Relatório Trabalho Prático LI1

Grupo 159

Gonçalo Faria e Gonçalo Pereira

26 de Janeiro de 2018

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

No âmbito da cadeira de Laboratórios de Informática 1 (1º semestre do 1º ano de MIEI), o método de avaliação escolhido pelos professores foi a realização de um trabalho em grupos de 2, onde o objetivo era programar de raiz o famoso jogo "Micro Machines", usando a linguagem de programação Haskell.

1.2 Motivação

O uso num contexto prático dos conhecimentos adquiridos ao longo do nosso percurso académico sempre foi algo que nos fascinou. Compreender as consequências e benefícios do paradigma de programação funcional é algo abstrato e um desafio de compreender, no entanto com a criação de aplicações estes são mais evidentes e permitem-nos assim desenvolver uma intuição de quando este poderá ser aplicado, no futuro na nossa vida profissional.

1.3 Objectivos

Os objetivos estabelecidos pelo grupo desde o início foram 4:

1. Ter um trabalho final apelativo;
2. Ter um trabalho final "smooth" com o mínimo de bugs possíveis;
3. Ter o código da programação o mais simples possível, de modo o jogo correr o mais rapidamente possível;
4. Conseguir realizar as 2 fases do trabalho antes do dia 15 de Dezembro;

Capítulo 2

Análise de Requisitos

2.1 Fase 1

1ª Fase do trabalho encontra-se subdividida em 3 Tarefas:

1. A Tarefa 1 tem como objetivo contruir um mapa para o jogo.

   Onde o mesmo vai seguir um caminho previamente indicado, sendo esse caminho constituído por inúmeros passos, dos quais podemos ter:

   Avanca; **Sobe**; **Desce**; **CurvaEsq**; **CurvaDir**;

   O mapa em si, consiste numa matriz de peças. Todas estas peças têm uma determinada altura e podem ser do tipo: **lava**; **recta**; **rampa**; **curva**; Sendo que as peças do tipo recta e curva têm um caraterística acrescida, a **orientação**.

   O foco nesta Tarefa seria receber uma lista de peças e conseguir peça a peça criar o mapa.

2. A Tarefa 2 tem como objetivo verificar se um dado mapa é válido ou não, de acordo com um conjunto de regras inicialmente estabelecidas.

   REGRAS TAREFA 2:

   - Existe apenas um percurso e todas as peças fora desse percurso são do tipo lava;
   - O percurso deve corresponder a uma trajetória, tal que começando na peça de partida com a orientação inicial volta-se a chegar à peça de partida com a orientação inicial;
   - A orientação inicial tem que ser compatível com a peça de partida. Como é sugerido na imagem abaixo, considera-se que a orientação é compatível com a peça se for possível entrar na peça seguindo essa orientação (note que esta questão só é relevante para as peças do tipo curva e rampa;
   - As peças do percurso só podem estar ligadas a peças do percurso com alturas compatíveis;
   - Todas as peças do tipo lava estão à altura 0;
   - O mapa é sempre retangular e rodeado por lava, ou seja, a primeira e última linha, assim como a primeira e última coluna são constituídas por peças necessariamente do tipo lava.

3. O objetivo da Tarefa 3 é começar implementação da mecânica do jogo, mais propriamente as movimentações do carro e as suas consequentes interações com o mapa.

   Para tal começamos por criar e modelar um Carro, caraterizado por três pontos: **Posição**; **Direção**; **Velocidade**;

   De seguida tivemos que, dado um estado inicial do carro, calcular o seu estado final após um determinado período de tempo.

2.2 Fase 2

A 2ª Fase do trabalho encontra-se subdividida também ela em 3 Tarefas:

1. O objetivo da Tarefa 4 é a partir de ações efetuadas por um jogador num período de tempo atualizar o estado do jogo .

   Para isso é necessário representar o **Jogo** (o estado interno do jogo) que deverá ser atualizado em cada instante e a **Ação** algo que vai indicar, por exemplo, se o carro está a acelerar, travar, ou curvar.

   O foco desta tarefa é garantir que o estado do jogo atualiza a cada momento corretamente, tendo em conta todas as possibilidades.

2. O objetivo desta Tarefa ao contrário das outras não é específico para todos os grupos, trata-se de uma Tarefa livre.

   Na Tarefa 5 é pedido que cada grupo implemente todo o jogo usando a biblioteca Gloss.

   A ferramenta Gloss irá tratar de construir a parte gráfica do jogo. Onde cada grupo tem a liberdade de desenhar o jogo à sua maneira.

   O foco principal na Tarefa 5 é criar o máximo de elementos gráficos apelativos e criativos, sem nunca perder o foco do jogo, uma corrida de carros.

3. O objetivo da Tarefa 6 é implementar um bot que jogue o Micro Machines automaticamente.

   Um bot que percorra o percurso da maneira mais eficiente possível, tendo em conta o que o rodeia. A estratégia de jogo a implementar fica ao critério de cada grupo, sendo que a avaliação automática será efetuada colocando o bot implementado a combater com diferentes bots de variados graus de "inteligência".

Capítulo 3

A Nossa Solução

Fase 1

3.1 Tarefa 1

Foco Principal:
Embora os docentes estejam a incutir nos alunos que esta tarefa deve ser resolvida usando listas de listas existem alternativas bem mais eficientes e acima de tudo mais rápidas e fáceis de a programar.

Desenvolvimento:
Inspiração
Embora na linguagem C existam matrizes, estas não são implementadas pelo compilador nesse formato. De forma a obter maior eficiência o compilador memoriza apenas o endereço do primeiro elemento e a dimensão desejada da matriz final. Isto é genial, pois, assim conseguem-se obter os benefícios adjacentes à manipulação de arrays com a simplicidade de gestão de dados das matrizes.

Haskell
Embora, Haskell não apoie arrays (com o Prelude) podemos usar este método para evitar o uso de listas de listas no processamento dos exercícios pois não só seria custoso no domínio da análise de complexidade do programa como cansativo para o programador.

A solução? Criar um tipo de Dados chamado 'Bidimentional list' que será composto por um tuplo de inteiros, designado dimensão e também uma lista.

```
data Bidimlist = Bid Dimensao [Peca]
```

A lista terá sempre, sendo a sua dimensão (x,y), x y elementos. Ou seja embora não estejamos a usar uma lista de listas a informação desta será guardada na íntegra (x y é o numero de elementos de uma matriz x por y).

A posição (i,j) da lista de listas será indexada no elemento (j x) + i .

Com estes elementos básico apresentados é agora possível estimar qual será então a solução proposta à tarefa .

Iniciar-se-á com a obtenção das dimensões iniciais do tabuleiro e do ponto de partida do caminho, usando as funções auxiliares dos docentes. A orientação inicial será, por convenção **Este** e a **Altura 0**.

Tendo como argumentos a ORIENTAÇÃO o PASSO e a ALTURA . Podemos obter qual é a peça que permitirá satisfazer estas condições.

Concorrentemente é também possível saber qual será a próxima posição. Pois é suficiente a orientação e a posição inicial . Todo este processo será recursivamente repetido para os restantes passos do caminho obtendo as sucessivas peças e posições.

Processamento

Tendo obtido apenas uma lista de Peças, à priori, muito inferior ao número de elementos do tabuleiro.

Como poderemos satisfazer os requisitos desta Tarefa?

Preencher a lista bidimensional

Com a fórmula mostrada anteriormente temos um método ideal de traduzir as coordenadas das posições em índices na lista bidimensional.

Os restantes índices, que um leitor mais precipitado rapidamente denunciaria, serão trivialmente preenchido com a peça lava.

Detalhes de implementação

De forma a diminuir o tempo de execução significativamente antes de preencher a lista bidimensional peças com a respetiva posição serão agregadas numa lista do duplo (Int , Peça) onde o inteiro é já a tradução das coordenadas da posição. Seguidamente, ocorrerá a ordenação desta lista de duplos em função desses Inteiros.

Ordenação

Foi usado o célebre algoritmo Quick Sort. Escolhido trivialmente pelo facto de já o ter implementado em haskell ao fazer um dos exercícios das 50 questões.

Não se encontra devidamente implementado pois o pivot será sempre o primeiro elemento da lista de cada recursão deste algoritmo.

Chegamos a esta conclusão pois existe a possibilidade do primeiro elemento ser ou o maior elemento da lista ou o menor (aproximadamente uma probabilidade de **(1/2) elevado a (n-2)** disto acontecer para uma lista de tamanho n), elementos estes que não assegurarão que a lista será devidamente separada (dividida a meio).

Ponto Final:

Após obtida a lista bidimensional ter-se-á unicamente de a converter em lista de lista nas dimensões em que foi originalmente declarada.

Isto faz-se segmentando a lista em y sub listas de tamanho x.

3.2 Tarefa 2

Foco Principal:

No todo que é o código da Tarefa 2, a função principal, 'valida', recebe um Mapa ao qual vai atribuir um valor Bool que será "True" caso o mapa seja válido ou "False" quando é inválido.

```
valida :: Mapa -> Bool
```

Desenvolvimento:

De maneira a determinar a veracidade do Mapa, tivemos que inicialmente estudar quais os casos que tornariam um mapa inválido, sendo eles:

- **Mapa Rodeado por lava:**

  Para o mapa ser válido o mesmo tem de ser rodeado por lava em todos os seus cantos.

  Sendo assim tivemos de criar uma função que verifica-se se a primeira e última coluna e a primeira e última linha eram constituídas necessariamente por uma peça do tipo lava.

- **Mapa Retagular:**

  O mapa é válido se e só se for retangular. Para confirmarmos se um mapa é retangular ou não, criámos uma função que recebe um Bidimlist e que lhe atribui um valor Bool.

  Será True toda a vez que o produto de x e y da Bidimlist for igual ao comprimento da lista da mesma.

- **Sequência lógica das Peças:**

  Como é óbvio o fator mais importante e mais detalhado da Tarefa 2 é a interação de um determinada peça com as suas peças adjacentes.

  Para deduzir as consequências de cada interação tivemos de em cada caso particular indicar a sua respetiva consequência.

  Temos este exemplo simples, em que as duas peças adjacentes são **Peca Recta 0** e **Peca Recta 1**, apesar de serem ambas peças reta como a altura de ambas é diferente o carro nunca poderá passar da primeira peça para a segunda, respetivamente.

  Identificada cada uma das possíveis interações, o passo seguinte foi criar uma função que fosse ler as peças da lista que constitui o mapa, peça a peça, e verificar se em cada caso a relação e interação entre elas era válida.

- **Início e fim do Mapa:**

  Um detalhe muito importante, também a ter em conta é o facto de o mapa ter de iniciar e acabar na mesma peça. Existindo um único caminho possível a percorrer pelo carro a fim de completar o mapa, cruzando a meta.

Ponto Final:

Com todos estes casos considerados e com o código bem estruturado e válido, a nossa função 'valida' será capaz de receber qualquer tipo de Mapa, lê-lo e no final atribuir-lhe um valor de válido ou inválido.

3.3 Tarefa 3

Foco Principal:

O objetivo principal na Tarefa 3 é implementar a função movimenta. Que irá devolver um valor Maybe carro, onde o maybe depende se o carro é destruído ou não.

```
movimenta :: Tabuleiro -> Tempo -> Carro -> Maybe Carro
```

Desenvolvimento:

Esta tarefa foi resolvida tentando dividir o problema de movimentar o carro por várias peças, em calcular o movimento do carro em múltiplas etapas.

A função que criámos para processar cada um dos movimentos foi a **Branch**.

```
branch :: Bidimlist -> Carro -> Tempo -> Posicao -> Maybe Carro
```

Isto acontece, pois, nós consideramos que a melhor solução seria aquela em que se dividia o problema de movimentar o carro por 3 pecas por partes.

Designámos por limite a extremidade de uma peça qualquer, ou seja, as paredes laterais, superiores e até mesmo a parede diagonal no caso das curvas.

Um dos nossos problemas nesta tarefa foi determinar o tempo que o carro demorava a atingir um limite. Portanto criámos a função **intersecT** que se limita a encontrar o tempo de interseção do carro com uma das quatro arestas.

```
intersecT :: Velocidade -> Ponto -> (Int,Int) -> Tempo
```

Uma situação específica é a qual em que o carro irá colidir com uma das duas possíveis diagonais de uma peça Curva.

Para calcularmos o tempo que o carro demora até interagir com uma dessas diagonais criámos a função **insecT**.

```
insecT :: Velocidade -> Ponto -> (Int,Int) -> Tempo
```

Um ponto importantíssimo é o facto da peça em que o carro se encontra ser fundamental, pois determina as diferentes funções que são executadas, visto que diferentes peças têm diferentes limites.

Após executada a função que irá processar movimentos na peça específica, vamos começar por ponderar a interseção do carro com todos os limites dessa peça e determinar se algum desses é possível.

Caso não seja possível a interseção do carro com um limite então esse carro vai movimentar-se livremente durante o instante indicado pois sabemos que não ocorreram adversidades. Caso contrário é calculado o movimento até esse limite e processada uma reação.

Os processamentos da reação no caso de colisão têm em conta que o ângulo refletivo é igual ao ângulo incidente e por isso procede à atualização da velocidade.

Uma função muito importante é a **collision**, que visa calcular o vetor velocidade após uma colisão.

```
collision :: Velocidade -> (Int,Int) -> Velocidade
```

Na eventualidade desse limite levar a uma outra peça onde é permissível a movimentação do carro, então o processo é repetido recursivamente até que o carro se destrua ou o tempo acabe.

Ponto Final:

Temos então uma função que é capaz de calcular qualquer interceção entre o carro e o mapa. Tendo em conta que o estado final do carro após o movimento também é estudado, podendo ele acabar inteirou ou destruído.

Fase 2

3.4 Tarefa 4

Foco Principal:

Criar uma função que tendo o estado atual do jogo, ao receber uma ação e um determinado período de tempo consiga atualizar o estado do jogo.

Desenvolvimento:

A função principal desta tarefa é a **atualiza**, usada para atualizar o estado do jogo dadas as ações de um jogador num determinado período de tempo.

```
atualiza :: Tempo -> Jogo -> Int -> Acao  -> Jogo
```

Para proceder à atualização da velocidade temos que usar a formula genérica **V = t*a + Vo**

Tivemos então que identificar a aceleração a que o carro se encontrava exposto e para isso tivemos de separadamente calcular os diferentes vetores aceleração.

Temos então o atrito lateral dos pneus, o atrito normal, a aceleração gravítica e a aceleração do motor.

Para calcular estas acelerações usámos as funções: **acgravitica**; **acPneus**; **acAcel**;

Para tal criámos a **acAtrito** que executa as computações correspondentes ao cálculo da aceleração do atrito no carro.

```
acAtrito :: Double -> Velocidade -> (Double , Double)
```

Seguidamente somámos todos esses vetores da aceleração obtendo assim a aceleração total do carro.

Sucedeu-se então à multiplicação do vetor aceleração total pelo tempo a que o carro estaria exposto à mesma. Por fim somámos à velocidade inicial do carro.

Uma das funções mais fundamentais desta Tarefa é a **newvelocity**, pois é nela que a grande maioria das velocidades são calculadas.

```
new_velocity :: Acao-> Propriedades ->Carro -> Tempo
->Maybe Orientacao -> Angulo ->Velocidade
```

Um ponto muito importante foi criar uma função que atualizasse o tempo de uso de nitro por parte de cada jogador. Criámos então a função **updateNitro**, que retribui o Jogo.

```
updateNitro :: Acao -> Tempo -> Jogo -> Int ->Jogo
```

Finalmente deu-se a atualização do histórico, onde verificamos se a posição atual do carro é diferente da posição à cabeça do histórico, caso seja verdade esta nova posição é adicionada à lista, caso não seja o histórico não ó alterado.

Para tal criámos a função **updateHist**.

```
updateHist :: Jogo -> Int -> Jogo
```

Ponto Final:

Para que seja possível que as mecânicas do jogo sejam de alguma forma realistas é crucial que esta componente da aplicação esteja extremamente bem definida e por essa razão nós dedicamos bastante tempo a obter as fórmulas das diferentes acelerações.

3.5 Tarefa 5

Foco Principal:

Criar um jogo visualmente apelativo, simples de manobrar e o mais completo e divertido possível.

Desenvolvimento:

Inicialmente antes de nos focarmos em tratar de criar imagens apelativas, colocámos como nossa prioridade tratar de escrever o código na sua totalidade

usando a ferramenta **"Gloss"** de modo a termos em primeira mão o nosso jogo a funcionar e a aparecer perante a nossa tela.

Como qualquer jogo teríamos de criar pelos menos 4 scenes,**Introdução do Jogo**, **Menu Inicial**, **Desenvolvimento do Jogo**, **Final do Jogo**. Acabámos no entanto por criar 5, acrescentando a possibilidade de poder escolher em que mapa pretende jogar.

Toda a Interface do jogo será controlada usando as **setas do Keyboard** e a tecla **"Enter"**.

No menu Inicial, temos o logotipo do jogo e de seguida surge um botão **"Start"** que permitirá dar início ao jogo.

Passamos então para a janela que permitirá ao jogador escolher entre 2 mapas. Escolhemos dois mapas, sendo um básico e um bem mais avançado.

Chegamos então à quarta tela, a mais detalhada e mais importante, o decorrer do jogo.

Nela podemos identificar: O **mapa previamente escolhido**, como plano de fundo; As **4 naves espaciais**, que irão competir entre si; **4 capacetes** ao qual corresponde uma **botija de Nitro**. É de referir que **qualquer capacete ficará cinzento caso a respetiva nave recebe uma penalização**. Já **a botija ficará cinzenta caso o respetivo veículo gaste todo o seu nitro**.

Para além de todos estas funcionalidades introduzimos também a **animação do nitro** que ocorre logicamente quando o nitro é ativado e também uma mecânica que dá a ilusão ao jogador que o seu carro está a ser seguido com uma câmara, pois colocamos o carro deste ampliado no centro do ecrã.

Por fim quando uma das 4 naves conclui a pista, a quarta tela surge.

A mesma encontra-se divida em duas possibilidades, onde a escolha de qual delas surge depende do resultado do jogador.

Se o mesmo ganhou a corrida irá aparecer **"You Win"** na tela, caso contrário irá aparecer **"You Lose"**.

No final da corrida o processo repete-se.

Ponto Final:

Com todos os nossos objetivos bem executados, logicamente o nosso jogo acabaria por ser apelativo e garantir ao jogador/jogadores um bom entretenimento. Tivemos, no entanto, a desilusão de não conseguir implementar o modo "Multiplayer",

Figura 3.1: 1ª Tela

Figura 3.2: 2ª Tela

Figura 3.3: 3ªTela

Figura 3.4: Possível 4ª Tela

Figura 3.5: Possível 5ªTela

3.6 Tarefa 6

Foco Principal:

O Foco principal desta Tarefa foi criar um bot que conseguisse ler cada situação específica em que o carro se encontrasse em qualquer ponto de um determinado mapa, de modo a rapidamente e facilmente deduzir qual a melhor ação que deve tomar de modo a percorrer o mapa o mais rapidamente possível.

Desenvolvimento:

A ação dada pela função **bot** serão o resultado da mistura de 2 mecanismos diferentes para o cálculo de decisões.

Estes mecanismos podem ser vistos como bots independentes, pois cada um irá produzir para qualquer estado de jogo uma ação, que podia por si só ser a solução a este problema.

No entanto, vários testes intuíram que a junção dos dois bots aumenta significativamente a performance do que ambos individualmente.

O primeiro bot designamos por **conservador**, pois ele foi criado pela necessidade de cumprir as diferentes regras criadas pelos professores (Não passar peças à frente, por exemplo).

No entanto este bot é regra geral extremamente lento, pois o seu objetivo é apenas percorrer a peça em que se encontra.

O segundo bot a que designamos **guloso**, usando sensores, tenta determinar qual é a direção que o permitirá evitar limites de forma a conseguir maximizar a velocidade.

O grande problema deste bot, e a principal razão pela qual não foi selecionado como única solução, foi que muitas vezes 'cortava-caminho' (não percorria todas as peças).

Isto não era problema para mapas mais restritos, no entanto para mapas mais complexos o bot chegava a evitar secções inteiras.

Esta ultima parte só foi problemática porque os professores nos torneios entre os diferentes alunos não permitiam que isto acontece-se.

As heurísticas para determinar qual dos dois usar foram bastante simples.

Era inicialmente determinando qual extremidade da peça de onde o carro

teria de sair.

Se o bot **guloso** obtive-se uma ação nessa direção essa seria a ação executada, senão seria a do bot **conservador**, que embora ineficiente consegue sempre percorrer o mapa.

Dado que estas ações são calculadas para frações de segundo, regra geral , uma pequena ação do bot **conservador** , é suficiente para catalisar uma sequência de decisões ideais ao bot **guloso**.

Ponto Final:

Embora as capacidades do bot não tenham alcançado as nossas expectativas este é na mesma um dos bots melhor colocados nos diferentes torneios que decorreram na plataforma de feedback e durante os testes internos que nós realizamo. Nenhum dos participantes conseguiu superar o desempenho deste. Embora métodos alternativos tenham sido testados os resultados desta implementação esclareceram qualquer tipo de dúvida quanto à estratégia a adotar.

Capítulo 4

Validação da Solução

Para cada uma das Tarefas os professores disponibilizaram um sistema online de feedback capaz de permitir a cada grupo, verificar a validade dos tipos requisitados nas diferentes tarefas, a partir do seu código.

O feedback referido encontra-se disponibilizado no site **i1.lsd.di.uminho.pf** .

Portanto para cada tarefa **exceto a 5 e a 6**, usufruímos desse feedback para determinar se o nosso código estava correto e se abrangia as enumeras possibilidades de combinações de cada Tarefa.

O nosso procedimento para a validação era bastante simples:

1. Testar se o código no total era corrido pelo **ghci** sem qualquer erro;
2. Elaborar diferentes Testes o máximo e mais diversificados uns dos outros possível;
3. Estudar cada um desses teste manualmente e anotar qual o resultado esperado no feedback;
4. Analisar o feedback online e comparar com os nossos resultados previstos manualmente;
5. Em caso de diferença, rever o código e alterá-lo de modo a levar o feedback a coincidir com os resultados estudados manualmente. (Que terão de estar 100 por cento corretos)

No caso da **Tarefa 5**, dado o facto de ser uma Tarefa aberta não há requisitos específicos como nas restantes tarefas.

Neste caso cada grupo tem a liberdade de criar a parte gráfica do jogo como bem entender, com os extras que pretender. Tendo em

conta que o aspeto gráfico tem de estar bem contruído e projetado, de modo a correr o jogo com os básicos necessários.

Na **Tarefa 6** como já tínhamos o aspeto gráfico do jogo podemos mais facilmente encontrar erros no funcionamento do bot e visto que na plataforma de feedback decorreram torneios, podemos assim facilmente identificar as nossas falhas e alterar as nossas estratégias.

Testes mais problemáticos:

- **Tarefa 2:**

  O mais problemática de verificar para nós, foi ver se o mapa tinha um ou mais possiveis caminhos que permitiriam ao carro percorrer a pista.

  O mapa é correto se tiver um e um só possível caminho possível por parte do carro de modo a completar a pista. Para tal, recorremos a um método em que assumimos a posição inicial e vamos percorrendo o mapa pelo caminho lógico até o completar. No final assumimos que quaisquer peças diferentes das que percorremos terão que ser especificamente lava.

- **Tarefa 3:**

  Situação específica em que nos deparamos com a seguinte sequência de peças: **Reta, Rampa, Rampa.**

  Onde as alturas suas alturas são respetivamente **x,x,(x+1).** As rampas estão dispotas com direção perpendicular à reta, tendo as mesmas sentidos opostos.

  A questão que torna esta situação problemática é o facto de o carro em certos casos conseguir percorrer as 3 peças e noutros casos chocar com uma parede, não o permitindo circular.

  O fator que irá determinar se ele passa ou não de uma peça para a outra é o desnivel entre o ponto de transição que o carro irá tomar entre a primeira e a segunda rampa.

  Sendo que passa apenas se o desnível for inferior a 1. Criámos então a função **'unTrapable'** que irá determinar o ponto onde a diferença entre rampas é 1.

  Sabendo as orientações da Rampa, saberemos portanto para que lado (esquerdo ou direito) desse ponto o carro será capaz de circular.

```
unTrapable :: Tempo -> Bidimlist -> Tempo -> Carro
-> (Int,Int) -> Maybe Carro
```

- **Tarefa 6:**

  Em alguns mapas o bot atingia velocidades demasiado altas o que fazia o carro colidir ou evitar peças. Para evitar foi necessário implementar

um mecanismo de travagem que caso os sensores detetem que o carro vai colidir em menos de um segundo, este desativa o nitro caso este esteja ativado e começava a travar.

Capítulo 5

Conclusão

Este projeto levou-nos a compreender que deliberar as diferentes formas de resolver um problema deve compreender a maioria do tempo na realização de um projeto, mas também que para conseguir terminar um projeto a tempo é crucial compromissos quer na qualidade do projeto, quer na quantidade de conteúdo.