Curso de Programación en C++.
TPL1-2021. Trabajo Práctico de Laboratorio. [2021-06-02]

PASSWD PARA EL ZIP: 24C5 85V4 56FQ

Ejercicios

[Ej. 1] **[bjgame]** La consigna consiste en escribir jugadores y un simulador para el juego del BlackJack (o también llamado 21). Usaremos una versión muy simplificada de las reglas.

- Hay sólo dos jugadores **p1** y **p2**.
- El juego consiste en una secuencia de manos en la cual los jugadores van pidiendo cartas del mazo.
- Los valores de las cartas van del 1 al 10 y todos los valores del 1 al 10 son igualmente probables. Los valores son generados en forma aleatorio.
- En cada mano cada uno de los dos jugadores va pidiendo cartas del mazo tratando de sumar la cantidad lo más elevada posible pero que no exceda de 21. El jugador puede decidir **plantarse** (es decir no pedir más cartas) por ejemplo cuando la suma llega a 18. Si el jugador *"se pasa"* de 21 su mano es inválida.
- Una vez que cada uno de los jugadores hizo su mano, pueden pasar alguna de las siguientes situaciones.
  - Si las manos de ambos jugadores son inválidas entonces empatan.
  - Si la mano de **p1** es inválida y la de **p2** no, entonces gana **p2** y viceversa.
  - Si la mano de ambos es válida entonces gana el que tiene la suma de cartas más alta.
  - Si ambos tienen la misma suma es empate.
- El juego consiste en realizar una cierta cantidad alta de manos (por ejemplo 1000) y determinar cual es la probabilidad de que gane **p1** o **p2** o haya un empate.
- Los jugadores se implementan derivando la clase virtual pura siguiente,

```
class player_t {
public:
  virtual bool wants_card(vector<int> &hand,
       history_t &me,history_t &you)=0;
  virtual string description() { return "unknown"; }
};
```

- El método que determina la estrategia es **`bool wants_card(vector<int> &hand,...);`** que retorna **`true`** si el jugador quiere otra carta o no. El **`vector<int> &hand`** contiene las cartas en el orden que fueron saliendo.
- El método **`string description()`** es auxiliar y retorna un string que debe ser una descripción de la estrategia utilizada. Tiene una definición por defecto, por lo tanto no es obligatorio implementarla.
- La consigna consiste en implementar una serie de estrategias de jugadores que explicitaremos a continuación.
- La estrategia más simple en plantarse en un cierto número de cartas, por ejemplo 4. Como las cartas van del 1 al 10, en promedio cada carta vale 5, entonces típicamente con 4 cartas llegamos a 20, muy cerca del máximo 21.

Consigna1: implementar una clase derivada **`fixed_t`** con un constructor **`fixed_t(int ncards)`** que debe tener un parámetro interno **`ncards`** que es el número de cartas que pide el jugador.

- Los parámetros **`history_t &me,history_t &you`** de **`wants_card()`** serán explicados después. Contienen la historia de juego de este jugador y el adversario. Entonces por ejemplo si determinamos que el adversario tiende a plantarse en sumas bajas podemos diseñar una estrategia que se plante en sumas bajas también. Total de todas formas ganaremos la mano, sin correr riesgo innecesario de pasarnos.
- **Consigna2:** Escribir una función **`void playbjgame(player_t &p1,player_t &p2, vector<double> &stats,int nhands);`** que toma dos jugadores por referencia y un número da manos y va haciendo jugar a los dos jugadores entre sí **`nhands`** veces.
  - El valor de retorno **`stats`** es la probabilidad de
    - **`stats[0]`** probabilidad de que **`p1`** gane (**`win1`**)
    - **`stats[1]`** probabilidad de que empaten (**`draw`**)
    - **`stats[2]`** probabilidad de que **`p2`** gane (**`win2`**)
  - Para jugar cada mano la función debe generar la mano de **`p1`**, después la de **`p2`**. Para "pedir una carta" puede usar la función

    ```
    int get_card() { return rand()%10+1; }
    ```

  - O sea que para generar cada mano **`playbjgame()`** debe ir llamando a **`wants_card()`** del jugador y pasándole la mano actual **`vector<int> cards`**. El jugador decide en base a la mano actual si se planta o no. En el caso de **`fixed_t(3)`** se planta (retorna **`false`**) si el número de cartas actual ya es 3.
- **Consigna3:** Utilizar la función **`playbjgame()`** para determinar las probabilidades de ganar entre los jugadores **`fixed_t(n)`** con **`n=2,3,4,5,6`**. Como verificación el resultado de hacer jugar a **`fixed_t(3)`** con **`fixed_t(2,4,5,6)`** es,

  ```
  fixed3 vs. fixed2: win1 0.611, draw 0.043, win2 0.346
  fixed3 vs. fixed4: win1 0.565, draw 0.120, win2 0.313
  fixed3 vs. fixed5: win1 0.717, draw 0.147, win2 0.135
  fixed3 vs. fixed6: win1 0.801, draw 0.158, win2 0.040
  ```

  Quiere decir que la mejor estrategia (para los **`fixed_t`**) es **`ncards=3`** ya que le gana a todos los otros.
- **Nota:** Para los jugadores **`fixed_t`** las historias **`history_t h1,h2;`** son irrelevantes, de manera que por el momento no es necesario implementarlas. Sólo basta con declarar a los objetos y pasárselos al jugador.
- Una estrategia más elaborada (**`target_t`**) consiste en plantarse cuando la suma de la mano es mayor que un valor **`targetsum`** predefinido. A primera vista el valor más apropiado para **`targetsum`** parece estar cerca de 16 ya que si ya tenemos 16 y pedimos una carta en promedio saldrá 5 con lo cual justo tendríamos el valor máximo 21.
  Consigna4: implementar una clase derivada **`target_t`** con constructor **`target_t(int targetsum)`** que realiza esta estrategia, es decir se planta cuando la suma de la mano es mayor o igual que **`targetsum`**.
- Hacer competir todos las variantes de **`target_t(n)`** entre si. Verificar que el óptimo es **`target_t(18)`**. Es decir, si enfrentamos a todos **`target_t`** entre sí podemos ver que **`target_t(18)`** vence a todos los otros **`target`**,

  ```
  | PROB(P1=wins)      | PROB(draw) | PROB(P2=wins)    |
  | target(18)* 0.582 | draw 0.024 | target(11) 0.395 |
  ```

```
| target(18)* 0.555 | draw 0.033 | target(12) 0.412 |
| target(18)* 0.528 | draw 0.051 | target(13) 0.421 |
| target(18)* 0.504 | draw 0.072 | target(14) 0.424 |
| target(18)* 0.471 | draw 0.104 | target(15) 0.425 |
| target(18)* 0.440 | draw 0.143 | target(16) 0.417 |
| target(18)* 0.413 | draw 0.187 | target(17) 0.400 |
| target(18)* 0.384 | draw 0.263 | target(19) 0.353 |
| target(18)* 0.430 | draw 0.296 | target(20) 0.274 |
| target(18)* 0.512 | draw 0.332 | target(21) 0.156 |
```

(**Nota:** en la tabla un `*` al lado del jugador indica que gana).

Estrategia adaptativa: Podemos tratar de diseñar un jugador *adaptativo* que modifica su `targetsum` de acuerdo a la estrategia del oponente.
Nota: *Lo que sigue es la motivación. Si quieren pueden saltar directamente a la* ***Consigna5****, más abajo.*

- Si sabemos que el oponente es (por ejemplo) un `target_t(13)` entonces la ganancia máxima no la obtiene `target_t(18)` (cuya probabilidad de ganar contra `target_t(13)` es 0.528, sino que la máxima probabilidad la tiene `target_t(16)` y es 0.594.

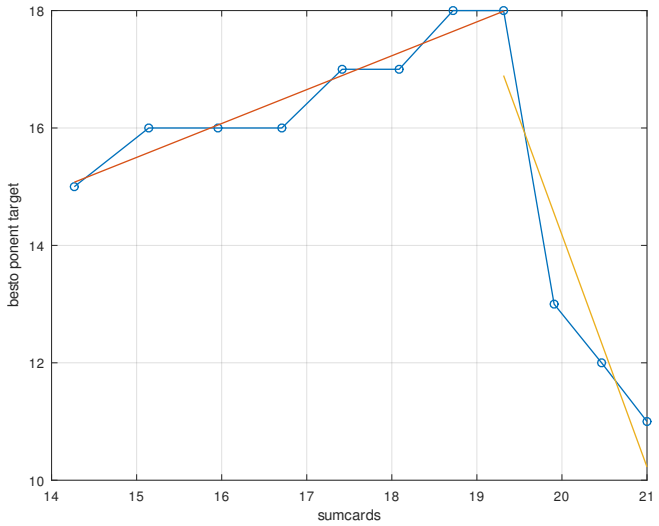
```
| PROB(P1=wins)     | PROB(draw) | PROB(P2=wins)     |
| target(13)* 0.632 | draw 0.092 | target(11) 0.276  |
| target(13)* 0.539 | draw 0.107 | target(12) 0.354  |
| target(13) 0.369  | draw 0.111 | target(14)* 0.520 |
| target(13) 0.328  | draw 0.095 | target(15)* 0.577 |
| target(13) 0.326  | draw 0.080 | target(16)* 0.594 |
| target(13) 0.356  | draw 0.065 | target(17)* 0.579 |
| target(13) 0.422  | draw 0.050 | target(18)* 0.528 |
| target(13)* 0.524 | draw 0.036 | target(19) 0.440  |
| target(13)* 0.655 | draw 0.027 | target(20) 0.318  |
| target(13)* 0.807 | draw 0.019 | target(21) 0.174  |
```

  Entonces esto sugiere que podemos desarrollar una estrategia **adaptativa**. Estudiando la historia de juego del adversario podemos tratar de adivinar su **target** y buscar cual es el target más adecuado, es el que tiene más probabilidad de ganar. Para esto relevamos la tabla siguiente, donde para cada player `target_t(n)` calculamos la probabilidad de pasarse `P(pasarse)` y la suma de cartas en el caso exitoso `sumcards`. Obviamente para un target `n`muy bajo (por ejemplo 11) la probabilidad de pasarse es muy baja, pero la suma de cartas también es en promedio baja: 14.27. Por el contrario contrario para un target alto (por ejemplo 21) la probabilidad de pasarse es muy alta (0.823) pero si no se pasa la suma de cartas es muy alta (justamente 21 en este caso).

```
target(n) P(pasarse) sumcards  BESTP2 P(win2)
   11     0.00000    14.26700   15    0.70700
   12     0.00000    15.14200   16    0.64900
   13     0.01700    15.95600   16    0.59400
   14     0.05100    16.70700   16    0.54200
   15     0.10400    17.41600   17    0.49400
   16     0.17500    18.08600   17    0.45500
   17     0.26900    18.72000   18    0.40900
```

18	0.41300	19.31400	18	0.42500
19	0.50900	19.90800	13	0.52300
20	0.66000	20.46600	12	0.66200
21	0.82300	21.00000	11	0.82200

Un jugador adaptativo **`adaptative_t p1`** puede diseñarse así, debe tratar de deducir la estrategia de su adversario (adivinar su **`p2.target`**) y en base a eso buscar el target más apropiado. Para adivinar el target del adversario, no podemos mirar su código, pero lo podemos tratar de adivinar a través de su historial de jugadas **`h2`**. Los historiales son simplemente **`vector<vector<int»`** donde cada entrada **`j`** son las cartas que jugó en esa mano. Entonces podemos esperar un cierto número de manos, por ejemplo 100, calcular la suma de cartas promedio **`sumcards`** de **`p2`** y con es valor el que más se aproxima en la 3ra columna de la tabla. A partir de ahí, deducimos cuál es el *target* del oponente (columna 1) y por lo tanto cuál es nuestro valor óptimo de *target* (columna 4). En la figura la curva azul de lineas y círculos muestra el target óptimo como función del sumcards.
Vemos que si el **`target`** del oponente es bajo (por ejemplo 11) **`sumcards`** será relativamente bajo también (14.27) entonces el óptimo es tomar **`target=15`** con lo cual probabilidad de ganar asciende a 0.707. Por el contrario si el oponente usa un target muy alto (por ejemplo 20) entonces tiene **`sumcards`** muy alto (20.4) pero falla muchas veces (66 %), entonces basta con tomar un target muy bajo (12) porque el 66 % de las veces el oponente se va a pasar.



Para evitar tener que cargar la tabla podemos aproximar el **`target`**, en función del **`sumcards`**, como dos tramos lineales (en rojo y anaranjado en la figura)

```
target=int(0.577*sumcards+6.842);   // si sumcards<19.5
target=int(-1.831*sumcards+49.462); // si sumcards>=19.5
```

- **Consigna5:** Implementar un jugador **`adaptative_t`** de acuerdo a la siguiente estrategia:
  - En las primeras 100 jugadas usa **`target=18`**.
  - A partir de las 100 jugadas calcula el **`sumcards`** del oponente y en base a ese valor usa las expresiones,
    ```
    target=int(0.577*sumcards+6.842);   // si sumcards<19.5
    target=int(-1.831*sumcards+49.462); // si sumcards>=19.5
    ```
    para adaptar su **`target`**. Por las dudas al finalizar el cálculo del **`target`** llevarlo al rango

`[11,18]`.

- *Cálculo del* **`sumcards`** *del oponente:* El jugador recibe en **`wants_card(hand,me,you);`** la historia de las jugadas en este partido, **`me`** son las mías, **`you`** las del oponente. Para implementar el **`adaptative_t`** sólo necesitamos las del oponente. Las historias son simplemente vectores de vectores de enteros:

```
typedef vector<vector<int>> history_t;
history_t h1,h2;
```

- Por supuesto para que los jugadores reciban las historias hay que modificar **`playbjgame()`** para que efectivamente mantenga las historias de ambos (hasta ahora sólo las declara pero están vacías). Para ello después de cada mano tiene que apendizar las manos de cada jugador a su historia. Notar que cuando llame a los jugadores tiene que pasarle en orden apropiado las historias. Es decir para
  - para **`p1`**: **`me`** es **`h1`** y **`you`** es **`h2`**
  - para **`p2`**: **`me`** es **`h2`** y **`you`** es **`h1`**

  Es decir que las llamadas deberían ser así,

```
history_t h1,h2;
// ... Dentro de un lazo le va preguntando a p1 y p2 si quieren
// más cartas
while (...) p1.wants_card(cards1,h1,h2);
while (...) p2.wants_card(cards2,h2,h1);
// Determina si gano p1 o p2 y actualiza stats[]...
// Apendiza cards1 a h1...
// Apendiza cards2 a h2...
```

- **Consigna6:** Realizar las tablas de probabilidad entre todos los jugadores: **`fixed_t(2-6)`**, **`target_t(11-21)`**, **`adaptative_t`**.

[Ej. 2] **[linearops]** Consideremos operaciones lineales homogéneas sobre vectores en $\mathbb{R}^n \to \mathbb{R}^n$, es decir están identificadas de la forma $y = Ax$, donde $A \in \mathbb{R}^{n \times n}$. Sin embargo, queremos representar estos operadores sólo por su acción sobre los vectores, es decir por la operación de aplicar el operador a un vector. Consideraremos entonces los operadores como clases derivadas de la clase virtual pura,

```
typedef vector<double> vd_t;
typedef vector<vd_t> vvd_t;

class linearop_t {
public:
  virtual void apply(vd_t &y,vd_t &x)=0;
  string description() { return "unknown"; }
};
```

A lo largo del ejercicio usaremos los **`typedef vd_t`** y **`vvd_t`** para vectores de dobles y vectores de vectores.

El método **`description()`** es opcional y debe retornar un string que describe a la acción de ese operador.

Implementar operadores:

- **Consigna1:** Implementar la clase **`zero_t`** que corresponde a la matriz $A = 0$, es decir por lo tanto que para todo **`x`** debe retornar el vector nulo.
- **Consigna2:** Implementar la clase **`identity_t`** que corresponde a la matriz identidad, es decir que $y = x$.
- **Consigna3:** Implementar la clase **`scale_t(a)`** que corresponde a la matriz $A = aI$, es decir un múltiplo escalar de la identidad. La operación corresponde a $y = ax$.
- **Consigna4:** Implementar la clase **`rot_t(int s)`** que hace un *shift* cíclico de los elementos del vector en **`s`** posiciones, es decir si **`x=[1,2,3]`** y **`s=1`** entonces debe dar **`y=[2,3,1]`**.
- **Consigna5:** Implementar la clase **`reflex_t(int axis)`** que cambia de signo la componente **`axis`**. Por ejemplo si **`x=[5,3,6,9]`** y **`axis=2`** entonces debe dar **`y=[5,3,-6,9]`**. Geométricamente corresponde a una reflexión según el plano perpendicular al eje **`axis`**.

Predicados:
Ahora nos podemos preguntar como implementar predicados, por ejemplo si es SPD (simétrica y positiva definida), si preserva la norma. También dados dos operadores si son iguales, etc...

- **Consigna6:** Implementar el predicado
  `bool is_spd(linearop_t &A,vvd_t &vectors,double tol=0.0);` que determina si el operador lineal **`A`** es SPD. Recordemos que una matriz se dice que es SPD si $(x, Ax) > 0,\ \forall x \in \mathbb{R}^n, x \neq 0$ donde $(x, y) = \sum_j x_j y_j$ denota el producto escalar. La verificación no se puede hacer sobre todos los vectores del espacio, porque son infinitos, pero lo hacemos sobre un cierto conjunto de vectores **`vvd_t &vectors`**. La función debe entonces recorrer los vectores de **`vectors`**, aplicarle el operador y luego hacer el producto escalar con **`x`** y verificar. Si es negativo al menos para un sólo **`x`** entonces retorna **`false`**. Caso contrario retorna **`true`**.
  - Verificar que la identidad es SPD.
  - Verificar que **`scale_t`** es SPD si **`a>0`**.
  - Verificar que **`scale_t`** no es SPD si **`a<0`**.
  - Verificar que **`rot_t(n)`** no es SPD a menos que **`n=0`** o **`n`** es un múltiplo de la dimensión, ya que en ese caso es la identidad.
  - Verificar que **`reflex_t`** no es SPD.

  Debido a errores de redondeo puede ser que el resultado de ligeramente negativo, por eso dejamos un argumento **`tol`** es decir que en el predicado se debe verificar que $(x, Ax) \geq -$**`tol`**. Si **`tol=0`** entonces estamos usando la versión exacta y si ponemos por ejemplo **`tol=1e-10`** le estamos tolerancia para el error de redondeo.
  Nota: En el **`program.cpp`** ya está implementada una función **`mkrandvectors()`** que genera los vectores aleatorios de prueba. Por ejemplo para verificar que **`scale_t(5.0)`** es SPD deberíamos hacer,

```
vvd_t vectors;
mkrandvectors(vectors);
scale_t smas(1.0),smenos(-1.0);
cout << "is_spd(smas) " << is_spd(smas,vectors) << endl;
cout << "is_spd(smenos) " << is_spd(smenos,vectors) << endl;
```

  y debe dar,

```
$ ./program.bin
is_spd(smas) 1
is_spd(smenos) 0
```

- **Consigna7:** Implementar el predicado
  `bool is_normpres(linearop_t &A,vvd_t &vectors,double tol=0.0);` que verifica si el operador *"preserva la norma"*, es decir que $|Ax| = |x|,\ \forall x$. Las reflexiones y rotaciones preservan la norma.
  - Verificar que la identidad preserva la norma.
  - Verificar que **`scale_t(a)`** no preserva la norma, salvo para **`a=-1,1`**.
  - Verificar que **`rot_t`** preserva la norma.
  - Verificar que **`reflex_t`** preserva la norma.

  La versión con tolerancia verifica que: $||y| - |x|| <$**`tol`**, donde $y = Ax$.
- **Consigna8:** Implementar el predicado **`bool are_equal(linearop_t &A1,linearop_t &A2, vvd_t &vectors,double tol=0.0);`** que toma dos operadores **`A1`**, **`A2`** y verifica si los operadores son iguales. Para ello recorre los vectores en **`vectors`** le aplica el operador **`A1`** y **`A2`** a cada uno de ellos y verifica si los resultados son iguales. Es decir si $y_1 = A_1x$, $y_2 = A_2x$ debe ser $y_2 = y_1$.
  La versión con tolerancia verifica que: $|y_2 - y_1| <$**`tol`**.
  Verificar que **`identity_t`** y **`scale_t(1)`** y **`rot_t(0)`** son todos iguales.

Composición de operadores:
Dada una serie de operadores $A_1, A_2...A_n$ podemos definir el operador compuesto A tal que $Ax = A_n(A_{n-1}(...(A_1x)))$, es decir vamos aplicando A_1 y luego A_2 hasta A_n.

Consigna9: Implementar la clase derivada **`compose_t`** que contiene un vector de punteros a operadores **`vector<linearop_t*> ops`** de manera que el método **`apply`** consiste en ir aplicando los operadores de **`ops`** (**Nota:** tal vez haga falta usar vectores temporarios.)

```
class compose_t : public linearop_t {
public:
  vector<linearop_t*> ops;
  void apply(vd_t &y,vd_t &x) {
    // Aplica los operadores de ops a x...
  }
};
```

- Verificar que la composición de un **`scale_t(5)`** y **`scale_t(0.2)`** es igual a un **`identity_t`**.
- Verificar que si la composición de **`n`** **`rot_t(1)`** es igual a un **`identity_t`**, donde **`n`** es la dimensión del vector.
  Nota: la dimensión de todos los vectores de prueba es 3, de manera que debe construir una composición de 3 **`rot_t(1)`** y chequear que ese operador es la identidad.
- Verificar que la composición de 2 **`reflex_t(axis=0)`** es igual a un **`identity_t`**, ya que dos reflexiones a través del mismo plano deja la identidad. Lo mismo para los otros planos, es decir 2 **`reflex_t(axis=1)`** o 2 **`reflex_t(axis=2)`** también deben dar la identidad.

mstorti@galileo/texstuff-1.4.0-67-ge52a7899/Tue Jun 22 08:48:50 2021 -0300