ICFPC-2012: Позорный провал за лямбдами

В этом году я участвовал в ICFPC уже в третий раз. Предыдущие мои отчеты можно посмотреть здесь: 2010, 2011.

Подготовка

В этом году организаторы сообщили, что тестирование (так же, как и в прошлом году) будет проводиться на виратульной машине под дебианом, и даже выложили образ этой виртуальной машины. В прошлом году я тщетно пытался поставить дистрибутив дебиана на свой старый ноут. В этом году все оказалось существенно проще: потребовалось только скачать и запустить образ в VirtualBox.

С виртуалбоксом я до этого не работал, поэтому понадобились ~~некоторые усилия~~ 5 минут гуглинга, чтобы настроить расшаренную папку. Делается это всего одной магической командой:

```
sudo mount -t vboxsf -o uid=1000,gid=1000 share ~/host
```

Однако, мне так и не удалось заставить работать copy+paste, а также понять, как набрать ~. Но кажется, что это не критично, так как всегда можно копировать скрипты через папочку shared.

// Про команду

План по организации заключался в том, чтобы как обычно собраться на квартире у Тарасова и трое суток хреначить код.

День 1

В 4 часа появилось задание. На первый взгляд, задание заключалось в написании бота для диггера. Передвигаемся по лабиринту – собираем алмазы, ставим лестницы, чтобы можно было ходить вверх и периодически возвращаемся на базу, чтобы продать найденное. Я распечатал 6 условий (на всякий случай, чтобы всем хватило) и поехал на место сбора. Уже читая условия в метро, я понял, что речь идет скорее об игре supaplex, только проще (так как нет бегающих ножниц, и лямбды не могут взрываться). Однако,

организаторы сделали лазейку в правилах: оставив за собой право менять условие (но не больше 4х раз). Поэтому вполне возможно, что в конечном варианте игра будет существенно сложнее supaplex.

Полное описание условия доступно здесь.

В скором времени стали подходить и другие члены команды. Еще раз все вместе вдумчиво читаем условие и обсуждаем возможные подходы к решению этой задачи. В этот раз условие оказалось на удивление простое, но как подбираться к самой задаче – было совсем непонятно.

В джаббере Олег говорил, что все такие задачи можно решать перебором + функция оценки. Но все равно непонятно, как должна выглядеть функция оценки и как организовывать перебор.

Также, пообсуждали физику игры и мелкие фишки. Все очень просто и наглядно: камни круглые, стены квадратные, поэтому камни скатываются с других камней во все стороны, но стоят на стенках. Лямбды ()– покатые, поэтому камни скатываются с них только вправо.

В правилах разбирается такая позиция:

```
#* *#
#* *#
#####
```

следующая позиция будет такой:

```
#   #
#***#
#####
```

То есть, вообще говоря, количество камней на поле может уменьшаться. Еще интересно, что если есть большая куча камней, то камни из нее будут падать с интервалом в одну клетку. И, например, можно встать в один из таких просветов и пройти дальше.

В результате обсуждений также и начала вырисовываться общая архитектура возможного решения: Есть некоторые цели (в первом приближении это просто какие-то клетки), которые надо посетить. Дальше запускается алгоритм выбора порядка обхода этих целей. В процессе обхода он получает некоторую информацию (например, выход из лабиринта завалило) и может добавлять новые цели или ставить частичный порядок на целях (например, что надо съесть некоторую землю раньше, чем идти к следующей лямбде). Все выбрано достаточно абстрактно и обще, чтобы ничего не потерять при таком сведении.

Примерно в этот момент становится понятно, что мы будем писать на C++.

Вообще говоря, язык особо выбирать не приходилось (лично я знаю достаточно хорошо только 2 языка: C++ и питон, ну и других членов команды ситуация примерно такая же).

Еще до контеста я думал писать традиционно писать на питоне. Но сейчас, в процессе обсуждении участники команды склонялись скорее к C++. Да и задача выглядела скорее сложной вычислительно, поэтому решено было использовать более быстрый язык, чтобы не пролететь из-за скорости. Казалось бы, C++ – это Тьюринг полный язык, поэтому мы не должны потерять в выразительности.

Тут стоить отметить, что скорость разработки в таком краткосрочном проекте – это очень немаловажный фактор. Недописанная программа на C++ работает в десятки миллионов раз медленнее, чем дописанная на питоне.

Узнав, что мы собираемся писать на C++, Леша почти сразу слился, решив, что с этого момента он будет только генерировать идеи. Забегая вперед скажу, что у него это неплохо получалось, но с их реализацией у нас что-то не пошло дело (на самом деле это и понятно: чужие идеи всегда сложнее реализовывать, а особенно если они еще и недостаточно формализованны).

Дальше было пора уже писать код. У меня Linux, у остальных – Windows, и язык в этот раз не питон, поэтому надо было сделать кроссплатформенную сборку. Я взял для этого cmake. Скопипастил с мануала простейший HelloProject и буквально за полчаса все заработало. В это время остальные настраивали just for fun проектор и традиционный для нашей команды dropbox (правда в этот раз он должен был использовался для C++, но кажется – невелика разница).

Миша сел писать pathfinder – алгоритм обхода целей с помощью A* (с функцией оценки расстояния – манхеттенской метрикой).

А я начал делать эмулятор игрового мира. Вначале решили писать самую простую версию (с тупым копированием всего поля при апдейте и поэлементным применением указанных правил as is), а потом если потребуется переписать на более оптимальный вариант (Сейчас кажется, что такая тактика была ошибочной – если сразу не заложить в архитектуру какое-то подобие оптимальной реализации, то потом будет сложно и лениво все переделывать).

Когда пишешь на C++, то сразу начинает тянуть к проектированию вместо решения поставленной задачи. Так у нас после недолгого единогласного обсуждения появились енумы `ECellType` и `EMove` (На питоне я бы не парился и просто бы использовал строковые константы для этого, но нет, блин, мы же пишем на C++: тут строгая типизация помогает уберечь себя от ~~выстрела в ногу~~ передачи внутрь функции хода вместо типа клетки. В итоге большую часть времени ты борешься с языком и думаешь как бы так реализовать то, что у тебя в голове давно уже придумано).

Дальше у нас появляется класс `TPosition` с функцией `TPosition Move(EMove)` и перегруженным оператором ^ (!), который означает манхеттенское расстояние между точками.

С реализацией игрового мира я справился довольно быстро (спецификация была очень подробная и понятная), и дальше стал делать управление роботом с клавиатуры.

Вскоре появилось и первое дополнение к правилам: наводнение. Посмотрев, что там все просто, я решил отложить реализацию этой фичи (так как мы все равно пока не понимаем, как собирать лямбды в самом простом случае).

Какое-то время пришлось повозиться с чтением клавиш с клавиатуры, так как все известные мне методы (`cin.get(1)`, `getchar()` ждали символа окончания строки). В итоге помог гугл рассказав про функцию `getch()`. Однако, ему требуется заголовочный файл `conio.h`, которого как я понял нет под линуксом. В общем, еще после некоторого общения с гуглом, мне удалось произвести такую кроссплатформенную магическую последовательность заклинаний:

```
#ifdef unix
#include <termios.h>
#include <unistd.h>
inline int getch()
{
    struct termios oldt, newt;
    int ch;
    tcgetattr(STDIN_FILENO, &oldt);
    newt= oldt;
    newt.c_lflag &= ~(ICANON| ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    ch= getchar();
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    return ch;
}
#else
#include <conio.h>
inline int getch() {
    return _getch();
}
#endif
```

С очисткой экрана, оказалось все гораздо проще:

```
#ifdef unix
inline void clear_screen() {
    system(”clear”);
}
#else
inline void clear_screen() {
    system(”CLS”);
}
#endif
```

Через какое-то время было готово управление робота по `wasd`. А ребята в это время придумали несколько подходов к решению и вроде даже начали их писать, но я уже был слишком уставшим, чтобы их обсуждать.

Мне понравилась идея с диаграммами Вороного:

1. Выбираем случайное множество точек
2. Разбиваем все подземелье на кластера по принципу близости к выбранным точкам
3. Дальше на этих локальных частях запускаем алгоритм обхода (???)
4. Глобально части тоже как-нибудь обходим (???)
5. Profit!

Диаграмма Вороного только звучит страшно. Здесь задача дискретная, поэтому она строится за один bfs.

День 2

Я проснулся раньше всех, и так как не мог дальше спать, то сел и написал простейшее жадное решение, которое шло к ближайшей лямбде. Если после применения пути нам не удавалось попасть туда, куда мы хотели, то все пересчитывалось и запускалось заново.

В само решение я встроил визуализацию, и было видно, что, в принципе, решение как-то работает.

Дальше просыпаются ребята и у нас есть следующие идеи, что можно делать:

- Миша хочет написать отжиг (выбираем случайные цели, обходим, по результатам как-то случайно меняем цели / добавляем новые)
- Родин будет писать диаграммы Вороного и двойного TSP: TSP в локальной части и обход по остовному дереву по всем частям (он дает решение не больше чем в 2 раза отличающееся от оптимального).
- Олег собирался сделать алгоритм определения камней, которые можно двигать, и которые нам как-то мешают.
- Я буду оптимизировать операции с состоянием игры (понятно как сделать обновление за O(количество камней)), а также добавлять необходимые функции в эмулятор on demand.

Я не помню почему, но я так и не добрался до оптимизации :), видимо пилил жадный алгоритм и делал какие-то инфраструктурные вещи: билд, визуализацию итп.

Примерно к середине дня я начинаю осознавать, что задачка мне не очень нравится. Как мне кажется главной причиной было то, что нет никого взаимодействия между разными командами (в частности нет scoreboard). Задачка очень сложная и ты не чувствуешь свой прогресс. (Есть high scores по тестовым картам, но он не очень показательный, так

как там половина решений набита руками, либо же ботом специально написанным под эту карту) Понятно, что все остальные команды испытывают те же самые трудности и проблемы, но ты не видишь этого и нет подбадривающего заряда, который ты получаешь делая какое-нибудь улучшение и поднимаясь вверх по турнирной таблице. (Сейчас, когда я пишу это, я понимаю, что значит надо было искусственно сделать себе такой прогресс и стимулы – то есть сгенерировать много своих карт и тестировать свои алгоритмы на них).

Немного слов про дропбокс. Обнаружилось главное отличие при использовании его для C++. Так как многие (в том числе и я) при написании кода очень любят нажимать ctrl+S, то в итоге получается, что недописанные строчки синхронизируются у других участников и билд ломается. Особых проблем это не вызывает: всегда можно вслух спросить: “кто, блин, опять билд сломал???”, и обычно он быстро чинится :). Ну и еще можно приучать себя сохранять файл, только когда написан компилируемый кусок. Зато получается такой реалтаймовый гит и не нужно ничего пушить - пулить. Иногда, конечно, случались конфликты, когда люди правят одновременно один файл (но крайне редко, так как IDE обычно успевает подхватить изменения).

Под вечер ребята убедили меня написать персистентный класс TBoard. Чтобы быстро обрабатывать изменения и задешево хранить все промежуточные состояния. В общем, профит становится неизбежен. Даже примерно понятно, как это сделать:

- Двумерное поле представляем как одномерный вектор.
- Одномерный вектор пишем, как персистентное дерево отрезков с операциями `get`, `set`.
- Надо еще персистентно помнить список всех камней. Здесь можно использовать персистентное декартово дерево по неявному ключу.

Я начал понемногу ковырять перистентность. Даже реализовал клас `TPersistentArray<T>`. Дальше, мой мозг стал совсем отключаться и я понял, что пора спать.

День 3

Утром нас снова ждал сюрприз от организаторов: появились бороды, которые клонировались каждые X ходов, и которые можно брить, собираю бритвы по уровню.

На свежую голову и учитывая нововведение я понимаю, что персистентную доску мы будем писать всей командой до конца контеста, попутно уклоняясь от выстрелов в ногу из-за выделения памяти через new. А у нас еще ни одной новой фичи не реализовано.

День 4

Проснувшись с утра, я почувствовал подъем сил, и желание доделать хоть какой-то вариант, чтобы отправить. В итоге, я стал писать солвер, который сначала делает жадный

поиск, а затем последовательно запускает отжиг с увеличивающимся числом итераций, пока не словит SIGINT.

Миша тоже довольно быстро появился в джаббере и в срочном порядке учит отжиг использовать новые добавленные фичи. Ну и несомненно, мы фиксили многочисленные обнаруживаемые баги.

Вот некоторые из багов, которые мы поправили.

1

```
for (TTeleports::TTo::const_iterator it = Board.GetTeleports().To.begin();
    it != Board.GetTeleports().To.end();
    ++it)
{
    Teleports.push_back(it->first);
}
```

Этот код зацикливался из-за пропущенного & в сигнатуре

```
const TTeleports TBoard::GetTeleports() const
```

2

```
inline int randMod(int N) {
    int res = 0;
    for (int i = 0; i < 2; ++i) {
        res = (res * RAND_MAX + rand()) % N;
    }
    return res;
}
```

Из-за этого ГСЧ, программа стабильно падала в корку в glibc. Как нетрудно видеть, такой генератор может выдавать отрицательные числа, которые не очень хорошо использовать как индексы вектора. Но так как вектор не проверяет выход за границу массива, то получалось портить память и ошибка возникала в самом неожиданном месте.

Дальше надо было прикрутить обработчик SIGINT. С этим пришлось повозиться. Я знал, как вызвать произвольную функцию-обработчик. Магическое заклинание примерно следующее:

```
#ifdef unix
    struct sigaction intAction;
    intAction.sa_handler = SigIntHandler;
    sigemptyset(&intAction.sa_mask);
    intAction.sa_flags = SA_RESTART;
    sigaction(SIGINT, &intAction, NULL)
#endif
```

Ну думаю, оберну общий цикл в `try ... catch`, а в обработчике брошу эксепшн. Такая штука не сработала: программа через раз просто завершалась со словами `Aborted`.

Еще обычно делают `volatile int` флаг, который `SigIntHandler` выставляет в `true`. А в цикле итерирования он проверяется и если пора, то брейкаемся. Но видимо, я не очень умею его готовить и такой способ у меня тоже не взлетел.

В итоге, добавил в `Solver` метод `GetBestAnswer`, который звался из обработчика сигнала, а потом просто делалось `exit(0)`. На самом деле, из правил хорошего тона было сделать обработчик реентерабельный, но организаторы обещали слать `SIGINT` ровно один раз.

В самом конце я быстренько наваял скрипт, который собирает бинарник на виртуалке и пакует его вместе с исходниками в архив.

Выводы