Uma possível abordagem para o ensino introdutório de algoritmos em cursos de Computação

Marcos A. Castilho, Fabiano Silva, Daniel Weingaertner
{marcos,fabiano,danielw}@inf.ufpr.br
Departamento de Informática/UFPR

RESUMO

Neste artigo discutimos uma possível abordagem para disciplinas introdutórias de algoritmos para o ensino superior em cursos de Computação e similares. Mostramos a experiência de 30 anos refinando conceitos, simplificando alguns pontos que nos permitem avançar em outros. Comentamos as motivações e as decisões tomadas ao longo de um curso de 60 horas. Estas decisões têm relação com o sequenciamento, com o uso de uma linguagem real como apoio e como é possível, ao final do curso, abordar assuntos relativamente avançados, mas de forma lúdica, até certo ponto.

CCS CONCEPTS

• **Social and professional topics** → Computing education.

PALAVRAS-CHAVE

Ensino de programação, Introdução à programação.

1 INTRODUÇÃO

O ensino de programação de computadores em cursos introdutórios de nível superior é considerado difícil e é tema de inúmeros artigos no mundo. Algumas dezenas de excelentes livros estão disponíveis nas bibliotecas das universidades, muitos deles também na Internet.

Este texto trata do contexto brasileiro, no qual poucos livros sobre o tema estão disponíveis em língua portuguesa. Algumas excelentes obras que costumamos adotar são:

- Sérgio Carvalho [1];
- Salveti e Barbosa [3];
- Harry Farrer e colegas [4];
- Marco Aurério Medina e Cristina Fertig [7];
- Jean-Paul Tremblay e Richard B. Bunt [8];
- Niklaus Wirth (um dos poucos traduzidos) [9].

Recentemente os autores deste artigo publicaram um livro [2] basicamente por dois motivos:

- Foi publicado sob licença Creative Commons;
- Reflete a maneira como entendemos o ensino inicial de algoritmos.

O primeiro ponto é muito importante para nós que acreditamos em software livre, em conhecimento livre e em livre acesso à Ciência. De fato, na ocasião da gênese deste material tínhamos um aluno com deficiência visual total e ele não tinha acesso à literatura impressa. Nós passávamos os fontes em LaTeX do livro e ele pode estudar com um pouco mais de qualidade.

O segundo ponto é relacionado com a maneira como entendemos um curso introdutório dentro da grade curricular do nosso curso, considerando a realidade sociogeográfica dos estudantes do curso de Bacharelado em Ciência da Computação da Universidade Federal do Paraná (UFPR), localizado em Curitiba, capital do estado do Paraná, e definitivamente não tem relação com qualquer tipo de questionamento da qualidade das obras existentes.

Existem várias formas de introduzir algoritmos aos estudantes. A título de exemplo, conhecemos a iniciativa do MIT em abordar linguagens funcionais em cursos introdutórios, alguns outros procuram ensinar orientação a objetos logo no primeiro período dos cursos, outros optam pelo uso de pseudocódigo, como *Portugol*, ou até mesmo usando somente fluxogramas, dentre outras iniciativas.

É bem verdade que todas estas iniciativas são interessantes e válidas, mas todas elas estão inseridas em um certo contexto sociogeográfico, consideram o plano pedagógico do seu curso, a grade curricular, e sobretudo, seu corpo docente. O ensino no primeiro período deve estar cuidadosamente ligado com as disciplinas subsequentes, tais como algoritmos e estruturas de dados mais avançadas.

Um ponto importante é que nós temos no nosso departamento um time de professores que discute esta temática há muitos anos. Esta equipe é constituída por pelo menos uma dezena de professores que tiveram suas formações em diversas instituições importantes no Brasil, tanto estaduais quanto federais. Muitos deles também vivenciaram experiências acadêmicas em outros países, tais como EUA, França, Alemanha, Espanha, Portugal, Japão e Inglaterra.

Porém, ocorreu que ao longo das nossa aulas neste tipo de disciplina, para calouros, os professores foram entendendo que era necessário adotar um padrão para as diversas turmas a cada semestre. Nós recebemos 110 estudantes por ano, por isso temos de duas a cinco turmas semestrais que ocorrem de forma simultânea.

Ao longo do tempo entendemos que era necessário seguir um mesmo ritmo, aplicar provas unificadas, conversar (pessoalmente ou por meios digitais), nos corredores, no café, enfim, o resultado foi a formação de um time de professores altamente interessados em melhorar o ensino de programação para calouros.

É importante mencionar que esta disciplina ocorre simultaneamente com uma outra de Circuitos Lógicos, na qual o nosso interesse é a representação de números binários e álgebra booleana, desta maneira podemos omitir este tipo de detalhe no início do curso.

Também vale mencionar que existe em nosso currículo uma importante disciplina subsequente, do segundo período, denominada Programação 1, é uma disciplina de laboratório cujo objeto principal são os tipos abstratos de dados implementados na linguagem C. Esta

EduComp'23, Abril 24-29, 2023, Recife, Pernambuco, Brasil (On-line)

é ministrada em paralelo com Algoritmos e Estruturas de Dados 2, na qual se abordam problemas de busca, ordenação e recursividade.

O texto que segue é o resultado materializado das incontáveis vezes que muitos professores discutiram o assunto e entenderam como sendo um bom curso de introdução aos Algoritmos e Estruturas de Dados, compatível com o nosso currículo. O artigo é escrito tomando-se o sequenciamento do livro como referência, mas o objeto principal é o sequenciamento das aulas e as decisões tomadas sobre o que omitir ou enfatizar ou sobre o que pode ser periférico mas pode ser apresentado. Também falaremos sobre a dosagem do tempo na apresentação dos conteúdos.

2 ESTRUTURA DO LIVRO - PARTE 1

Conforme explicamos, o livro segue fielmente a sequência de aulas de uma disciplina de 60 horas/aula, ministrada duas vezes por semana (4 horas/aula por semana) em um total de 30 aulas. Elas são divididas em três partes de aproximadamente 10 aulas cada:

(1) A parte 1 contém os fundamentos dos algoritmos e é acompanhada também dos fundamentos de uma linguagem de programação, no nosso caso *Pascal*. É considerada a parte fundamental e normalmente é ministrada em 10 aulas, sendo a décima primeira a prova 1.
(2) A parte 2, ministrada em 9 aulas, sendo a décima a prova 2, trata de funções e procedimentos e de vetores, incluindo aqui a caracterização do que é uma estrutura de dados.
(3) A parte 3 ocorre nas 9 últimas aulas, sendo a última aula a prova 3. Aqui tratamos de matrizes, registros e uma breve introdução aos tipos abstratos de dados. As duas últimas aulas têm uma parte lúdica, motivadora e importante para os estudantes, que é a implementação de um jogo simples baseado em matrizes, nada gráfico, tudo textual. Em geral os discentes gostam muito e se motivam bastante.

Dado este contexto, explicamos brevemente o sequenciamento das ideias distribuídas em 13 capítulos, sendo o primeiro uma introdução. O livro propriamente dito inicia no capítulo 2.

2.1 Primeira parte da Parte I

A parte 1 do livro contém 6 capítulos (de 2 a 7), sendo que o principal são os capítulos de 5 a 7, que são programados para 8 aulas. Os capítulos de 2 a 4, ministrados em no máximo 3 aulas, vão da discussão de problemas em altíssimo nível até o funcionamento da máquina com a apresentação do Modelo Von Neumann.

No capítulo 2 tratamos um problema muito simples, que é contar pessoas presentes em um evento. Fazemos uma grande discussão de várias maneiras de contar, procuramos formalizar em alto nível cada solução, discutimos brevemente a eficiência, a qualidade e a viabilidade da aplicação da técnica em diversos contextos. Tratamos de como algoritmos não são necessariamente bons ou ruins, apenas se aplicam melhor, ou não, em certas situações.

Normalmente os estudantes ficam impressionados com a quantidade de soluções para um problema tão simples. O ápice deste capítulo é a apresentação de um algoritmo logarítmico, o qual entendemos como muito importante para mostrar de maneira bastante intuitiva o poder e a elegância dos algoritmos. Mostramos que é teoricamente possível contar toda a população do planeta em alguns poucos passos (cerca de 20 ou 21).

Acreditamos que esta discussão, logo na primeira aula, mostra a beleza da Ciência da Computação. Ainda que introdutoriamente, discutimos o que é de fato fazer Computação e procuramos convencê-los de que Computação é uma bela ciência.

O capítulo 3 motiva para a apresentação do modelo Von Neumann que está no capítulo 4. O objetivo é mostrar que é preciso estabelecer um ponto mínimo de detalhamento de uma solução. O exemplo usado é o da receita de bolo.

Este exemplo está presente em vários outros livros, mas aqui o foco é diferente. Normalmente, na literatura que conhecemos, este exemplo é usado para apresentação da ideia do que é um algoritmo. Nós também fazemos isso, mas vamos um pouco além.

Mostramos não apenas que um algoritmo é escrito por uma pessoa para ser executada por outra, mas fundamentalmente que estas duas pessoas devem ter aproximadamente o mesmo nível de conhecimento sobre o assunto sendo discutido. De fato, se o algoritmo é escrito por uma pessoa experiente em cozinha, mas será executado por um aprendiz que mal sabe fritar um ovo, então ele não vai conseguir seguir os passos.

De maneira lúdica, brincamos que o cozinheiro experiente, em algum momento, perde a paciência em detalhar alguma parte do processo e ele obriga o aprendiz a pelo menos aprender o básico, senão ele não ajuda mais. Este é o gancho para o modelo Von Neumann, que é o nosso limite inferior de conhecimento.

Finalmente, esta parte termina com o modelo Von Neumann. A pergunta é: vale a pena mostrar este modelo? Acreditamos que sim.

Este modelo é uma clara fronteira entre a máquina e o ser humano. Ele deixa claro, de maneira intuitiva e simplificada, o que é de fato e como funciona a máquina. Ele desmistifica o conceito de computador e de computação também. Entendemos que este conhecimento é valioso para os discentes.

Uma grande vantagem é que a "caixa preta" do processo de computação fica plenamente caracterizada como algo que processa entrada e gera uma saída por manipulações de dados que estão em memória. Aliás, a memória é apresentada como endereços que têm conteúdos. O principal ganho é que a noção de *variável*, que é introduzida no capítulo seguinte, se torna clara: é nada mais nada menos do que um nome associado a um endereço de memória.

A apresentação em si do modelo é bastante simplificada, mostramos uma máquina hipotética com somente 9 instruções, mas que permite ver o processo da computação de um tipo de equação do segundo grau resolvida pelo método de Bhaskara. Na aula pode ser usada inclusive uma apresentação um pouco teatral no qual discentes e docentes fazem o papel de unidade de controle, de unidade lógica e aritmética e dos registradores.

Os estudantes só descobrem que se trata da fórmula de Bhaskara ao final da aula, antes eles acompanham o maçante e exaustivo processo no qual o computador "não tem ideia" do que está fazendo. Vários níveis de abstração levam à questão do que é um compilador e do que é uma linguagem de alto nível. Ao final da aula temos um código em *Pascal* que pode ser compilado e experimentado.

2.2 Motivações para a segunda parte da Parte I

A segunda parte da Parte I, a qual consideramos a principal do livro, será discutida detalhadamente a seguir, por enquanto comentamos algumas motivações e decisões que tomamos.

A parte I é fundamental para o aprendizado, ela é chave para que nossas ideias da disciplina como um todo possam ocorrer. É muito difícil aprender os conceitos básicos, existem muitos estudos a respeito. Por isso a redação, o sequenciamento, o *timing* e o nível de profundidade dos assuntos abordados mereceu extrema atenção para chegarmos no que consideramos ideal para nosso caso.

São três capítulos, com títulos bem sugestivos:

Cap 5: Conceitos elementares;
Cap 6: Técnicas elementares;
Cap 7: Aplicações das técnicas elementares.

Estes conteúdos estão presentes na maior parte da literatura, mas nós procuramos encontrar um sentido lógico para os temas, problemas e algoritmos apresentados. Por isso separamos um capítulo inicial (cap. 5) que basicamente apresenta comandos e conceitos de qualquer linguagem de programação (no nosso caso seguimos o paradigma de programação estruturada):

- O fluxo de execução de um programa;
- Comandos: entrada e saída, atribuição, repetição e desvio condicional;
- Expressões: aritméticas e lógicas.

Um fundamento defendido é a adoção de uma linguagem de programação real. A linguagem escolhida foi *Pascal*. Este ponto sempre é objeto de muitas discussões com todo o time de professores. Sempre há quem defenda que:

- não se deve trabalhar com uma linguagem real, mas com pseudocódigo ou fluxogramas. Esta linha defende que aprender algoritmos já é difícil o suficiente e que paralelamente aprender uma linguagem também já é difícil, então acreditam que aprender ambas ao mesmo tempo é aumentar demais a natural dificuldade do aprendizado dos algoritmos;
- sim, deve ser adotada uma linguagem e sim, deve ser mantido o paradigma estruturado, mas a linguagem deve ser outra. Exemplos frequentemente citados são *C* e *Python*;
- programação estruturada é coisa do passado, deve-se adotar desde cedo o paradigma de orientação a objetos e linguagens modernas como *Java*, afinal, é o que o mercado usa.

A nossa decisão, que reflete a sala de aula, é a opinião da maioria e tem os seguintes argumentos favoráveis:

- Programação estruturada é mais simples conceitualmente do que orientação a objetos;
- A maior parte da literatura clássica e de qualidade também se baseia em programação estruturada, por isso o discente sempre tem opções de leitura complementar;
- *Pascal* foi criada para fins didáticos, tem um compilador bem comportado, uma estrutura simples, as mensagens de erro do compilador são razoavelmente autoexplicativas;
- *Pascal* é uma linguagem fortemente tipada e nós acreditamos que isso é importante para aprendizes de computação;
- A linguagem "esconde" do estudante algumas coisas que podem se tornar complexas, tais como ponteiros e endereços. De fato, em *C* todos reclamam do `&` obrigatório em um `scanf`;
- Uma outra coisa ruim para quem aprende *C* como primeira linguagem é perder horas procurando o problema no código porque usou `if (a = 0)` ao invés de `if (a == 0)`. O uso do símbolo `=` na forma simples só dá um *warning*, mas não dá erro, sendo que o correto é usar `==`. Em *Pascal* é mais claro com o uso da notação `:=`;
- *Pascal* permite estudar de maneira simples as semânticas de passagem de parâmetro por valor e por referência, bem como a noção de escopo de variáveis, caracterizando as variáveis globais e locais.
- Por fim, o uso em uma linguagem antiga e em desuso força os estudantes a aprenderem outras linguagens de programação nas disciplinas subsequentes do curso, o que acreditamos ajudar na habilidade de se adaptar rapidamente a novas tecnologias. No nosso caso, a disciplina Programação 1 do segundo período exige que os programas sejam feitos em *C*. Acreditamos que a obrigatoriedade de se aprender outra linguagem é benéfica.

Além disso, nossa abordagem é *bottom-up*. Queremos que os programadores entendam como funciona o computador e o saibam programar com uma linguagem de baixo nível. A linguagem alvo para os nossos bacharelados é C/C++, pois é importante para algumas disciplinas estratégicas tais como Sistemas Operacionais, Redes, Introdução à Computação Científica, dentre outras.

Entretanto, a linguagem C possui muitas variações (*C, C++09, C++11, C++14, C++17, C++20*) que, quando os alunos buscam informações na Internet, são apresentadas como sendo equivalentes ou misturadas. Isso acaba dificultando porque confunde o aluno.

A linguagem *Pascal* possui as características que precisamos de uma linguagem de baixo nível (tipos bem definidos, alocação "manual" de memória, mas por não ser tão largamente utilizada, possui bem menos variações e os compiladores possuem um comportamento previsível. Além disso, como em seguida os alunos devem produzir trabalhos em *C*, passam pelo processo de troca de linguagem e percebem na prática que as diferenças são pequenas.

Utilizar linguagens como *Python* ou *Java* são interessantes para processos de aprendizado *top-down*, no qual o aluno consegue com poucas linhas de código escrever programas complexos. Estas abordagens costumam ser mais atraentes para os alunos, pois eles conseguem produzir conteúdo muito mais sofisticado em pouco tempo. Entretanto, uma vez que tenham se acostumado com as facilidades deste tipo de linguagem, há uma grande resistência em "baixar o nível", ou até mesmo em compreender a razão de se aprender uma linguagem de baixo nível.

Continuamos precisando de pessoas que sejam capazes de programar um computador atentando para cada byte de memória utilizada ou para cada instrução executada, para o uso eficiente de registradores. Afinal, alguém ainda precisa escrever as rotinas dos sistemas operacionais, ou os compiladores, ou até mesmo implementar os interpretadores das linguagens de alto nível.

Nossa hipótese (que tem se mostrado acertada) é que iniciar com uma linguagem de baixo nível não impede o desenvolvimento em direção ao uso de linguagens de alto nível, o que acaba sendo a escolha da maioria dos alunos. Mas permite que alguns descubram os encantos de se trabalhar próximo ao hardware, compreendendo as sutilezas acessíveis apenas pelas linguagens de baixo nível.

Estes argumentos não são suficientes para a questão apresentada acima, sobre a suposta dificuldade em aprender simultaneamente duas coisas difíceis. Mas nós entendemos que se os conceitos forem apresentados corretamente, com um bom sequenciamento e com o

devido tempo, o fato do aluno programar permite que ele valide seus algoritmos. O estudante pode compilar, rodar o programa, testar, editar novamente, recompilar, etc.

Ao mesmo tempo, permite o uso de ferramentas existentes para correção automática de códigos, existem muitas disponíveis hoje em dia. Nós adotamos a ferramenta *FARMA-ALG* [5, 6], pois entendemos que ela é mais didática do que o uso de outras no estilo do BOCA, usada em competições de programação. O BOCA responde sim ou não, e entendemos que isso não é didático.

Uma vez estas importantes decisões tomadas, passamos ao novo desafio, que já foi mencionado acima e repetimos aqui por questão de clareza: Como fazer um bom sequenciamento e equilibrar o tempo necessário para a absorção dos conteúdos?

2.3 Capítulo 5: conceitos elementares

Entendemos como conceitos elementares: o fluxo de execução de um programa, as expressões aritméticas e lógicas e os comandos que modificam o fluxo de execução do programa, além dos comandos que permitem a interação do usuário com a máquina e que manipulam a memória propriamente dita.

Vale reforçar que nesta parte o tempo de apresentação é fundamental. O conteúdo apresentado a seguir é feito em 8 dias de aulas. Alguns conceitos são deliberadamente omitidos para podermos usar melhor o tempo com as ideias fundamentais de desvio e repetição.

De todos os conceitos os mais simples são os comandos de entrada e saída, então optamos por iniciar por este último, apresentando a partir de um "hello world" algumas evoluções dos conceitos da linguagem *Pascal*. Mostramos por exemplo, com cuidado, a diferença entre escrever `writeln ('2 + 2');` e `writeln (2 + 2);`, isto é, com e sem aspas. Apresentamos o compilador e mostramos:

- As diferentes saídas para os exemplos acima;
- Erros que podem ocorrer e como corrigi-los, entendendo as mensagens diferentes que aparecem na tela. são eles: erros de compilação; erros de execução (*runtime errors*); erros de lógica.

Introduzimos cuidadosamente um comando de leitura, usando somente o `read`, ignorando completamente a forma `readln`. Esta parte é delicada, pois já envolve o uso de variáveis, lembrando que ele é facilitado pela apresentação anterior do modelo Von Neumann.

Diversos pequenos programas com entrada e saída são apresentados, mas aos poucos vamos construindo expressões aritméticas simples. Entre elas, já mostramos uma divisão por zero, motivando o aluno para as aulas futuras sobre desvios.

Discutimos tipos de dados, mas contrariamente ao que é comum na literatura, optamos por mostrar somente dois deles: `integer` e `real`. É suficiente para eles entenderem a noção de representação numérica e os conceitos de linguagem tipada e fortemente tipada. Os tipos `char` e `string` são dispensáveis para o aprendizado inicial de algoritmos e nos permite ignorar o `readln`. O tipo `boolean` aparecerá em breve, mas não agora.

Progressivamente passamos às expressões aritméticas e ao comando de atribuição. Mostramos precedência de operadores e trabalhamos diversos exemplos bastante simples. Temos o cuidado de dedicar uma seção para uma atribuição bastante curiosa: `i:= i + 1;`. A prática de ensino nos mostrou que é uma atribuição estranha para aprendizes, então esclarecemos desde cedo.

É importante mencionar que neste momento já existe no *FARMA-ALG* a primeira lista de exercícios, que envolve somente os conceitos já vistos até o momento. São exercícios para manipulação simples de expressões aritméticas, atribuições, entrada e saída simples, basicamente leitura, um cálculo simples seguido de impressão. Reforça também a diferença entre os tipos inteiros e reais.

Ao introduzirmos as expressões booleanas usamos exemplos simples, idealmente com o uso de um computador com a tela sendo projetada, para os estudantes verem as diferentes formas de escrever código e entender os erros.

Um ponto delicado são expressões tais como $0 \leq x \leq 10$, que não podem ser escritas em linguagens de programação típicas. Consequentemente, explicamos cuidadosamente os operadores lógicos `AND, OR, NOT`. Lembramos que concomitantemente existe a disciplina de circuitos lógicos, que aborda este assunto em detalhes.

Uma importante decisão nossa veio de uma suposição há alguns anos e foi validada em sala de aula: falar de repetições antes de desvios. De fato, observamos que a noção de repetição em algoritmos é melhor absorvida quando apresentada antes dos desvios.

O princípio da apresentação é o de iniciar sempre pelos conteúdos que eles já sabem, progressivamente mostrando que não é mais possível escrever algoritmos que não têm repetição.

Outra importante decisão nossa: apresentamos *um único* comando de repetição, que é o `while`. O `for` e o `repeat` podem esperar um pouco para aparecerem. O `while` é o mais genérico de todos e é suficiente para a compreensão da noção de repetição.

Eles aparecerão mais tarde, após a primeira prova. Quando isto for feito, bastarão 15min de exposição para cada comando, pois eles já deverão ter a noção de repetição compreendida.

Discutimos em detalhes questões fundamentais, tais como a variável de controle do laço bem como critérios alternativos de parada. São sempre dois pontos extremante complexos no processo de aprendizagem e ao mesmo tempo fundamentais no contexto de algoritmos que repetem códigos. Procuramos caracterizar muito bem a diferença entre repetir algo um número fixo de vezes com relação a repetir algo um número indefinido de vezes. Discutimos a questão do "em tempo de compilação" em contrapartida a "em tempo de execução".

Com isso resta apenas o comando de desvio condicional, que como é apresentado após a repetição se torna mais simples. O cuidado é mostrar que não há repetição, só desvio. Mostramos exemplos da diferença entre repetir (`while`) e desviar (`if/then/else`).

Novamente outra decisão nossa: não mostramos o comando `case`, que também é desnecessário neste ponto por se tratar de uma representação alternativa a uma série de `if/then/else`'s. Por outro lado, apresentamos rapidamente o comando `goto`, que nos permite discutir pontos importantes:

- O conceito de programação estruturada em si, que fundamentalmente significa programar sem `goto`;
- A partir da apresentação de dois códigos equivalentes, um com `while` e outro com uma combinação complexa e não intuitiva que usa `if` e `goto`, tentamos convencer o aprendiz de qual é o problema que motivou a programação estruturada, afinal, quem programa e quem lê um programa são seres humanos. Mostrar a diferença entre máquinas e humanos é importante desde a primeira aula.

Mostramos dois algoritmos para o problema de contar números, um sem `goto` (Código 1) e outro com `goto` (Código 2). Procuramos mostrar a simplicidade de um e a complexidade do outro e tentamos justificar que não é boa prática o uso deste comando.

```
program imprimir_de_1_ate_30;
var i: integer;

begin
    i:= 1;
    while i <= 30 do
    begin
        writeln (i);
        i:= i + 1;
    end;
end.
```

Código 1: Contar em programação estruturada

```
program imprimir_de_1_ate_30_com_goto;
label 10;
var i: integer;

begin
    i:= 1;

10: writeln (i);
    i:= i + 1;

    if i <= 30 then
        goto 10;
end.
```

Código 2: Contar usando goto

Finalmente, a segunda lista na ferramenta de correção de códigos já está liberada. Ela trata de programas simples iniciando com repetições simples, desvios simples, depois uma mistura evolutiva deles, mas não ainda com repetições aninhadas.

2.4 Capítulo 6: técnicas elementares

Neste capítulo nós abordamos uma série de problemas amplamente encontrados na literatura. A diferença é que nós procuramos dar nomes para as técnicas de soluções. Acreditamos que dar nomes para elas ajuda o aluno a fixar o conteúdo em aulas futuras, quando formos construir soluções mais complexas, como no caso de séries, que usa basicamente o que chamamos de técnica do acumulador.

Um bom exemplo que podemos comentar é a famosa, e comumente presente na literatura, sequência de Fibonacci. Sua importância é justamente relacionada ao uso de duas variáveis de controle. Os problemas tratados até antes desta sequência eram controlados por uma única variável, por isso demos o nome de "lembrar de mais de uma informação".

Antes de mostrarmos as técnicas propriamente ditas, achamos importante discutir o que significa "lógica de programação". Este termo está presente nos títulos de muitas obras importantes na literatura e merece uma explicação. Isto é seguido de uma brevíssima formalização do teste de mesa, pois neste ponto do curso eles já iniciam o tratamento de problemas um tanto mais sofisticados. Os nomes das técnicas e os problemas usados como exemplo foram:

- Técnica do acumulador: somar vários números com alguns critérios de parada diferentes. Usa repetição e atribuição, sem desvios condicionais do tipo `if`;
- Árvores de decisão: encontrar o menor de três números. Usa somente aninhamentos de desvios condicionais, mas também reforça o conceito de lógica de programação;
- Definir a priori e depois corrigir: encontrar o menor número dentre vários números lidos. Pela primeira vez misturamos repetições com desvios condicionais, reforçamos a lógica de programação;
- Lembrar de mais de uma informação: sequência de Fibonacci. Já explicado acima;
- Processar parte dos dados de entrada: imprimir os números positivos lidos. Serve para reforçar o desvio condicional no escopo da repetição;
- Processar parte dos dados de um modo e outra parte de outro. Generalização do problema anterior para fixação de um desvio condicional com `else`, no escopo da repetição;
- Finalmente, múltiplos acumuladores: histogramas. Serve para generalizar todas as técnicas anteriores.

Os problemas de somar números e encontrar o menor serão retomados várias vezes ao longo do curso. Por exemplo, ao apresentarmos funções, vetores e matrizes, mas também na parte final de tratamento de imagens e na implementação dos jogos. Isto deverá ficar mais claro ao longo deste texto.

Com isso encerramos o capítulo tendo mostrado diversas formas de aninhamento de comandos: desvios no escopo de repetições, repetições no escopo de desvios, desvios no escopo de desvios. Observem uma decisão muito importante que tomamos: *Não* falamos ainda de laços aninhados, pois eles são extremamente difíceis para os estudantes, fato vindo da nossa experiência em sala de aula. Consideramos que isso pode esperar mais um pouco para aparecer.

Colocamos aqui um exemplo que será utilizado algumas vezes ao longo da disciplina. Isso faz parte da nossa "didática da novidade", que ficará clara ao longo do texto. O problema é o de encontrar o menor elemento de um conjunto de dados (Código 3).

```
program encontra_menor;
var n, menor: integer;
begin
    read (n);
    if n <> 0 then (* se o primeiro for zero nao faz nada *)
    begin
        menor:= n; (* chutamos que o menor eh o primeiro lido *)
        while n <> 0 do
        begin
            if n < menor then (* aqui testamos a nova informacao *)
                menor:= n;    (* e corrigimos se for o caso      *)
            read (n);
        end;
        writeln (menor);
    end;
end.
```

Código 3: Encontrando o menor elemento de um conjunto de dados

2.5 Algumas reflexões antes de continuar

Este conteúdo foi apresentado de maneira muito lenta, mas progressiva. Tipicamente usamos 8 das 10 aulas previstas antes da primeira prova, que sempre é aplicada na aula 11.

Usamos problemas que aparecem frequentemente na literatura com alguma formalização alternativa que procuramos dar, como os nome das técnicas elementares. Eliminamos comandos que não são necessários nesta fase da disciplina, tais como os já citados `readln`, `repeat` e `for` bem como completamente ignoramos tipos de dados como `char` e `string`.

Invariavelmente levamos os estudantes ao laboratório para acompanhamento e verificação da evolução deles. Nestas aulas de laboratório ajudamos a entender a ferramenta de correção automática.

Cuidamos para não abordarmos laços duplos, mas apresentamos todas as outras combinações possíveis em dois níveis no máximo, a menos do caso da árvore de decisão, que tem três níveis.

Um ponto importante ainda não mencionado é o cuidado com os exercícios, tanto os presentes no livro quanto os usados no *FARMA-ALG*. São exercícios coletados da literatura, de sites de outras instituições, nacionais ou não, mas tivemos o cuidado de reescrever e padronizar todos os enunciados, eliminar ambiguidades e sobretudo definir um sequenciamento progressivo.

Os exercícios propostos na ferramenta são um subconjunto dos que aparecem no livro, com raros exercícios adicionais. Na ferramenta, extremo cuidado foi dado nos casos de testes, procurando escolher os que cobrem uma gama de problemas que podem ocorrer quando se programa, como por exemplo, um caso de teste que força o aluno a pensar numa possível divisão por zero. A propósito, neste ponto a lista 3 da ferramenta já está liberada, com exercícios compatíveis com o que foi apresentado.

2.6 Capítulo 7: aplicações das técnicas elementares

Este capítulo aborda tanto problemas amplamente encontrados na literatura, como por exemplo o cálculo do MDC pelo método de Euclides, tabuadas e fatoriais, mas também optamos por uma série de exercícios que lidam com operações de divisão e resto de divisão inteira. Achamos que estes exercícios mostram que saber os conceitos e as técnicas elementares não habilita o aprendiz a resolver algoritmos em geral. O último tipo de problema mostrado é o de séries, que consideramos os mais difíceis para o aprendizado, embora o problema de determinar se um número é primo também não seja trivial. Mas achamos importante apresentá-los.

Os problemas mostram que encontrar um algoritmo para um problema não é trivial, exige muita criatividade, compreensão da lógica de programação e muito esforço intelectual. A questão é que os estudantes acham que estão entendendo, pois veem os docentes escreverem o código. Um código escrito se torna para eles "simples" e "aprendido", mas falsamente aprendido. Mas, enfim, são exemplos muito interessantes de como resolver problemas um pouco mais complexos do que aqueles vistos até o momento. Por outro lado, a experiência mostra que duas aulas são suficientes para todo o conteúdo, dado que o aluno tenha de fato praticado sozinho em casa ou no laboratório, com o uso da ferramenta de correção automática. O capítulo traz 11 problemas:

- Inverter um número de três dígitos e sua generalização. Este é um problema interessante para mostrar que lidamos com problemas e não com instâncias de problemas. Desta forma, uma solução trivial para inverter um número de três dígitos é péssima para inverter números com outras quantidades de dígitos. Mostramos que usar a técnica dos acumuladores é boa neste caso. Mostramos como decompor um número da direita para a esquerda e processá-los com a conhecida técnica, ao mesmo tempo introduzindo multiplicações sucessivas por 10 para reconstruir o número invertido;
- Conversão para binário. Basicamente usa a mesma técnica anterior, mas também mostra como usar a definição de números binários, encontrar uma potência de 2 do número e trabalhar desta vez da esquerda para a direita;
- Cálculo do MDC por Euclides. Apresentamos o que talvez seja o primeiro algoritmo registrado na forma escrita. Antes apresentamos a definição de MDC e comentamos que a implementação do cálculo pela definição é difícil e complexa. A elegância e eficiência do método de Euclides é discutida;
- Tabuada e Fatorial. Ambos tratam de introduzir laços duplos, mas com um destaque: enquanto o primeiro problema adapta-se facilmente de um laço simples para duplo, para mostrar a evolução da tabuada de um único número para a de muitos números, o mesmo não ocorre para o segundo problema, no qual a aplicação da mesma técnica gera um algoritmo ineficiente. Mostramos esta questão e apresentamos um algoritmo com laço único que resolve bem o problema;
- Revisitamos os números de Fibonacci para reforçar alternativas para parada de uma repetição. Aproveitamos para comentar sobre a beleza desta sequência e mostramos o número Áureo. Neste ponto exploramos a sequência como forma também de preparar o estudante para a aula sobre séries;
- Séries. A partir deste ponto são algoritmos mais complexos. Embora as soluções para séries usem laços simples, a transformação de um termo em outro não é óbvia e exige muito cuidado na hora da apresentação. Por isso decidimos mostrar como a solução para a série Harmônica, mais simples, pode evoluir para a implementação do cálculo do seno;
- Maior segmento crescente. Difícil e interessante problema que é cuidadosamente explicado com o objetivo de mostrar a aplicação de diversas técnicas estudadas, uso de acumuladores, chutar e depois corrigir, dentre outras;
- Primos entre si. Serve para depois motivarmos o uso de funções em capítulo subsequente. É uma aplicação imediata do método de Euclides;
- Números primos. Considerado por muitos o problema mais difícil desta fase do curso. Um dos objetivos é mostrar como deixar um pouco mais eficiente a solução que é baseada em resto de divisão por números. O algoritmo evolui do trivial até uma solução que vai até a raiz quadrada do número testando basicamente só os ímpares e porquê isso funciona. O único primo par é o 2.

Apresentamos o algoritmo de Euclides no Código 4 pois além de ser um ícone dos algoritmos, ele nos servirá mais à frente.

O algoritmo que determina todos os números que são primos entre si presente em um intervalo de dois valores dependem do

```
program mdcporeuclides;
var a, b, resto: integer;

begin
    read (a,b);
    if (a <> 0) and (b <> 0) then
    begin
        resto:= a mod b;
        while resto <> 0 do
        begin
            a:= b;
            b:= resto;
            resto:= a mod b;
        end;
        writeln ('mdc = ', b);
    end
    else
        writeln ('o algoritmo nao funciona para entradas nulas.');
end.
```

Código 4: Algoritmo de Euclides para cálculo do MDC

algoritmo que calcula se o MDC. Este ponto será importante mais a frente, quando falarmos sobre funções e procedimentos. Este algoritmo está no Código 5 e tem este destaque nos comentários do Código 4. Comentamos com os estudantes que este código será melhor escrito quando o assunto de modularidade for abordado.

```
program primosentresi;
var i, j, a, b, resto: integer;
begin
    i:= 2;
    while i <= 100 do
    begin
        j:= i;
        while  j <= 100 do
        begin
            a:= i; b:= j;          (* inicio bloco euclides  *)
            resto:= a mod b;       (*         *              *)
            while resto <> 0 do    (*         *              *)
            begin                  (*         *              *)
                a:= b;             (*         *              *)
                b:= resto;         (*         *              *)
                resto:= a mod b;   (* termino bloco euclides *)
            end;
            if b = 1 then
                writeln (i,j); (* se o mdc = 1 *)
            j:= j + 1;
        end;
        i:= i + 1;
    end;
end.
```

Código 5: Gerando todos os primos entre si de 1 a 100

Destacamos que todos os problemas tratados usam números inteiros, raramente booleanos e quase nunca números reais, a menos das situações de cálculos de médias, por exemplo. Assim eles não têm que se preocupar muito com formatação de saída.

3 ESTRUTURA DO LIVRO - PARTE 2

A parte 2 pode ser vista em duas subpartes, a que trata de funções, procedimentos e vetores, e o restante do livro, que aborda matrizes, uma aplicação em jogos e tipos abstratos de dados.

3.1 Funções e procedimentos

Discutimos as noções de subprogramas em *Pascal* e três conceituações importantes: funções ou procedimentos; variáveis globais ou locais; e passagem de parâmetros por valor ou por referência.

É importante mencionar que lidaremos com os mesmos problemas de antes, nenhum novo é proposto, o foco é nos três novos conceitos para que o estudante se sinta seguro e aprenda com propriedade quando deve usar funções ou procedimentos, quanto deve usar variáveis globais ou locais e como é a passagem parâmetros por valor ou por referência.

Destacamos que o modelo Von Neumann ajuda neste momento quando falamos de passagem de parâmetros por referência.

Os problemas são muito simples, como paridade, uso do método de Euclides, o método de Bhaskara, todos já conhecidos, mas agora usando funções. O foco é a modularidade e a clareza no código. O novo conhecimento é o destaque, por isso trabalhamos exatamente as mesmas técnicas com problemas conhecidos.

Apresentamos aqui a refatoração do problema de determinar se dois números são primos entre si, desta vez com a novidade sendo as funções. Lembremos do algoritmo apresentado no Código 5 e agora em sua nova forma de função, o que é apresentado no Código 6.

```
program primosentresi_v2;
var i, j: integer;

function primos_entre_si (a, b: integer): boolean;
var resto: integer;
begin
    primos_entre_si:= false;
    if (a <> 0) and (b <> 0) then
    begin
        resto:= a mod b;
        while resto <> 0 do
        begin
            a:= b;
            b:= resto;
            resto:= a mod b;
        end;
        if b = 1 then
            primos_entre_si:= true;
    end;
end;

begin
    i:= 2;
    while i <= 100 do
    begin
        j:= i;
        while  j <= 100 do
        begin
            if primos_entre_si (i,j) then
                writeln (i,j);
            j:= j + 1;
        end;
        i:= i + 1;
    end;
end.
```

Código 6: Gerando todos os primos entre si revisitado

Optamos por apresentar a solução do cálculo do MDC pela definição por dois motivos: primeiramente porque é possível implementar um código complexo que se torna mais legível com o uso de funções/procedimentos. Em segundo lugar para mostrar que, neste caso, existe um algoritmo muito melhor: o de Euclides. "Nem tudo

que é possível é bom" é a mensagem que fica de recado. Um último comentário: a experiência mostra que iniciar com funções é melhor.

3.2 Vetores

Aqui a nossa "didática da novidade" começa a ficar mais clara. Ela tem base na pergunta: Qual é a novidade[1]? A novidade são os vetores, então usamos os mesmos problemas já conhecidos, tais como somar vários números ou encontrar o menor de vários números.

Temos que ter foco na diferença entre índice e conteúdo acessado pelo índice. Temos que ter foco também na sintaxe, que usa colchetes para acessar os conteúdos. Tudo isso já é suficientemente complexo. Por isso, após uma breve apresentação conceitual, trabalhamos com leitura e impressão de vetores, incluindo imprimir ao contrário. Neste momento achamos que os números reais são relevantes, para reforçar a diferença entre índice e conteúdo.

Esta diferença é amplamente explorada, imprimindo-se os elementos pares, depois os conteúdos dos índices pares, e assim por diante. Resgatamos o já conhecido algoritmo de somar vários números, de encontrar o menor, todos agora vistos usando-se vetores. Apresentamos versões com e sem o uso de funções/procedimentos. Sempre fiéis à novidade, que são os vetores.

Um exemplo desta técnica é a nova forma do algoritmo apresentado no Código 3, agora com vetores e funções, o que pode ser visto no programa apresentado no Código 7.

```
function menor_dos_lidos (var v: vetor_r; n: integer): real;
var i: integer; menor: real;
begin
    menor:= v[1];
    for i:= 2 to n do
        if v[i] < menor then
            menor:= v[i];
    menor_dos_lidos:= menor;
end;
```

Código 7: Achar o menor de N números lidos, com vetor

Quando a novidade foi absorvida pelos estudantes o processo ganha outra dimensão. Podemos iniciar uma nova fase da disciplina que pode discutir intuitivamente noções de complexidade, algoritmos lineares, quadráticos, logarítmicos.

Este processo se inicia com problemas simples em vetores, tais como somar elementos de vetores, fazer produto escalar (fundamental para multiplicação de matrizes, em aula futura) e demais problemas simples, para quem entendeu vetores.

A partir deste ponto, normalmente um pouco antes da aula 20, na qual ocorre a segunda prova, pode-se iniciar o que chamamos de uma disciplina de Ciência da Computação de verdade. De fato, mostramos através de problemas simples, tais como inserir, remover ou buscar elementos em vetores, que as operações têm algum custo. Definitivamente não falamos nada relacionado com notações do tipo O, Ω nem Θ, mas somente noções intuitivas de que os processos podem estar relacionados com o próprio tamanho do vetor, em funções lineares ou quadráticas.

Iniciamos de forma intuitiva o estudo do que significa melhor caso, pior caso e caso médio, enfatizando que focaremos no pior caso, que é mais fácil de entender. O conteúdo das aulas se torna um pouco mais ambicioso, aproveitando o entusiasmo da turma, que já entendeu que a Computação é mais do que escrever desvios e repetições, que os algoritmos podem ser eficientes e elegantes.

Mostramos o caso dos vetores ordenados e revisitamos os problemas da inserção, remoção e busca, chegando até na busca binária e enfatizando o seu comportamento logarítmico. É muito interessante observar que neste momento do curso "as fichas caem", para usar uma expressão popular.

Comparamos as complexidades de inserir, remover e buscar em vetores com as duas estruturas de dados, vetores ordenados e não ordenados. Aproveitamos para motivar para a nossa disciplina de algoritmos 3 (no 3o período), que trata justamente de algoritmos eficientes para estes três problemas.

Antes de finalizar, mostramos um único método de ordenação, o método da seleção. Muito antigamente mostrávamos também o da inserção e o da bolha, mas os abandonamos. O principal motivo é relacionado com a disciplina de algoritmo 2 (no 2o período), que tem praticamente metade de seu conteúdo em algoritmos diversos de ordenação. Poderíamos ter optado por mostrar outro método quadrático, como o da inserção ou o da bolha.

Tudo isso serviria para terminar esta parte da disciplina, mas frequentemente ministramos uma aplicação de vetores no problema matemático das permutações. Casualmente, um dos autores procurava uma aplicação e encontrou esta. A aula foi fantástica, com muitos estudantes sugerindo vários algoritmos diferentes para este problema e rendeu duas boas aulas. Na verdade este estudo nasceu de um problema de uma competição da maratona de programação da SBC/ACM que envolvia o cálculo eficiente da ordem de uma permutação. Mais curiosamente ainda, a solução envolve o nosso conhecido algoritmo de Euclides.

Terminamos a parte 2 do conteúdo e estamos aptos a entrar na parte 3, já tendo aplicado a prova 2, mais ainda nos restam 9 aulas antes da prova derradeira, a prova 3. Claro que o número de aulas pode ser aumentado ou diminuído, isto depende muito do sentimento dos professores sobre o andamento da turma, e neste caso, graças à ferramenta de correção automática, podemos ter este sentimento de uma forma um pouco mais concreta.

4 ESTRUTURA DO LIVRO - PARTE 3

Esta parte é opcional, como dissemos poderíamos ter terminado na parte 1 ou na parte 2. Tudo é uma questão de analisar as turmas a cada semestre letivo. Mas frequentemente vamos até o final da parte 3, que se inicia com matrizes, que é uma x generalização de vetores, registros, rapidamente alguns tipos abstratos de dados e o desenvolvimento de um jogo simples que é baseado em matrizes. Tudo isso parece muito ambicioso, mas a leitura do texto que segue pode ajudar a entender como é possível.

4.1 Matrizes

Iniciando com matrizes, retomamos os mesmos problemas já abordados. Matrizes é muito fácil de aprender para quem já aprendeu vetores. Basta lidar com um índice a mais.

O foco agora é no conceito de matrizes, então, novamente, lemos, imprimimos, somamos, multiplicamos (já tendo visto produto escalar), achamos o menor elemento e todos os outros algoritmos

[1] Gilberto Gil disse que a novidade era o máximo.

exaustivamente já vistos anteriormente, para focar na novidade. Apresentamos abaixo o Código 8 que é a nova forma dos algoritmos 3 e 7. Notem que o comando `for` apareceu agora.

```
function acha_menor_matriz (var w: matriz; n,m: integer): integer;
var i, j: integer;
    menor: integer;
begin
    menor:= w[1,1];
    for i:= 1 to n do
        for j:= 1 to m do
            if w[i,j] < menor then
                menor:= w[i,j];
    acha_menor_matriz:= menor;
end;
```

Código 8: Encontrando o menor elemento de uma matriz

Decidimos iniciar uma parte lúdica, mas repleta de aprendizado e bastante apreciada pelos discentes. Mostramos o formato de imagens PGM e como lidar com problemas que nos permitem: clarear, fazer *zoom* e obter as bordas de uma imagem.

Para clarear basta somar um valor fixo nos elementos da matriz; Fazer *zoom* consiste em técnicas de percorrer submatrizes e fazer médias dos vizinhos; Finalmente, para obter as bordas o mais complicado é convencer os estudantes da noção de gradiente.

Uma das etapas do processo de *zoom* é encontrar o maior valor na matriz de pixels. O algoritmo é exatamente o mesmo do encontrar o menor a menos da desigualdade, que troca o < pelo >, mantendo a mesma técnica apresentada nos algoritmos dos Códigos 3, 7 e 8. O resultado é o algoritmo apresentado no Código 9.

```
function maior_valor (var O: imagem; l,c: integer): integer;
var i,j, m: integer;
begin
    m:= O[1,1];
    for i:= 1 to l do
        for j:= 1 to c do
            if O[i,j] > m then
                m:= O[i,j];
    maior_valor:= m;
end;
```

Código 9: Cálculo do valor do maior pixel

Nós trabalhamos com imagens reais e mostramos como tudo funciona bem. Isso permite depois generalizar a busca por elementos para problemas do tipo "onde está Wally", que nada mais é que o mesmo problema de busca, mas desta vez por uma submatriz. Tudo ocorre de forma natural e lúdica.

4.2 Registros

O assunto de registros, em *Pascal* `record`, está presente em várias obras da literatura básica de introdução aos algoritmos. A novidade é ter cuidado com a notação, que não usa mais `[]`, e sim usa um ponto `.` para acessar os elementos do registro.

Optamos por mostrar duas formas de uso de registros e vetores simultaneamente: na primeira, registros que têm vetores como um campo; e na segunda, vetores que têm registros como elementos.

A grande dificuldade curiosamente não são os algoritmos, que são sempre os mesmos de sempre, localizar o menor por exemplo. Mas no lugar de procurar o menor inteiro em um vetor de inteiros, mostramos como encontrar o menor saldo presente em um vetor de registros de clientes. O ponto delicado é mostrar aonde tem que colocar `[]` e aonde tem que colocar `.` (ponto). Em outras palavras, é uma mera questão de *notação*.

4.3 Tipos abstratos de dados

Temos vetores de registros e registros com vetores. Isto nos permite apresentar o conceito de tipos abstratos de dados (TAD). Observamos que não vamos apresentar nenhum algoritmo novo, todos os problemas vão usar algo já estudado, que é inserir, remover, buscar em vetores. A novidade é a abstração.

Mostramos o TAD pilha, que é o mais simples, pois insere na última posição, retira da última posição, controla o tamanho.

É tão curioso que apesar de termos aumentado o nível de abstração, os algoritmos ficaram mais simples. De fato, na pilha não há busca, só inserção e remoção, sem necessidade de deslocamentos de elementos. Mas como já falamos de modularidade, de funções e de procedimentos, esta parte não é empecilho para ninguém.

Na verdade, o tipo pilha será usado na construção do jogo, no próximo e último capítulo do livro.

Mas para bem apresentar os tipos abstratos de dados, mostramos a noção de um conjunto matemático, uma coleção de elementos sem repetição. A novidade aqui é a não repetição, pois quase todos os algoritmos básicos já foram vistos no capítulo de vetores. É uma continuidade do aprendizado de vetores.

Por isso mostramos que apenas com os protótipos de funções, tais como intersecção e cardinalidade, é possível resolver o problema de encontrar o vencedor da megassena. Mostramos que também é possível resolver um problema de maratona de programação: encontrar uma celebridade, sem escrever código das funções.

Em seguida apresentamos duas formas de implementar conjuntos, uma usa vetores não ordenados e a outra usa vetores ordenados. Isso nos dá a possibilidade de comparar, intuitivamente, as complexidades dos mesmos procedimentos que usam estruturas de dados diferentes. Observamos que estes problemas já foram estudados antes quando tratamos de vetores. A diferença agora é o uso dos mesmos algoritmos, mas sobre o tipo abstrato de dados, que no fundo ainda é um vetor.

Na verdade, as implementações das operações de intersecção, união, diferença, dentre outras, são extremamente elegantes quando se usa vetor ordenado. Consideramos isso um grande ponto no livro, acreditamos que a implementação com vetores não ordenados é bastante simples e ao alcance dos estudantes, mas não resistimos a mostrar, e a comentar, o caso dos vetores ordenados.

4.4 Aplicação em jogos

Desta forma chegamos ao fim do curso com uma ou duas aulas de sobra para podermos fazer aulas práticas no laboratório, ou dependendo do interesse deles, implementar um jogo.

O livro mostra duas implementações de jogos, embora mostremos somente um deles. Elas usam o TAD pilha como parte da solução (o conhecido minas e um outro chamado *floodfill*) e existem para celulares nas lojas de aplicativos. Desenvolvemos a interface textual,

nada de bibliotecas gráficas, no máximo usamos a biblioteca CRC do *Pascal*, que tem um limpador de tela (`clrscr`) e um atraso na impressão dela (`delay`).

Mas já tivemos oportunidade, no passado, de implementar em no máximo duas aulas: Snake, PacMan, 2048 e 4 em linha, todos podem ser encontrados na Wikipédia.

A novidade são as funções que lidam com tela textual, como limpar a tela. A implementação do jogo é o resultado natural da aplicação de técnicas de manipulação de matrizes já vistas. Outra novidade é apresentação da técnica de desenvolvimento *top-down* utilizada para mostrar como escrever um código um pouco maior.

5 CONCLUSÃO

Apresentamos aqui a nossa forma de introduzir conceitos de algoritmos e programação para calouros na Universidade Federal do Paraná para os cursos de Ciência da Computação e de Informática Biomédica.

O nosso método é fruto de um grande esforço no aperfeiçoamento das nossas ideias ao longo de 30 anos. Muitos professores e professoras participaram das experiências e tentativas, incluindo especialmente egressos do nosso curso que se tornaram colegas e que hoje fazem parte da equipe. Isto é importante pois eles trazem a visão de quem foi aluno e teve aulas segundo esta proposta e por isso contribuem valiosamente para o aperfeiçoamento das ideias.

Ao mesmo tempo, as opiniões de quem vivenciou experiências em diversas universidades tanto brasileiras quanto de fora permitem que nosso método seja uma amálgama de outros métodos aplicados em diversos cursos mundo afora.

A proposta parece a primeira vista um pouco ambiciosa, mas procuramos mostrar ao longo deste artigo, não apenas que é possível atingir as metas sugeridas, mas também mostramos como fazer isso. Um ponto importante a destacar é a temporização sugeria na apresentação de novos conceitos, aliado ao fato de insistirmos nos mesmos algoritmos, dando foco na novidade da vez.

As aplicações em jogos são factíveis, desde que se tome bastante cuidado com a parte inicial. A parte lúdica envolvendo processamento de imagens atrai o estudante e o resto é mera consequência.

Uma das nossas maiores contribuições foi investir tempo no sequenciamento dos exercícios, quase todos eles encontrados na Internet, mas nós procuramos uniformizar os enunciados e adotar técnicas de redações destes a partir da metodologia seguida nas redações de problemas em maratonas de programação, principalmente mostrando casos de testes em praticamente todos os enunciados.

Nosso método também conta com a valorosa ferramenta de submissão de códigos, o *FARMA-ALG*, já citado.

É preciso deixar claro que o ensino introdutório de algoritmos e programação não é simples, esperamos ter dado alguma contribuição neste sentido a partir de nossas experiências. O importante a dizer é que os nossos egressos, normalmente, não têm geralmente maiores problemas em seguir as outras disciplinas subsequentes nos aspectos que envolvem programação.

Finalmente, esta sugestão proposta está encaixada em um projeto pedagógico de curso que evolui ao longo do tempo, mas o mais importante é que esta disciplina é considerada como um dos pilares do curso como um todo. Isto significa dizer que a grade curricular tem forte base no que temos feito aqui. Talvez seria importante como trabalho futuro discorrer um pouco mais sobre este ponto.

Um breve resumo das nossas principais ideias segue abaixo.

- Escolha da linguagem *Pascal*;
- Apresentação do modelo Von Neumann;
- Reserva de mais tempo para os conceitos introdutórios, omitindo em alguns momentos alguns conceitos, tais como *char* e *string* e alguns comandos, tais como *for*, *repeat* e *case*;
- Manter a opção por números (na maior parte dos casos inteiros e eventualmente reais) em praticamente todos os exercícios. A propósito, convertemos vários exercícios encontrados na literatura que tratavam de *strings* e os remodelamos em termos de inteiros;
- Apresentar repetições antes de desvios condicionais;
- Adiar a apresentação de laços duplos;
- Focar na novidade e usar os mesmos problemas em diversos contextos diferentes, como foi feito com o problema de encontrar o menor número de uma coleção dada como entrada;
- No momento oportuno, logo após vetores, discutir intuitivamente noções de complexidade de algoritmos;
- Apresentar problemas concretos sobre processamento de imagens, procurando de forma lúdica motivar o estudante;
- Discutir brevemente tipos abstratos de dados.

Como trabalhos futuros pretendemos analisar com mais rigor metodológico os pontos principais que propomos. Existe farta literatura que pode explorar melhor os aspectos teóricos e metodológicos para melhor embasar nossos pontos.

Outros trabalhos podem discorrer sobre o impacto do grande esforço feito com os exercícios, o sequenciamento deles, os casos de testes e a qualidade do texto dos enunciados. Isto pode ser feito em conjunto com um estudo do real impacto da ferramenta de correção automática que adotamos, o *FARMA-ALG*.

REFERÊNCIAS

[1] Sérgio Carvalho. 1982. *Introdução à Programação com Pascal.* Editora Campus.

[2] Marcos A. Castilho, Fabiano Silva, and Daniel Weingaertner. 2020. *Algoritmos e Estruturas de Dados I.* Universidade Federal do Paraná. https://www.inf.ufpr.br/marcos/livro_alg1/livro_alg1.pdf Licença Creative Commons BY-NC-ND.

[3] Lisbete Madsen Barbosa Dirceu Douglas Salvetti. 1998. *Algoritmos.* Makron Books.

[4] Harry Farrer et alii. 1999. *PASCAL Estruturado.* Editora Guanabara Dois. 3a edição Guanabara Dois.

[5] Alexander R. Kutzke and Alexandre I. Direne. 2015. Farma-alg: An application for error mediation in computer programming skill acquisition. *International Conference on Artificial Intelligence in Education*, 690–693.

[6] Alexander R. Kutzke and Alexandre I. Direne. 2018. Em Direção à Mediação do Erro por Meio de Um Arcabouço de Sistema Computacional. *Revista Brasileira de Informática na Educação* 26.03, 139.

[7] Marco Aurélio Medina and Cristina Fertig. 2006. *Algoritmos e Programação: Teoria e Prática.* Novatec.

[8] Jean-Paul Tremblay and Richard B. Bunt. 1983. *Ciência dos Computadores.* McGraw-Hill.

[9] Niklaus Wirth. 1978. *Programação Sistemática em PASCAL.* Editora Campus.