DotAI - Projet de semestre

Thomas Ibanez

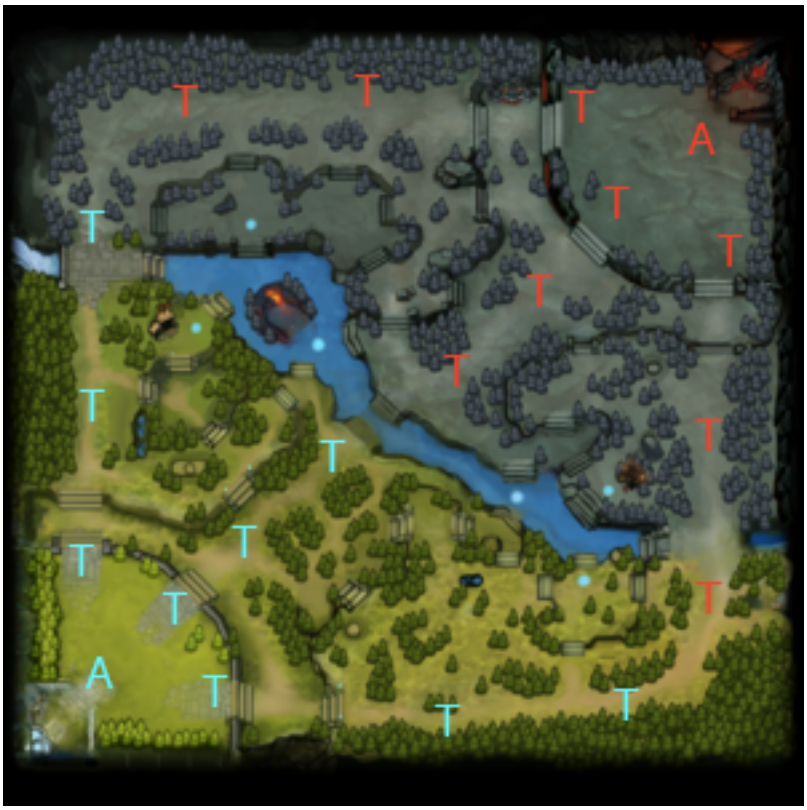
25 février 2018

1 Introduction

Le but de ce projet de semestre est de créer un ensemble de logiciels permettant de présenter les données contenues sur `opendota.com` pour le développement d'un logiciel d'apprentissage automatique sur le jeu DotA 2.

1.1 DotA 2

DotA 2 est jeu-vidéo multijoueur de type bataille en arène (MOBA) où 2 équipes s'affrontent. Chaque équipe est composée de 5 joueurs chacun controllant un héro choisi parmis les héros disponnibles dans le jeu (il ne s'agit donc pas d'un avatar personnalisable comme dans le cas des MMORPG). A l'heure actuelle le jeu propose 115 héros, chacuns disposant de 5 attaques différentes. La partie se déroule sur une carte symétrique constituté de 3 "lanes" sur lesquelles sont disposées des tours dont voici une vue aérienne :



Chaque équipe commence dans un coin de la carte (en bas à gauche et en haut à droite), le but pour chaque héros est de controler une partie de la carte et détruire les tours ennemies (T) afin d'arriver jusqu'a la base adverse (A) et de la détruire.

En plus des héros chaque base va périodiquement créer des "creeps", sortes de petit monstres qui vont se déplacer sur une lane.

2 Recupération des informations d'OpenDota

Opendota collecte une quantitée énorme de parties de DotA 2 jouées par des joueurs du monde entier. Le site présente une API permettant la récupération de matchs selon certains critères. Une documentation sur le genre de requêtes faisable est disponible sur `https://docs.opendota.com/`. Dans le cadre de se projet, nous allons utiliser un filtre permettant de séléctionner un héros dont on voudrait qu'il soit présent dans le match.
Dans un premier temps M. Malaspinas et moi-même pensions que les données étaient directement exploitable depuis opendota, malheureusement c'est plus compliqué. En effet OpenDota stocke uniquement des données statistiques sur la partie jouée mais pas les actions et positions des entitées à chaque instant. Heureusement le résultat contient un lien vers un fichier *.dem* qui lui contient toutes ces informations.

3 Analyse des fichiers .dem

Les fichiers .dem sont des fichiers contenant toutes les informations dont on pourrait vouloir sur une partie allant de entités présentes sur la carte jusqu'au commentaires audio si disponnibles. Un fichier .dem pour une partie standard (~30 Minutes) pèse ~50Mo.

3.1 Format

Le format global de ces fichiers est assez simple. DotA utilise protobuf, une technologie de sérialization développée par google, pour encoder ces fichiers. Le protocole est maintenu à jour par l'équipe SteamRE sur `https://github.com/SteamRE/SteamKit/tree/master/Resources/Protobufs/dota`
Chaque fichier commence par une en-tête, soit "PBDEMS2" pour les replay datant d'après la mise à jour de DotA sur le moteur de jeu Source 2. Soit "PBUFDEM" pour les replay plus anciens. Dans le cadre de ce projet nous nous concentrons uniquement sur les replay d'après Source 2 car ce sont ceux qui sont émis en ce moment.
Une fois ce header lu le fichier contient 8-bytes dont l'utilitée n'est pas connue.

3.1.1 Format global

La suite du fichier est une série d'entrées de ce type :

Nom	Type
ID	VarInt
Tick	VarInt
Taille	VarInt
Données	Bytes

Note : Les varints sont des entiers encodés sur une longeur variable de bits. Cet encocodage est détaillé sur `https://developers.google.com/protocol-buffers/docs/encoding#varints`

L'ID va indiquer le type du message parmis la liste définie dans le protocole :

Messages

```
enum EDemoCommands {
  DEM_Error = -1;
  DEM_Stop = 0;
  DEM_FileHeader = 1;
  DEM_FileInfo = 2;
  DEM_SyncTick = 3;
  DEM_SendTables = 4;
  DEM_ClassInfo = 5;
  DEM_StringTables = 6;
  DEM_Packet = 7;
  DEM_SignonPacket = 8;
  DEM_ConsoleCmd = 9;
  DEM_CustomData = 10;
  DEM_CustomDataCallbacks = 11;
  DEM_UserCmd = 12;
  DEM_FullPacket = 13;
  DEM_SaveGame = 14;
  DEM_SpawnGroups = 15;
  DEM_Max = 16;
  DEM_IsCompressed = 64;
}
```

La seule exception est *DEM_IsCompressed* qui n'est pas un type de message mais qui défini un bit (6ème bit) qui s'il est égal à 1, alors le message doit être décompressé via la librairie snappy avant d'être interpreté.

Le Tick va définir le moment dans la partie ou le message arrive.

Le champ Taille indique la taille en bytes des données de ce message.

Une fois ces informations connues nous pouvons lancer le décodage via protobuf afin d'obtenir le message, du moins c'est le cas pour tout les ID sauf DEM_SignonPacket, DEM_Packet, DEM_FullPacket et DEM_SaveGame. Pour ces messages-ci il va falloir lire les données intégrées.

3.1.2 Données intégrées

Les données intégrées sont basiquement un message encodé dans un autre message. Dans notre cas un DEM_Packet et un DEM_FullPacket sont définis comme :

Packets

```
message CDemoPacket {
  optional int32 sequence_in = 1;
  optional int32 sequence_out_ack = 2;
  optional bytes data = 3;
}
```

```
message CDemoFullPacket {
  optional .CDemoStringTables string_table = 1;
  optional .CDemoPacket packet = 2;
}
```

Les DEM_SignonPacket n'ont pas de définition distincte, leur structure est la même qu'un DEM_Packet. Les DEM_FullPacket sont en fait composés d'une *StringTable* et d'un DEM_Packet qu'il faudra interprêter comme les autres DEM_Packet.

Comme on peut le constater un packet n'est en vérité que composé de numéros de séquences (qui ne sont pas intéressants dans notre cas) et d'une suite de bytes qu'il nous faut interprêter. L'intérprétation de ce champ est la suivante : Il s'agit d'une série de sous-messages dont encodés d'une manière ressemblant au format global du fichier :

Nom	Type
ID	Bitvar
Taille	VarInt
Données	Bytes

Ce qu'il faut remarqué c'est que l'ID est encodé en tant que "bitvar" et non pas en tant que VarInt. Ce format est un petit peu particulier et implique un désalignement :

```
Byte No: |       0       ||       1       ||       2       ||       3       ||      4  //
Bit No:  |0 1 2 3|4 5 6 7||0 1 2 3|4 5 6 7||0 1 2 3|4 5 6 7||0 1 2 3|4 5 6 7||0 1      //
         |-------|-------||-------|-------||-------|-------||-------|-------||---     //
Valeur:  |X Y 0 0|0 0 1 1||1 1 2 2|2 2 3 3||3 3 3 3|3 3 3 3||3 3 3 3|3 3 3 3||3 3     //
```

Les bits "X", "Y" et "0" doivent obligatoirement être lus. Ensuite :
- Si Y vaut 1, il faut lire 4 bits supplémentaires (les bits "1")
- Si X vaut 1, il faut lire 8 bits supplémentaires (les bits "1" et "2")
- Si X et Y valent 1, il faut lire 28 bits supplémentaires (les bits "1", "2" et "3")

Note : Comme on peut le constater, la lecture de cette variable ne s'arrête non pas à la fin d'un byte mais bel et bien au millieu. Nous ne sommes donc plus alignés sur une base de 8 bits ce qui m'a obligé à créer un flux de lecture bit par bit au lieu d'utiliser un des flux disponnibles avec java qui sont tous byte par byte.

La valeur est ensuite construite de la manière suivante : Les 4 bits "0" sont les bits de poids faible, les autres bits restent dans l'ordre dans lequel ils sont écrits (les "1" en poids fort).

Le champs taille défini la taille des données de ce sous-packet. Les données sont à nouveau une structure encodée via protobuf, cette structure peut être de type NET_Message ou SVC_Message (également d'autres types possible mais ils ne nous intéressent pas).

Voici les enums SVC_Messages et NET_Messages :

Messages

```
enum SVC_Messages {
  svc_ServerInfo = 40;
  svc_FlattenedSerializer = 41;
  svc_ClassInfo = 42;
  svc_SetPause = 43;
  svc_CreateStringTable = 44;
  svc_UpdateStringTable = 45;
  svc_VoiceInit = 46;
  svc_VoiceData = 47;
  svc_Print = 48;
  svc_Sounds = 49;
  svc_SetView = 50;
  svc_ClearAllStringTables = 51;
  svc_CmdKeyValues = 52;
  svc_BSPDecal = 53;
  svc_SplitScreen = 54;
  svc_PacketEntities = 55;
  svc_Prefetch = 56;
  svc_Menu = 57;
  svc_GetCvarValue = 58;
  svc_StopSound = 59;
  svc_PeerList = 60;
  svc_PacketReliable = 61;
  svc_HLTVStatus = 62;
  svc_FullFrameSplit = 70;
}

enum NET_Messages {
  net_NOP = 0;
  net_Disconnect = 1;
  net_SplitScreenUser = 3;
  net_Tick = 4;
  net_StringCmd = 5;
  net_SetConVar = 6;
  net_SignonState = 7;
  net_SpawnGroup_Load = 8;
  net_SpawnGroup_ManifestUpdate = 9;
  net_SpawnGroup_SetCreationTick = 11;
  net_SpawnGroup_Unload = 12;
  net_SpawnGroup_LoadCompleted = 13;
}
```

Dans ce projet, étant donné que nous ne cherchons pas à créer un interpreteur complet, ce sont les messages svc_CreateStringTable, svc_UpdateStringTable, svc_PacketEntities et svc_ServerInfo qui vont nous intéresser par le suite.

3.2 Données

Revenons au début du fichier, directement après le magic et les 8-bytes à sauter nous allons trouver un message de type DEM_FileHeader. Ce message contient des méta-données sur le fichier, par exemple :

```
demo_file_stamp: "PBDEMS2\000"
network_protocol: 44
server_name: "Valve Dota 2 Europe Server (srcds122.133.57)"
client_name: "SourceTV Demo"
map_name: "start"
game_directory: "/opt/srcds/dota/dota_v2432/dota"
fullpackets_version: 2
allow_clientside_entities: true
allow_clientside_particles: true
addons: ""
```

Ces informations ne nous sont cependant pas utiles.
Nous trouvons ensuite une série de DEM_SignonPacket, l'un d'entre eux contient un sous-message de type svc_ServerInfo dont voici la structure :

ServerInfo

```
message CSVCMsg_ServerInfo {
  optional int32 protocol = 1;
  optional int32 server_count = 2;
  optional bool is_dedicated = 3;
  optional bool is_hltv = 4;
  optional bool is_replay = 5;
  optional int32 c_os = 6;
  optional fixed32 map_crc = 7;
  optional fixed32 client_crc = 8;
  optional fixed32 string_table_crc = 9;
  optional int32 max_clients = 10;
  optional int32 max_classes = 11;
  optional int32 player_slot = 12;
  optional float tick_interval = 13;
  optional string game_dir = 14;
  optional string map_name = 15;
  optional string sky_name = 16;
  optional string host_name = 17;
  optional string addon_name = 18;
  optional .CSVCMsg_GameSessionConfiguration game_session_config = 19;
  optional bytes game_session_manifest = 20;
}
```

Le champ game_dir va nous permettre de déterminer la version de DotA 2 utilisée, ce qui nous sera utile par la suite. Le contenu du champ ressemble à ça "/opt/srcds/dota/dota_v2432/dota" (à noter que ce champ est également présent dans le message DEM_FileHeader. La partie qui nous interesse est "v2432" qui nous indique que le replay à été créer avec la version 2432 de DotA 2.

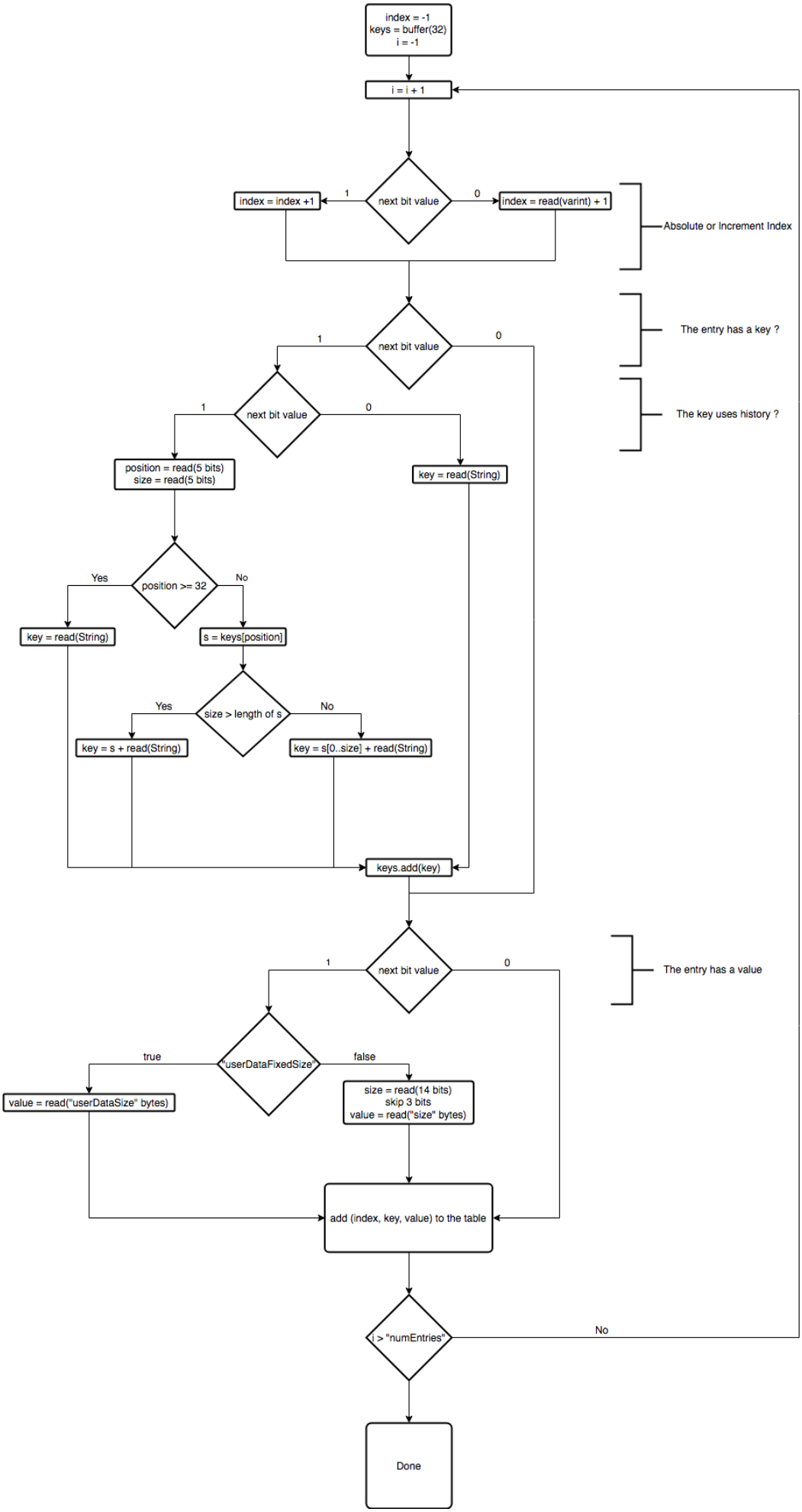
Nous trouvons ensuite des DEM_Packets qui contiennent des sous-messages de type svc_CreateStringTable qui vont servir, comme leur nom l'indique à créer des "String Tables".

3.2.1 String Tables

Une String Table est une table nomée, composée d'une série d'entrées ayant un index (entier), une clef (chaine de caractères) et une valeur (chaine de bytes) bien que le clef comme la valeur puisse ne pas être définie (null). Voici pour illustrer une petite partie de la String Table "EntityNames" (la table entière contient 251 entrées)

Index	Clef	Valeur
24	dark_troll_warlord_ensnare	null
25	dark_troll_warlord_raise_dead	null
26	polar_furbolg_ursa_warrior_thunder_clap	null

La totalité du contenu de la String Table est englobé dans le champ *StringData*, cependant il est possible que ce champ soit compressé avec la librairie Snappy pour les replay plus récents ou avec LZSS pour les plus anciens. Pour savoir quelle méthode est utilisée on peut vérifier si les 4 premiers bytes composent la chaine de caractères "LZSS".
Une fois décompressé si besoin nous pouvons commencer la lecture, il nous faut dans un premier temps déclarer un buffer circulaire de 32 éléments qui va constituer un historique des clefs lues.


index = -1
keys = buffer(32)
i = -1
i = i + 1
next bit value
1
0
index = index +1
index = read(varint) + 1
Absolute or Increment Index
next bit value
1
0
The entry has a key ?
next bit value
1
0
The key uses history ?
position = read(5 bits)
size = read(5 bits)
key = read(String)
position >= 32
Yes
No
key = read(String)
s = keys[position]
size > length of s
Yes
No
key = s + read(String)
key = s[0..size] + read(String)
keys.add(key)
next bit value
1
0
The entry has a value
"userDataFixedSize"
true
false
value = read("userDataSize" bytes)
size = read(14 bits)
skip 3 bits
value = read("size" bytes)
add (index, key, value) to the table
i > "numEntries"
No
Done

Une fois toutes ces String tables lues nous allons recevoir un packet contenant les "Send Tables" qu'il va falloir lire également.

3.2.2 Send Tables

La définition d'une SendTable est très simple :

SendTable

```
message CDemoSendTables {
  optional bytes data = 1;
}
```

Comme on peut le constater il n'y a qu'un champ *data* qui est une suite de bytes, il va donc falloir le décoder.
En vériter le décodage du champs *data* est simple, il faut d'abort lire un varint qui va nous indiquez une taille, puis lire cette taille (en bytes) afin d'obtenir une structure de type svc_FlattenedSerializer

FlattenedSerializer

```
message CSVCMsg_FlattenedSerializer {
  repeated .ProtoFlattenedSerializer_t serializers = 1;
  repeated string symbols = 2;
  repeated .ProtoFlattenedSerializerField_t fields = 3;
}

message ProtoFlattenedSerializerField_t {
  optional int32 var_type_sym = 1;
  optional int32 var_name_sym = 2;
  optional int32 bit_count = 3;
  optional float low_value = 4;
  optional float high_value = 5;
  optional int32 encode_flags = 6;
  optional int32 field_serializer_name_sym = 7;
  optional int32 field_serializer_version = 8;
  optional int32 send_node_sym = 9;
  optional int32 var_encoder_sym = 10;
}

message ProtoFlattenedSerializer_t {
  optional int32 serializer_name_sym = 1;
  optional int32 serializer_version = 2;
  repeated int32 fields_index = 3;
}
```