2ª Aula Prática – Algoritmos gananciosos/Algoritmos de retrocesso

Instruções

- Faça download dos ficheiros **cal1516_fp02.zip** da página da disciplina e descomprima-o (contém os ficheiros **Labirinth.h, Labirinth.cpp**, **Sudoku.h, Sudoku.cpp** e **Test.cpp**).
- Abra o eclipse e crie um novo projecto C++ do tipo Cute Project (File/New/C++ Project/Cute Project) com o nome CalFp02 e usando o MinGW GCC.
- Inclua os *headers* da biblioteca Boost no projeto.
- Importe para a pasta *src* do projecto, os ficheiros extraídos: (Import/General/File System)
    - Aparecendo a mensagem a perguntar se quer fazer Overwrite ao ficheiro Test.cpp diga sim.
    - Compile o projecto.
    - Execute o projecto como CUTE Test (Run As/CUTE Test). Se surgir a pergunta de qual compilador usar, escolha MinGW gdb.
- ***Deverá realizar esta ficha respeitando a ordem das alíneas.*** Poderá executar o projecto como CUTE Test quando quiser saber se a implementação que fez é suficiente para passar no teste correspondente.
- Considere implementar as várias soluções baseando-se nas indicações dos ficheiros ".h" respectivos.

Enunciado

1. Labirinto *(Labirinth.h, Labirinth.cpp)*

Pretende-se encontrar a saída de um labirinto de 10 por 10. A posição inicial é sempre nas coordenadas (1, 1). O tabuleiro é representado por uma matriz de inteiros de 10 por 10, em que 0 indica uma parede, 1 um espaço livre e 2 a saída.

a. Implemente a rotina *findGoal* (ver Labirinth.h e Labirinth.cpp) que descobre o caminho para a saída usando algoritmos de retrocesso. A função deve-se chamar recursivamente até descobrir o caminho até à solução. Em cada ponto de decisão no labirinto, só é possível andar para a esquerda, direita, baixo ou cima (ver Labirinth.h e Labirinth.cpp). Quando a saída for encontrada deve imprimir uma mensagem de sucesso. Sugere-se a implementação de uma matriz de casas visitadas.
b. Qual é a eficiência temporal do algoritmo, no pior caso?

2. Sudoku *(Sudoku.h, Sudoku.cpp)*

O objectivo do Sudoku [http://en.wikipedia.org/wiki/Sudoku] é completar uma matriz 9×9 com números de 1 a 9, sem repetir números em cada linha, coluna ou bloco 3×3.

a. Implementar a rotina *solve()* por forma a resolver automaticamente (e eficientemente) Sudokus de qualquer grau de dificuldade, com base num algoritmo com retrocesso (ver esqueleto de algoritmo com retrocesso nas aulas teóricas). O algoritmo deve funcionar de forma recursiva, isto é, cada

chamada deve procurar preencher uma célula e deve chamar-se recursivamente para preencher o resto. Seguir o seguinte algoritmo ganancioso para escolher a célula a preencher: escolher uma célula não preenchida que aceite um número mínimo de nºs candidatos (idealmente 1).

b. Desenvolver e testar uma rotina capaz de determinar a multiplicidade de soluções de um Sudoku (sem solução, com uma só solução, com mais do que uma solução). Sugestão: adaptar a rotina *solve()*.

c. Desenvolver e testar uma rotina capaz de gerar automaticamente Sudokus. Sugestão: começando num sudoku vazio, escolher aleatoriamente uma célula e um nº; se a célula não estiver preenchida e aceitar o nº gerado, preencher a célula e analisar a multiplicidade de soluções do novo sudoku usando a rotina da alínea anterior; se o sudoku se tornar impossível, voltar a limpar a célula; se o sudoku passar a admitir uma única solução, terminar; caso contrário continuar o processo.

3. Escalonamento de atividades

Em relação ao problema de escalonamento de atividades abordado nas aulas teóricas (ou seja, o problema de selecionar um número máximo de atividades não sobrepostas de um conjunto de atividades com uma hora de início e fim definida para cada atividade), provar que o critério ganancioso de "início mais tarde" (isto é, começar por selecionar a atividade com o início mais tarde) garante a solução ótima.