Universidade do Minho

Relatório de Projeto

DSL para Árvores de Comportamento (Behavior Trees)

Autores:
Miguel Oliveira
Pedro Mimoso Silva
Pedro Moura

Supervisores:
Pedro Rangel Henriques
José João Almeida

Projeto realizado no âmbito da UC de Laboratórios de Engenharia Informática, pertencente ao Mestrado em Engenharia Informática

do

Departamento de Informática da Universidade do Minho

5 de Maio de 2020

UNIVERSIDADE DO MINHO

Resumo

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

BT	**B**ehavior **T**ree
DSL	**D**omain **S**pecific **L**anguage
NPC	**N**on-**P**layable **C**haracter

Capítulo 1

Introdução

O presente relatório descreve o desenvolvimento do projeto da UC de Laboratórios em Engenharia Informática, integrada no 1º ano do Mestrado em Engenharia Informática da Universidade do Minho.

Este projeto consiste no desenvolvimento de uma linguagem textual de domínio específico (DSL) para representar árvores de comportamento (às quais nos referiremos daqui em diante por *Behavior Trees*) e um compilador capaz de traduzir esta linguagem para uma linguagem de programação já existente (por exemplo *Python*).

De modo conciso, *Behavior Trees* são formalismos para descrever comportamentos reativos de atores autónomos, como um robô, ou um NPC num jogo. Com esta linguagem, pretendemos criar uma forma simples e intuitiva de especificar estas árvores, e ao mesmo tempo acrescentar novas funcionalidades que, a nosso ver, podem melhorar a qualidade da implementação de comportamentos nestes atores.

- Estrutura do relatório
  - Behavior Trees - Básicos
  - Linguagem
  - Compilador
  - Implementação
  - Caso de Estudo
  - Conclusão

Capítulo 2

Behavior Trees - Básicos

2.1 O que são *Behavior Trees*?

Behavior Trees, ou BT, são estruturas que controlam a execução de um conjunto de ações por parte de um agente autónomo, de forma a descreverem o seu comportamento. Este tipo de estrutura começou por ser utilizado especialmente em videojogos (para simular o comportamento de NPCs), mas com o passar do tempo áreas como a Robótica ou a Inteligência Artificial também o começaram a usar, devido à sua capacidade modular e reativa.

Formalmente, uma BT é uma árvore com raiz onde os nodos internos são chamados *control flow nodes* e as folhas são chamadas de *execution nodes*. Para cada nodo conetado usamos a terminologia de *parent* (pai) e *child* (filho). A raiz é o único nodo que não tem pais, todos os outros nodos têm exatamente um pai. Os *control flow nodes* têm pelo menos um filho, já os *execution nodes* não têm nenhum.

A figura 2.1 mostra um exemplo de uma BT, onde o nodo a verde é a raiz, os nodos a cinzento são *control flow nodes* e os restantes são *execution nodes*.

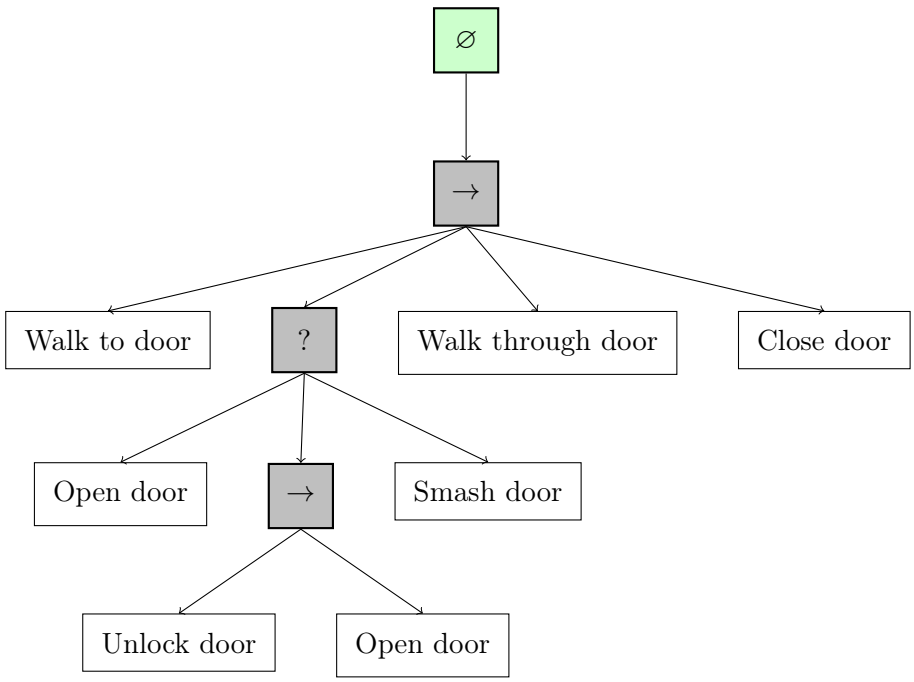


Figura 2.1: Exemplo de uma BT.

2.2 Execução

Uma BT começa a sua execução pelo nodo raiz que gera sinais chamados *ticks* com uma determinada frequência, que são enviados para os seus filhos. Estes sinais permitem a execução dos nodos.

Qualquer nodo, não importa o tipo, ao ser executado retorna um de três estados:

- *Success* - foi executado com sucesso;
- *Failure* - não conseguiu ser executado;
- *Running* - ainda está a executar.

2.3 Nodos

2.3.1 *Control Flow Nodes* - Clássicos

Control flow nodes são nodos estruturais, que não têm qualquer impacto no estado global do sistema. Na formulação clássica, existem 4 categorias deste tipo de nodo: *Sequence*, *Selector (ou Fallback)*, *Parallel* e *Decorator*.

Sequence Um nodo *Sequence* visita (envia *ticks*) todos os filhos por ordem, começando pelo primeiro, e se este retornar *Success*, chama o segundo, e por aí em diante. Caso um filho falhe, o nodo *Sequence* retorna *Failure* imediatamente. Caso todos os filhos sucedam, o nodo retorna *Success*. Caso um filho retorne *Running*, o nodo retorna também *Running*.

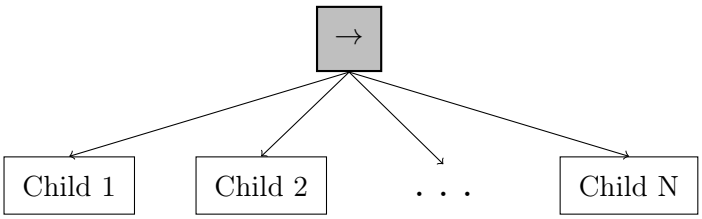


FIGURA 2.2: Estrutura de um nodo *Sequence*.

Selector Tal como o *Sequence*, o nodo *Selector* visita todos os filhos por ordem, mas só avança para o próximo se o filho que está a ser executado retorne *Failure*. Caso um filho suceda, o nodo *Sequence* retorna *Sucess* imediatamente. Caso todos os filhos falhem, retorna *Failure*. Caso um filho retorne *Running*, o nodo retorna também *Running*.

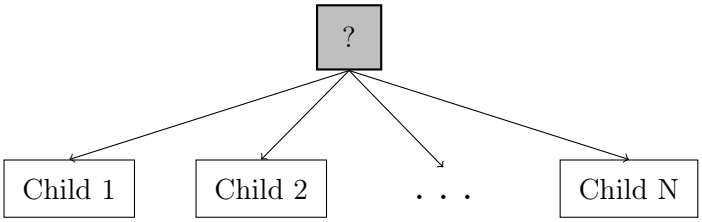


FIGURA 2.3: Estrutura de um nodo *Selector*.

Parallel Um nodo *Parallel*, como o nome indica, visita todos os filhos paralelamente. Para $M \leq N$, retorna *Success* caso M filhos sucedam, e retorna *Failure* caso $N-M+1$ filhos retornem *Failure*. Caso contrário, retorna *Running*.

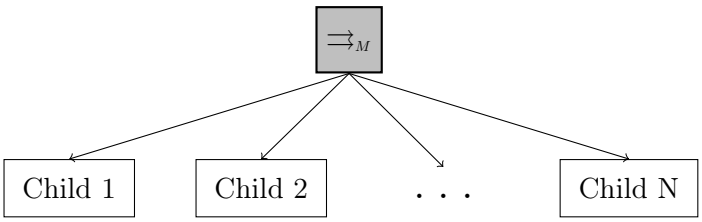


FIGURA 2.4: Estrutura de um nodo *Parallel* com taxa de sucesso M.

Decorator Um nodo *Decorator* tem um único filho, e utiliza uma política (conjunto de regras) definida pelo utilizador para manipular o estado de retorno desse filho ou controlar a sua execução.

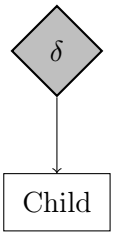


FIGURA 2.5: Estrutura de um nodo *Decorator* com uma política δ.

Exemplos de nodos *Decorator*:

- *inverter* - inverte o estado *Success/Failure* retornado pelo filho;
- *max-N-tries* - o filho só pode falhar N vezes, depois retorna sempre *Failure* sem enviar *ticks* para o filho.
- *max-T-seconds* - deixa o filho correr durante T segundos. Se depois disto o filho retornar *Running*, o nodo retorna *Failure* sem correr o filho.

2.3.2 *Execution Nodes*

Execution nodes são os nodos mais simples, porém os mais poderosos, pois contêm os testes ou ações que serão implementados pelo sistema. Existem 2 categorias para este tipo de nodos: *Action* e *Condition*.

Action Quando recebe *ticks*, um nodo *Action* executa um ou mais comandos. Retorna *Success* se a ação foi corretamente completada ou *Failure* se a ação falhou. Enquanto está a ser executado, retorna *Running*.

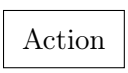


FIGURA 2.6: Estrutura de um nodo *Action*.

Condition Quando recebe *ticks*, um nodo *Condition* verifica uma proposição. Retorna *Success* ou *Failure* dependendo se a proposição é válida ou não. De notar que um nodo *Condition* nunca retorna um estado *Running*.

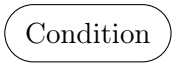


FIGURA 2.7: Estrutura de um nodo *Condition*.

NOTA: *Quando juntamos o nodo Condition com os nodos Sequence e Selector podemos criar uma expressão condicional (if-then-else). Vejamos o seguinte exemplo:*

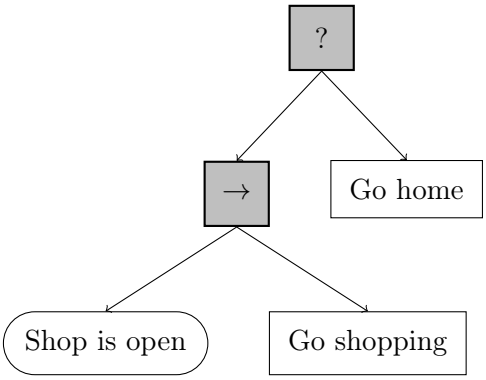


FIGURA 2.8: Exemplo de uma expressão condicional numa BT.

Como podemos ver, a escolha da ação a executar alterna entre ir às compras ou ir embora consoante a loja esteja aberta ou não. Ou seja,

***if** Shop is open **then** Go shopping **else** Go home*

2.3.3 *Control Flow Nodes* - Novos

Para além dos *control flow nodes* existentes na formulação clássica das BTs, podemos ter outras categorias de nodos, de forma a proporcionar uma melhoria a tanto a nível de AI, como a nível de imprevisibilidade, ou até mesmo apenas algo que facilite a implementação de certos comportamentos por parte do programador. Visto isto, decidimos introduzir um novo nodo: *Probability Selector*.

Probability Selector Um nodo *Probability Selector* pode ser visto como uma "extensão" ao nodo *Selector*. A única diferença é que, enquanto o *Selector* clássico visita os filhos por ordem, este novo nodo executa-os de uma forma aleatória tendo em conta que cada filho tem uma certa probabilidade, definida pelo utilizador, de ser escolhido.

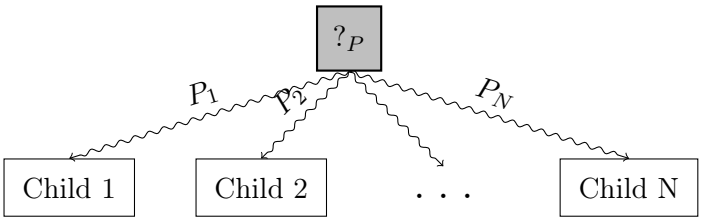


FIGURA 2.9: Estrutura de um nodo *Probability Selector*.

Com este novo nodo, conseguimos atingir um novo nível de AI, pois permite-nos construir comportamentos não só mais imprevisíveis, como também influenciados por fatores externos. Analisemos a seguinte situação:

Num videojogo, o nosso herói quer convidar um certo NPC para o acompanhar numa missão muito perigosa. No entanto há um problema... o NPC é pouco corajoso.

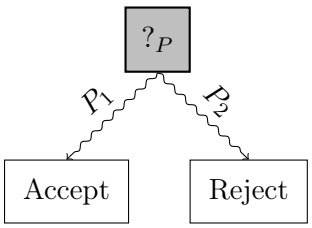
Se olharmos para esta situação de forma racional, é óbvio que ele irá rejeitar este convite. Mas não existirá na mesma a possibilidade de ele por acaso aceitar? Vejamos o seguinte:

Mas por obra do acaso o dia até está a correr bem ao NPC, e naquele momento ele sente-se confiante e, contra todas as espetativas, decide aceitar o convite.

Ok, temos aqui novas informações acerca do comportamento do NPC, mas nada demais. Construindo um *if-then-else* com a nossa BT conseguimos resolver este problema. Se o NPC estiver confiante aceita, senão rejeita.

Mas e se mesmo que esteja confiante, o medo leva a melhor e ele decide rejeitar? No mundo real, não conseguimos prever com 100% de certeza o comportamento de uma certa pessoa. Existe aqui um nível de imprevisibilidade que não é atingível utilizando apenas os nodos clássicos das BT.

Posto isto, vamos adicionar esta subárvore com o nosso novo nodo ao comportamento do NPC:



$$P_1 = \frac{C}{10}$$
$$P_2 = 1 - \frac{C}{10}$$

onde $C \in [0, 10]$ é um valor , criado pelo programador, que representa o nível de coragem do NPC.

O que esta subárvore diz é que, mesmo que seja muito pouco provável (porque tem um C reduzido), existe a possibilidade do NPC aceitar o convite, quer seja por estar confiante, ou por outra razão qualquer que não diz respeito ao jogador.

Mesmo não sendo muito correto (porque não sabemos que fatores levam à existência desta possibilidade), é uma aproximação interessante à tal imprevisibilidade que existe na vida real, permitindo a criação de comportamentos ainda mais complexos.

2.4 Exemplo - *PACMAN*

Para percebermos melhor as aplicações das BTs, decidimos escolher como exemplo aplicacional o jogo do PAC-MAN da *Namco's*.

O jogo do PAC-MAN é um jogo *single player*, em que se passsa num labirinto que vai mudando a medida que o jogador passa de nivel. O jogador controla um boneco chamado de PAC-MAN, que tem como objectivo comer comida(Eat Pills) no labirinto e fugir dos inimigos chamados de fantasmas(Ghosts). Esta comida(Pills) será representada por uma bolinha, sendo que existem 2 tipos de comida(Pills), a comida(Pills) "normal"é representada por uma bolinha mais pequena, sendo que a bolinha maior após ingerida torna os fantasmas vulneráveis(Ghosts Scared) podendo o jogador comer os fantasmas(Chase Ghosts) durante um determinado tempo, após esse tempo os fantasmas regressam ao seu estado inicial. Os fantasmas irão perseguir o jogador, sendo que o objectivo deste será fugir destes(Avoid Ghosts) ou come-los caso estes estejam vulneráveis. Por fim o jogador transita para o proximo nivel assim que comer toda a comida existente no labirinto.

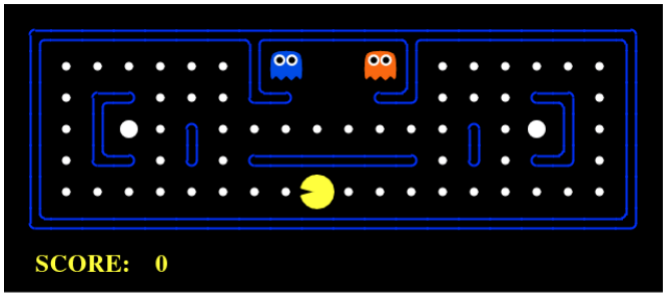


Figura 2.10: Exemplo do jogo de PAC-MAN

Foi importante perceber como funciona o jogo numa fase inicial pois, para definirmos uma BT para o jogador precisamos de perceber qual o seu comportamento. Dito isso, percebemos que numa fase inicial o jogador deverá verificar se não se encontra nenhum fantasma perto(Ghost Close), caso se encontre deverá fugir deste(Avoid Ghosts), caso estes se encontrem vulneráveis o jogador deverá come-los(Chase Ghosts). Caso não se encontre nenhum fantasma por perto o jogador deverá comer a comida(Eat Pills) que se encontra no labirinto. Após a análise do comportamento, iremos em seguida mostrar a respectiva BT.

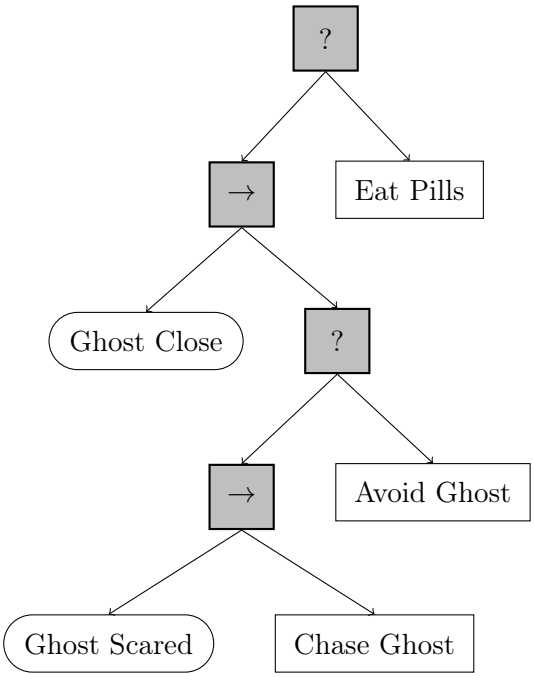


FIGURA 2.11: Exemplo da BT de um jogador de PAC-MAN.

Capítulo 3

Linguagem

TODO: breve introdução ao capítulo

3.1 Estrutura de um ficheiro

Um comportamento pode ser visto como um programa. Cada ficheiro, ao qual designamos de "ficheiro de comportamento", descreve um e só um comportamento.

Um ficheiro de comportamento está dividido em 3 partes:

- *Comportamento* - especificação da BT que descreve o comportamento representado pelo ficheiro. É análogo à função *main* de um programa.
- *Definições* (opcional) - definições de nodos (ou sub-árvores) que podem ser referenciadas noutros nodos ou na árvore de comportamento.
- *Código* - código em *Python* onde são descritos os nodos de execução, e código extra que o programador deseje utilizar.

3.2 Sintaxe

3.2.1 Comportamento

Um comportamento é especificado da seguinte forma:

```
behavior : [
    tree
]
```

onde `tree` é um dos nodos referidos anteriormente, cada um com a sua própria sintaxe (seguindo a estrutura do capítulo anterior):

```
sequence : [
    child1,
    child2,
    ...
    childn
]
```

```
selector : [
    child1,
    child2,
    ...
    childn
]
```

```
prob_selector : [
    P1 -> child1,
    P2 -> child2,
    ...
    Pn -> childn
]

decorator : POLICY [
    child
]
```

```
parallel : M [
    child1,
    child2,
    ...
    childn
]

condition : $CONDITION

action : $ACTION
```

Se repararmos, os *execution nodes* são definidos por um nome antecedido de um '$', que é uma referência para algo. Neste caso este algo é uma função, que obrigatoriamente tem de estar definida na parte do *código* do ficheiro.

Os *control flow nodes* também podem ser escritos com esta notação, sendo que nome em questão tem de existir no conjunto de *definições* do ficheiro.

3.2.2 Definições

As definições são *control flow nodes* etiquetados com um nome. A sua sintaxe é bastante semelhante à apresentada em cima:

```
sequence NAME : [
    child1,
    child2,
    ...
    childn
]

prob_selector NAME : [
    P1 -> child1,
    P2 -> child2,
    ...
    Pn -> childn
]

decorator NAME : POLICY
    [
    child
]
```

```
selector NAME : [
    child1,
    child2,
    ...
    childn
]

parallel NAME : M [
    child1,
    child2,
    ...
    childn
]
```

3.2.3 Código

Nesta parte do ficheiro escrevemos código *Python* que será compilado juntamente com o ficheiro de comportamento. Aqui têm de estar definidos quaisquer nodos de execução usados no comportamento ou nas definições.

Capítulo 4

Compilador

4.1 Análise léxica *(tokenization)*

A primeira etapa no desenvolvimento de um compilador consiste na criação de um analisador léxico, ou *lexer*, que é responsável por converter uma sequência de caracteres numa sequência de *tokens*. Um *token* é uma *string* com um significado atribuído, e pode ser estruturado como um par que contém um nome e um valor opcional.

A seguinte tabela de valores mostra os *tokens* utilizados para esta análise:

Tokens	
Nome	**Valor**
literais	`([]),:%`
RIGHTARROW	`->`
BEHAVIOR	`\bbehavior\b`
SEQUENCE	`\bsequence\b`
SELECTOR	`\bselector\b`
PROBSELECTOR	`\bprobselector\b`
PARALLEL	`\bparallel\b`
DECORATOR	`\bdecorator\b`
CONDITION	`\bcondition\b`
ACTION	`\baction\b`
INVERTER	`\bINVERTER\b`
MAXTRIES	`\bMAXTRIES\b`
MAXSECONDS	`\bMAXSECONDS\b`
INT	`\d+`
VAR	`$\w+`
NODENAME	`\b\w+\b`
CODE	`%%(.|\n)+`

Tabela 4.1: Tabela de valores do *lexer*.

4.2 Análise sintática *(parsing)*

A análise sintática é o processo de analizar uma sequência de símbolos consoante uma gramática. De uma forma muito simples, uma gramática é um conjunto de regras que nos diz se uma determinada *string* pertence a uma certa linguagem ou se está gramaticalmente incorreta.

Os símbolos podem ser de dois tipos: *terminais* ou *não terminais* (*tokens*). Os símbolos terminais, ou *tokens*, são os símbolos elementares da linguagem. Os não

terminais são substituídos pelos terminais de acordo com as regras de produção da gramática.

Para a nossa linguagem, temos a seguinte gramática, onde os símbolos em minúsculo são os não terminais e os que estão em maiúsculo são os terminais:

```
root : behavior CODE
     | behavior definitions CODE
     | definition behavior CODE

behavior : BEHAVIOR ':' '[' node ']'

node : SEQUENCE ':' '[' nodes ']'
     | SEQUENCE ':' VAR
     | SELECTOR ':' '[' nodes ']'
     | SELECTOR ':' VAR
     | PROBSELECTOR ':' '[' prob_nodes ']'
     | PROBSELECTOR ':' VAR
     | PARALLEL ':' INT '[' nodes ']'
     | PARALLEL ':' VAR
     | DECORATOR ':' INVERTER '[' node ']'
     | DECORATOR ':' MAXTRIES '(' INT ')' '[' node ']'
     | DECORATOR ':' MAXSECONDS '(' INT ')' '[' node ']'
     | DECORATOR ':' VAR
     | CONDITION ':' VAR
     | ACTION ':' VAR

nodes : nodes ',' node
      | node

prob_nodes : prob_nodes ',' prob_node
           | prob_node

prob_node : VAR RIGHTARROW node

definitions : definitions definition
           | definition

definition : SEQUENCE NODENAME ':' '[' nodes ']'
          | SELECTOR NODENAME ':' '[' nodes ']'
          | PROBSELECTOR NODENAME ':' '[' prob_nodes ']'
          | PARALLEL NODENAME ':' INT '[' nodes ']'
          | DECORATOR NODENAME ':' INVERTER '[' node ']'
          | DECORATOR NODENAME ':' MAXTRIES '(' INT ')'
            '[' node ']'
          | DECORATOR NODENAME ':' MAXSECONDS '(' INT ')'
            '[' node ']'
```

4.3 Análise semântica

Capítulo 5

Caso de Estudo

Capítulo 6

Conclusão

Apêndice A

Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:
`\hypersetup{urlcolor=red}`, or
`\hypersetup{citecolor=green}`, or
`\hypersetup{allcolor=blue}`.
If you want to completely hide the links, you can use:
`\hypersetup{allcolors=.}`, or even better:
`\hypersetup{hidelinks}`.
If you want to have obvious links in the PDF but not the printed text, use:
`\hypersetup{colorlinks=false}`.