Berdes

Zeste de savoir

Advent of Code 2020, jour 8 : approche, solutions et bonus

11 décembre 2020

Table des matières

@nohar a déjà écrit deux billets à propos de l'Advent of Code (jours 1–5 et 6–10). Je vais donc vous laisser lire ses billets si vous voulez une présentation de l'événement.

Sur les problèmes des 10 premiers jours, celui du jour 8 m'a semblé être le plus intéressant, d'autant qu'il laisse la porte ouverte à quelques extensions. Je voulais donc présenter la manière dont j'ai initialement abordé le problème et le processus que j'ai suivi pour améliorer ma solution.

Je vais me focaliser un maximum sur l'aspect algorithmique, en illustrant avec du C++.

1. Présentation du problème

Pour faire court, on a une machine virtuelle avec 2 registres (un pointeur d'instruction et un accumulateur) et 3 instructions différentes qui ont chacune un nombre entier en paramètre :

— `nop` : augmente le pointeur d'instruction de 1, ignore son paramètre.
— `acc` : ajoute son paramètre à l'accumulateur, augmente le pointeur d'instruction de 1.
— `jmp` : ajoute son paramètre au pointeur d'instruction.

Un programme est simplement une liste d'instruction. Pour l'exécuter, il faut initialiser les deux registres à 0 et exécuter l'instruction pointé par le pointeur d'instruction jusqu'à ce qu'il pointe juste après la dernière instruction du programme.

Un programme peut se modéliser comme ceci :

```
1 struct Instruction {
2   enum Type { NOP, ACC, JMP };
3   Type type;
4   int argument;
5 };
6
7 using Program = std::vector<Instruction>;
```

En supposant que l'on veuille retourner la valeur de l'accumulateur lorsque le programme se fini, on se retrouve avec le code suivant :

```cpp
int RunProgram(const Program& prog) {
  int ip = 0;   // instruction pointer
  int acc = 0;  // accumulator
  while (ip != prog.size()) {
    switch (prog[ip].type) {
      case Instruction::NOP:
        ++ip;
        break;
      case Instruction::ACC:
        acc += prog[ip].argument;
        ++ip;
        break;
      case Instruction::JMP:
        ip += prog[ip].argument;
        break;
    }
  }
  return acc;
}
```

J'ai omis de vérifier que le pointeur d'instruction ne se retrouve pas en dehors du programme à un moment pour garder les choses les plus simples possible et je vais continuer à omettre ce genre de vérification pour la suite.

2. Partie 1

Dans la première partie, on nous dis que le programme que l'on a ne fini pas et va rentrer dans une boucle infini.

La machine virtuelle que l'on a n'a aucun moyen de faire un branchement conditionnel. Cela veut dire que si l'on se retrouve à un moment à exécuter une instruction que l'on a déjà exécuté, on est dans un scénario de boucle infini. La méthode la plus simple pour détecter ce genre de cas est de garder une mémoire de toutes les instructions exécutés précédemment.

La réponse à la première partie est donnée par le contenu de l'accumulateur juste **avant** que l'on exécute une instruction pour la deuxième fois. L'implémentation n'est pas plus compliqué que pour l'exécution d'un programme normal :

```cpp
int Part1(const Program& prog) {
  int ip = 0;   // instruction pointer
  int acc = 0;  // accumulator
  std::vector<bool> executed_instructions(prog.size(), false);
```

```
  while (!executed_instructions[ip]) {
    executed_instructions[ip] = true;
    switch (prog[ip].type) {
      case Instruction::NOP:
        ++ip;
        break;
      case Instruction::ACC:
        acc += prog[ip].argument;
        ++ip;
        break;
      case Instruction::JMP:
        ip += prog[ip].argument;
        break;
    }
  }
  return acc;
}
```

3. Partie 2

3.1. Première version

Dans la deuxième partie du problème, on va essayer de réparer le programme. On nous dit que soit une instruction `nop` à été changé en `jmp`, soit c'est l'inverse et une instruction `jmp` à été changé en `nop`. En trouvant cette instruction et en la réparant, le programme doit s'exécuter correctement, c'est-à-dire arriver jusqu'à la fin.

La première idée que j'ai eu est d'imaginer que l'on a N versions du programme, une pour chaque instruction qui peut être changé, ainsi qu'une version normale. Lors de l'exécution du programme, à chaque fois que l'on rencontre une instruction qui peut être changé on a alors un ou deux choix :

— On peut exécuter l'instruction normalement.
— Si on est dans la version normale du programme, on a aussi la possibilité de passer dans une version alternative du programme en changeant l'instruction courante et en l'exécutant.

Avant d'exécuter une instruction, on a 4 cas possibles :

— Soit le pointeur d'instruction est juste après la dernière instruction du programme : on a alors trouvé la solution. La réponse du problème est le contenu de l'accumulateur.
— Soit le pointeur d'instruction est complètement en dehors du programme : l'instruction que l'on a "corrigé" nous donne un programme invalide, on peut abandonner l'exécution.
— Soit le pointeur d'instruction pointe vers une instruction qui a déjà été exécuté : on est dans une boucle infini, on peut abandonner l'exécution.
— Soit le pointeur d'instruction pointe sur une instruction valide qui n'a pas encore été exécuté : il faut l'exécuter.

Pour gérer les multiples états potentiels générés lorsque l'on rencontre une instruction `nop` ou `jmp` et détruits lorsque l'on arrive dans un cas invalide, on va simplement utiliser une pile. Dans chaque état, on va garder 3 infos : le pointeur d'instruction, l'accumulateur et la version actuelle du programme (-1 pour le programme normal).

Pour garder une trace de toutes les instructions qui ont été exécuter, on va avoir un tableau de booléen (comme dans la partie 1) pour chaque version du programme[1].

En code, ça donne ça :

```cpp
int Part2v1(const Program& prog) {
  // The key is the position of the inversed instruction or -1 for the normal
  // version.
  std::unordered_map<int, std::vector<bool>> executed_instructions;
  // Populate the map with false everywhere.
  executed_instructions[-1].resize(prog.size(), false);
  for (int i = 0; i < prog.size(); ++i) {
    if (prog[i].type != Instruction::ACC) {
      executed_instructions[i].resize(prog.size(), false);
    }
  }

  struct State {
    int ip, acc, version;
  };
  std::stack<State> states;
  states.push({0, 0, -1});  // Start of the normal program.
  while (!states.empty()) {
    const State st = states.top();
    states.pop();
    if (st.ip == prog.size()) {
      // Found the solution, return it.
      return st.acc;
    } else if (st.ip < 0 || st.ip > prog.size()) {
      // The instruction pointer is out of bounds. Abandon current execution.
      continue;
    } else if (executed_instructions[st.version][st.ip]) {
      // We already executed this instruction. Abandon current execution
      continue;
    } else {
      executed_instructions[st.version][st.ip] = true;
      switch (prog[st.ip].type) {
        case Instruction::NOP:
          states.push({st.ip + 1, st.acc, st.version});
          if (st.version == -1) {
            // We're in the normal program, we can also try doing a jmp instead.
```

```
37              states.push({st.ip + prog[st.ip].argument, st.acc,
                    st.ip});
38            }
39            break;
40          case Instruction::ACC:
41            states.push({st.ip + 1, st.acc + prog[st.ip].argument,
                  st.version});
42            break;
43          case Instruction::JMP:
44            states.push({st.ip + prog[st.ip].argument, st.acc,
                  st.version});
45            if (st.version == -1) {
46              // We're in the normal program, we can also try doing a
                    nop instead.
47              states.push({st.ip + 1, st.acc, st.ip});
48            }
49            break;
50        }
51      }
52    }
53    // We tried all branches but didn't find the solution :(
54    std::cerr << "Coundn't find solution to Part 2\n";
55    exit(2);
56  }
```

En terme de complexité, générer le tableau des instructions visités est déjà $O(n^2)$ avec n le nombre d'instructions du programme. C'est vraiment pas génial, mais ça marche.

3.2. Deuxième version

Bon, on va peut-être essayer de trouver une meilleur solution.

Étant donné que l'on sait qu'il y a une et une seule instruction fautive, on peut classer les instructions en 3 catégories : 1. les instructions atteignables depuis le début du programme. 2. les instructions qui mènent à la fin du programme. 3. les autres instructions.

Pour trouver quel instruction corriger, il suffit de trouver une instruction qui fait partie de la catégorie 1 et qui nous amène à une instruction de la catégorie 2 lorsqu'elle est corrigé.

Trouver les instructions de la catégorie 2 va nécessiter d'exécuter le programme à l'envers, ce qui est plus compliqué qu'une exécution normale étant donné que plusieurs instructions peuvent amener à une seule instruction et qu'il faut toutes les visiter. Pour pouvoir trouver tous les antécédents possibles il faudra lire toutes les instructions du programme et générer un tableau qui nous donne pour chaque instruction la liste des antécédents (qui inclue l'instruction précédente si et seulement si c'est un `nop` ou un `acc`[2]). La bonne nouvelle, c'est qu'on n'a pas à se soucier de boucles infini puisqu'une instruction qui fait partie d'une boucle infini ne peut pas amener à la fin. Là encore, on va utiliser une pile pour se souvenir des instructions à visiter lors de l'exécution inverse.

Pour calculer facilement la réponse à la solution du problème, on va aussi enregistrer la valeur de l'accumulateur lors de la classification des instructions. La solution sera alors la valeur de l'accumulateur au niveau de l'instruction à corriger (catégorie 1) plus la valeur de l'accumulateur au niveau de l'instruction suivante (catégorie 2).

```
int Part2v2(const Program& prog) {
  struct InstructionData {
    // 1 is for the instructions that are reachable from the start.
    // 2 is for the instructions that lead to the end.
    // 0 is for the rest.
    int category;
    // Accumulator value when the instruction was reached for
        category 1
    // instructions.
    // Accumulator value obtained at the end for category 2
        instructions.
    // 0 for category 0 instructions.
    int acc;
  };
  std::vector<InstructionData> data(prog.size() + 1, {0, 0});

  // Find all instructions reachable from the start.
  int ip = 0;
  int acc = 0;
  while (data[ip].category == 0) {
    data[ip].category = 1;
    data[ip].acc = acc;
    switch (prog[ip].type) {
      case Instruction::NOP:
        ++ip;
        break;
      case Instruction::ACC:
        acc += prog[ip].argument;
        data[ip].acc = acc;
        ++ip;
        break;
      case Instruction::JMP:
        ip += prog[ip].argument;
        break;
    }
  }

  // For each instruction, list all instructionst that could lead
      to it.
  std::vector<std::vector<int>> antecedents(prog.size() + 1,
                                            std::vector<int>());
  for (int i = 0; i < prog.size(); ++i) {
    const auto& [type, arg] = prog[i];
```

```
41      if (type == Instruction::JMP && i + arg >= 0 && i + arg <=
            prog.size()) {
42        antecedents[i + arg].push_back(i);
43      } else {
44        antecedents[i + 1].push_back(i);
45      }
46    }
47
48    // Find all instructions leading to the end.
49    struct State {
50      int ip, acc;
51    };
52    std::stack<State> states;
53    states.push({prog.size(), 0});
54    while (!states.empty()) {
55      const State st = states.top();
56      states.pop();
57      data[st.ip].category = 2;
58      data[st.ip].acc = st.acc;
59      for (const int antecedent : antecedents[st.ip]) {
60        int new_acc = st.acc;
61        if (prog[antecedent].type == Instruction::ACC) {
62          new_acc += prog[antecedent].argument;
63        }
64        states.push({antecedent, new_acc});
65      }
66    }
67
68    // Find the instruction to bridge the two categories.
69    for (int i = 0; i < prog.size(); ++i) {
70      if (data[i].category != 1) {
71        continue;
72      }
73      if (prog[i].type == Instruction::NOP) {
74        int next_instruction = i + prog[i].argument;
75        if (next_instruction >= 0 && next_instruction <= prog.size()
              &&
76            data[next_instruction].category == 2) {
77          // Found the instruction to fix.
78          return data[i].acc + data[next_instruction].acc;
79        }
80      } else if (prog[i].type == Instruction::JMP) {
81        if (i + 1 >= 0 && i <= prog.size() && data[i + 1].category ==
              2) {
82          // Found the instruction to fix.
83          return data[i].acc + data[i + 1].acc;
84        }
85      }
86    }
```

```
87    // No instruction can be fixed to go from category 1 to category
          2.
88    std::cerr << "Coundn't find solution to Part 2\n";
89    exit(2);
90  }
```

L'implémentation de cette solution est plus longue que celle de la précédente. En revanche, la complexité est nettement meilleur. Il n'est pas difficile de se convaincre que chaque boucle va s'exécuter au maximum n fois, avec n le nombre d'instruction du programme. Cela nous donne une complexité en $O(n)$ au lieux du $O(n^2)$ que l'on avais précédemment.

4. En allant plus loin

C'est bien jolie tout ça, mais il se passe quoi si notre programme a plusieurs instructions corrompus ? La deuxième approche utilisé semble assez difficile à généraliser. Peut-être qu'on pourrais reprendre la première approche, l'améliorer et la généraliser ?

Si une seule instruction doit être changé, on n'a pas besoin de garder la liste des instructions déjà exécutés pour chaque version du programme parce qu'une seule liste partagée par toutes les versions est suffisante. Pour s'en convaincre on peut réfléchir de la manière suivante. Mettons que l'on exécute une version V et qu'on arrive à une instruction visité dans une version V'. On note M l'instruction modifié dans V et M' l'instruction modifié dans V'. On a 5 cas de figures :

- soit V' termine, auquel cas on n'a pas besoin d'explorer plus loin.
- soit V' boucle sans passer par M ou M' : ça va boucler aussi pour nous (toutes les instructions sont identiques).
- soit V' boucle en arrivant à M : on retombe sur nos pas, donc on boucle.
- soit V' boucle en arrivant à M' : ça veut dire qu'on arrive sur un segment exécuté dans la version normale. Si on est avant M, on va finir par y arriver et on va boucler. Si on est après M, soit on boucle comme la version normale, soit la boucle de la version normale nous ramène sur M et on boucle aussi.

Si V ou V' est la version normale, ça élimine certains cas, mais ça ne change pas le fait que l'on n'a pas besoin de continuer à explorer.

Et du coup, il se passe quoi si on commence à avoir plusieurs instructions à corriger ? Est-ce qu'on peut toujours conserver une seule liste d'instructions visités ? La réponse est oui, mais on va orienter le problème différemment pour le démontrer.

Si on imagine le programme comme un graphe où chaque instruction est un nœud et les instructions sont reliés par des arrêtes orientés qui indique à quel instruction on va arriver après avoir exécuté l'instruction d'un nœud. Par exemple, avec le programme suivant :

1. Ce n'est en fait pas nécessaire et j'expliquerais pourquoi un peu plus tard. Si l'on ne garde qu'un seul tableau pour toutes les versions, cette solution est équivalente à celle de @nohar.
2. Ce n'est pas strictement nécessaire d'enregistrer les `nop` et `acc` dans la liste des antécédents vu que ça peut se trouver facilement lors de l'exécution à l'envers, mais ça simplifie nettement le code.

```
acc +1
jmp +2
nop -1
jmp -3
```

on a les arrêtes suivantes :

- `acc +1` -> `jmp +2`
- `jmp +2` -> `jmp -3`
- `nop -1` -> `jmp -3`
- `jmp -3` -> `acc +1`

Se demander si on peut atteindre une instruction depuis une autre, reviens à se demander s'il existe un chemin qui va d'une instruction à une autre.

Pour modéliser le fait de changer une instruction, on peut ajouter une arrête pour cette instruction modifié. Avec l'exemple précédent, ça ajoute 3 arrêtes :

- `jmp +2` -> `nop -1`
- `nop -1` -> `jmp +2`
- `jmp -3` -> fin du programme

Chercher à corriger le programme en modifiant le moins d'instructions possible, ça reviens à chercher un chemin qui va du début à la fin du programme et qui passe par le moins d'arrêtes liés à une modification possible. Vu que toutes les modifications se valent, on peut leur donner un coût de 1 et les arrêtes normales un coût de 0.

On se retrouve alors avec le problème bien connu de recherche du plus court (ou moins coûteux) chemin dans un graphe. L'algorithme standard dans ce cas est l'algorithme de Dijkstra . Dans notre cas où le nombre d'arrête est compris entre n et $2n$, toujours avec n le nombre d'instructions, la complexité est de $O(n \times \log(n))$.

Le principe de l'algorithme est de maintenir une file à priorité des prochains sommets à visiter. La priorité est donné par la distance (on visite en priorité les sommets les plus proches du départ). À chaque fois que l'on visite un sommet, on ne fais rien s'il a déjà été visité et on ajoute tous ses voisins avec pour priorité la distance du départ au sommet actuel plus la distance du sommet actuel au voisin. On visite $O(n)$ sommets et on insère $O(n)$ éléments dans la file, chaque insertion étant en $O(\log(n))$, on se retrouve donc bien en $O(n \times \log(n))$.

Cependant, avec notre problème de programme à corriger, on est dans un cas assez particulier étant donnés que tous les poids ont soit la valeur 0, soit la valeur 1. On peut donc se contenter d'utiliser deux piles pour les sommets à visiter : une pour des sommets à visiter qui sont à une distance N du début du programme et une pour les sommets à visiter qui sont à une distance N+1 du départ. Une fois que la pile des sommets à une distance N est vide, on passe à la pile de distance N+1 et on créé une pile pour les sommets à une distance N+2.

Une autre solution, encore plus simple est d'utiliser une file double dans laquelle on insère les sommets qui sont à une distance 0 du sommet actuelle au début de la file et ceux à une distance 1 à la fin de la file. On visite à chaque fois le sommet qui est au début de la file.

Que ce soit avec deux piles ou avec une double file, le coût d'enregistrer un sommet à visiter passe de $O(\log(n))$ à $O(1)$. Donc la solution pour trouver un moyen de corriger le programme via un nombre minimum de corrections devient $O(n)$.

Niveau implémentation, c'est assez proche de la toute première solution :

```
int Part2v3(const Program& prog) {
  struct State {
    int ip, acc;
  };
  std::vector<bool> visited(prog.size(), false);
  std::deque<State> to_visit;
  to_visit.push_front({0, 0});
  while (!to_visit.empty()) {
    const State st = to_visit.front();
    to_visit.pop_front();
    if (st.ip == prog.size()) {
      // Found the solution.
      return st.acc;
    }
    // Discard bogus instruction pointers and visited instructions.
    if (st.ip < 0 || st.ip > prog.size() || visited[st.ip]) {
      continue;
    }
    visited[st.ip] = true;
    switch (prog[st.ip].type) {
      case Instruction::NOP:
        to_visit.push_front({st.ip + 1, st.acc});
        to_visit.push_back({st.ip + prog[st.ip].argument, st.acc});
        break;
      case Instruction::ACC:
        to_visit.push_front({st.ip + 1, st.acc +
            prog[st.ip].argument});
        break;
      case Instruction::JMP:
        to_visit.push_front({st.ip + prog[st.ip].argument,
            st.acc});
        to_visit.push_back({st.ip + 1, st.acc});
        break;
    }
  }
  // We tried all paths but didn't find the solution :(
  std::cerr << "Coundn't find solution to Part 2\n";
  exit(2);
}
```

Voilà, j'espère que ça vous a intéressé. Hésitez pas à me dire s'il y a des points à améliorer en particulier.

4. En allant plus loin

Ça fait aussi beaucoup trop longtemps que j'ai pas écris un truc aussi long en français, donc j'espère qu'il n'y a pas trop d'anglicismes, de tournures grammaticales douteuses et de fautes en tous genres 🍊 .