PCC170 - Projeto e Análise de Experimentos Computacionais

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

1 Boas práticas de implementação

Fonte

Este material é parcialmente baseado no relatório técnico

- Ian P. Gent, Stuart A. Grant, Ewan MacIntyre, Patrick Prosser, Paul Shaw, Barbara M. Smith and Toby Walsh. *How Not To Do It*. Research Report 97.27, School of Computer Studies, University of Leeds. May 1997.

Aviso

Este material é em grande parte baseado em minha experiência em pesquisa.

Particularmente, os tópicos de implementação consistem em sugestões aos alunos e refletem tão somente minha visão pessoal.

Licença

Boas práticas de implementação

How not to do it

Don't trust yourself: *bugs* acontecem, esteja atento;

Do it fast enough: seu código não precisa ser o mais rápido do mundo, precisa ser rápido o suficiente;

Do use version control: incluindo *backup*;

Do be paranoid: sempre verifique duas vezes;

Boas práticas de implementação

How not to do it

Do check your solutions: verifique manualmente a consistência das soluções de seus programas;

Do be stupid: não ignore idéias "bobas";

Don't trust your source of random numbers: muito cuidado com sementes, números cíclicos e experimentos em série;

Boas práticas de implementação

Sugestão de padronização

- Linguagem C/C++, padrão C++ 17;
- *flag* de compilação -O3 no g++;
- *flag* de compilação -march=native no g++;
- Utilização da STL em sua totalidade (contêineres, algoritmos e iteradores);
- Usar alocação dinâmica de contêineres;
- NUNCA use *goto*[a].

[a] Se você não sabe o que é, continue sem saber.

Boas práticas de implementação

Boas práticas de implementação

Programas híbridos C/C++

Apesar de programarmos em C++, podemos continuar programando de maneira estruturada e usando apenas os objetos que nos interessam da biblioteca padrão.

Não é necessário criar classes, *getters*, *setters*, nem mesmo separar o código em arquivos diferentes ou usar padrões de projetos.

Variáveis globais podem ser utilizadas, etc, menos *goto*.

O intuito é minimizar o *overhead* da implementação.

Esteja atento à complexidade das suas implementações também.

Entrada e saída

A primeira coisa a ser implementada em um método computacional é a entrada e saída do problema, padronizando-os.

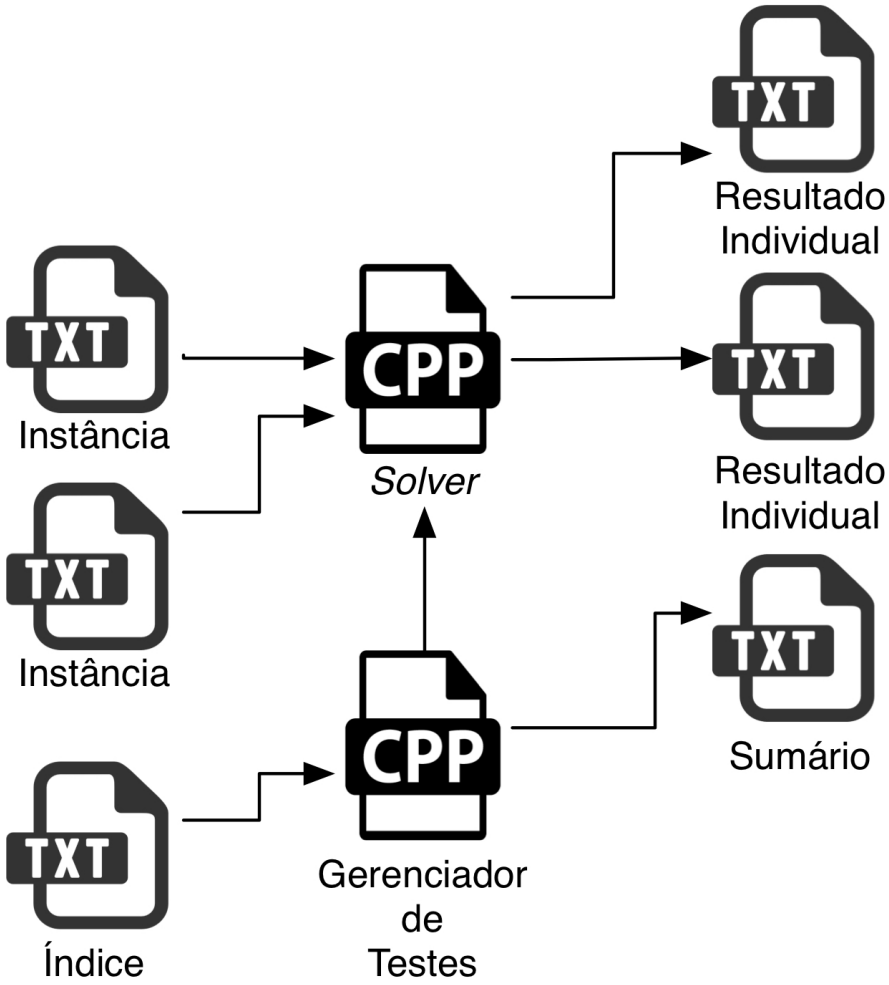
Note que isto independe de qual método será escolhido para solução do problema em si.

Para testes extensivos, crie um programa gerenciador de testes, que alimenta o *solver* com instâncias e colhe dados para o log sintético.

Um arquivo de índice contém o nome das instâncias a serem consideradas. O gerenciador invoca o solver para cada uma das instâncias.

O *solver* é o algoritmo em si, recebe uma instância (ou seu nome apenas), resolve o problema e gera arquivos de log individuais.

Análise do desempenho de métodos heurísticos



Esquema para realização de entrada de dados e geração de arquivos de log.

Validação

Todo componente do código deve ser validado.

Para algumas instâncias pequenas verifique **manualmente** se o resultado das funções está correto, como leitura da entrada, preenchimento de estruturas, cálculos, etc.

Para o caso geral, crie um programa validador de soluções que considere arquivos de log individuais.

Análise do desempenho de métodos heurísticos

Tradeoff entre memória e processamento

Processamento é mais caro que memória, também em termos de desempenho.

Todavia, movimentar grandes quantidade de dados exige esforço computacional.

Estruturas de dados espertas ajudam a ganhar em desempenho.

Análise do desempenho de métodos heurísticos

Tradeoff entre memória e processamento

Ao invés de copiar dados, crie ponteiros para os dados.

Operações pesadas e frequentes de manipulação de dados podem ser substituidas por dados duplicados, como índices reversos.

Esteja atento a complexidade assintótica de operações em estruturas, como remoções em *vector* e *list*.

Boas práticas de implementação

Inicialização e terminação

Todos os contêineres devem ser inicializados logo no início da execução do código e limpos ao final.

Estabeleça dinamicamente o tamanho de cada contêiner com base nos dados da instância e outros parâmetros.

Zere todas as variáveis numéricas.

Ao final, esvazie todos os contêineres e libere qualquer memória utilizada.

Boas práticas de implementação

Inicialização e terminação

Embora as próprias linguagens cuidem deste pontos ao construir e destruir objetos por exemplo, são comuns os erros ocasionados por lixo em variáveis globais em testes extensivos.

Nestes experimentos, normalmente o código binário é executado uma única vez, embora o algoritmo em si seja executado diferentes vezes por instância.

Desta forma, todas as variáveis e objetos permanecem na memória e podem ser reutilizados com valores antigos.

Boas práticas de implementação

Função de avaliação

A função de avaliação literalmente avalia o custo de uma solução calculada por um método.

Algumas são simples expressões matemáticas e outras envolvem algoritmos adicionais.

Ao implementar a função de avaliação, verifique manualmente se o cálculo está correto para algumas instâncias.

Teste para instâncias de conjuntos diferentes e com características diferentes.

Boas práticas de implementação

Função de avaliação

Em métodos baseados em buscas locais, é essencial implementar uma **função de avaliação incremental**.

Dados uma solução s e um movimento m, uma função $\Delta(s, m)$ avalia somente a transformação da solução s após a aplicação do movimento m, utilizando informações anteriores.

Desta forma, a avaliação é mais leve e consome menos tempo, porém, estas vantagens são obtidas ao preço de maior consumo de memória.

Boas práticas de implementação

Função de avaliação

Há um exemplo clássico para o método *2-opt*[a].

Na impossibilidade de uma avaliação incremental, uma alternativa é uma avaliação aproximada, que seja mais rápida, porém, imprecisa.

---

[a] Talbi, El-Ghazali. Metaheuristics: from design to implementation. Vol. 74. John Wiley & Sons, 2009.

Boas práticas de implementação

Pré-processamento

Métodos de pré-processamento são aplicados antes da resolução uma instância e tem por finalidade reduzir o tamanho da instância pela remoção de informações redundantes ou irrelevantes e também detectar estruturas específicas que facilitem a solução de um problema.

Boas práticas de implementação

Pré-processamento

Modelar um problema usando grafos é uma boa maneira de detectar estruturas específicas.

Por exemplo, problemas classificados como NP-difícil podem ter solução em tempo determinístico polinomial caso o grafo resultante seja uma árvore, 1-árvore, grafo completo, etc.

Outro exemplo, mais comum ainda, ocorre quando a solução do problema é realizada por uma busca no grafo: se o grafo possuir mais de um componente, então a instância pode ser dividida em duas ou mais de tamanho menor, em que cada problema é um dos componentes do grafo.

Boas práticas de implementação

Pré-processamento

Verifique na sua revisão da literatura se algum método de pré-processamento já foi utilizado.

Novos métodos de pré-processamento constituem contribuições científicas relevantes.

Geração de números aleatórios

A biblioteca *random* da STL possui geradores de diversas distribuições de números.

Para gerar um número dentro de um intervalo, normalmente utilizamos a distribuição uniforme[a] ou normal[b].

Entretanto, verifique qual distribuição deve ser utilizada (bernoulli, binomial, geométrica, poisson, exponencial, etc).

Na dúvida, utilize o *engine* padrão *default_random_engine generator*.

---

[a] Veja o exemplo: `https://bityli.com/zOfac`
[b] Veja o exemplo: `https://bityli.com/TkNpd`

Seleção aleatória de elementos

No caso de precisar selecionar aleatoriamente elementos de um conjunto (um vetor por exemplo) sem poder repetí-los, utilize a função *shuffle*.

Esta função embaralha um vetor aleatoriamente. Depois, simplesmente o percorra linearmente[a].

Não confundir com *random_shuffle*.

Na dúvida, utilize o *engine* padrão *default_random_engine generator*.

[a] Veja o exemplo em `https://bityli.com/jpszj`

Boas práticas de implementação

Seleção aleatória de tuplas de elementos

O mesmo princípio se aplica à seleção de pares, trios ou tuplas de elementos.

Preencha o vetor com as tuplas, embaralhe-o e depois, simplesmente o percorra linearmente.

O consumo de memória aumenta, mas o tempo de execução não.

Tomada de tempo

Verifique qual é o padrão de tomada de tempo dos outros métodos utilizados em seus experimentos.

Normalmente, o tempo de execução não engloba entrada e saída de dados, somente a execução do algoritmo em si.

Porém, se seu código realiza pré-processamento ao ler a entrada, este o tempo deve ser considerado, uma vez que seu algoritmo de solução já está sendo executado.

Use o objeto *high_resolution_clock* da biblioteca *chrono* da STL[a].

[a] Veja o exemplo em `https://bityli.com/SCSxe`

Boas práticas de implementação

Limitante inferior para a solução

Um limitante inferior é um valor de referência, possivelmente não viável, que é menor ou igual ao valor da solução ótima, no caso de um problema de minimização.

Normalmente pode ser obtido por meio da relaxação de alguma restrição.

Caso haja um limitante inferior trivial para o valor da solução, de acordo com a sua revisão da literatura, implemente-o.

Boas práticas de implementação

Limitante inferior para a solução

Em métodos que buscas locais ou fases de aprimoramento (*polishing*) são aplicadas sucessivamente, um critério de parada comum é quando a solução obtida for igual ao limitante inferior.

Nesse caso, a solução obtida é comprovadamente ótima.

Boas práticas de implementação

How not to do it

Do check your health regularly: verifique a consistência das suas soluções em relação às da literatura;

Do look at the raw data: dados resumidos escondem os fatos, dê uma olhada nos dados brutos;

Do look for good views: procure um ponto de vista interessante para os seus dados;

Do report negative results: não varra a sujeira para debaixo do tapete ou ajuste suas conclusões de acordo com os dados.

Leitura recomendada

- Ian P. Gent, Stuart A. Grant, Ewan MacIntyre, Patrick Prosser, Paul Shaw, Barbara M. Smith and Toby Walsh. *How Not To Do It*. Research Report 97.27, School of Computer Studies, University of Leeds. May 1997. Disponível em `https://ipg.host.cs.st-andrews.ac.uk/papers/97_27.ps.gz`.

Dúvidas?