, un entier $w \leqslant n$ et un syndrome $\mathbf{u} \in \mathbb{F}_2^m$, le problème consiste à trouver un vecteur $\mathbf{x} \in \mathbb{F}_2^n$ solution de l'équation $\mathbf{H} \cdot \mathbf{x} = \mathbf{u}$ et tel que son poids de Hamming soit fixé à w.

Comparaison de tailles de clés

La taille des clés est importante pour évaluer l'efficacité des systèmes de chiffrement, et s'avère critique sur les systèmes embarqués comme les cartes à puces où la mémoire est critique. Il existe d'autres métriques, comme les temps pour chiffrer/déchiffrer, mais elles sont difficiles à évaluer puisqu'il est possible d'avoir des performances incomparables par l'utilisation de circuits programmables par exemple. Une caractéristique des cryptosystèmes post-quantiques est de fournir des clés plus grandes que celles utilisées par leurs concurrents (comme le chiffrement El Gamal sur courbes elliptiques par exemple).

Nom de l'algorithme	*Type d'hypothèses*	*Taille de la clef* publique *(en bits)*	*Taille de la clef* privée *(en bits)*
Ring-LWE	Réseaux euclidiens	6 595	14 000
NTRU	Réseaux euclidiens	6 130	6 743
Rainbow	Polynômes multivariés	991 000	740 000
Hash signature	Fonctions de hachage	36 000	36 000
McEliece sur des codes de Goppa	Codes correcteurs	8 373 911	92 027
McEliece sur des codes MDPC quasi-cycliques	Codes correcteurs	65 542	4 384
SIDH	Isogénie de courbes	6 144	6 144

TABLE 1 – Comparaison de tailles de clés

Ces clés sont évaluées pour un niveau de sécurité de 128 bits face à un attaquant quantique (256 bits face à un attaquant classique). À titre équivalent, un module RSA pour 256 bits de sécurité classique est estimé à 15 360 bits (clef publique) [6].

SUJET POSÉ EN 2026

Soutenabilité, coûts et consommation d'énergie

La généralisation de nouveaux algorithmes cryptographiques, dans les nœuds IoT[3] notamment, entraînera deux coûts majeurs, via d'une part l'augmentation de la consommation énergétique liée à l'usage d'algorithmes résistants aux attaques quantiques, et d'autre part la mise à niveau de l'infrastructure existante (les puces de nombreux dispositifs ne sont pas prévues pour la cryptographie post-quantique, et parfois la bande passante devra être élargie), en lien avec le concept d'*agilité cryptographique* (capacité d'un système à changer d'algorithme sans changer le matériel) que beaucoup de dispositifs n'ont pas, d'où une obsolescence matérielle précoce, augmentant le coût environnemental global via l'énergie grise.[4]

Quels que soient les choix retenus, la cryptographie post-quantique (CPQ) repose a priori sur des calculs plus volumineux qu'avec la cryptographie actuelle, et elle implique des échanges de données plus lourds que les méthodes antérieures. Il s'ensuit une augmentation de la consommation d'énergie du fait que les algorithmes de CPQ manipulent des clés et des signatures nettement plus volumineuses que la cryptographie actuelle.

Une augmentation de la consommation d'énergie touchera aussi les centres de données et les millions à milliards d'appareils connectés (IoT, smartphones, ordinateurs, tablettes, montres connectées, véhicules autonomes ou connectés, portefeuilles numériques, etc.), et ne sera pas soutenable sans introduire une approche novatrice de « cadre de sécurité multi-niveau conscient de l'énergie » intégrant des systèmes « basse consommation. »

Cette hausse potentielle est liée à plusieurs tendances technologiques.

Trading à très haute fréquence. L'arbitrage se fait à la milliseconde, ce qui implique qu'on utilise souvent des réseaux privés physiquement sécurisés où le chiffrement est parfois minimal pour gagner du temps. Le surcoût énergétique concernera surtout la phase du post-trade (règlement et livraison) et les transactions sur blockchains publiques, où l'authentification forte est obligatoire — pour le trading en lui-même, l'enjeu est la latence (de l'ordre de la nanoseconde), ce qui forcera l'achat de nouveaux processeurs (CAPEX[5]) plus que la consommation électrique pure (OPEX[5]). Au-delà de la latence, le passage à la CPQ là où il faut une authentification forte augmente la taille des paquets de données (les clés post-quantiques peuvent être 10 à 100 fois plus grandes que les clés ECC), ce qui freinera le flux des échanges. Concernant les blockchains, chaque transaction (de cryptomonnaies ou autre) doit être signée, puis vérifiée par des algorithmes de signature asymétrique. Leur remplacement par des algorithmes post-quantiques de chiffrement (e.g. CRYSTALS-Kyber) ou de protection des signatures (e.g. Dilithium ou Falcon) augmentera la taille des signatures et des transactions, ainsi que — légèrement — le coût de calcul de leur vérification au détriment de la bande passante et du stockage des nœuds du réseau.

Essor massif[6] des véhicules autonomes et de l'IoT. L'augmentation de la taille des clés et des messages affectera la consommation d'énergie du module radio des objets IoT via une

3. *Internet of Things* (internet des objets)

4. L'énergie *grise* est la quantité d'énergie consommée lors du cycle de vie d'un matériau ou d'un produit à l'exception notable de l'utilisation. Elle couvre donc l'énergie nécessaire à l'extraction des matières premières, les processus de transformation, fabrication, production, le transport, la mise en œuvre et l'entretien et enfin le recyclage (notoirement difficile pour les déchets électroniques). Elle est est cachée et indirecte, au contraire de l'énergie liée à l'utilisation, que le consommateur connaît (ou du moins peut connaître) aisément.

5. CAPEX /OPEX (*CAPital / OPerational EXpenditure*) : dépenses d'investissement / d'exploitation

6. 19 et 22 milliards dans le monde (surtout Chine et USA) en 2026, estimé à 39 milliards en 2030 et 50+ milliards en 2050 (les chiffres varient selon que l'on inclut ou non les capteurs passifs de type RFID/NFC)

EDITION 2026

augmentation de la durée d'émission et de réception pour l'objet connecté, plutôt que via l'augmentation du calcul CPU. Cette consommation radio est due à la taille des paquets. En CPQ, une clef peut dépasser la taille d'un paquet MTU standard (1 500 octets), forçant à faire une fragmentation, laquelle est le véritable « tueur d'énergie » car elle multiplie les handshakes et les réémissions en cas de perte de signal. Selon une étude de 2026 concernant les systèmes IoT *extreme-edge* — i.e. très petits, très peu puissants et très proches du capteur qui doivent fonctionner longtemps avec très peu d'énergie — moins de 1 % de la batterie serait surconsommée dans ces cas, un pourcentage qui parait faible, mais doit être jugé à l'aune du nombre d'objets connectés, en hausse rapide. Ces nouveaux sytèmes de cryptage vont aussi solliciter davantage les processeurs pour des opérations de hachage répétées. Et toute augmentation de la taille des clés ou du flux et du volume des messages implique pour les centres de données spécialisés une dépense énergétique (consommation électrique et besoin de refroissement) accrue, ici donc à grande échelle.

IA agentique et embarquée. Leur généralisation ajoute des couches d'interfaçage et de traitement supplémentaire, devant — elles aussi — être sécurisées par CPQ, qui aggravera la consommation énergétique des IA et des réseaux de transmission.

L'impact global de la CPQ ne sera pas uniforme : sur les serveurs modernes, on peut espérer que l'augmentation de consommation sera supportable, et à long terme peut-être compensée par des optimisations matérielles et logicielles. Mais pour les appareils IoT et les véhicules, la consommation d'énergie des serveurs, l e besoin de haut-débit et de bande passante (5G, 6G) pourrait augmenter, en raison de messages plus lourds, plus ou moins selon les technologies utilisées pour réduire la consommation d'énergie, e.g. regrouper les messages, réduire les handshakes pour limiter la consommation radio, utiliser un accélérateur cryptographique et des modes basse-consommation avec mise en veille, veille profonde, etc.

Des facteurs imprévisibles ou imprévus comme une crise de l'hélium pourraient aggraver les choses : les serveurs de calcul intensif pour la CPQ et l'IA agentique chauffent davantage. Les centres de données utilisent de nombreux disques durs scellés à l'hélium (gaz qui y réduit les frottements et améliore grandement leur refroidissement interne, ce qui diminue le besoin de refroidissement des racks de serveurs ou des systèmes de refroidissement cryogénique pour des processeurs quantiques hybrides.

Références

[1] Ryan Babbush, Adam Zalcman, Craig Gidney, Michael Broughton, Tanuj Khattar, Hartmut Neven, Thiago Bergamaschi, Justin Drake, & Dan Boneh. *Securing Elliptic Curve Cryptocurrencies against Quantum Vulnerabilities : Resource Estimates and Mitigations* (white paper, 57 pages).

[2] Carl Pomerance, « Analysis and comparison of some integer factoring algorithms, » in *Computational Methods in Number Theory*, Mathematical Centre Tracts 154, p. 89-139, Mathematisch Centrum Amsterdam, 1982.

[3] Daniel J. Bernstein, Nadia Heninger, Paul Lou & Luke Valenta, « Post-quantum RSA, » in *Post-Quantum Cryptography*, LNCS vol. 10346, p. 311-329, Springer, 2017.

[4] Robert J. McEliece, « A Public-Key Cryptosystem Based on Algebraic Coding Theory », *Jet Propulsion Laboratory DSN Progress Report*, p. 42-44, 1978.

[5] Raphael Overbeck & Nicolas Sendrier, « Code-based cryptography, » in Daniel J. Bernstein, Johannes Buchmann & Erik Dahmen (éditeurs), *Post-Quantum Cryptography*, p. 95-145 (chapitre 3), Springer, 2009.

[6] Elaine Barker, *Recommendation for Key Management — Part 1 : General*, NIST SP 800-57, 2016.

SUJET IPOPT N 2026