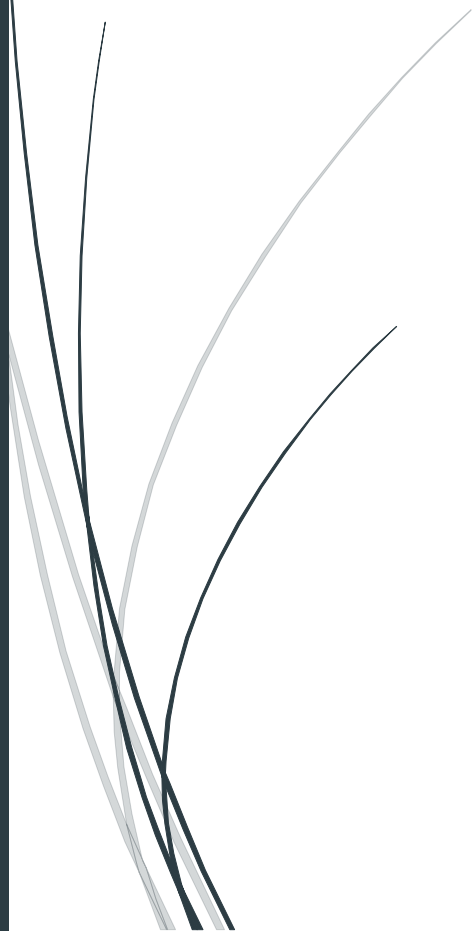
Algoritmos e Estruturas de Dados

2022/2023

universidade de aveiro

Professor
Tomás Silva

Speed Run



Diogo Silva-107647 -> 33.07%
Rafael Vilaça-107476 -> 33.6%
Miguel Cruzeiro-107660 ->33.33%

Índice

Introdução

Problema Proposto

Inicialmente, temos uma estrada dividida em vários segmentos de igual tamanho e sabemos que cada segmento de estrada tem um limite de velocidade. A velocidade é definida pelo número de segmentos de estrada que o carro avança num único movimento. Em cada movimento o carro tem 3 opções: reduzir a sua velocidade; manter a sua velocidade ou aumentar a sua velocidade.

No início o carro é colocado no 1º segmento de estrada com uma velocidade de 0 e tem como objetivo chegar ao último segmento com velocidade de 1 (para assim poder reduzir a sua velocidade e parar).

O objetivo deste trabalho é determinar o número mínimo de movimentos que são necessários para o carro chegar á posição final.

Métodos Utilizados

Brute Force

O "Brute Force" é um método direto e intuitivo em que para cada movimento são calculadas todas as possibilidades seguintes e para o movimento seguinte é sempre escolhida a melhor possibilidade.

1. *primeiro* (*P*): gera o primeiro candidato à solução de *P*.
2. *próximo* (*P*, *c*): gera o próximo candidato de *P* depois de *c*.
3. *válido* (*P*, *c*): checa se o candidato *c* é a solução de *P*.
4. *saída* (*P*, *c*): usa a solução *c* de *P* como for conveniente para a aplicação.

Imagem retirada de:"https://pt.wikipedia.org/wiki/Busca_por_força_bruta"

Embora seja um método fácil de implementar (em termos de código) e chega sempre á solução correta este método não é eficiente tendo uma complexidade algorítmica de O(n*n!).

Branch and Bound

O algoritmo "Branch and Bound" é similar ao algoritmo "Brute Force" tendo como grande diferença o mesmo não analisar soluções que através de conhecimento previamente adquirido já não seriam soluções logo só são contabilizadas as possibilidades que podem ser solução.

A complexidade algorítmica para este algoritmo não é um valor exato podendo mudar dependendo do contexto do problema/programa.

Outros métodos e notas importantes

Devido á falta de tempo ou na dificuldade na resolução do problema dado, alguns métodos não foram implementados, no entanto achámos importante referi-los.

A programação dinâmica é uma técnica eficiente de programação que nos ajuda a resolver qualquer problema que possa ser dividido em sub-problemas (estes sub-problemas são ainda divididos em sub-problemas mais pequenos).

Este método recorre a que tenhamos de calcular repetidas vezes o valor de um mesmo sub-problema até achar a melhor solução.

No contexto deste problema seria criado um array com a posição e a velocidade como índices em que para cada segmento seria guardado o melhor número de movimentos até chegar a essa posição com uma determinada velocidade.

De seguida, esse array seria percorrido do fim para o início de modo a descobrir o caminho até ao final com o menor número de movimentos possível.

Soluções

Brute force

```
static void solution_1_recursion(int move_number, int position, int speed, int final_position)
{

  int i, new_speed;
  // record move
  solution_1_count++;
  solution_1.positions[move_number] = position;
  // is it a solution?
  if (position == final_position && speed == 1)
  {
    // is it a better solution?
    if (move_number < solution_1_best.n_moves)
    {
      solution_1_best = solution_1;
      solution_1_best.n_moves = move_number;
    }
    return;
  }
  // no, try all legal speeds
  for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      if (i > new_speed)
        solution_1_recursion(move_number + 1, position + new_speed, new_speed, final_position);
    }
}
```

Nesta primeira solução começa se por criar uma variável(count) para guardar o número de vezes que é chamada a função recursiva e um array com o número de movimentos para cada posição.

De seguida verifica se se está na posição final e se a velocidade é um. Caso seja verdade e se não houver nenhuma solução melhor, atualiza se a variável "solution_1_best.n_moves" com o número de movimentos necessário até chegar ao final e termina se esta iteração.

Se ainda não estivermos na posição final é executado um "Loop" for para definir a próxima velocidade que pode ser maior, menor ou igual do que atual em uma unidade.

Dentro desse for verifica se que a velocidade é maior que zero e se é menor ou igual do que a velocidade máxima da estrada na próxima posição para chamar a função recursiva e calcular essa nova posição.

Brunch and bound (incomplete)

```
static void solution_2_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_2_count++;
  solution_2.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_2_best.n_moves)
    {
      solution_2_best = solution_2;
      solution_2_best.n_moves = move_number;
    }
    return;
  }
  if (move_number > best_move_number_2[position][speed] && best_move_number_2[position][speed] != 0)
    return;
  best_move_number_2[position][speed] = move_number;
  for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      if (i > new_speed)
        solution_2_recursion(move_number + 1, position + new_speed, new_speed, final_position);
    }
}
```

Para a segunda solução é inicialmente criado um array com a posição e a velocidade como índices. Inicia se também todos os valores do array com valor zero.

Diferente da primeira solução, nesta são atualizados os valores do array para a posição em que se encontra se ainda não existir um valor melhor de movimentos no array. Se os valores para esta posição já tiverem sido calculados e forem melhores do que o atual, termina se a iteração.

Brunch and bound (complete) brake first

```
static void solution_3_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_3_count++;
  solution_3.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_3_best.n_moves)
    {
      solution_3_best = solution_3;
      best_move_number_3[position][speed] = move_number;
      solution_3_best.n_moves = move_number;
    }
    return;
  }
  if (move_number == best_move_number_3[position][speed]){
    counter_sol_3 +=1;
  }
  if (move_number >= best_move_number_3[position][speed]){
    return;
  }
  best_move_number_3[position][speed]=move_number;
  for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      {
        if (i > new_speed)
        {
          solution_3_recursion(move_number + 1, position + new_speed, new_speed, final_position);
        }
      }
    }
}
```

Neste caso o array "best_move_number" foi inicializado com um valor elevado (10000) podendo assim evitar problemas futuros.

Foi também criada uma variável (counter_sol_3) que representa o número de soluções para uma determinada posição e velocidade, ou seja, o número de vezes que, fazendo a recursão chegou ao mesmo valor de movimentos para a mesma posição e velocidade. Com esta variável é possível verificar o número de vezes que foi possível evitar que a recursão fosse chamada em relação à solução anterior, visto que nesta solução todos os valores do número de movimentos que forem iguais aos existentes no array já não serão calculados.

É criada também uma condição "IF" que verifica se para cada vez que a recursão é chamada, se aquela posição e velocidade já foram calculadas previamente com um número de movimentos inferior. Caso isso se verifique termina a iteração.

Desta forma é possível ver uma melhoria significativa de performance porque muitos valores são guardados ao serem calculados previamente evitando assim que sejam calculados novamente.

Brunch and bound (complete) accelerate first v1

```
static void solution_4_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_4_count++;
  solution_4.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_4_best.n_moves)
    {
      solution_4_best = solution_4;
      best_move_number_4[position][speed] = move_number;
      solution_4_best.n_moves = move_number;
    }
    return;
  }
  if (move_number == best_move_number_4[position][speed]){
    counter_sol_4 +=1;
  }
  if (move_number >= best_move_number_4[position][speed]){
    return;
  }
  best_move_number_4[position][speed]=move_number;
  for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      {
        if (i > new_speed)
          { solution_4_recursion(move_number + 1, position + new_speed, new_speed, final_position);
        }
      }
    }
}
```

Para esta solução foi realizada apenas uma alteração, sendo a mesma para que seja testada primeiro uma velocidade superior à atual. É de notar uma melhoria substancial após esta alteração, porque a mesma começa sempre com a maior velocidade possível diminuindo apenas se necessário.

Brunch and bound (complete) accelerate first v2

```
static void solution_5_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_5_count++;
  solution_5.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_5_best.n_moves)
    {
      solution_5_best = solution_5;
      best_move_number_5[position][speed] = move_number;
      solution_5_best.n_moves = move_number;
    }
    return;
  }
  if (move_number == best_move_number_5[position][speed]){
    counter_sol_5 +=1;
  }
  if (move_number >= best_move_number_5[position][speed]){
    return;
  }
  best_move_number_5[position][speed]=move_number;
  for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      {
        if (i > new_speed)
          {
            if (best_move_number_5[position+new_speed][new_speed] == 10000){
              solution_5_recursion(move_number + 1, position + new_speed, new_speed, final_position);
            }
          }
        }
      }
}
```

Para a segunda solução foi adicionada uma condição "if" para que no momento que é testada uma nova velocidade é verificado se o valor da posição seguinte e a sua velocidade ainda não foram calculados anteriormente. Esta verificação é efetuada averiguando a existência destes valores no array (best_move_number_5).

Se ainda não tiver sido calculado é chamada a função para calcular os valores para essa nova posição.

Esta solução é capaz de reduzir o número de vezes que a função é chamada para menos de metade melhorando assim o tempo de execução.

Brunch and bound (complete) accelerate first v3

```
static void solution_6_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_6_count++;
  solution_6.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_6_best.n_moves)
    {
      solution_6_best = solution_6;
      best_move_number_6[position][speed] = move_number;
      solution_6_best.n_moves = move_number;
    }
    return;
  }
  best_move_number_6[position][speed]=move_number;
  for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position )
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      {
        if (best_move_number_6[position+new_speed][new_speed] <= move_number+1 && position+new_speed != final_position)
        {
          return;
        }
        else if (i > new_speed)
          {
            solution_6_recursion(move_number + 1, position + new_speed, new_speed, final_position);
          }

      }

  }
}
```

No caso da solução 6, quando se testa uma nova velocidade, foi acrescentada uma condição "if" para verificar se o valor do número de movimentos até chegar à posição seguinte já foi calculado e existe no array (best_move_number_6).

Se já existir, caso o número de movimentos for superior ao número de movimentos da posição atual mais uma unidade (próxima posição) e a posição não for a posição final termina-se a iteração.

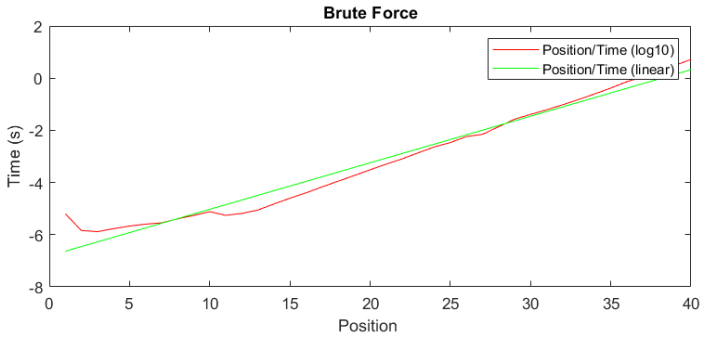
Com isto é possível reduzir ainda mais o número de vezes que é necessário chamar a função recursiva.

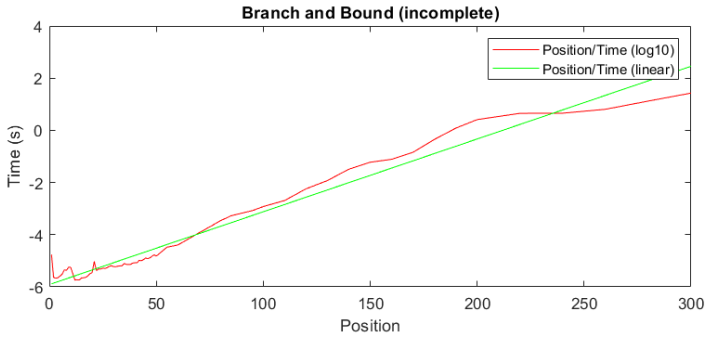
Resultados Obtidos

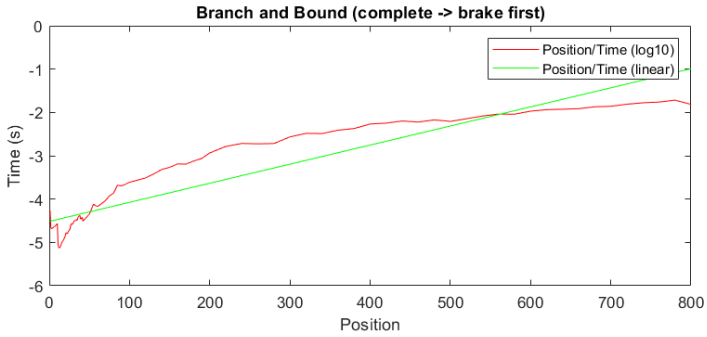
Comparação da eficiência dos métodos

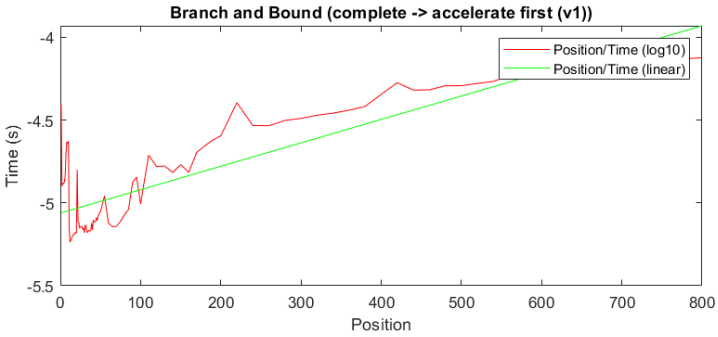
Todos os gráficos que serão apresentados de seguida foram obtidos ao executar o programa com o numero mecanográfico 107647

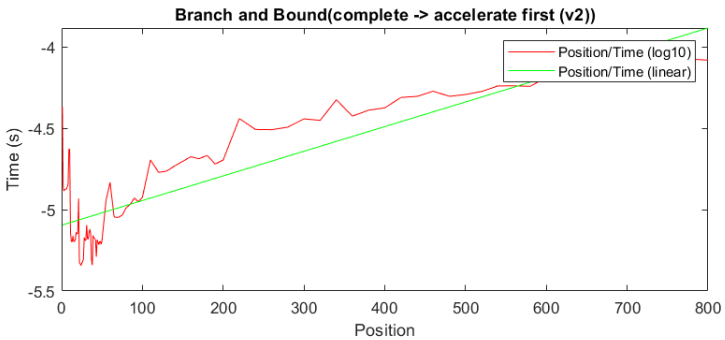
Run Time

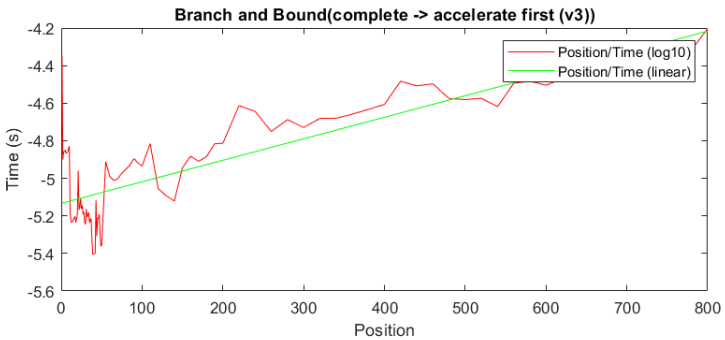


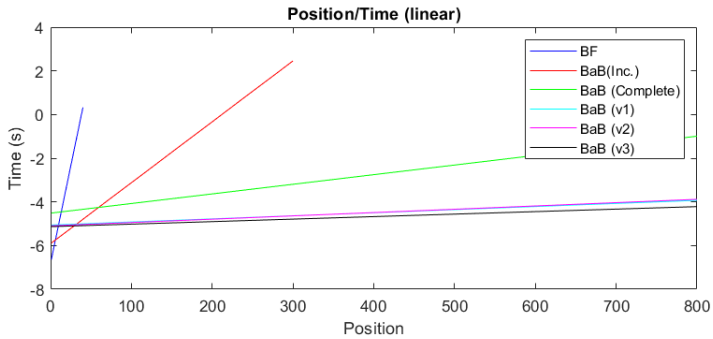










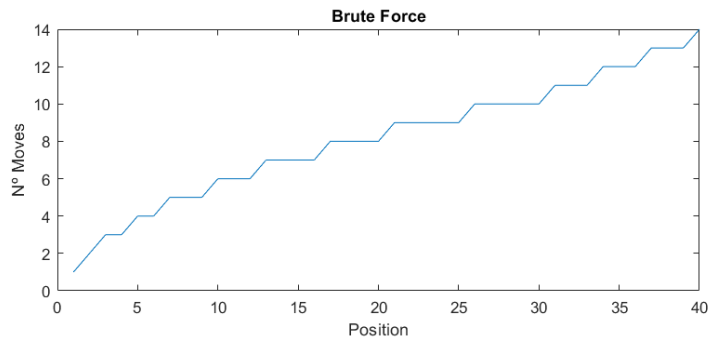


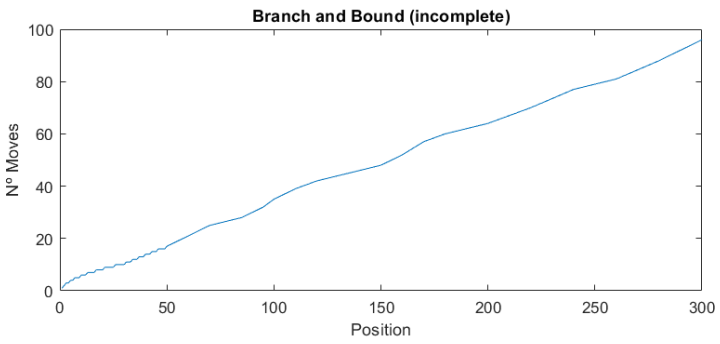
Como podemos ver pelos gráficos de run time(s), podemos concluir que o método "Brute Force" é o que se torna mais demorado com o aumento da posição o que comprova a respetiva complexidade logarítmica de **n*n!**

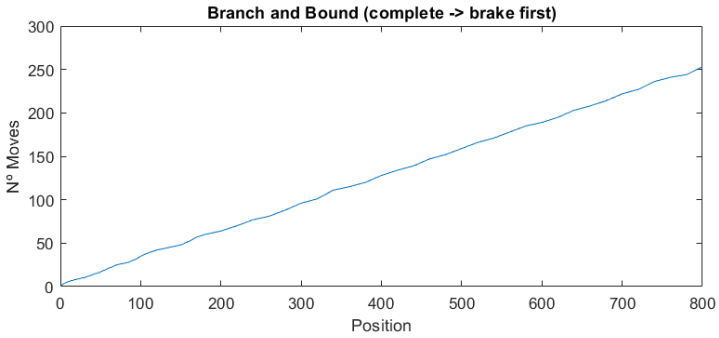
Em relação a todas as versões do "Branch and Bound" podemos concluir como referido anteriormente que a cada versão existe uma diminuição substancial do run time.

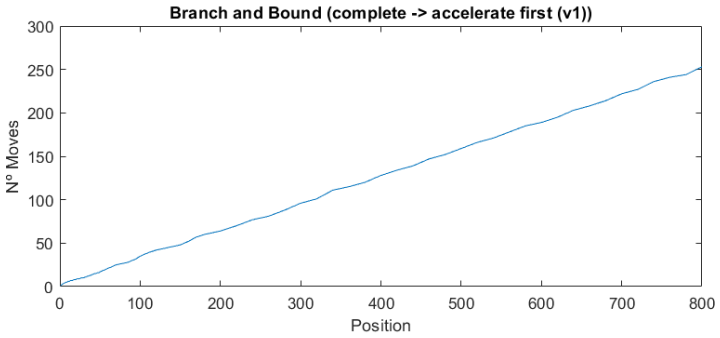
Para as duas primeiras soluções, como não é recomendado esperar pelo cálculo do run time para a última posição final (800), este valor é então estimado usando o matlab, o mesmo sendo respetivamente para o algoritmo Brute Force (34 minutos e 43 segundos) e para o algoritmo Branch and Bound (Incomplete) (25 minutos e 9 segundos).

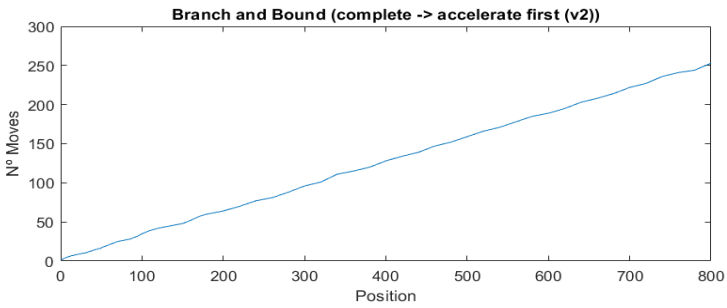
Número de Movimentos

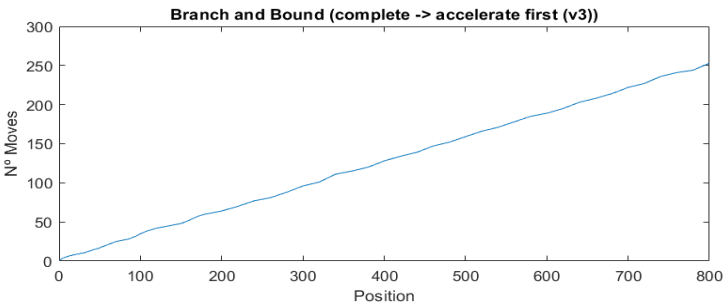


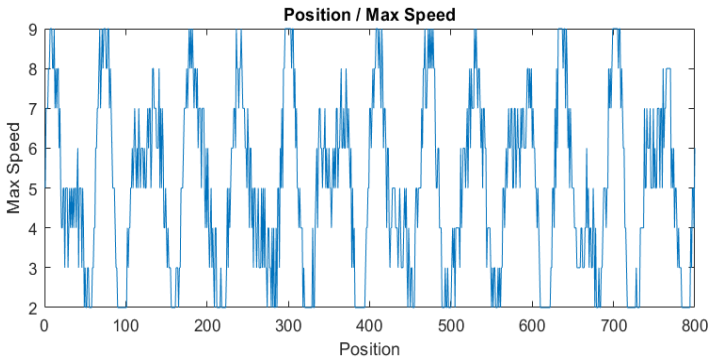


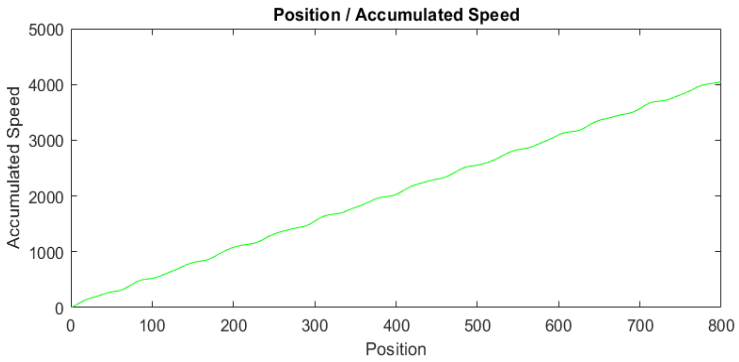












Nos gráficos acima estão representados o número de movimentos em função da posição e os gráficos da posição em relação com o MaxSpeed e Accumulated Speed.

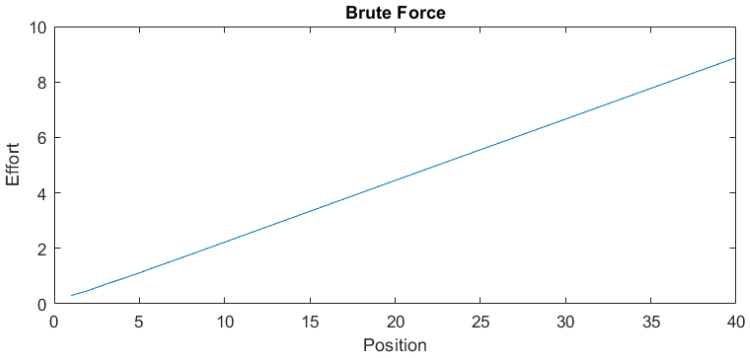
Como podemos ver nos gráficos acima, o número de movimentos vai ser idêntico para todos os métodos utilizados tendo apenas o primeiro método um move number estimado errado (223.96), provavelmente porque o mesmo não calcula (no tempo desejado) posições suficientes para estimar para a posição final.

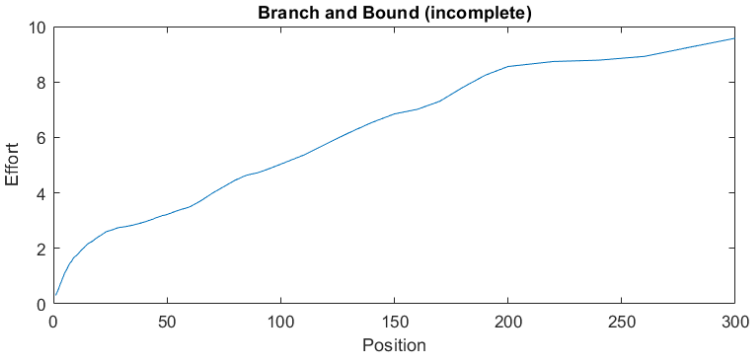
Esse problema é resolvido no segundo método que, apesar de não calcular a posição final (no tempo desejado), apresenta um move number estimado correto (252.53).

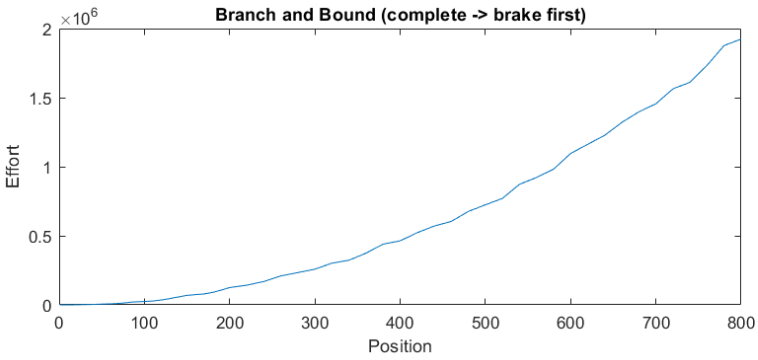
Os restantes métodos, como os mesmos terminam a sua execução e os valores de move number são idênticos, estão corretos (para qualquer método utilizado o valor do número de movimentos deve ser igual para a mesma posição).

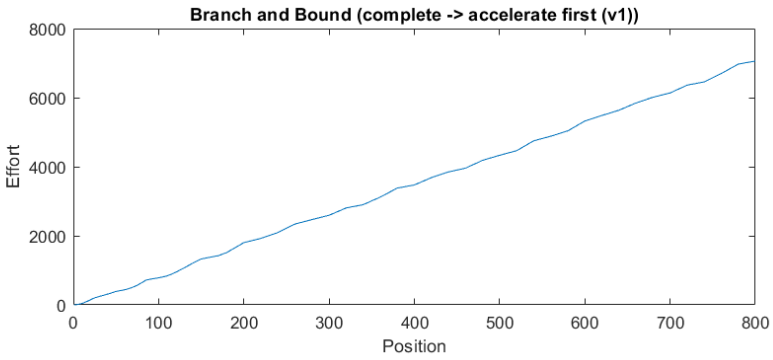
No gráfico da MaxSpeed é visível a velocidade máxima que pode ser atingida para cada posição. É de notar que em diversas posições, devido a esta característica, é verificada uma mudança nos gráficos. Por exemplo, nos gráficos do run time, logo após a posição 200 com uma velocidade máxima próxima da mínima possível, como é necessário fazer um número maior de cálculos, existe um run time ligeiramente mais demorado. Enquanto na posição 300 com a velocidade máxima verificamos uma pequena diminuição de calculo e tempo necessário.

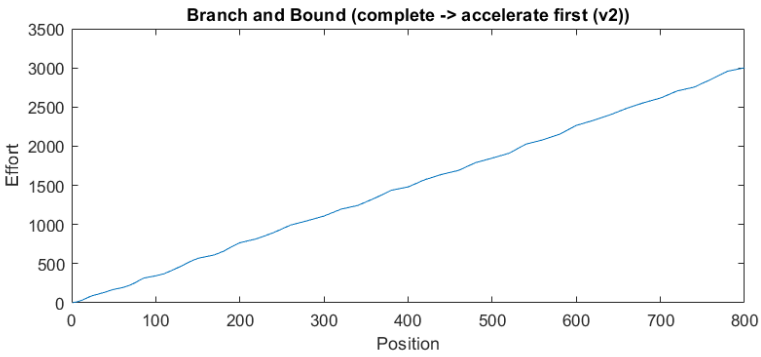
Esforço

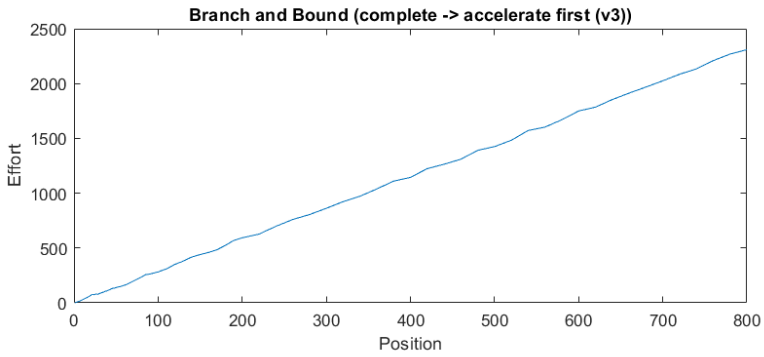












Nestes gráficos podemos ver o esforço (quantidade de vezes que a função é chamada) em função da posição para cada método.

No primeiro gráfico, para ter um aspeto mais linear, é utilizado o logaritmo e é também estimado o esforço necessário para calcular a última posição (800) nesse método que é 5.03e^09.

Para os restantes, não foi efetuada nenhuma mudança para modificar o seu aspeto, porque os mesmos, sendo apenas estimado o esforço para a posição final (800) no método 2 que é 3.45e^09.

Destes 4 métodos restantes, o primeiro tem o pior aspeto e o maior esforço devido ao mesmo calcular as posições sempre com uma velocidade (velocidade-1) inferior à desejada. Esse cálculo é ineficiente porque o objetivo do programa é chegar o mais rapidamente possível á posição final.

Muitas vezes a solução correta é a velocidade+1, logo os cálculos que são efetuados inicialmente são desnecessários. Isto transforma-se num efeito cascata onde para cada cálculo desnecessário são calculados mais cálculos desnecessários.

Código completo

Código em C

```c
// AED, August 2022 (Tomás Oliveira e Silva)
//
// First practical assignement (speed run)
//
// Compile using either
//   cc -Wall -O2 -D_use_zlib_=0 solution_speed_run.c -lm
// or
//   cc -Wall -O2 -D_use_zlib_=1 solution_speed_run.c -lm -lz
//
// Place your student numbers and names here
//   N.Mec. XXXXXX  Name: XXXXXXX
//

//
// static configuration
//

#define _max_road_size_ 800 // the maximum problem size
#define _min_road_speed_ 2  // must not be smaller than 1, shouldnot be smaller than 2
#define _max_road_speed_ 9  // must not be larger than 9 (only because of the PDF figure)

//#define sol_1
//#define sol_2
//#define sol_3
//#define sol_4
//#define sol_5
#define sol_6

//
// include files --- as this is a small project, we include the PDF generation code directly from make_custom_pdf.c
//

#include <math.h>
#include <stdio.h>
#include "elapsed_time.h"
#include "make_custom_pdf.c"

//
// road stuff
//

static int max_road_speed[1 + _max_road_size_]; // positions 0.._max_road_size_
```

```c
static void init_road_speeds(void)
{
  double speed;
  int i;

  for (i = 0; i <= _max_road_size_; i++)
  {
    speed = (double)_max_road_speed_ * (0.55 + 0.30 * sin(0.11 * (double)i) + 0.10 * sin(0.17 * (double)i + 1.0) + 0.15 * sin(0.19 * (double)i));
    max_road_speed[i] = (int)floor(0.5 + speed) + (int)((unsigned int)random() % 3u) - 1;
    if (max_road_speed[i] < _min_road_speed_)
      max_road_speed[i] = _min_road_speed_;
    if (max_road_speed[i] > _max_road_speed_)
      max_road_speed[i] = _max_road_speed_;
  }
}

//
// description of a solution
//

typedef struct
{
  int n_moves;                          // the number of moves (the number of positions is one more than the number of moves)
  int positions[1 + _max_road_size_]; // the positions (the first one must be zero)
} solution_t;

//
// the (very inefficient) recursive solution given to the students
//

static solution_t solution_1, solution_1_best;
static double solution_1_elapsed_time; // time it took to solve the problem
static unsigned long solution_1_count; // effort dispended solving the problem
```

```c
static void solution_1_recursion(int move_number, int position, int speed, int final_position)
{

  int i, new_speed;
  // record move
  solution_1_count++;
  solution_1.positions[move_number] = position;
  // is it a solution?
  if (position == final_position && speed == 1)
  {
    // is it a better solution?
    if (move_number < solution_1_best.n_moves)
    {
      solution_1_best = solution_1;
      solution_1_best.n_moves = move_number;
    }
    return;
  }
  // no, try all legal speeds
  for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      if (i > new_speed)
        solution_1_recursion(move_number + 1, position + new_speed, new_speed, final_position);
    }
}

static void solve_1(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_1: bad final_position\n");
    exit(1);
  }
  solution_1_elapsed_time = cpu_time();
  solution_1_count = 0ul;

  solution_1_best.n_moves = final_position + 100;
  solution_1_recursion(0, 0, 0, final_position);
  solution_1_elapsed_time = cpu_time() - solution_1_elapsed_time;
}
```

```c
static solution_t solution_2, solution_2_best;
static double solution_2_elapsed_time; // time it took to solve the problem
static unsigned long solution_2_count; // effort dispended solving the problem
static int best_move_number_2[_max_road_size_][_max_road_speed_];


static void solution_2_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_2_count++;
  solution_2.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_2_best.n_moves)
    {
      solution_2_best = solution_2;
      solution_2_best.n_moves = move_number;
    }
    return;
  }
  if (move_number > best_move_number_2[position][speed] && best_move_number_2[position][speed] != 0)
    return;
  best_move_number_2[position][speed] = move_number;
  for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      if (i > new_speed)
        solution_2_recursion(move_number + 1, position + new_speed, new_speed, final_position);
    }
}

static void solve_2(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_2: bad final_position\n");
    exit(1);
  }
  solution_2_elapsed_time = cpu_time();
  solution_2_count = 0ul;
  solution_2_best.n_moves = final_position + 100;
  solution_2_recursion(0, 0, 0, final_position);
  solution_2_elapsed_time = cpu_time() - solution_2_elapsed_time;
}
```

```c
static solution_t solution_3, solution_3_best;
static double solution_3_elapsed_time; // time it took to solve the problem
static unsigned long solution_3_count; // effort dispended solving the problem
static int best_move_number_3[_max_road_size_+1][_max_road_speed_+1];
static long counter_sol_3 = 0;


static void solution_3_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_3_count++;
  solution_3.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_3_best.n_moves)
    {
      solution_3_best = solution_3;
      best_move_number_3[position][speed] = move_number;
      solution_3_best.n_moves = move_number;
    }
    return;
  }
  if (move_number == best_move_number_3[position][speed]){
    counter_sol_3 +=1;
  }
  if (move_number >= best_move_number_3[position][speed]){
    return;
  }
  best_move_number_3[position][speed]=move_number;
  for (new_speed = speed - 1; new_speed <= speed + 1; new_speed++)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      {
        if (i > new_speed)
        {
          solution_3_recursion(move_number + 1, position + new_speed, new_speed, final_position);
        }
      }
    }
}
```

```c
static void solve_3(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_3: bad final_position\n");
    exit(1);
  }
  solution_3_elapsed_time = cpu_time();
  solution_3_count = 0ul;
  for (int b = 0 ;b<=_max_road_size_;b++)
  {
    for (int f=0; f<=_max_road_speed_;f++)
      best_move_number_3[b][f]=10000;
  }
  solution_3_best.n_moves = final_position + 100;
  solution_3_recursion(0, 0, 0, final_position);
  solution_3_elapsed_time = cpu_time() - solution_3_elapsed_time;
}

static solution_t solution_4, solution_4_best;
static double solution_4_elapsed_time; // time it took to solve the problem
static unsigned long solution_4_count; // effort dispended solving the problem
static int best_move_number_4[_max_road_size_+1][_max_road_speed_+1];
static long counter_sol_4 = 0;
```

```c
static void solution_4_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_4_count++;
  solution_4.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_4_best.n_moves)
    {
      solution_4_best = solution_4;
      best_move_number_4[position][speed] = move_number;
      solution_4_best.n_moves = move_number;
    }
    return;
  }
  if (move_number == best_move_number_4[position][speed]){
    counter_sol_4 +=1;
  }
  if (move_number >= best_move_number_4[position][speed]){
    return;
  }
  best_move_number_4[position][speed]=move_number;
  for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      {
        if (i > new_speed)
          { solution_4_recursion(move_number + 1, position + new_speed, new_speed, final_position);
          }
      }
    }
}
```

```c
static void solve_4(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_4: bad final_position\n");
    exit(1);
  }
  solution_4_elapsed_time = cpu_time();
  solution_4_count = 0ul;
  for (int b = 0 ;b<=_max_road_size_;b++)
  {
    for (int f=0; f<=_max_road_speed_;f++)
      best_move_number_4[b][f]=10000;
  }
  solution_4_best.n_moves = final_position + 100;
  solution_4_recursion(0, 0, 0, final_position);
  solution_4_elapsed_time = cpu_time() - solution_4_elapsed_time;
}

static solution_t solution_5, solution_5_best;
static double solution_5_elapsed_time; // time it took to solve the problem
static unsigned long solution_5_count; // effort dispended solving the problem
static int best_move_number_5[_max_road_size_+1][_max_road_speed_+1];
static long counter_sol_5 = 0;


```

```c
static void solution_5_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_5_count++;
  solution_5.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_5_best.n_moves)
    {
      solution_5_best = solution_5;
      best_move_number_5[position][speed] = move_number;
      solution_5_best.n_moves = move_number;
    }
    return;
  }
  if (move_number == best_move_number_5[position][speed]){
    counter_sol_5 +=1;
  }
  if (move_number >= best_move_number_5[position][speed]){
    return;
  }
  best_move_number_5[position][speed]=move_number;
  for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position)
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      {
        if (i > new_speed)
          {
              if (best_move_number_5[position+new_speed][new_speed] == 10000){
                solution_5_recursion(move_number + 1, position + new_speed, new_speed, final_position);
              }
          }
      }
    }
}
```

```c
static void solve_5(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_4: bad final_position\n");
    exit(1);
  }
  solution_5_elapsed_time = cpu_time();
  solution_5_count = 0ul;
  for (int b = 0 ;b<=_max_road_size_;b++)
  {
    for (int f=0; f<=_max_road_speed_;f++)
      best_move_number_5[b][f]=10000;
  }
  solution_5_best.n_moves = final_position + 100;
  solution_5_recursion(0, 0, 0, final_position);
  solution_5_elapsed_time = cpu_time() - solution_5_elapsed_time;
}

static solution_t solution_6, solution_6_best;
static double solution_6_elapsed_time; // time it took to solve the problem
static unsigned long solution_6_count; // effort dispended solving the problem
static int best_move_number_6[_max_road_size_+1][_max_road_speed_+1];
static long counter_sol_6 = 0;


```

```
static void solution_6_recursion(int move_number, int position, int speed, int final_position)
{
  int i, new_speed;
  solution_6_count++;
  solution_6.positions[move_number] = position;

  if (position == final_position && speed == 1)
  {
    if (move_number < solution_6_best.n_moves)
    {
      solution_6_best = solution_6;
      best_move_number_6[position][speed] = move_number;
      solution_6_best.n_moves = move_number;
    }
    return;
  }
  best_move_number_6[position][speed]=move_number;
  for (new_speed = speed + 1; new_speed >= speed - 1; new_speed--)
    if (new_speed >= 1 && new_speed <= _max_road_speed_ && position + new_speed <= final_position )
    {
      for (i = 0; i <= new_speed && new_speed <= max_road_speed[position + i]; i++)
        ;
      {
        if (best_move_number_6[position+new_speed][new_speed] <= move_number+1 && position+new_speed != final_position)
        {
          return;
        }
        else if (i > new_speed)
          {
            solution_6_recursion(move_number + 1, position + new_speed, new_speed, final_position);
          }

      }

    }
}

static void solve_6(int final_position)
{
  if (final_position < 1 || final_position > _max_road_size_)
  {
    fprintf(stderr, "solve_4: bad final_position\n");
    exit(1);
  }
  solution_6_elapsed_time = cpu_time();
  solution_6_count = 0ul;
  for (int b = 0 ;b<=_max_road_size_;b++)
  {
    for (int f=0; f<=_max_road_speed_;f++)
      best_move_number_6[b][f]=10000;
  }
  solution_6_best.n_moves = final_position + 100;
  solution_6_recursion(0, 0, 0, final_position);
  solution_6_elapsed_time = cpu_time() - solution_6_elapsed_time;
}

//
// example of the slides
//
static void example(void)
{
  int i, final_position;

  srandom(0xAED2022);
  init_road_speeds();
  final_position = 30;
  solve_3(final_position);
  make_custom_pdf_file("example.pdf", final_position, &max_road_speed[0], solution_3_best.n_moves, &solution_3_best.positions[0], solution_3_elapsed_time, solution_3_count, "Plain recursion");
  printf("mad road speeds:");
  for (i = 0; i <= final_position; i++)
    printf(" %d", max_road_speed[i]);
  printf("\n");
  printf("positions:");
  for (i = 0; i <= solution_3_best.n_moves; i++)
    printf(" %d", solution_3_best.positions[i]);
  printf("\n");
}

//
// main program
//

int main(int argc, char *argv[argc + 1])
{
#define _time_limit_ 3600.0
  int n_mec, final_position, print_this_one;
  char file_name[64];

  // generate the example data
  if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'e' && argv[1][2] == 'x')
  {
    example();
    return 0;
  }
  // initialization
  n_mec = (argc < 2) ? 0xAED2022 : atoi(argv[1]);
  srandom((unsigned int)n_mec);
  init_road_speeds();
  // run all solution methods for all interesting sizes of the problem
  final_position = 1;
  solution_1_elapsed_time = 0.0;
  solution_2_elapsed_time = 0.0;
  solution_3_elapsed_time = 0.0;
  solution_4_elapsed_time = 0.0;
  solution_5_elapsed_time = 0.0;
  solution_6_elapsed_time = 0.0;
  printf("    +--- ---------------- --------- +\n");
  printf("    |                plain recursion |\n");
  printf("+--- +--- ---------------- --------- +\n");
  printf("|  n | sol            count  cpu time |\n");
  printf("+--- +--- ---------------- --------- +\n");
  while (final_position <= _max_road_size_ /* && final_position <= 20*/)
  {
    print_this_one = (final_position == 10 || final_position == 20 || final_position == 50 || final_position == 100 || final_position == 200 || final_position == 400 || final_position == 800) ? 1 : 0;
    printf("%3d |", final_position);

```

```
  // first solution method (very bad)
#ifdef sol_1
    if (solution_1_elapsed_time < _time_limit_)
    {
      solve_1(final_position);
      if (print_this_one != 0)
      {
        sprintf(file_name, "%03d_1.pdf", final_position);
        make_custom_pdf_file(file_name, final_position, &max_road_speed[0], solution_1_best.n_moves, &solution_1_best.positions[0], solution_1_elapsed_time, solution_1_count, "Plain recursion");
      }
      printf(" %3d %16lu %9.3e |", solution_1_best.n_moves, solution_1_count, solution_1_elapsed_time);
    }
    else
    {
      solution_1_best.n_moves = -1;
      printf("                            |");
    }
#endif

  // second solution method (little better than the first one))
#ifdef sol_2
    if (solution_2_elapsed_time < _time_limit_)
    {
      solve_2(final_position);
      if (print_this_one != 0)
      {
        sprintf(file_name, "%03d_2.pdf", final_position);
        make_custom_pdf_file(file_name, final_position, &max_road_speed[0], solution_2_best.n_moves, &solution_2_best.positions[0], solution_2_elapsed_time, solution_2_count, "Plain recursion");
      }
      printf(" %3d %16lu %9.3e |", solution_2_best.n_moves, solution_2_count, solution_2_elapsed_time);
    }
    else
    {
      solution_2_best.n_moves = -1;
      printf("                            |");
    }
#endif
```

```
  // third solution method (little worst than second one)
#ifdef sol_3
    if (solution_3_elapsed_time < _time_limit_)
    {
      solve_3(final_position);
      if (print_this_one != 0)
      {
        sprintf(file_name, "%03d_3.pdf", final_position);
        make_custom_pdf_file(file_name, final_position, &max_road_speed[0], solution_3_best.n_moves, &solution_3_best.positions[0], solution_3_elapsed_time, solution_3_count, "Plain v3 recursion");
      }
      printf(" %3d %16lu %9.3e %81d |", solution_3_best.n_moves, solution_3_count, solution_3_elapsed_time,counter_sol_3);
    }
    else
    {
      solution_3_best.n_moves = -1;
      printf("                            |");
    }
#endif
#ifdef sol_4
    if (solution_4_elapsed_time < _time_limit_)
    {
      solve_4(final_position);
      if (print_this_one != 0)
      {
        sprintf(file_name, "%03d_4.pdf", final_position);
        make_custom_pdf_file(file_name, final_position, &max_road_speed[0], solution_4_best.n_moves, &solution_4_best.positions[0], solution_4_elapsed_time, solution_4_count, "Plain v3 recursion");
      }
      printf(" %3d %16lu %9.3e %51d |", solution_4_best.n_moves, solution_4_count, solution_4_elapsed_time,counter_sol_4);
    }
    else
    {
      solution_4_best.n_moves = -1;
      printf("                            |");
    }
#endif
```

```
#ifdef sol_5
    if (solution_5_elapsed_time < _time_limit_)
    {
      solve_5(final_position);
      if (print_this_one != 0)
      {
        sprintf(file_name, "%03d_5.pdf", final_position);
        make_custom_pdf_file(file_name, final_position, &max_road_speed[0], solution_5_best.n_moves, &solution_5_best.positions[0], solution_5_elapsed_time, solution_5_count, "Plain v4 recursion");
      }
      printf(" %3d %16lu %9.3e |", solution_5_best.n_moves, solution_5_count, solution_5_elapsed_time);
    }
    else
    {
      solution_5_best.n_moves = -1;
      printf("                            |");
    }
#endif
#ifdef sol_6
    if (solution_6_elapsed_time < _time_limit_)
    {
      solve_6(final_position);
      if (print_this_one != 0)
      {
        sprintf(file_name, "%03d_6.pdf", final_position);
        make_custom_pdf_file(file_name, final_position, &max_road_speed[0], solution_6_best.n_moves, &solution_6_best.positions[0], solution_6_elapsed_time, solution_6_count, "Plain v4 recursion");
      }
      printf(" %3d %16lu %9.3e |", solution_6_best.n_moves, solution_6_count, solution_6_elapsed_time);
    }
    else
    {
      solution_6_best.n_moves = -1;
      printf("                            |");
    }
#endif
    // done
    printf("\n");
    fflush(stdout);
    // new final_position
    if (final_position < 50)
      final_position += 1;
    else if (final_position < 100)
      final_position += 5;
    else if (final_position < 200)
      final_position += 10;
    else
      final_position += 20;
  }
  printf("--- + --- ---------------- --------- + --- ---------------- --------- +\n");
  return 0;
#undef _time_limit_
}
```

Código em MatLab

```
load run1
load run2
load run3
load run4
load run5
load run6

% run time
r1_time = run1(:,4);
r1_pos = run1(:,1);
lt = log10(r1_time);
X=[r1_pos,0*r1_pos+1];
w = pinv(X)*r1_time;
w(1)*800+w(2)
A=X*w;
subplot("Position",[0.10 0.55 0.36 0.30])
plot(r1_pos,lt,"r",r1_pos,A,"g")
xlabel("Position ")
ylabel("Time (s)")
title("Brute Force")
legend("Position/Time (log10)","Position/Time (linear)")

r2_time = run2(:,4);
r2_pos = run2(:,1);
lt = log10(r2_time);
X=[r2_pos,0*r2_pos+1];
w = pinv(X)*r2_time;
B=X*w;
w(1)*800+w(2)
subplot("Position",[0.53 0.55 0.36 0.30])
plot(r2_pos,lt,"r",r2_pos,B,"g")
xlabel("Position")
ylabel("Time (s)")
title("Branch and Bound (incomplete)")
legend("Position/Time (log10)","Position/Time (linear)")

r3_time = run3(:,4);
r3_pos = run3(:,1);
lt = log10(r3_time);
X=[r3_pos,0*r3_pos+1];
w = pinv(X)*lt;
C=X*w;
subplot("Position",[0.10 0.07 0.36 0.30])
plot(r3_pos,lt,"r",r3_pos,C,"g")
xlabel("Position")
ylabel("Time (s)")
title("Branch and Bound (complete -> brake first)")
legend("Position/Time (log10)","Position/Time (linear)")


r4_time = run4(:,4);
r4_pos = run4(:,1);
lt = log10(r4_time);
X=[r4_pos,0*r4_pos+1];
w = pinv(X)*lt;
D=X*w;
subplot("Position",[0.53 0.07 0.36 0.30])
plot(r4_pos,lt,"r",r4_pos,D,"g")
xlabel("Position")
ylabel("Time (s)")
```

```
61  title("Branch and Bound (complete -> accelerate first (v1))")
62  legend("Position/Time (log10)","Position/Time (linear)")
63
64
65  r5_time = run5(:,4);
66  r5_pos = run5(:,1);
67  lt = log10(r5_time);
68  X=[r5_pos,0*r5_pos+1];
69  w = pinv(X)*lt;
70  E=X*w;
71  figure (2)
72  subplot("Position",[0.10 0.55 0.36 0.30])
73  plot(r5_pos,lt,"r",r5_pos,E,"g")
74  xlabel("Position")
75  ylabel("Time (s)")
76  title("Branch and Bound(complete -> accelerate first (v2))")
77  legend("Position/Time (log10)","Position/Time (linear)")
78
79
80  r6_time = run6(:,4);
81  r6_pos = run6(:,1);
82  lt = log10(r6_time);
83  X=[r6_pos,0*r6_pos+1];
84  w = pinv(X)*lt;
85  F=X*w;
86  subplot("Position",[0.53 0.55 0.36 0.30])
87  plot(r6_pos,lt,"r",r6_pos,F,"g")
88  xlabel("Position")
89  ylabel("Time (s)")
90  title("Branch and Bound(complete -> accelerate first (v3))")
```

```
91  legend("Position/Time (log10)","Position/Time (linear)")
92
93  subplot("Position",[0.10 0.07 0.36 0.30])
94  plot(r1_pos,A,"blue",r2_pos,B,"red",r3_pos,C,"green",r4_pos,D,"cyan",r5_pos,E,"magenta",r6_pos,F,"black")
95  title("Position/Time (linear)")
96  xlabel("Position")
97  ylabel("Time (s)")
98  legend("BF","BaB(Inc.)","BaB (Complete)","BaB (v1)","BaB (v2)","BaB (v3)")
99  %%
100 % move number
101 load run1
102 load run2
103 load run3
104 load run4
105 load run5
106 load run6
107
108 r1_nmoves = run1(:,2);
109 r1_pos = run1(:,1);
110 X=[r1_pos,0*r1_pos+1];
111 w = pinv(X)*r1_nmoves;
112 w(1)*800+w(2)
113
114 figure(1)
115 subplot("Position",[0.10 0.55 0.36 0.30])
116 plot(r1_pos,r1_nmoves)
117 xlabel("Position")
118 ylabel("Nº Moves")
119 title("Brute Force")
120
```

```
r2_nmoves = run2(:,2);
r2_pos = run2(:,1);
X=[r2_pos,0*r2_pos+1];
w = pinv(X)*r2_nmoves;
w(1)*800+w(2)

subplot("Position",[0.53 0.55 0.36 0.30])
plot(r2_pos,r2_nmoves)
xlabel("Position")
ylabel("Nº Moves")
title("Branch and Bound (incomplete)")
r3_nmoves = run3(:,2);
r3_pos = run3(:,1);

subplot("Position",[0.10 0.07 0.36 0.30])
plot(r3_pos,r3_nmoves)
xlabel("Position")
ylabel("Nº Moves")
title("Branch and Bound (complete -> brake first)")


r4_nmoves=run4(:,2);
r4_pos=run4(:,1);


subplot("Position",[0.53 0.07 0.36 0.30])
plot(r4_pos,r4_nmoves)
xlabel("Position")
ylabel("Nº Moves")
title("Branch and Bound (complete -> accelerate first (v1))")
```

```

r5_nmoves=run5(:,2);
r5_pos=run5(:,1);

figure (2)
subplot("Position",[0.10 0.55 0.36 0.30])
plot(r5_pos,r5_nmoves)
xlabel("Position")
ylabel("Nº Moves")
title("Branch and Bound (complete -> accelerate first (v2))")



r6_nmoves=run6(:,2);
r6_pos=run6(:,1);

subplot("Position",[0.53 0.55 0.36 0.30])
plot(r6_pos,r6_nmoves)
xlabel("Position")
ylabel("Nº Moves")
title("Branch and Bound (complete -> accelerate first (v3))")


%%
% effort
load run1
load run2
load run3
load run4
load run5
```

```
load run6

r1_effort = run1(:,3);
r1_pos = run1(:,1);
lt = log10(r1_effort);
X=[r1_pos,0*r1_pos+1];
w = pinv(X)*r1_effort;
w(1)*800+w(2)


figure(1)
subplot("Position",[0.10 0.55 0.36 0.30])
plot(r1_pos,lt)
xlabel("Position")
ylabel("Effort")
title("Brute Force")

r2_effort = run2(:,3);
r2_pos = run2(:,1);
lt=log10(r2_effort);
X=[r2_pos,0*r2_pos+1];
w = pinv(X)*r2_effort;
w(1)*800+w(2)

subplot("Position",[0.53 0.55 0.36 0.30])
plot(r2_pos,lt)
xlabel("Position")
ylabel("Effort")
title("Branch and Bound (incomplete)")

```

```
r3_effort = run3(:,3);
r3_pos = run3(:,1);

subplot("Position",[0.10 0.07 0.36 0.30])
plot(r3_pos,r3_effort)
xlabel("Position")
ylabel("Effort")
title("Branch and Bound (complete -> brake first)")

r4_effort = run4(:,3);
r4_pos = run4(:,1);
subplot("Position",[0.53 0.07 0.36 0.30])
plot(r4_pos,r4_effort)
xlabel("Position")
ylabel("Effort")
title("Branch and Bound (complete -> accelerate first (v1))")


r5_effort = run5(:,3);
r5_pos = run5(:,1);
figure (2)
subplot("Position",[0.10 0.55 0.36 0.30])
plot(r5_pos,r5_effort)
xlabel("Position")
ylabel("Effort")
title("Branch and Bound (complete -> accelerate first (v2))")



r6_effort = run6(:,3);
```

```
r6_pos = run6(:,1);
subplot("Position",[0.53 0.55 0.36 0.30])
plot(r6_pos,r6_effort)
xlabel("Position")
ylabel("Effort")
title("Branch and Bound (complete -> accelerate first (v3))")

%%
% maxspeed
load maxspeed1

maxspeed=maxspeed1(:,2);
position=maxspeed1(:,1);
accumulatedspeed=maxspeed1(:,3);
bestnmoves=maxspeed1(:,4);

figure(1)
subplot("Position",[0.10 0.55 0.36 0.30])
plot(position,maxspeed)
xlabel("Position")
ylabel("Max Speed")
title("Position / Max Speed")

subplot("Position",[0.53 0.55 0.36 0.30])

plot(position,accumulatedspeed,"green")
xlabel("Position")
ylabel("Accumulated Speed")
title("Position / Accumulated Speed")
```

Link do ficheiro com o código em C e o código usado para fazer os gráficos em MatLab:

https://drive.google.com/file/d/1VduVaTrAJ7OXOWHCXFrzH6XOa58zKVzH/view?usp=share_link

Conclusão

Este problema foi resolvido com sucesso o que significa que o programa deu o resultado esperado com um tempo de execução bastante inferior à solução inicial disponibilizada pelo professor.

Este trabalho foi feito com base em muita pesquisa, mas a maior parte do conhecimento necessário foi adquirido nas aulas práticas e teóricas. Graças a este trabalho foi adquirido um maior conhecimento sobre a linguagem em questão, possibilitando o desenvolvimento de outros projetos no futuro.

Bibliografia

https://pt.wikipedia.org/wiki/Busca_por_for%C3%A7a_bruta

https://www.geeksforgeeks.org/branch-and-bound-algorithm/

https://www.geeksforgeeks.org/dynamic-programming/

https://stackoverflow.com/questions/842626/branch-and-bound