Dokumentation zur Projektaufgabe für das Modul Künstliche Intelligenz für Spiele im Wintersemester 2022/23

Oliver Jakobs

20. Februar 2023

Thema und Motivation

Als Thema für meine Modulaufgabe habe ich mich für **Decision Trees** entschieden. Dafür habe ich ein Programm geschrieben, das einen Entscheidungsbaum aus einer XML-Datei einliest und daraus eine passende Datenstruktur generiert. Dann kann diese Datenstruktur über das Terminal in einer Art Frage-Antwort Spiel durchlaufen werden.

Um ein Beispiel zu haben, mit dem ich mein Programm testen kann habe ich online nach einfachen Entscheidungsbäumen gesucht und hier (`https://heartbeat.comet.ml/understanding-the-mathematics-behind-decision-trees-22d86d55906`) einen gefunden, mit dem ich (fast) alle geplanten Features demonstrieren kann.

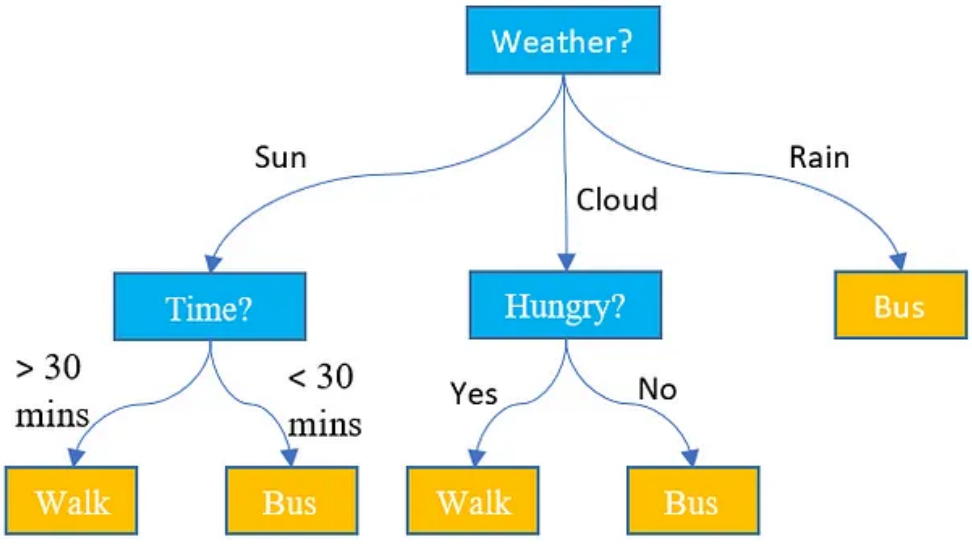

Abbildung 1: tree.xml

Zwei Features haben mir noch gefehlt, aber diese konnte ich mit kleinen Anpassungen (ohne dabei die Ergebnisse zu verändern) auch noch darstellen. Mit diesem Baum habe ich dann meine XML-Struktur definiert und danach mein Programm implementiert.

Allgemeine Designentscheidungen

Mein Programm lässt sich in zwei Abschnitte aufteilen. Der erste Teil ist die Implementiation des Entscheidungsbaumes. Dieser wird aus einer .xml-Datei eingelesen und in eine passenden Datenstruktur geladen. Den zweiten Teil habe ich TreeWalker gennant. Dieser Teil beschäftigt sich mit Kommunikation zwischen dem Nutzer und dem Baum.

Das Programm habe ich bewusst im C-Style programmiert. Das heißt ich habe mich bei der Struktur des Programmes an C gehalten und habe lediglich einzelene Features (z.B. `std::vector` oder `std::variant`) aus C++ verwendet. Der Grund für diese Entscheidung war, dass ich privat hauptsächlich in C programmiere und so am komfortabelsten mit diesem Stil bin. Außerdem habe ich so weniger Aufwand, wenn ich die Implementation anpassen muss, wenn ich Decision Trees in meinen C-Programmen verwenden will.

Entscheidungen zur XML-Datei

In der XML-Datei kann neben dem Entscheidungsbaum auch noch Zusatzinformationen angegeben werden, um die Interaktion mit dem Entscheidungsbaum zu vereinfachen. Das Root-Element der XML-Datei habe ich `decisiontree`. Aber da ich dafür keine Abfrage implementiert habe, kann man das im Grunde so nennen wie man möchte.

Zusatzinformationen
Mit dem Tag `intro` kann ein Einleitungstext angegeben werden. Mit dem Tag `prompt` kann definiert werden welche Frage der TreeWalker für eine Entscheidung stellen soll. Und mit dem Tag `result` können Beschreibungen der Ergebnisse angegeben werden. Hier muss dann noch der Name des Ergebnisses als Attribut angegeben werden.

Entscheidungsbaum
Der eigentliche Entscheidungsbaum besteht aus vier Tags (`decision`, `option`, `final` und `invalid`). Diese Tags entsprechen dem Type der TreeNodes, die aus diesem Element generiert werden. Außerdem können Elemente mit diesen Tags noch einen Namen (`name`) und einen Wert (`value`) als Attribut haben.

Beispiel
Wi oben erwähnt muss der Beispielsbaum noch bisschen angepasst werden. Die erste Anpassung nutzt den `invalid` Tag um bei der Entscheidung *Time?* negative Antworten zu verhindern. Die zweite Anpassung habe ich letztendlich nur gemacht um ein Interval als Antowrtmöglichkeit angeben zu können. Wenn man dann den angepassten Baum dann mit meiner XML-Struktur darstellt, kann das wie folgt aussehen:

```
<decisiontree>
  <intro>Deciding whether to walk or take the bus.</intro>
  <option name="weather">
    <prompt>How is the weather? (sunny/cloudy/rainy)</prompt>
    <decision value="sunny" name="time">
      <prompt>How much time do you have (in minutes)?</prompt>
      <invalid value="&lt;0"/>
      <final value="0:29" name="walk" />
      <final value="&gt;=30" name="bus" />
    </decision>
```

```
    <option value="cloudy" name="hungry">
      <prompt>Are you hungry? (yes/no)</prompt>
      <final value="yes" name="walk" />
      <final value="no" name="bus" />
    </option>
    <final value="rainy" name="bus" />
  </option>

  <result name="walk">You should walk.</result>
  <result name="bus">You should take the bus.</result>
</decisiontree>
```

Entscheidungen zur DecisionTree Implementation

Der DecisionTree wird aus mehreren TreeNodes aufgebaut. So eine TreeNode hat einen Type, einen Namen, einen Wert und eine Liste an Nachfolgern. Anhand der Types werden folgende Arten von TreeNodes unterschieden:

Decision
TreeNodes mit dem Type `decision` bilden den Grundbaustein meiner Implementation. An sich lässt sich alle Funktionalität von DecisionTrees nur mit TreeNodes dieser Art umsetzen.

Der Wert einer `decision` ist eine Boolean Expression, in die zum Zeitpunkt der Entscheidung eine Variable eingesetzt werden kann. Operatoren in diesen Expressions sind die Standartoperatoren ($<, <=, >, >=$) und ein Intervall Operator (angegeben mit $v_1 : v_2$). Gibt die Expression `True` zurück, so ist die entsprechende TreeNode die richtige Antwort.

In meiner Implementation sind alle numerischen Werte `Integers`, da `Float`-Werte den Umfang und den damit verbundenen Aufwand sehr stark in die Höhe treiben würden. Außerdem ist die Umsetzung von Float-Werten (bis auf das Erkennen und Parsen) gleich mit der Umsetzung der Integer-Werte und bringt daher keine neuen Herausforderungen.

Option
Bei einer TreeNode mit dem Type `option` können die Antwortmöglichkeiten als Zeichenketten angegeben werden. Die gegebene Antwort wird mit allen diesen Zeichenketten (mit `std::string::compare`) verglichen und die erste passende Option wird ausgewählt.

Final
Eine TreeNode ist `final`, wenn sie keine Nachfolger hat (also ein Blatt ist). In der XML-Datei kann der entsprechende Tag verwendet werden, um das zu verdeutlichen.

Invalid
Mit einer `invalid` TreeNode können bestimmte Werte blockiert werden. So können zum Beispiel alle Werte < 0 als ungültige Antwort makiert werden.

Wenn über einen Knoten im Baum entschieden wird, wird jeder Nachfolger nacheinander angeschaut. Dabei wird der erste Treffer wird akzeptiert. Die Antwort ist ungültig, wenn kein passender Knoten gefunden wurde, wenn die Eingabe das falsche

Format hat (z.B. kein `int`-Wert als Eingabe für eine `decision`) oder wenn der (erste) gefundene Knoten `invalid` ist.

Parsen von Boolean Expressions

Ein großer Teil von meiner Implementation ist ein Parser um aus einer Zeichenkette eine Boolean Expression zu parsen, in die dann nach Eingabe einer Variablen auch evaluiert werden kann.

Um das umzusetzen wird eine Zeichenkette in sogenannte Tokens übersetzt, aus denen dann der Ausdruck generiert wird. Die Funktion `next_token` erzeugt aus einer Zeichenkette einen Token und gibt den verbleibenden, noch nicht betrachteten Teil der Zeichenkette zurück. Dieser Token kann alle Bestandteile eines Ausruckes darstellen. Eine besondere Art ist der `END`-Token. Dieser kennzeichnet das Ende der Zeichenkette. Um einen gültigen Ausruck zu generieren, müssen die entsprechenden Token in der richtigen Reihenfolge gelesen werden. Für die einfachsten Ausdrücke (z.b. x¡0) muss der erste Token ein Operator, der Zweite dann eine Zahl und der Dritte dann der `END`-Token sein. Werden all diese Token erfolgreich gelesen hat man einen gültigen Ausdruck. Dieser kann dann ausgeführt werden, indem man eine Variable mit dem vorher geparsten Wert, auf eine vom Operator abhängige Art, vergleicht. Dafür kann man z.B. ein `switch`-Statement verwenden.

Mit diesen einfachen Ausdrücken könnte man schon viele Bäume umsetzen, aber ich habe noch zwei weitere Arten von Ausdrücken erlaubt.

Die erste Art ist ein Shortcut für den Gleichheits-Operator. Wenn die Zeichenkette aus einem einzigen Token (der `END`-Token ausgenommen) besteht und dieser eine Zahl ist, wird der Gleichheits-Operator verwendet und ein entsprechender Ausdruck generiert. Die zweite Art ist der Interval-Operator. Dieser wird angegeben mit $v_1 : v_2$. So ein Ausdruck ist ein bisschen aufwendiger. Vorallem mit dem Shortcut für den Gleichheits-Operator wird das ganze Parsen um einiges komplexer.

Der komplette Parsing-Vorgang läuft im Grunde so ab:

1. Ersten Token ist Operator (Außer interval):
   (a) Wenn zweiter Token Zahl und dritter Token `END` ist `->` Parsen erfolgreich.
   (b) Alles andere `->` Ungültiger Ausdruck.
2. Erster Token ist eine Zahl:
   (a) Wenn zweiter Token `END` ist `->` Parsen erfolgreich.
   (b) Wenn zweiter Token Interval-Operator ist:
      i. Wenn dritter Token Zahl und vierter Token `END` ist `->` Parsen erfolgreich.
      ii. Alles andere `->` Ungültiger Ausdruck.
   (c) Alles andere `->` Ungültiger Ausdruck.
3. Alles andere `->` Ungültiger Ausdruck.

Bedienung

Um das Programm zu starten muss die Datei `DecisionTree.exe` ausgeführt werden. Dafür wird neben der `.exe` auch der Ordern `./res/` mit Inhalt benötigt. Beim Starten

über die Kommandozeile kann noch ein Pfad zu einer XML-Datei als Argument angegeben werden, um einen eigenen Entscheidungsbaum zu laden.

```
.\DecisionTree.exe [filename]
```

Wird kein Argument angegeben wird der oben vorgestellte Beispielsbaum geladen.

Die Ausführung des Programms findet in einem `read-eval-print loop (REPL)` statt. Dem Nutzer wird eine Frage gestellt, auf die er dann antworten muss. Ist die Antwort ungültig, das heißt die Antwort entspricht nicht der erwarteten Form oder ist keine der möglichen Antwortmöglichkeiten, so wird der Nutzer erneut aufgefordert zu antworten. Wenn ein Blatt des Entscheidungsbaumes erreicht wird, wird das Endergebnis angezeigt und das Programm beendet.

Wenn zur Compile-Zeit `RUN_TESTS` definiert ist, wird anstelle des *normalen* Programmes der Entscheidungsbaum mit den in der Funktion `run_tests` angegebenen Tests getestet. Diese Tests sind für den Beispielbaum und müssten für andere Enscheidungsbäume angepasst werden.

Neben dem oben vorgestellten Entscheidungsbaum liegen noch zwei weitere Beispiele in dem Ordner `./res/`. Den Baum in der Datei `job.xml` habe ich hier **hier** gefunden und den Baum in der Datei `badminton.xml` **hier**.

Hier sind beide Bäume nochmal grafisch dargestellt:

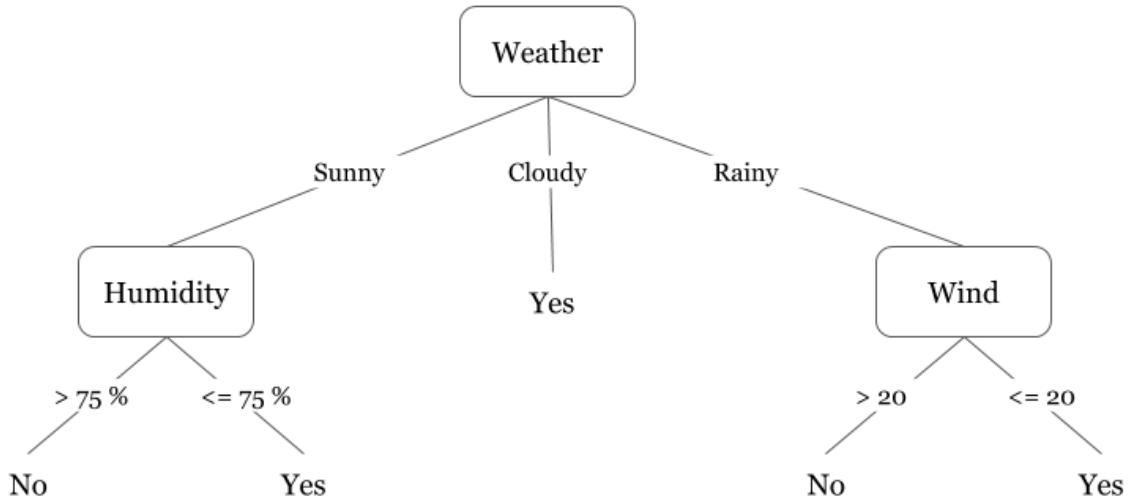

Abbildung 2: badminton.xml

Bibliotheken

Da dieses Projekt textbasiert ist benötige ich nur eine Möglichkeit XML-Dateien einzulesen. Für diese Funktionalität habe ich mich für die Bibliothek **TinyXML-2** (`https://github.com/leethomason/tinyxml2`) entschieden. Die Bibliothek besteht aus zwei Dateien (einer `.h` und einer `.cpp`), die ich einfach in mein Projekt eingefügt habe.

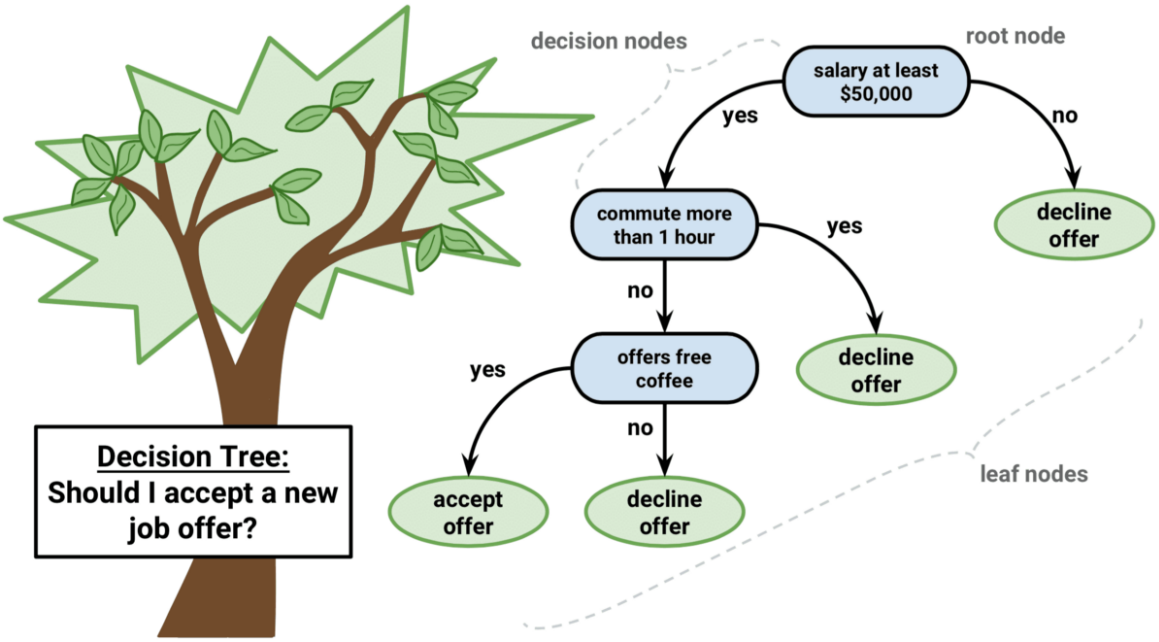

Abbildung 3: job.xml

Build

Zur Project Generation benutze ich Premake (`https://premake.github.io/`). Ich habe die entsprechenden Scripts meiner Abgabe beigefügt. Falls also Probleme mit der VisualStudio Solution autreten sollten, kann mit dem folgenden Befehl das Projekt neu generiert werden und so die Probleme hoffentlich gelöst werden:

```
.\premake\premake5.exe [action]
```

Für mein Projekt habe ich `vs2019` als `action` verwendet. Andere Möglichkeiten sind hier `https://premake.github.io/docs/Using-Premake` aufgelistet.