Instituto Superior Técnico – Universidade Técnica de Lisboa
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Ano Lectivo 2012 / 2013
Lisboa, 12 de Dezembro de 2012
Algoritmos e Estruturas de Dados

ALGORITMOS E ESTRUTURAS DE DADOS

Relatório do Projecto

Cross Sums

Grupo nº24

Maria Margarida Reis
nº 73099 ; e-mail: margarida.reis@ist.utl.pt
Ricardo Filipe Amendoeira
nº 73373 ; e-mail: ricardofilipeamendoeira@gmail.com

Índice

Descrição do problema

Neste projecto foi-nos pedido que desenvolvêssemos um programa que fosse capaz de resolver um puzzle comummente designado de *kaku*ro ou *cross sums*.

Neste tipo de puzzle tem-se uma grelha de dimensões variáveis com células de três tipos: brancas, pretas e triangulares. O objectivo ao longo do jogo é preencher as células brancas com números de 1 a 9 de forma que a soma de um conjunto de células brancas seguidas corresponda ao valor inserido na célula triangular correspondente. As somas podem ser feitas na vertical e na horizontal, sendo que se forem na horizontal a triangular correspondente é a que está à esquerda do conjunto de brancas e se forem na vertical a triangular correspondente é a que está por cima do conjunto de brancas. No primeiro caso o valor da soma é o que está na metade triangular superior e no segundo caso na metade triangular inferior. De referir ainda que em cada conjunto de células brancas só pode haver uma instância de cada número de 1 a 9.

Para o projecto, com o fornecimento de um ficheiro de entrada contendo a configuração do puzzle o esperado é que o programa o analise e gere de seguida um ficheiro de saída que para além de conter um esquema do puzzle contenha também a solução deste. Se o puzzle tiver mais que uma solução só é necessário apresentar uma e se não tiver de todo solução o ficheiro de saída deve ser igual ao de entrada.

Abordagem e resolução do problema

Numa primeira fase decidiu-se representar todo o puzzle em duas matrizes - uma que guarda, em cada entrada, as informações pertinentes para cada célula, variando estas com o facto de ser uma branca, uma preta ou uma triangular e outra matriz que longo da resolução do puzzle armazena os valores já calculados correctamente para cada branca. De seguida decidiu-se que o método mais eficaz de começar a resolver o puzzle seria arranjar o máximo de restrições entre as somas e entre as várias células brancas de modo a tentar resolver o maior número possível de brancas sem ser necessário recorrer a métodos de tentativa-e-erro. As várias restrições foram feitas com base em algoritmos que foram implementados de maneira recursiva pois percebeu-se que seria a melhor maneira de ao resolver uma casa conseguir resolver outra e assim sucessivamente.

Porém, a certa altura ao correr o programa para vários puzzles uns estavam inteiramente resolvidos e outros não e percebeu-se que com as restrições já não era possível avançar mais na resolução destes e tinha de se recorrer a um método de tentativa-e-erro. Na aplicação deste método ao tentar resolver um puzzle com a escolha

de um número possível numa dada branca, antes de se avançar de novo com outra branca com outro número tentado decidiu-se que recorrer aos algoritmos recursivos em primeiro lugar seria a melhor hipótese e só se a resolução do puzzle ficasse encravada de novo se recorreria outra vez a esse método. Depois de ter feito uma versão deste método que funcionava e resolvia todos os puzzles propostos chegou-se à conclusão que não era eficiente devido ao tempo de resolução e à memória que ocupava para resolver alguns puzzles e procedeu-se a uma alteração. Acabou por se decidir que era melhor fazer uma escolha estratégica das casas onde se faz tentativa-e-erro, com o objectivo de melhor a eficácia do programa. Este método e esta escolha estratégica serão explicados mais à frente no relatório em detalhe.

Por fim, tomou-se esta abordagem de tentar arranjar o maior número de restrições para ajudar na resolução do puzzle com o objectivo de melhorar o tempo de execução deste pois com tentativa-e-erro leva mais tempo e por vezes pode nem ser necessário se se conhecer bem as restrições do puzzle que ajudam a dizer directamente qual o número que deve ser em cada branca.

Arquitectura do programa

O fluxograma da função *main* segue em anexo (Anexo 1).

O fluxograma mostra a arquitectura do programa e como se vai resolvendo o puzzle através de várias funções que são responsáveis pela chamada de outras funções, resumindo quais os objectivos ao longo dos vários passos que se vai tomando.

Ao longo do relatório vai-se explorar melhor o que fazem as várias funções, e como ajudam a resolução do problema, quais as estruturas de dados utilizadas e como estas são manipuladas ao longo do programa.

Descrição das estruturas de dados

As estruturas de dados utilizadas para o programa têm por base matrizes a duas dimensões, todas com tamanho (*altura * largura*) sendo que o valor destas variáveis é obtido através da leitura das duas primeiras linhas do ficheiro de entrada. Assim, só depois de efectuado esse passo se declararam as matrizes no programa, pois dependendo de cada puzzle o seu tamanho é dinâmico. Usou-se também uma fila que é útil para o caso do método de tentativa-e-erro ter de ser aplicado.

tipo	matriz de inteiros
declaração no programa	int matriz[altura][largura]
função	representação gráfica do tabuleiro de jogo na forma de uma matriz sendo esta utilizada depois como auxílio na criação de outras matrizes
valor na entradas	→ 0 – representa uma célula preta → (-1) – representa uma célula branca por preencher → outro número entre 1 a 4 algarismos – representa uma célula triangular e este número é o que se lê no ficheiro de entrada para estas, não fazendo ainda a distinção entre o valor da soma superior e inferior

tipo	matriz de inteiros
declaração no programa	int matriz_solucao[altura][largura]
função	armazenar ao longo da resolução do puzzle o valor que as casas brancas tomam e é feita também como uma representação gráfica do tabuleiro de jogo, tendo a sua criação sido feita com recurso à estrutura struct matriz
valor nas entradas	→ 0 – representa até ao momento da sua solução uma célula branca de valor a definir, pretendendo-se assim que no final do programa não haja zeros nesta matriz → (-1) – representa uma célula preta ou triangular, nesta matriz não se faz distinção entre estas células → algarismo de 1 a 9 – representa uma célula branca já resolvida

tipo	estrutura
declaração no programa	struct triangular
função	armazenar todos os dados considerados relevantes em relação a uma célula triangular sendo de notar que se uma triangular tem soma inferior e superior então será representada por duas estruturas deste tipo
valor nas entradas	→ int soma – soma associada a essa triangular → char orientacao – ‘h’ se for uma soma feita na horizontal (triangular superior) ou ‘v’ se for uma soma feita na vertical (triangular inferior) → int num_casas – número de casas brancas que constituem uma dada soma → int pos_y – posição no eixo Oy da célula (em função do índice dela

	na vertical na estrutura struct matriz) → int pos_x – posição no eixo Ox da célula (em função do índice dela na horizontal na estrutura struct matriz)

tipo	estrutura
declaração no programa	struct branca
função	armazenar todos os dados considerados relevantes em relação a uma célula branca
valor nas entradas	→ int valores[9] – vector de inteiros de tamanho 9 que irá armazenar os valores possíveis que essa branca pode tomar tendo em conta a(s) soma(s) de que faz parte, sendo que se um valor for possível coloca-se um 1 nesse vector e um 0 se for impossível (o vector actuará como uma máscara então) → triangular * superior – ponteiro para a estrutura struct triangular que vai armazenar os dados da triangular superior que tem a branca como parte da sua soma → triangular * inferior – ponteiro para a estrutura struct triangular que vai armazenar os dados da triangular inferior que tem a branca como parte da sua soma → int pos_y – posição no eixo Oy da célula (em função do índice dela na vertical na estrutura struct matriz) → int pos_x – posição no eixo Ox da célula (em função do índice dela na horizontal na estrutura struct matriz)

tipo	estrutura de ponteiros para estruturas
declaração no programa	struct celula_matriz
função	guardar os dados de cada célula através de várias estruturas, sendo que se for uma branca os dois ponteiros relativamente a células triangulares serão NULL, se for uma preta serão todos NULL, se for uma triangular superior apenas esse ponteiro será diferente de NULL, se for uma triangular inferior apenas esse ponteiro será diferente de NULL e se for uma triangular dos dois tipos apenas os dois ponteiros relativamente a células triangulares serão diferentes de NULL

valor nas entradas	→ triangular * triangulo_superior – ponteiro para a estrutura struct triangular que guarda os dados da triangular superior → triangular * triangulo_inferior – ponteiro para a estrutura struct triangular que guarda os dados da triangular inferior → branca * casa – ponteiro para a estrutura struct branca que guarda os dados da célula branca

tipo	matriz de estruturas
declaração no programa	celula_matriz matriz_estruturas[altura][largura]
função	representar mais detalhadamente o puzzle de uma maneira a auxiliar a resolução deste, sendo que esta matriz foi construída com recurso à estrutura struct matriz
valor nas entradas	cada entrada é do tipo struct celula_matriz que já foi explicada anteriormente

tipo	fila
declaração no programa	struct fila_casas
função	armazenar as coordenadas (índices na vertical e na horizontal das matrizes) das brancas em que se chama a função responsável pela aplicação do método de tentativa-e-erro
valor nas entradas	→ int i – coordenada no eixo Oy da branca a que se vai aplicar o método de tentativa-e-erro → int j – coordenada no eixo Ox da branca a que se vai aplicar o método de tentativa-e-erro → struct fila_casas * anterior – ponteiro para o elemento do mesmo tipo (struct fila_casas) que é anterior ao elemento em questão

Optou-se pela utilização das matrizes por serem fácil de manipular e permitirem uma boa visualização espacial do puzzle, sendo que o objectivo da matriz é, ao nunca ser alterada, poder-se sempre ter acesso à configuração do puzzle como estava no início do programa. Assim, quando se quer verificar se a matriz de solução está certa pode-se recorrer à primeira que tem os valores de cada soma e basta verificar se os valores das brancas coincidem. Em relação às estruturas que definem as peças brancas e as

triangulares decidiu-se que deviam ser diferentes pois as informações que se quer guardar para cada caso são diferentes.

A fila foi escolhida pois da maneira como se implementou a função que aplica o método de tentativa-e-erro só vai interessar, sempre, aceder ao último elemento desta e uma pilha com estas características é uma fila.

Descrição dos algoritmos

Os principais algoritmos utilizados ao longo do programa prendem-se com as restrições aplicadas no puzzle e com o método de tentativa-e-erro. Inicialmente, depois de criada a estrutura struct matriz vai-se criar a estrutura struct matriz_estruturas sendo que esta já é mais detalhada e já vai conter dados como o vector com os números possíveis para cada branca. Assim, dentro da função que cria a matriz de estruturas chama-se o primeiro algoritmo (que será responsável por determinar essas hipóteses de cada branca) assim como todos os outros que vão sendo chamados.

- cruza_somas

O objectivo deste algoritmo é para uma dada branca analisar a pista vertical (valor da soma inferior) e a pista horizontal (valor da soma superior) e a partir de cada uma delas ver quais que os valores que ela poderia tomar e então cruzar essas duas hipóteses e ver a que hipóteses poderá a casa ter ficado reduzida.

Algoritmo:

para cada casa

para a pista vertical

ver qual o valor da soma, ver qual o número de casas brancas que constituem essa soma e ver quais as hipóteses possíveis para essas brancas[1]

para a pista horizontal

ver qual o valor da soma, ver qual o número de casas brancas que constituem essa soma e ver quais as hipóteses possíveis para essas brancas

cruzar as duas hipóteses

verificar se a casa ficou reduzida a uma hipótese

se sim recorrer à função resolve_casas_adjacentes

[1]Repara-se que dentro desta função é onde se calcula quais as hipóteses possíveis para cada branca e isso é feito com recurso a outra função, a função combinacoes. Esta

função tem declarados oito vectores de tamanho variável (o número de somas possíveis para esse número de casas) e é um vector para cada número de casas possíveis para uma soma, ou seja, a começar em duas casas e acabar em nove. Dentro de cada vector tem-se para todas as somas possíveis com esse número de casas os valores que as brancas podem tomar (isto sem ter em conta arranjos e permutações de hipóteses).

- resolve_casas_adjacentes

Este algoritmo tem por base o facto de se saber que num conjunto de brancas pertencentes à mesma soma não pode haver números repetidos e assim esta função só é chamada quando se tem a certeza de um número numa dada branca e vai retirar esse número (se existente) às hipóteses das outras brancas dessa soma.

Algoritmo:

verificar qual o número resolvido para a branca em questão
para a soma na vertical
percorrer as brancas dessa soma
retirar o número às hipóteses delas
verificar se alguma branca ficou resolvida
se sim por recursividade chamar de novo a função resolve_casas_adjacentes
para a nova branca resolvida
para a soma na horizontal
idêntico ao supra mencionado para a soma na vertical

- procura_soma_triangulares

Este algoritmo efectua vários passos, todos dentro de um ciclo e só sai desse ciclo quando detecta que já não consegue fazer mais alterações no puzzle, seja por este estar completo, seja por ter de recorrer agora a método de teste de hipóteses. Para se perceber melhor como funciona este algoritmo recorreu-se a um fluxograma.

O fluxograma deste algoritmo segue em anexo (Anexo 2)

- chama_algoritmos

A chamada dos algoritmos que se vai explicar mais abaixo depende cada uma de certas condições que se verifiquem na resolução do puzzle e assim esta função está encarregue de analisar essas condições e consoante as que se verifiquem chamar os algoritmos correspondentes. Para se perceber melhor quais as condições que esta função analisa e quais os algoritmos a ser chamados recorreu-se a um fluxograma.

O fluxograma deste algoritmo segue em anexo (Anexo 3).

- completar_soma

Este algoritmo é chamado sempre com o intuito de resolver mais uma casa no puzzle pois como o nome indica vai completar uma soma, seja ela na vertical ou na horizontal e assim já tem todos os dados de que precisa para completar mais uma casa, diga-se, o valor da soma e o valor das brancas dessa soma já preenchidas.

Algoritmo:

verificar qual o número que falta para a branca que ainda está vazia nessa soma
se a soma for na vertical
percorrer as brancas dessa soma até encontrar a branca vazia
colocar o número em falta como solução da mesma
chamar a função resolve_casas_adjacentes
se a soma for na horizontal
idêntico ao supra mencionado para a soma na vertical

- combinações_pares_reais

Este algoritmo é feito quando numa dada soma faltam preencher duas casas apenas e quer-se saber se nessa nova soma a duas casas os números que anteriormente se tinham como hipóteses para essas brancas ainda são possíveis.

Algoritmo:

verificar qual a nova soma feita a duas casas
se a soma for na vertical
percorrer as brancas dessa soma até encontrar as duas brancas vazias
para a primeira branca ver os valores possíveis desta e para cada um deles verificar se o valor correspondente necessário para fazer a nova soma está na segunda branca
se encontrar algum valor que não esteja na segunda branca retirar o valor que se testou na primeira branca às hipóteses possíveis desta
retirar também às hipóteses da segunda branca os valores que não podem fazer a nova soma a duas casas por a primeira branca não ter os números correspondentes e não os ter testado sequer
verificar uma a uma se as duas brancas ficaram reduzidas a uma hipótese
se sim chamar a função resolve_casas_adjacentes
se a soma for na horizontal

idêntico ao supra mencionado para a soma na vertical

- possibilidades_exclusivas

Este algoritmo pretende identificar se existe numa soma possibilidades que estejam afectas a certas casas e assim poder retirá-las das casas que não as tenham como possibilidades exclusivas.

Tome-se como exemplo uma soma a cinco casas e em três casas dessas cinco são possíveis os valores 1, 4 e 6 sabe-se por este algoritmo que esses três valores ao terem de existir obrigatoriamente nessa soma são exclusivos dessas três casas e podemos retirar essas hipóteses às outras duas casas.

Algoritmo:

se a soma for na vertical
percorrer as brancas dessa soma
para cada branca verificar quantas hipóteses tem
percorrer as brancas da soma a partir da branca que se está a testar
verificar quantas mais brancas tem o mesmo número e as mesmas hipóteses que a branca que se está a testar
se forem tantas brancas quanto o número de hipóteses da branca de teste
retirar as hipóteses da branca de teste às brancas que são diferentes dela
verificar se alguma dessas brancas diferentes ficou resolvida
se sim chamar a função resolve_casas_adjacentes
se a soma for na horizontal
idêntico ao supra mencionado para a soma na vertical

- eliminar_maximos_minimos

Se para uma dada célula branca, ao usar o máximo dela somado com o mínimo das restantes brancas da soma der uma soma maior que a pretendida então com este algoritmo sabe-se que esse número é demasiado grande, logo já não fará parte das hipóteses dessa branca. Também, se for feito o mesmo com o mínimo da casa somado aos máximos das restantes e não chegar ao valor da soma então essa possibilidade é demasiado pequena e também não pode ser usada. De notar que este algoritmo pode ser aplicado a somas que já tenham casas resolvidas, sendo que nesses casos calcula-se o valor da nova soma pretendida e recorre-se então ao teste dos máximos e dos mínimos para as casas que não estão resolvidas.

Algoritmo:

se a soma for na vertical

percorrer as brancas dessa soma

para cada branca que não estiver resolvida

testar o máximo dessa branca com o mínimo das outras brancas também não resolvidas

verificar se essa soma seria possível em relação à soma pretendida

se não for possível retirar essa hipótese à branca com que se testou e verificar se esta poderá ter ficado resolvida

se sim chamar a função resolve_casas_adjacentes

testar o mínimo dessa branca com o máximo das outras brancas também não resolvidas

verificar se essa soma seria possível em relação à soma pretendida

se não for possível retirar essa hipótese à branca com que se testou e verificar se esta poderá ter ficado resolvida

se sim chamar a função resolve_casas_adjacentes

se a soma for na horizontal

idêntico ao supra mencionado para a soma na vertical

- valores_max_min

Este algoritmo vai verificar se a soma do número máximo possível para cada branca num conjunto de brancas corresponde à soma que se pretende e irá efectuar o mesmo para os mínimos e se num dos casos for de facto verdade então descobriu-se o valor da soma, ou é feita dos máximos de cada casa ou do mínimo delas. De notar que este algoritmo só é feito se as brancas tiverem todas mínimos e máximos diferentes e se uma dada casa já está resolvida o valor a considerar nesse caso é o valor que ela toma.

Algoritmo:

se a soma for na vertical

percorrer as brancas dessa soma

para cada branca retirar o seu valor máximo e mínimo

verificar qual a soma que os máximos dão e o mesmo para a soma dos mínimos

se os máximos são todos diferentes e soma destes é igual à soma da triangular

para cada branca resolvê-la de acordo com o seu máximo respectivo

chamar a função resolve_casas_adjacentes

se os mínimos são todos diferentes e soma destes é igual à soma da triangular
para cada branca resolvê-la de acordo com o seu mínimo respectivo
chamar a função resolve_casas_adjacentes
se a soma for na horizontal
idêntico ao supra mencionado para a soma na vertical

- n_possibilidades_n_casas

Este algoritmo é aplicável quando se tem uma soma a "n" casas e quando se verifica quais os valores possíveis para cada uma dessas "n" casas existem "n" possibilidades e assim cada casa vai ter obrigatoriamente um valor desses "n" valores possíveis. O algoritmo vai tentar determinar se sabe o arranjo que essas "n" hipóteses tomam na soma.

Veja-se um exemplo: se houver uma soma de 24 a três casas cada casa toma como hipóteses de resolução os valores 7, 8 e 9 e assim já sabemos que numa casa dessas três casas estará um 7, noutra o 8 e na que sobra o 9, isto obrigatoriamente. Suponha-se que depois de cruzar cada casa com a sua outra respectiva pista fica-se que na primeira branca os valores possíveis são o 7 e o 8, na segunda o 7, o 8 e o 9 e na última de todas de novo o 7 e o 8. Com a aplicação deste algoritmo sabemos que na segunda casa o valor final é o 9 pois este tem de existir obrigatoriamente nessas três casas.

De notar que esta função já recebe como argumento os "n" valores possíveis para as "n" brancas e a função só é de facto chamada quando se sabe que se verifica esta situação.

Algoritmo:

se a soma for na vertical
escolher o primeiro desses "n" valores possíveis
percorrer as "n" brancas dessa soma
verificar em quantas dessas "n" brancas o valor escolhido é possível
se só for possível numa delas resolvê-la de acordo com esse valor
chamar a função resolve_casas_adjacentes
repetir para o próximo dos "n" valores até se ter repetido o algoritmo para todos
se a soma for na horizontal
idêntico ao supra mencionado para a soma na vertical

- resolver_como_soma_menor

Para que este algoritmo seja chamado basta que uma casa esteja preenchida e o que ele faz é ao valor total da soma retirar o valor da(s) casa(s) que está(ão) preenchida(s) e com a nova soma verificar quais as novas hipóteses que cada branca vazia ia tomar e cruzá-las com os valores já lá existentes a ver se poderá ter resolvido alguma célula. De notar que esta função já recebe como argumento os valores possíveis para as brancas de acordo com a nova soma e com o novo número de casas.

Algoritmo:

se a soma for na vertical
percorrer as brancas dessa soma
para cada branca não resolvida ainda cruzar as hipóteses que esta já lá tinha com as hipóteses da nova soma menor
verificar para cada branca se ficou resolvida
se sim chamar a função resolve_casas_adjacentes
se a soma for na horizontal
idêntico ao supra mencionado para a soma na vertical

- resolve_casa_teste

Esta função vai percorrer as matrizes à procura da primeira branca não resolvida que tiver o número mínimo de hipóteses possíveis de solução. O objectivo disto é escolher estrategicamente em que casa a função que aplica o método de tentativa-e-erro deve ser aplicada. De notar que este algoritmo é feito até o número de hipóteses que se procura numa dada branca ser 9 pois esse é o número máximo de hipóteses de resolução que ela pode ter.

Algoritmo:

número de hipóteses que se quer para uma dada branca começa a 2
percorrer a matriz de solução à procura de brancas não resolvidas
encontrou uma branca não resolvida
verificar a matriz de estruturas na posição dessa branca e ver quantas hipóteses possíveis de solução esta tem
verificar se o número de hipóteses dessa branca corresponde ao número de hipóteses que se quer
se sim armazenar as coordenadas dessa branca e sair da função

se não encontrar nenhuma branca com o mesmo número de hipóteses que se quer incrementa-se essa última variável

- testa_hipoteses

Este é o algoritmo responsável pelo método de tentativa-e-erro, ou seja, pelo método de tentar resolver o puzzle escolhendo numa dada branca um número que se sabe ser possível nesta. Por vezes pode bastar um só número por tentativa-e-erro e esse estar logo certo e resolver logo o puzzle todo (isto porque também se faz recurso às restrições), mas noutras ocasiões, mesmo depois de uma aplicação deste método e das devidas restrições, a resolução do puzzle pode ficar encravada e recorre-se a nova iteração deste método. Para se perceber melhor como funciona este método recorreu-se a um fluxograma.

O fluxograma deste algoritmo segue em anexo (Anexo 4).

Descrição dos subsistemas funcionais

O programa *crosssums* é constituído por quatro subsistemas – "kakuro", "matrizes", "combinações" e "resolve".

O subsistema "kakuro", constituído pelos ficheiros *kakuro.c* e *kakuro.h*, engloba o *main* (onde são chamadas às funções de auxílio a resolução do programa) e as funções de leitura do ficheiro de entrada e escrita do ficheiro de saída. As estruturas de dados referentes às células brancas e triangulares e à estrutura que engloba estas estão declaradas no *kakuro.h*. O subsistema "matrizes", constituído pelos ficheiros *matrizes.c* e *matrizes.h*, engloba todas as funções relacionadas com a criação e análise de matrizes, uma função responsável por libertar todo o espaço que foi alocado para a criação destas e duas funções associadas a operações de *pop* e *push* de elementos na fila. O subsistema "combinações", constituído pelos ficheiros *combinacoes.c* e *combinacoes.h*, engloba duas funções relacionadas com o cálculo das hipóteses possíveis para uma dada branca. O subsistema "resolve", constituído pelos ficheiros *resolve.c* e *resolve.h*, inclui todas as funções associadas com o uso de restrições no puzzle para resolver o máximo possível deste e ainda a função que corresponde ao método de tentativa-e-erro.

- *kakuro.c*

→ void **ler_ficheiro** (FILE * fpin, int altura, int largura, int matriz[altura][largura]);

lê o ficheiro de entrada com a configuração do puzzle e para cada linha do puzzle cria a linha correspondente na matriz com os valores apropriados

→ void **escreve_solucao** (FILE * fpin, char * argv[], int solucao, int altura, int largura, int matriz[altura][largura], int matriz_solucao[altura][largura]);

escreve a configuração do puzzle e a solução final deste (se existir) no ficheiro de saída

- *matrizes.c*

→ branca ***** **cria_branca** (int pos_y, int pos_x, int altura, int largura, celula_matriz matriz_estruturas[altura][largura]);

aloca o espaço necessário para uma estrutura do tipo struct branca e inicializa-a com os valores pretendidos

→ triangular ***** **cria_triangulo** (int valor, char orientacao, int pos_y, int pos_x, int altura, int largura, int matriz[altura][largura]);

aloca o espaço necessário para uma estrutura do tipo struct triangular e inicializa-a com os valores pretendidos

→ void **cria_matriz_estruturas** (int altura, int largura, int matriz[altura][largura], celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

preenche a matriz de estruturas do tipo struct celula_matriz com os valores pretendidos, incluindo as hipóteses que cada branca toma em função da(s) soma(s) a que pertence

→ void **cria_matriz_solucao** (int altura, int largura, int matriz[altura][largura], int matriz_solucao[altura][largura]);

preenche a matriz de solução com os valores adequados a partir da leitura da matriz

→ int **analisa_matriz** (int altura, int largura, int matriz[altura][largura], celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

a partir da leitura da matriz e da matriz de estruturas verifica quais os valores finais que as brancas tomam e escreve-os na matriz de solução

→ int **analisa_matriz_solucao** (int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

analisa a matriz de solução e verifica se está completamente preenchida ou não e se estiver verifica se os números colocados nas células brancas estão correctos de acordo com as somas do puzzle

→ void **cria_matrizes_temporarias** (int altura, int largura, int matriz[altura][largura], celula_matriz matriz_estruturas[altura][largura], celula_matriz matriz_estruturas_temporaria[altura][largura], int matriz_solucao[altura][largura], int matriz_solucao_temporaria[altura][largura]);

se for necessário recorrer ao método de testar hipóteses esta função preenche uma nova matriz de estruturas do tipo struct celula_matriz com os valores pretendidos (é uma cópia da matriz de estruturas criada primeiramente) e preenche também uma nova matriz de solução com os valores pretendidos (é uma cópia da matriz de solução de como ela está no momento)

→ fila_casas * **push_coordenadas** (int i, int j, fila_casas * ultima);

função que insere preenche os dados de novos elementos que vão ser inseridos na fila sendo que cada novo elemento é inserido no fim

→ fila_casas * **pop_coordenadas** (int * i, int *j, fila_casas * ultima);

função que lê e armazena em variáveis passadas por referência os dados pretendidos do último elemento da fila e depois liberta a memória por esta alocada

→ void **limpa_memoria** (int altura, int largura, celula_matriz matriz_estruturas[altura][largura]);

liberta todo o espaço que foi alocado na matriz de estruturas

- *combinacoes.c*

→ void **combinacoes** (int soma, int casas, int vector[9]);

função que consoante o número de casas que constituem uma dada soma e qual o valor da mesma altera um vector de 9 posições passado por referência colocando 1s nos valores possíveis para as brancas dessa soma e 0s nos impossíveis

→ int **combinacoes_tentativa_erro** (int soma, int num_casas);

função que devolve 1 ou 0 consoante uma dada soma é possível com um dado número de casas ou não, respectivamente

- *resolve.c*

→ int **resolve_casas_adjacentes** (int y, int x, int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função recursiva que quando é descoberto o valor de uma casa retira esse valor às hipóteses das casas adjacentes e verifica se ficaram resolvidas mais casas, sendo de notar que esta é a função que se chama de cada vez que se descobre uma casa

→ int **cruza_somas** (int y, int x, int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que cruza para uma dada branca as hipóteses que esta toma dependendo da pista vertical com as hipóteses da pista horizontal

→ void **completar_soma** (triangular * celula, int valores_descobertos[9], int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que se para uma dada soma existir apenas uma branca por preencher calcula qual o valor que ela toma

→ int **combinacoes_pares_reais** (triangular * celula, int valores_descobertos[9], int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que verifica numa soma a duas casas se os dois números necessários de cada arranjo (que são os valores que podem dar a soma) existem ora numa casa ora na outra

→ int **eliminar_maximos_minimos** (triangular * celula, int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que testa se o máximo de uma branca mais os mínimos das outras brancas da soma assim como o mínimo de uma branca mais os máximos das outras vai dar um valor possível no contexto da soma

→ int **n_possibilidades_n_casas** (triangular * celula, int novas_possibilidades[9], int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que verifica se numa soma a "n" casas com cada casa a ter "n" possibilidades se sabe qual o valor desses "n" que pertence a cada casa, ou seja, se sabe qual a sua ordem

→ int **resolver_como_soma_menor** (triangular * celula, int novas_possibilidades[9], int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que vai calculando para somas já com casas preenchidas qual o valor da nova soma e então qual seriam as novas hipóteses para cada branca dessa nova soma a menos casas e cruzar as novas hipóteses com as já existentes

→ int **possibilidades_exclusivas** (triangular * celula, int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que analisa se numa dada soma existem valores exclusivos a várias brancas retirando-os então às possibilidades das outras

→ int **valores_max_min** (triangular * celula, int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que verifica se a soma do número máximo possível para cada branca num conjunto de brancas corresponde à soma que se pretende e efectua o mesmo para os mínimos a ver se poderá ter resolvido essas casas

→ int **chama_algoritmos** (triangular * celula, int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que dependendo do contexto que se verificar para uma dada soma chama as funções acima analisadas do ficheiro *resolve.c*

→ int **procura_soma_triangulares** (int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que até detectar que já não se pode avançar mais no puzzle com as restrições acima descritas ou que o puzzle já está resolvido chama a função imediatamente acima descrita

→ void **encontra_casa_teste** (int * pos_y, int * pos_x, int altura, int largura, celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura]);

função que percorre as matrizes à procura da primeira branca não resolvida que tiver o número mínimo de hipóteses possíveis

→ int **testa_hipoteses** (int altura, int largura, int matriz[altura][largura], celula_matriz matriz_estruturas[altura][largura], int matriz_solucao[altura][largura], celula_matriz matriz_estruturas_temporaria[altura][largura], int matriz_solucao_temporaria[altura][largura]);

função que vai aplicar o método de tentativa-e-erro e depois da aplicação deste vai recorrer a todas as restrições já explicadas e verificar se resolveu o puzzle e se isso não se verificar chama-se recursivamente

Análise dos requisitos computacionais

Ao longo do programa acede-se várias vezes às diversas matrizes criadas, a de estruturas e a de solução, e dentro da matrizes de estruturas acede-se ainda várias vezes, para a estrutura das células brancas, a um vector de 9 posições que contém as hipóteses possíveis para uma dada branca. Analise-se então melhor a complexidade em torno do acesso às várias estruturas de dados.

- struct matriz e struct matriz_solucao (são ambas matrizes de inteiros, caso será igual)

Nestas matrizes, sempre que se quer ler ou escrever o valor de uma casa acede-se directamente à matriz nos índices respectivos dessa célula para lá efectuar a operação pretendida e nesses casos a complexidade é *O(1),* pois o acesso a essa posição é feito de uma maneira directa (com os índices) e não a percorrer toda a matriz. Nos casos em que se percorre a matriz com recurso a um ciclo de duplo *for*, um que começa com "i" a 0 e que sai quando o "i" for menor que a altura e outro que começa com "j" a 0 e que sai quando "j" for menor que a largura, a complexidade de acesso à matriz será *O(i*j).* Nessas situações faz-se o mesmo acesso que anteriormente a uma posição específica da matriz, mas isso agora para as "n" casas, que correspondem a i*j. Se a matriz for toda

percorrida “i” corresponderá à altura do puzzle e “j” à largura deste, se forem feitos *breaks* para interromper esses *for*, até ele sair terá percorrido i*j casas.

A memória alocada é igual a (*número de casas que o puzzle tiver * o tamanho de um inteiro na arquitectura do sistema que está a correr o programa*). O tamanho do inteiro pode ser 32bit ou 64bit.

- matriz de estruturas

O caso acima explicado das complexidades de acesso para a matriz e para a matriz de solução verifica-se também para os acessos à matriz de estruturas que podem ser acedidas directamente ou através de um ciclo que a percorre.

Porém, ainda na matriz de estruturas pode-se aceder ao vector da estrutura das brancas que guarda as hipóteses possíveis para estas. O acesso ao vector é também com complexidade *O(1)*. Note-se que mesmo quando se efectua um ciclo para percorrer o vector este ciclo nunca e interrompido a meio como pode acontecer com os ciclos que percorrem a matriz, pelo que para percorrer este vector de inteiros o número de operações será sempre constante (9), e então a complexidade é *O(1)*.

A memória alocada na matriz de estruturas é (*32bit * 3 ponteiros = 96bit)*. Isto porque cada célula contém ponteiros para três estruturas e cada ponteiro ocupa 32bit (supondo ser essa a arquitectura do computador).

- struct branca

A estrutura struct branca é parte da matriz de estruturas e a memória que ocupa depende do tipo de informações que guarda. Veja-se então – o vector de nove inteiros ocupa (*9 * 32bits*), as duas coordenadas correspondem a dois inteiros e assim essa parte ocupa (*2 * 32bits*) e ainda nessa estrutura tem-se dois ponteiros para outras estruturas que ocupam (*2 * 32bits*), sendo então o total de memória alocada para esta estrutura de dados de 416bits.

- struct triangular

A estrutura struct triangular é parte da matriz de estruturas e a memória que ocupa depende do tipo de informações que guarda. Veja-se então – os quatro inteiros ocupam (*4 * 32bits*) e existe ainda nessa estrutura um carácter que ocupa 8bits sendo então o total de memória alocada para esta estrutura de dados de 136bits.

- struct fila_casas

A complexidade associada a esta fila é de *O(1)* uma vez que se acede sempre ao último elemento desta e nunca é necessário percorrê-la à procura de um elemento específico.

A estrutura da fila é uma estrutura constituída por dois inteiros que ocupam (*2 * 32bit*) e por um ponteiro para um elemento do mesmo tipo, e como é um ponteiro ocupa também 32bit sendo o total de memória alocada de 96bits.

- função resolve_casas_adjacentes

Esta função percorre as casas da soma vertical e da soma horizontal da branca para a qual foi chamada, fazendo um número de operações constantes em cada casa, mas estas dependendo do número de casas que a soma vertical e horizontal têm, o que resulta numa complexidade de *O(número de casas na soma vertical + número de casas na soma vertical)*.

- função cruza_somas

Esta função tem complexidade *O(1)* pois cruza os vectores das hipóteses de uma branca para a pista vertical e para a pista horizontal mas multiplica elemento do vector a elemento do vector, isto para os nove elementos do vector, sendo assim um número de operações constantes.

- funções completar_soma, combinacoes_pares_reais, eliminar_maximos_minimos, n_possibilidades_n_casas, resolver_como_soma_menor, possibilidades_exclusivas, valores_max_min e chama_algortimos

Estas funções executam cada uma um número diferente de operações por cada casa porém, essas operações dependem do número de casas na soma o que resulta numa complexidade *O(número de casas na soma)*.

- função procura_soma_triangulares

Esta função corresponde a um percorrer da matriz pelo que a sua complexidade é igual à complexidade explicada anteriormente para esta estrutura, ou seja, *O(i*j)*.

- função testa_hipóteses

Para esta função é difícil de prever a sua complexidade uma vez que não se sabe quantas operações são feitas pois a função pode ser interrompida em vários pontos, como pode nunca ser interrompida e ser chamada depois recursivamente, como pode ainda também correr uma vez e não voltar a ser chamada.

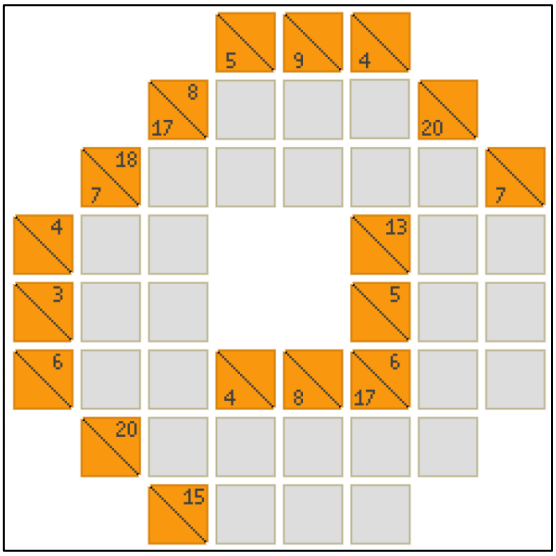
Análise crítica do programa

Da primeira vez que se fez *upload* dos ficheiros o programa passou em todos os testes fornecidos pelo corpo docente porém o grupo achou que estava a demorar muito tempo para resolver alguns dos puzzles e decidiu-se rever o código. Depois da adição da estrutura de dados da fila e da revisão do método de tentativa-e-erro o programa ficou mais eficiente em termos de memória e de tempo de execução e aí o grupo decidiu que já não iria alterar mais nada, uma vez que os 254 puzzles fornecidos pelo corpo docente ficaram resolvidos em 41segundos nos computadores do SCDEEC.

Tentou-se arranjar o máximo de restrições, criando as respectivas funções, com o objectivo de melhorar a eficiência do programa e evitar recorrer ao método de testar hipóteses por este poder aumentar o tempo que o programa demora a resolver um puzzle. No final, o grupo pensa que consegui implementar bem o pretendido e de uma forma eficaz com um bom recurso às várias estruturas de dados e à maneira como foram manipuladas.

Exemplo de aplicação

Para demonstrar uma aplicação de um programa escolhemos um kakuro que aplica várias restrições e ainda precisa de uma aplicação do método de teste de hipóteses. A configuração inicial do puzzle é a seguinte.



Primeiramente cria-se a matriz com a configuração do puzzle, a matriz de solução e a matriz de estruturas. Tome-se um exemplo para cada tipo de casa.

Para a primeira triangular do puzzle (soma 5) as estruturas de dados ficariam de seguinte da maneira:

int matriz[altura][largura]	5
int matriz_solucao[altura][largura]	(-1)
struct triangular	→ int soma = 5 → char orientação = 'v' → int num_casas = 2 → int pos_y = 0 → int pos_x = 3
struct celula_matriz	→ triangular * triangulo_superior = NULL → triangular * triangulo_inferior = aponta para a estrutura struct triangular enunciada acima na tabela → branca * casa = NULL

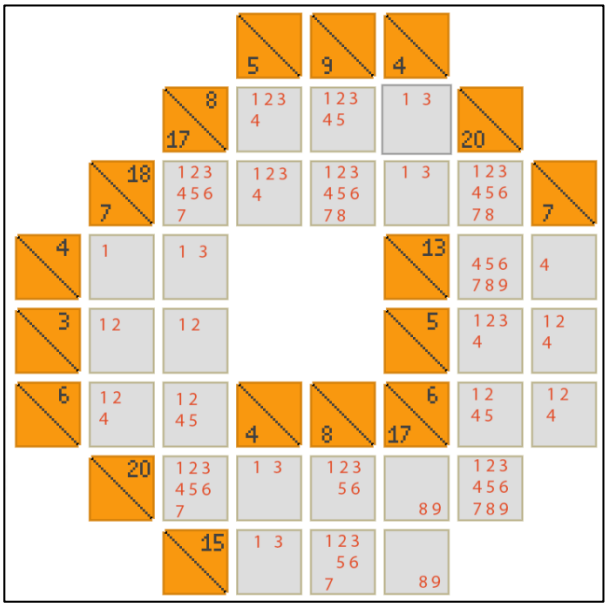
Se fosse uma triangular dupla seriam criadas duas estruturas do tipo struct triangular e na struct celula_matriz os dois primeiros ponteiros iriam apontar respectivamente para essas estruturas e o ponteiro para a estrutura do tipo célula branca seria NULL.

Para a primeira branca do puzzle as estruturas de dados ficariam da seguinte maneira:

int matriz[altura][largura]	-1
int matriz_solucao[altura][largura]	0
struct branca	→ int valores[9] = {1,1,1,1,1,1,1,1,1} → triangular * superior = aponta para a estrutura struct triangular da branca superior (soma 8) → triangular * inferior = aponta para a estrutura struct triangular da branca superior (soma 5) → int pos_y = 1 → int pos_x = 3
struct celula_matriz	→ triangular * triangulo_superior = NULL → triangular * triangulo_inferior = NULL → branca * casa = aponta para a estrutura struct branca enunciada acima na tabela

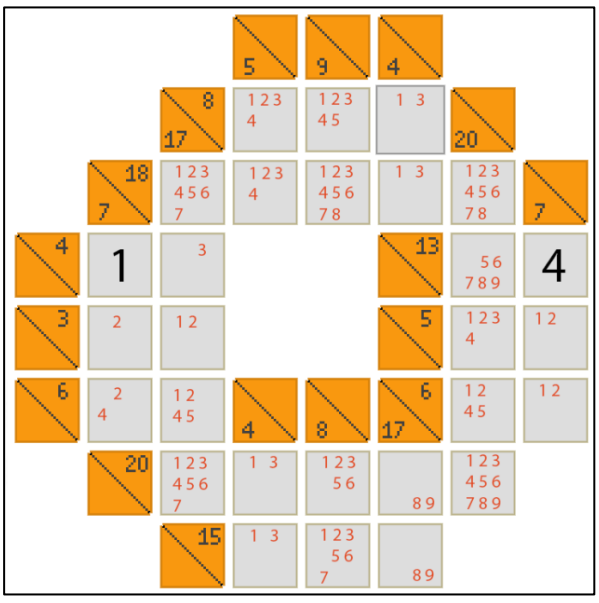
Ao longo do programa para as células triangulares nenhuma das estruturas de dados será alterada. Já para as células brancas vai-se alterar o valor da matriz de solução para o valor final que cada branca vai tomar e na estrutura struct branca o vector de inteiros também se vai alterando ao longo do programa à medida que se vão reduzindo as hipóteses que cada vai tomando.

Depois vai-se recorrer a função que calcula todas as combinações possíveis para as células brancas de acordo com a pista vertical e a pista horizontal e cruza-as (função cruza_somas). No final deste passo o puzzle terá o seguinte aspecto em termos de resolução.



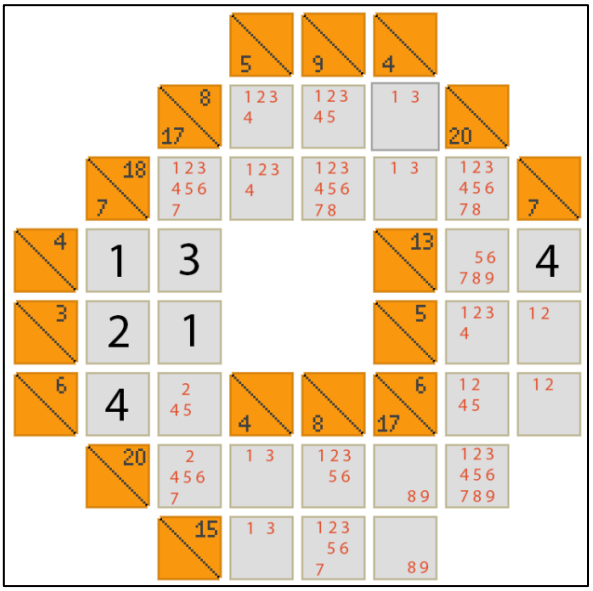
Neste caso, no nosso exemplo, para a estrutura struct branca da primeira branca o vector de inteiros ficaria alterado e passaria a tomar os valores {1,1,1,1,0,0,0,0,0}, assim como o vector das restantes brancas para as respectivas máscaras dos números que se vêm na figura.

Como se vê pelo resultado do algoritmo cruza_somas resolveram-se casas e assim iria recorrer-se à função resolve_casas_adjacentes e já com esse algoritmo aplicado o puzzle ficaria da seguinte maneira.

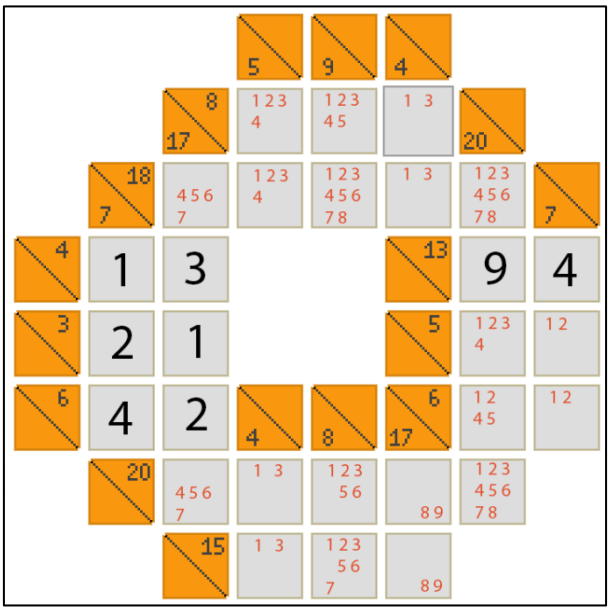


De notar que com a resolução destas casas a matriz de solução iria ser alterada nas posições respectivas e na estrutura struct branca o vector de inteiros passaria a ter apenas um único 1, sendo essa então a única mudança verificada na matriz de estruturas.

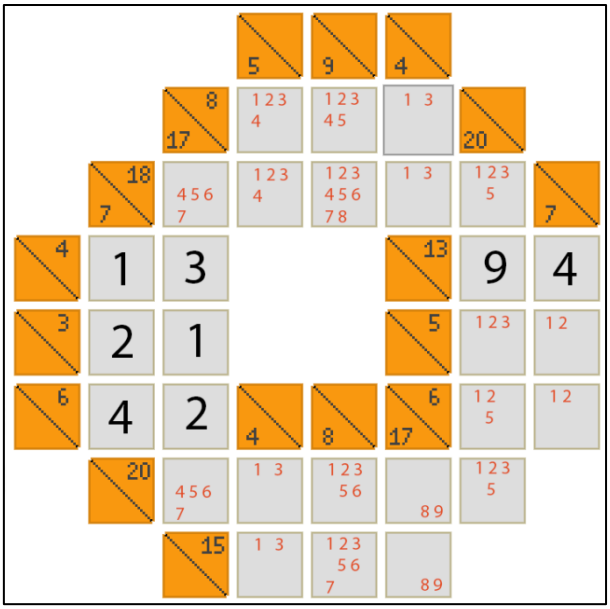
Como se vê pela figura ao resolver-se o 1 para a branca do lado esquerdo retirou-se este às hipóteses das casas abaixo e da casa à direita e o mesmo se fez mas para o 4 descoberto do lado direito. Com a aplicação deste algoritmo resolveram-se novas casas e a função está programa para ser chamada recursivamente nesses casos pelo que depois disso o puzzle ficaria mais resolvido.



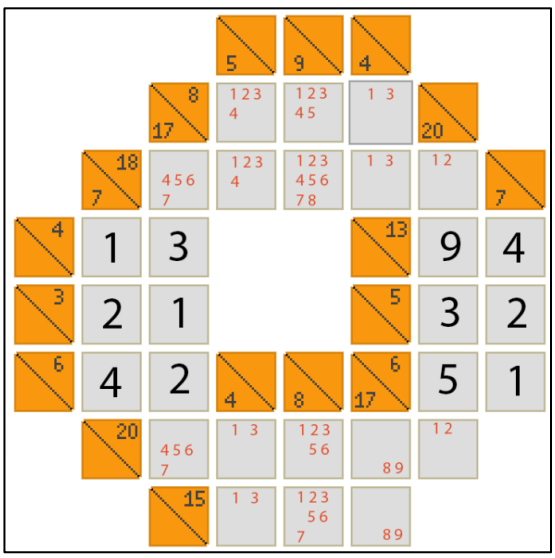
Pela análise do estado do puzzle vemos que a função chamar_algortimos iria recorrer à função completar_soma e aí de novo à resolve_casas_adjacentes. Assim o puzzle ficaria da seguinte maneira.



No contexto do problema como está existem somas com casas já resolvidas nas quais é possível aplicar o algoritmo de resolve_como_soma_menor. Do lado direito o algoritmo ia fazer como nova soma 17-3-1-2 = 11 e calcular quais as hipóteses que duas as brancas iam tomar. Do lado esquerdo, uma das novas somas seria 20-9=11 e a função iria ver qual o valor que as quatro brancas iriam tomar e faria também 7-4=3 e ver quais as novas hipóteses para as duas brancas. Assim, depois de aplicado o algoritmo o puzzle ficaria da seguinte maneira.

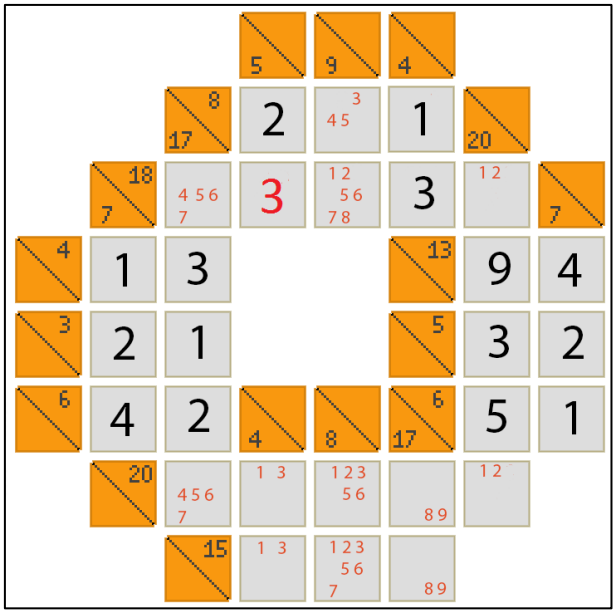


Ainda neste contexto é possível aplicar a função combinacoes_pares_reais pois existem várias somas a duas casas. Do lado direito a soma horizontal 5 iria ficar resolvida pois para estar o número 1 na primeira branca tinha de estar um 4 na outra e tal não se verifica e assim o 1 desaparecia das hipóteses da primeira e o mesmo se iria verificava com o 2 que não encontra o correspondente 4 na outra casa. Assim, já se sabe que o valor da primeira branca é o 3 e da segunda o 2. Na soma horizontal 6 iria-se aplicar o mesmo algoritmo e essas duas casas também ficavam resolvidas. De notar, que ao resolver-se casas chama-se a função resolve_casas_adjacentes e o puzzle ficaria como se enuncia abaixo.



Neste momento o programa iria tentar aplicar mais das restrições que já se explicaram mas não iria conseguir e ia verificar que o puzzle ainda está incompleto pelo que recorreria à função de testar hipóteses. Esta função vai procurar a primeira branca que tiver menos hipóteses e testar primeiramente a sua hipótese mais baixa, sendo esta branca a que está na 2ª linha e 6ª coluna, onde vai testar o número 1. Aqui já está criada uma nova matriz de estruturas e uma nova matriz de solução, iguais à maneira como as originais estavam quando foram copiadas. O espaço alocado para estas matrizes só será libertado quando a função sair do contexto onde estava.

Depois de testado o 1 nessa casa são aplicadas as restrições no puzzle até que este volta a encravar e assim a próxima casa onde a função se chama de novo fica na mesma linha que a anterior e é logo a primeira dessa soma horizontal. Testa aí o 2, chama as restrições e encontra um erro na segunda linha, porque iria precisar de um 3 na segunda casa dessa mesma linha mas o 3 já não é uma das hipóteses para essa branca.



Com esse verificado, tenta então o 3 e o 4 na primeira casa da 2ª linha (onde estava anteriormente) e ambos vão também encontrar erros, pelo que tem de regressar à primeira casa testada, a terceira casa da 2ª linha. A próxima e última hipótese possível desta casa é o 3 pelo que a função continua a resolver o puzzle com esse número, sabendo que já é o valor correcto dessa casa. Aqui se percebe a vantagem de escolher estrategicamente a casa inicial para a função de testar as hipóteses. Depois desta casa resolvida usa-se as restrições e o puzzle fica resolvido na totalidade, como se vê na figura abaixo.

Depois de resolvido o puzzle é libertado o espaço alocado pelas matrizes temporárias e de seguida pelas matrizes originais.

De notar que a fila é preenchida, porém não é necessário recorrer-se a esta uma vez que o puzzle é de pequenas dimensões e tinha várias restrições aplicáveis. Em puzzle de maiores dimensões a fila já é necessária para este método.