Projet Worddle

On se propose de réaliser un résolveur de grilles de jeu *Boggle* dans le cadre du projet de *structures de données*.

1 Règles du jeu

1.1 Origines du jeu

Boggle est un jeu de mots créé par Allan Turoff et commercialisé à l'origine par *Parker Brothers*. Le terme *Boggle* est une marque déposée, c'est pourquoi il sera préférable d'utiliser un autre nom pour désigner notre projet : par exemple Worddle.

1.2 Règles

Le Worddle se présente sous la forme d'une boîte accueillant 4 rangées de 4 dés (soit au total 16 dés) : sur chacune des faces des dés figure une lettre. On ferme la boîte avec un couvercle en plastique, puis on la secoue afin de mélanger les dés : la face supérieure des 16 dés forme une matrice 4x4 de lettres. L'objectif est alors de trouver en un temps limité (usuellement 3 minutes – un sablier étant généralement utilisé –) le maximum de mots (présents dans un dictionnaire de référence) obtenus par juxtaposition de cellules adjacentes de la matrice (réutiliser plusieurs fois la même cellule étant interdit).

Il est possible de jouer au Worddle en solitaire, mais aussi en groupe. Plusieurs variantes existent pour compter les points dans un jeu en groupe : généralement, on n'attribue aucun point pour les mots trouvés par au moins deux personnes ; les joueurs ayant trouvé un mot en exclusivité se voient attribuer un score qui est une fonction croissante de la taille du mot. Le gagnant est bien sûr le joueur ayant obtenu le score le plus élevé.

Exemple de jeu Un exemple de grille obtenu avec les dés du Worddle anglais est indiqué en figure 1.2. En utilisant un dictionnaire de mots anglais[1], on trouve que cette grille permet de former 2 mots de 7 lettres (*abelson*, *besomed*), 6 mots de 6 lettres (*abases*, *anabel*, *basest*, *deleon*, *moslem*, *santos*) ainsi que d'autres mots de longueur inférieure.

d	m	t	m
e	s	o	t
l	e	b	n
g	s	a	a

Figure 1: Exemple de grille 4x4 de Worddle (obtenue avec les dés anglais)

[1] On utilise le dictionnaire /usr/share/dict/american-english de la distribution Linux Debian.

2 Le projet

Le projet consiste à implanter un résolveur de grilles de Boggle fournies par l'utilisateur avec utilisation d'un dictionnaire de mots donnés. Vous avez tout loisir pour utiliser l'algorithme de votre choix : nous vous conseillons cependant de suivre les conseils donnés dans cet énoncé.

2.1 Arbre lexicographique du dictionnaire

Tout d'abord, nous vous proposons de réaliser un arbre lexicographique (appelé également *trie* par les anglophones) d'un dictionnaire : cette structure s'avérera utile afin de trouver les mots d'une grille de Worddle.

Vous devez donc créer, à partir d'une liste de mots présentée sous la forme d'un fichier ASCII[2] avec un mot par ligne, le délimiteur de fin de ligne étant le caractère '\n'. Vous pourrez trouver des exemples de dictionnaires de différentes langues dans le répertoire `/usr/share/dict` de toute distribution Linux qui se respecte.

Par exemple, construisons l'arbre lexicographique (voir la figure 2.1) du dictionnaire suivant :

```
chien
chat
niche
nicher
zoo
```

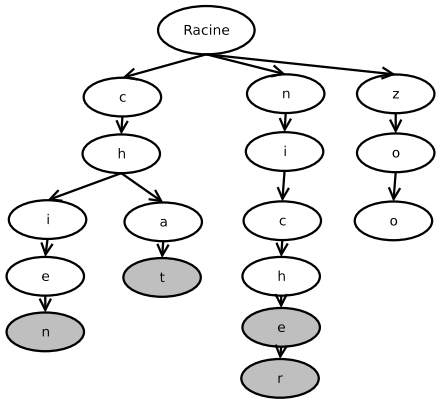

Figure 2: Arbre lexicographique du dictionnaire `chien, chat, niche, nicher, zoo`

On remarquera en particulier qu'il existe deux types de nœuds au sein d'un arbre lexicographique :

- Les nœuds intermédiaires,
- et les nœuds terminaux qui marquent la fin d'un mot et qui ne sont pas nécessairement des feuilles de l'arbre (lorsqu'un mot est préfixe d'un autre, comme niche et nicher ici). Si l'on souhaite que les nœuds terminaux soient nécessairement des feuilles, il est possible d'utiliser un caractère de fin de mot qui ne soit pas une lettre (par exemple `$`).

Vous avez toute liberté pour choisir la structure de données en C la plus adaptée à l'implantation de l'arbre lexicographique : inutile cependant de chercher à tout prix l'optimisation en mémoire utilisée et en temps nécessaire pour le parcours, contentez-vous d'une structure simple.

[2]En fait il s'agit plutôt d'un fichier en codage iso-8859-15 afin de représenter notamment les mots en langue française, mais l'essentiel est de retenir qu'il s'agit d'un sur-ensemble du codage ASCII et qu'un caractère y est codé sur un octet.

Test de l'arbre lexicographique Afin de tester la (bonne) construction de l'arbre lexicographique, il est fortement conseillé de procéder au parcours de l'arbre pour relever tous les mots qu'il contient : on peut alors effectuer l'étape inverse de transformation de l'arbre en liste de mots. On vérifie ensuite si cette liste de mots correspond à la liste originelle utilisée pour créer l'arbre : à cet effet, on trie les deux fichiers de mots (en utilisant la commande `sort` par exemple) et on les compare : on utilise la commande `cmp` pour la comparaison ; si les deux fichiers sont différents, on peut utiliser `diff` pour visualiser les différences.

Sauvegarde de l'arbre lexicographique Certains langages de programmation (tels que Java, OCaml ou C#) disposent dans leur API standard d'un moyen simple de sauvegarder linéairement dans un fichier le contenu d'une structure de données (sérialisation). Malheureusement, il n'existe pas de moyen aisé de réaliser une sérialisation en C. On aimerait cependant pouvoir sauvegarder l'arbre construit dans un fichier pour éviter de devoir le reconstruire pour résoudre chaque problème de Worddle. Vous pouvez donc imaginer un moyen de procéder à la sérialisation de l'arbre dans un fichier et à sa restauration en mémoire : cette exigence est néanmoins facultative, le gain de temps engendré par la restauration de l'arbre construit par rapport à la reconstruction étant faible. Un bonus sera attribué aux étudiants mettant en place une telle solution. Si vous souhaitez vous attaquer à ce problème, il est conseillé de bien réfléchir à la structure de donnée utilisée pour implanter l'arbre lexicographique (des choix judicieux peuvent faciliter l'écriture des fonctions de sauvegarde et de restauration de l'arbre).

Utilisation d'un automate Il existe des structures plus compactes en mémoires qu'un arbre lexicographique pour représenter un dictionnaire, en particulier les automates déterministes minimaux. Toutefois la construction d'une telle structure est plus compliquée qu'un arbre lexicographique : des méthodes directes peuvent être employées ou alors indirectes par la construction intermédiaire de l'arbre lexicographique. Nous vous déconseillons de vous aventurer dans la voie de l'utilisation d'un automate sauf si vous êtes vraiment très très motivé.

2.2 Générateur de grilles de Worddle

Nous mettons à votre disposition un générateur de grilles ainsi que quelques grilles types avec les solutions trouvées par notre implantation de référence que vous pourrez trouver sur le forum.

2.3 Résolveur de grilles

La résolution d'une grille nécessite la construction préalable de l'arbre lexicographique du dictionnaire. Une grille de Worddle peut être vu comme un graphe non-orienté : chaque dé est un sommet et tous les sommets adjacents sont reliés par des arêtes. L'adjacence de sommets se définit dans le jeu classique en utilisant la 8-connexité : soit (i, j) le sommet de la ligne i et de la colonne j ; il est relié par une arête aux sommets (s'ils existent) $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, $(i, j+1)$, $(i-1, j-1)$, $(i-1, j+1)$, $(i+1, j-1)$, $(i+1, j)$ (voir la figure 2.3).

Plusieurs possibilités existent pour représenter la grille : soit représenter simplement la grille avec des relations d'adjacence implicites (les fonctions de parcours de graphe prennent en compte cette adjacence implicite), soit utiliser une structure de graphe explicite. Chaque méthode a ses avantages et inconvénients : la première est plus directe, tandis que la deuxième permet éventuellement de changer la définition d'adjacence sans avoir à modifier les méthodes de parcours de graphe utilisées.

Finalement, la grille étant vue comme un graphe, et disposant de l'arbre lexicographique du dictionnaire, nous pouvons nous attaquer à la recherche exhaustive de mots dans le graphe. L'algorithme proposé est le suivant (en pseudo-code) :

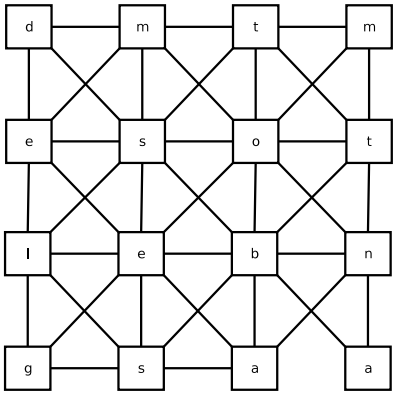

Figure 3: Graphe induit de la grille de Worddle 1.2

Fonction resolveur(G: graphe) : **Mots**
    $M \leftarrow \{\}$
    Pour chaque sommet S de G **faire**
        $M \leftarrow M \cup mots(S)$
    Fin Pour
    Retourner M
Fin

Fonction mots(S: sommet, A: trie, V: sommets visités, P: préfixe mot) : **Mots**
    $M \leftarrow \{\}$
    Si (A' = A[S.lettre] existe) **Alors**
        P ← P . S.lettre
        Si (A' est un noeud terminal) **Alors**
            $M \leftarrow M \cup \{P\}$
        Fin Si
        Pour chaque sommet S' adjacent à S avec $S' \notin V$ **faire**
            $M \leftarrow M \cup mots(S', A', V + S, P)$
        Fin Pour
    Fin Si
    Retourner M
Fin

Cet algorithme consiste donc, en partant d'un sommet, à parcourir le graphe par suivi des sommets adjacents avec parcours en parallèle de l'arbre lexicographique : lorsque l'on ne trouve pas de branche dans l'arbre lexicographique correspondant au sommet en cours, on arrête le parcours. Si l'on arrive à un noeud terminal de l'arbre, on a trouvé un mot de la grille. On mémorise les sommets déjà visités afin de ne pas les réutiliser.

2.4 Généralisation en n dimensions

Il pourrait être intéressant (mais ce n'est nullement obligatoire dans le cadre de ce projet) de généraliser le fonctionnement du programme pour des grilles en 3, voire n dimensions : il faut alors s'intéresser à la notion de voisinage d'une cellule en n dimensions. Cette généralisation est loin d'être triviale.

2.5 Tri des mots trouvés

La liste des mots trouvés doit être triée par score décroissant, à score égal (mots de même longueur), le tri s'effectue par ordre lexicographique croissant (on pourra utiliser la fonction *strcmp*). Vous êtes libre d'utiliser la méthode de votre choix pour effectuer le tri : soit implanter votre algorithme de tri, soit utiliser par exemple la fonction `qsort` de la bibliothèque standard (qui permettra au passage d'aborder les pointeurs de fonctions en C). Une fois les mots triés, il est nécessaire de supprimer les doublons (mots pouvant être obtenus de plusieurs façons).

3 Consignes et conseils divers

3.1 Programme *worddle*

Vous devez implanter le constructeur d'arbre lexicographique et le résolveur de grille (en utilisant l'algorithme proposé ou un autre plus performant de votre cru). Le programme résolveur de grille devra être nommé `worddle` et accepter les arguments suivants :

- -d `fichier` : pour spécifier le dictionnaire (liste de mots) à utiliser
- -s `entier` : pour indiquer la taille de la grille

La grille est fournie ligne par ligne sur l'entrée standard et est exprimée sous forme d'une suite de lettres, chacune séparée soit par une espace, soit par un `\n`. La liste de mots obtenus par résolution de la grille est imprimée sur la sortie standard (un mot par ligne, séparation par `\n`) et doit être triée par longueur de mot décroissante puis par ordre lexicographique pour les mots de même longueur, avec suppression des doublons.

Un fichier *Makefile* doit être livré avec les sources : ce fichier devra comporter une règle *all* permettant de compiler votre projet. Le projet doit être rendu compressé dans un fichier *tar.gz* (comprenant aussi le rapport PDF) dont le nom suit la syntaxe suivante : *worddle.login.tar.gz* où *login* est votre nom d'utilisateur sur le réseau de l'université.

3.2 Rapport

Un rapport de quelques pages au format PDF décrivant vos choix d'implantation pour le projet doit être rendu avec vos sources.

3.3 Date limite de rendu

Non encore clairement définie : *a priori* celle-ci devrait se situer aux alentours de la date du dernier TP. Le projet doit être rendu par email à votre chargé de TP. Un malus de 2 points sera retenu par journée de retard.

3.4 Forum

Pour toute question sur le projet, pour récupérer d'éventuelles mises à jour du présent sujet ou pour des demandes d'aide ou d'éclaircissement, vous pouvez vous rendre sur le forum à l'adresse suivante : `http://etudiant.univ-mlv.fr/~sdumazet/forum/`.

3.5 Quelques conseils en vrac...

- Commentez utilement vos sources : décrivez le rôle de chacune de vos fonctions, levez les passages obscurs de votre source par un petit commentaire. Il est inutile par contre d'associer des commentaires à du code qui n'en a pas besoin (exemple de commentaire inutile : `i++; /* Incrémentation de la variable i */`). Vous pouvez (si vous le souhaitez) générer une documentation à partir de votre code-source en utilisant un outil tel que *Doxygen*.

- Soyez homogène dans vos notations : soit vous choisissez de coder avec des noms de variables (et commentaires) en français, soit en anglais, mais pas les deux à la fois. Les seules langues acceptées sont le français et l'anglais avec des sources en ASCII : les sources en japonais avec codage UTF-16 ne seront pas acceptées (par exemple).
- Indentez correctement votre code. Au pire, si vous n'aimez pas indenter votre code, il est toujours possible de tricher en utilisant *indent* juste avant le rendu.
- Organisez votre code-source en plusieurs fichiers plutôt que de faire un fichier fourre-tout avec des dizaines de fonctions et des milliers de lignes.
- Coder en C présente l'inconvénient (ou l'avantage, selon les points de vue) de devoir gérer soi-même la mémoire : les erreurs liées à la gestion de mémoire sont fréquentes, même chez les programmeurs les plus expérimentés. Il est donc très fortement conseillé d'utiliser un outil de détection de mauvais usages de la mémoires tel que *valgrind* : ce logiciel indispensable à tout programmeur C fait tourner les programmes dans une machine virtuelle et repère les tentatives de lecture ou écriture sur des zones de mémoire non-allouées. L'absence de plantage par faute de segmentation ne dispense pas d'utiliser *valgrind* : il est possible que de tels problèmes passent inaperçus jusqu'au jour (par la loi de Murphy) où le correcteur le fera tourner.
- Commencez à réfléchir et à implanter le projet tôt afin d'éviter de le faire rapidement (et salement) à la dernière minute. Dans un premier temps, implantez uniquement la partie obligatoire, puis songez à des améliorations citées ici ou de votre cru afin de récolter des bonus. Il est toujours préférable d'avoir un projet qui marche mais sans fioritures plutôt qu'un projet avec des tas d'améliorations mais qui engendre des erreurs de segmentation aléatoires.
- Bon courage !