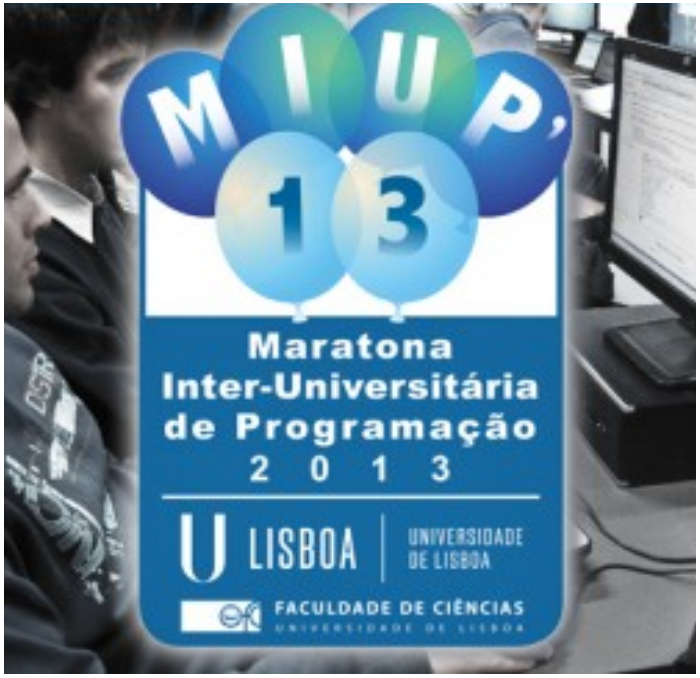


Sessão de Discussão dos Problemas

Pedro Ribeiro (DCC/FCUP)

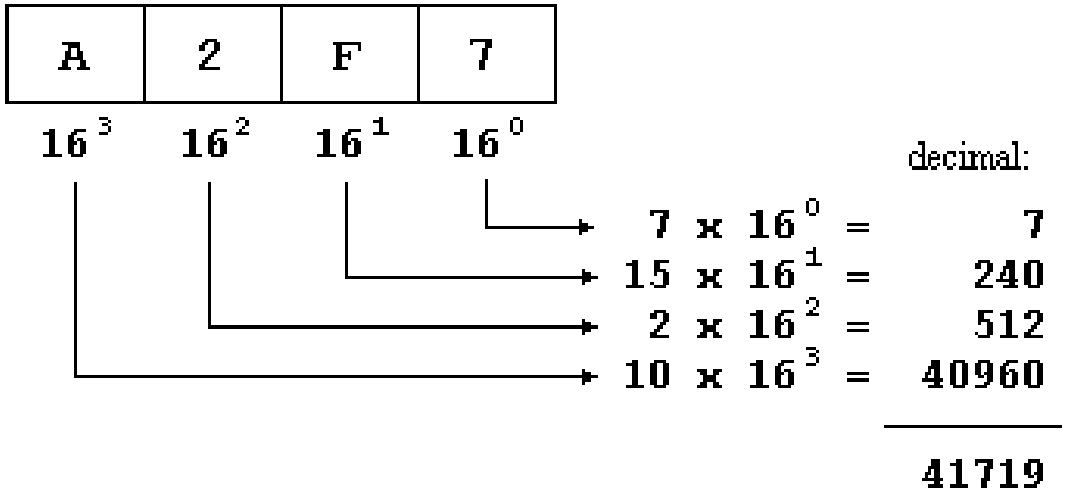
Uma visão geral (e pessoal)

#	Nome	Tipo	Dificuldade				Ling.	#Linhas
			En.	Alg.	Imp.	Tot.		
A	Speed Leaks	Ordenação por vários critérios	5	5	5	15	C	71
B	Say Geese	Teoria de Grafos	5	8	6	19	C	67
C	Roman Warfare	Programação Dinâmica	6	7	6	18	C	51
D	Base Arithmetic	Matemática / Conversão de Bases	5	4	4	13	C	39
E	Boundaries	Flood Fill / Compressão Coordenadas	6	7	7	20	C++	113
F	Twenty Thirteen	Ad Hoc / Dígitos	5	7	6	18	C	76
G	Polygon Phobia	Grafos / Ciclos / Union-Find	5	6	6	17	C	59
H	No Music, no Life	Exact Cover / Pesquisa com Cortes	5	5	6	16	C	53
I	The Magic Potion	Grids/Caminhos/Pesq. com Cortes	5	6	6	17	C	100

En. (Enunciado) - Até que ponto era fácil de perceber o enunciado e o problema em si.
Alg. (Algoritmo) - A dificuldade "mental" de perceber um algoritmo correcto e suficientemente eficiente para resolver o problema.
Imp. (Implementação) - Até que ponto era complicado implementar realmente em código a solução desenhada.

D – Base Arithmetic

- **Autor:** Rui Mendes (Universidade do Minho)
- **Input:** Quatro números *a*, *b*, *c* e *d*
- **Output:** Qual a base onde é verdade que $a + b = c \times d$
- **Limites:** Base entre 2 e 36



D – Base Arithmetic

- Provavelmente o problema mais fácil dos 9 propostos!

- É necessário saber o valor de um número numa dada base
  - Todos sabem fazer isso?

    Ex: **246** em octal = **2** x 8^2 + **4** x 8^1 + **6** x 8^0
- Só existem 35 bases para testar (2 a 36) logo podiamos fazer pesquisa exaustiva e ver se nessa base $a + b = c \times d$
- Cuidado para apenas testar as bases possíveis:
  - Se existir dígito de valor *V*, começar na base *V+1*
  - Base mínima é sempre 2 (mesmo que teste seja 0 0 0 0)

A – Speed Leaks

- **Autor:** Pedro Guerreiro (Universidade do Algarve)
- **Input:** Conjunto de Registos de matrícula e velocidade
- **Output:** “História” ordenada de matrículas e suas velocidades
- **Limites:** 20 000 registos

A – Speed Leaks

- Era um dos problemas mais “fáceis” da MIUP
- Essencialmente o problema dividia-se em duas partes:
  - (1) Agrupar os registos por matrícula
  - (2) Ordenar de forma “customizada”

- Saber ordenar é um das bases de qualquer programado
  - Quando não sabes o que fazer... ordena!
- Saber usar as funções de ordenação da linguagem: O (n log n)
- (1) podia ser feito... ordenando :) (por matrícula)
- (2) função comparadora que tenha em conta o enunciado

H – No Music, no Life

- **Autor:** Cristina Vieira (Universidade do Algarve)
- **Input:** Orquestras e instrumentos que sabem
- **Output:** Nº de conjuntos de orquestras que cobrem exatamente todos os instrumentos
- **Limites:** 25 orquestras e 25 instrumentos

H – No Music, no Life

- Este problema corresponde a um **exact cover**, que é um dos originais 21 problemas **NP-completos**.

- Solução polinomial... não existe!

- Era necessário fazer uma pesquisa “inteligente” (com alguns cortes)
- O tamanho era suficientemente pequeno para não serem necessárias heurísticas “fortes”

H – No Music, no Life

- Força bruta... bruta!
  - Testar todos os conjuntos possíveis: 2^25
- Corte Trivial
  - Quando uma orquestra usa um instrumento já “tocado”, não a incluir!
- Como testar quando adicionamos nova orquestra?
  - Se usarmos os bits, podemos usar operações binárias:
    - AND para testar se existe “overlap”
    - OR para adicionar

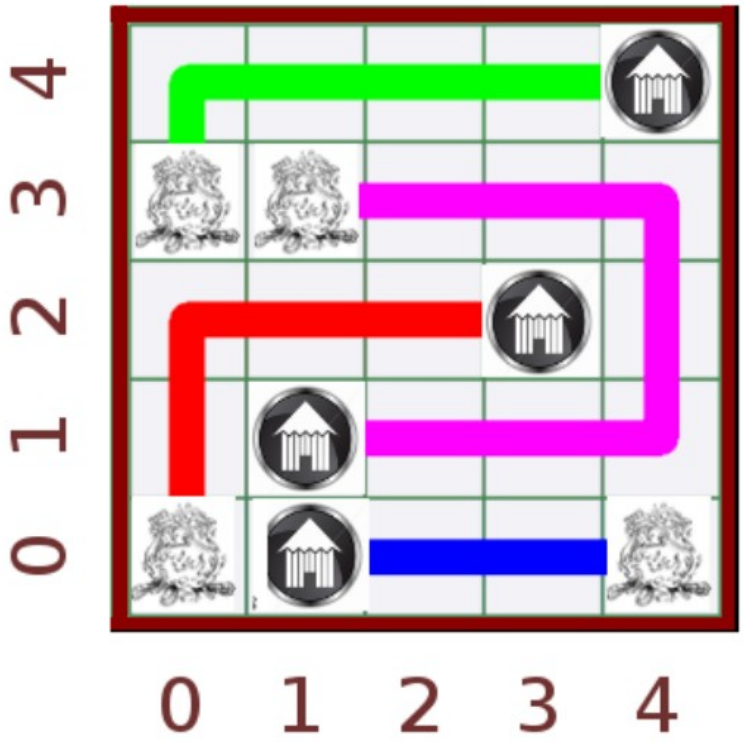
H – No Music, no Life

- Isto já chegava, mas como melhorar ainda mais?
  - Vários tipos de heurísticas possíveis
    - Ex: Procurar adicionar ao instrumento que tem menos opções (diminuir abertura da árvore de pesquisa)

- Podem espreitar também o **“algoritmo X”** de Knuth :)
  - Implementação com “Dancing links”

I – The Magic Potion

- **Autor:** Vítor Nogueira e Vasco Pedro (Universidade de Évora)
- **Input:** Uma grid e um conjunto de pares de pontos
- **Output:** Caminhos sem interseções a ligar cara par de pontos
- **Limites:** Grid 7x7, 7 pares de pontos



I – The Magic Potion

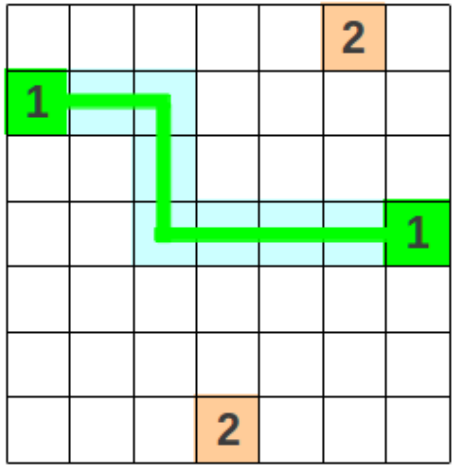
- Existe um jogo muito conhecido para Android e IOS que é muito semelhante: **Flow Free**
- O puzzle “original” chama-se **Numberlink**

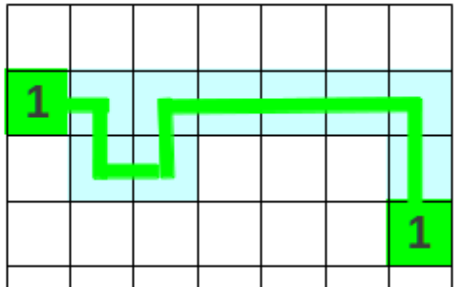
- Resolver o problema é computacionalmente difícil sem nenhum algoritmo polinomial
  - Já existiu um... concurso para o programa que conseguisse ser o mais rápido a resolver
    - No puzzle “original” é preciso preencher completamente a grid (aqui podem ficar células por preencher)

I – The Magic Potion

- Resolver passa por fazer uma pesquisa pelos vários caminhos possíveis
  - ex: Usar DFS com backtracking e ligar par a par

- É necessário fazer cortes!
  - Se um novo caminho "desliga" um par, não vale a pena continuar:
    - Flood-Fill/DFS/BFS entre par
  - Evitar caminhos "entrelaçados"
    - Não andar por células adjacentes



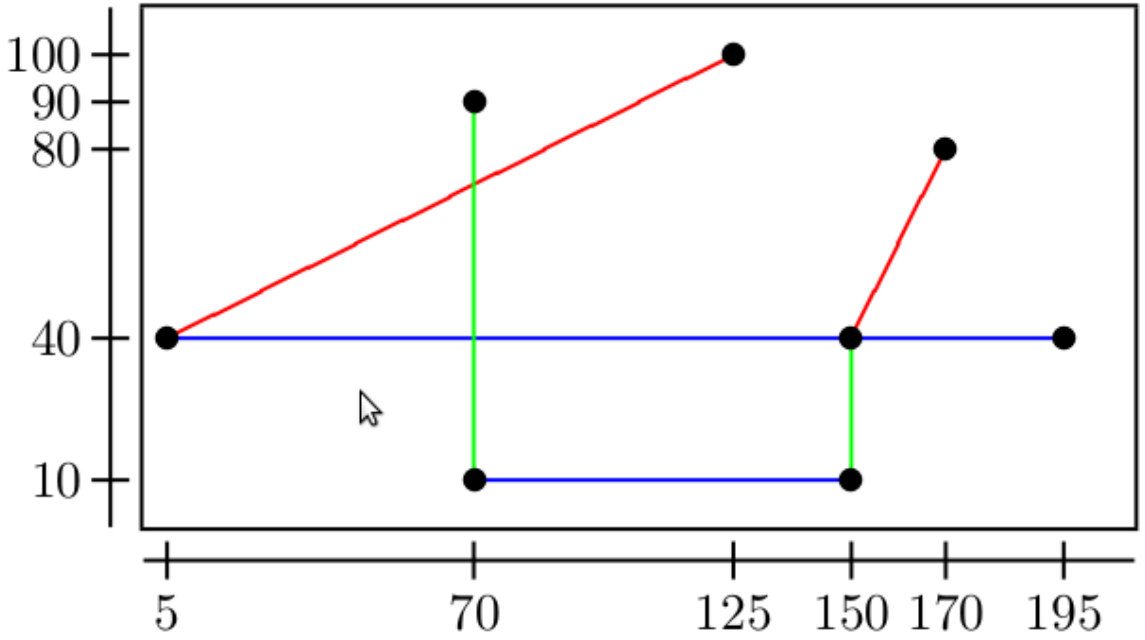


I – The Magic Potion

- Isto já bastava, mas muito mais podia ser feito:
  - Existem células que obrigatoriamente pertencem a um dado par (todos os caminhos passam por lá)
  - Porque que ordem devemos pesquisar os pares?
  - Como usar os cantos?
  - Começar em vários ao mesmo tempo?
  - Como detectar que dois pares são “incompatíveis”?
  - Como imitar as “heurísticas” que um humano usa ao jogar?

G – Polygon Phobia

- **Autor:** Margarida Mamede (Universidade Nova de Lisboa)
- **Input:** Sequências de S segmentos de linha
- **Output:** Nº de segmentos quando inseridos não criam cadeias
- **Limites:** 100 000 segmentos



G – Polygon Phobia

- Os segmentos podem ser pensados como um grafo
  - Nós/Vértices são os pontos
  - Arestas/Ligaçoes são os segmentos

- Deste “ponto de vista” uma cadeia é... um **ciclo**!
- Detectar ciclos é um problema “tradicional” de grafos

- Precisamos de estrutura que faça o seguinte:
  - Mantenha conjuntos de pontos ligados (componentes conexas)
  - Permita inserir e verificar se dois “endpoints” não pertencem ao mesmo componente (nesse caso teríamos um ciclo)

G – Polygon Phobia

- Precisamos de fazer o que se chama de **union-find (disjoint set)**
  - **Find:** descobrir se nó pertence a um componente
  - **Union:** Unir dois componentes
  - Resolver o problema é fazer o seguinte:
    - No início cada ponto pertence ao seu próprio componente
    - Para cada segmento verifica-se (com find) em que componente estão os endpoints:

      - Em componentes diferentes: unir

      - Em componentes iguais: ciclo! (não adicionar)

G – Polygon Phobia

- Como implementar de forma eficiente union-find?
  - É um problema “clássico”
  - Pesquisa linear não chega! Seria O(S^2)
  - Ideia: usar **florestas disjuntas**
    - Cada componente é uma árvore
    - Representante de componente é a raíz da árvore
    - **Find** é apenas seguir os “pais” do nó até chegar à raíz
    - **Union** é juntar uma raíz à raíz de outra

G – Polygon Phobia

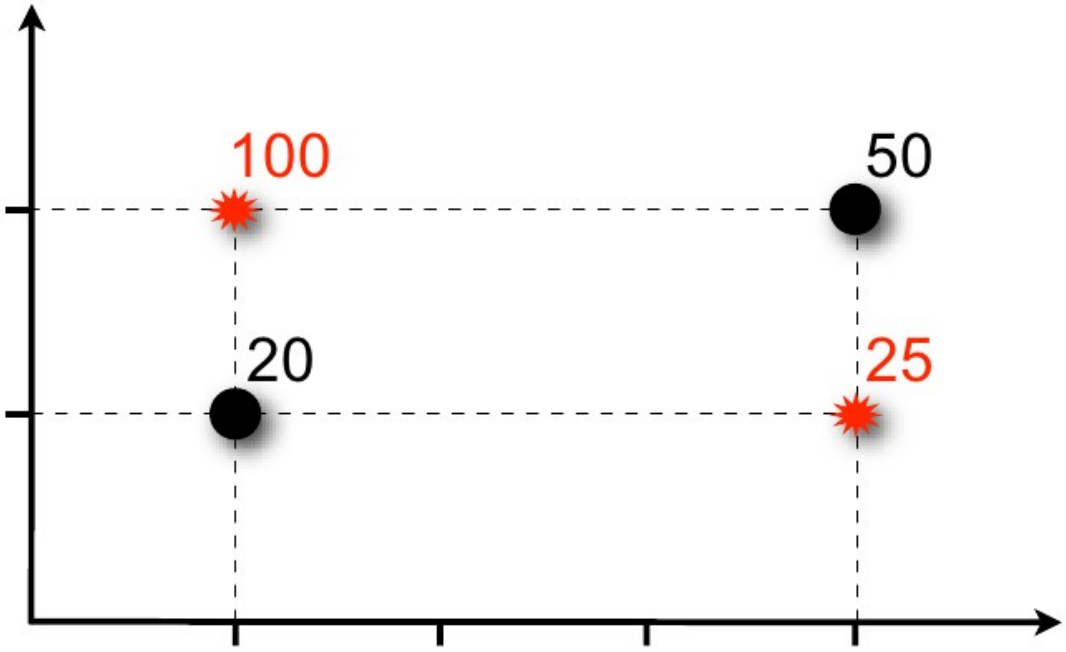
- Sem manter as árvores equilibradas usar as florestas disjuntas pode ser tão mau como pesquisa linear
  - Árvore com altura igual a número de nós

- Duas optimizações que melhoram o equilíbrio:
  - **Union by rank:** unir a árvore mais pequena à raíz da maior
  - **Path Compression:** ao fazer um find, ligar cada nó directamente à raíz (árvore fica mais "comprimida")

- Análise de complexidade fora do âmbito mas é interessante verem para perceber as garantias que dá (**análise amortizada**)

C – Roman Warfare

- **Autor:** Hugo Torres Vieira (Fac. Ciências Universidade Lisboa)
- **Input:** Localização e custo/valor de exércitos e populações
- **Output:** melhor maneira de alocar exércitos a populações
- **Limites:** 4000 exércitos (A), 4000 populações (P)



C – Roman Warfare

- Objectivos:
  - Maximizar valor conquistado
  - Em caso de empate, minimizar distância percorrida
  - Em caso de novo empate, minimizar custo

- Observação chave:
  - 2 exércitos com custo crescente, só podem atacar 2 localidades com custo crescente
  - Se um exército está alocado a uma dada localidade
    - Exércitos que custam < só podem atacar localidades que valem <
    - Exércitos que custam > só podem atacar localidades que valem >

C – Roman Warfare

- Se o único objectivo fosse maximizar valor conquistado?
  - Fácil! É só ordenar as localidades por ordem decrescente e conquistar as possíveis: min(A,E)
- O “problema” é que temos de ter em conta as distâncias!
- Pesquisa exaustiva não passa no tempo!
  - Bruto: todas as possíveis alocações (exponencial)
  - Cortes não chegam
  - Temos de pensar em solução polinomial

C – Roman Warfare

- Em problemas de optimização, onde é preciso **minimizar** ou **maximizar** algo (e também em problemas de **contagem**) muitas vezes pode-se usar **programação dinâmica (PD)**.
  - A sua ideia base é reutilizar resultados de subproblemas

- Vamos formular o problema de uma maneira que aceite PD!
- Vamos assumir que os exércitos e as localidades estão ordenados por ordem decrescente de custo/valor

C – Roman Warfare

- Seja **best[i][j]** a melhor maneira de alocar os primeiros *j* exércitos às primeiras *i* localidades
  - A solução fica guardada em best[min(P,A)][A]
  - best[i][j] = melhor entre

    best[i][j-1] ← não usar exército *j*

    aux: ← usar exército *j* na localidade *i*

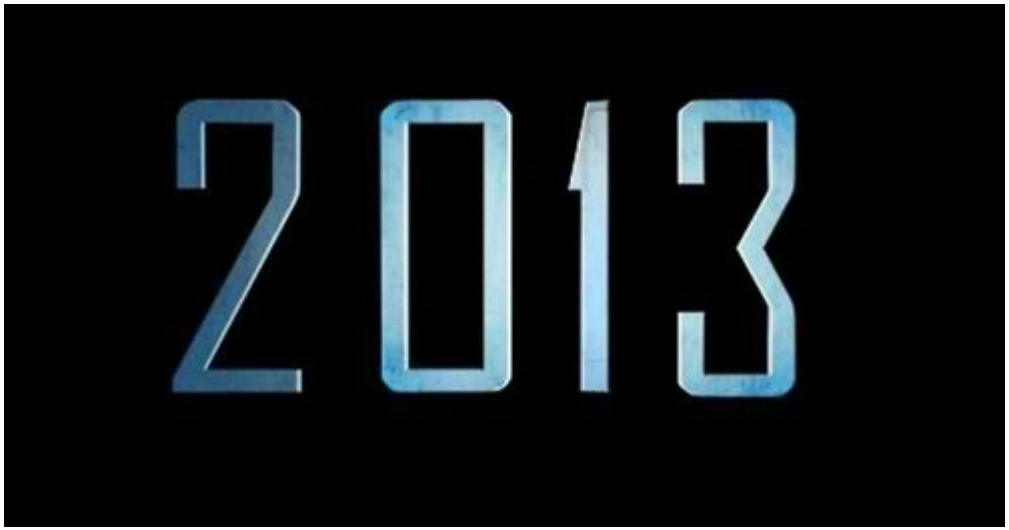
    aux.valor = best[i-1][j-1].valor + valor(i)

    aux.dist = best[i-1][j-1].dist + |i.x–j.x| + |i.y-j.y|

    aux.custo = best[i-1][j-1].custo + custo(j)
  - Posso preencher best[][] passando uma única vez por cada posição em O(A*P)!

F – Twenty Thirteen

- **Autor:** André Restivo (Fac. Engenharia Universidade Porto)
- **Input:** Um número (representando um ano)
- **Output:** Próximo número com dígitos diferentes e sequenciais
- **Limites:** 10 000 perguntas, número < 9876543210



F – Twenty Thirteen

- Pesquisa exaustiva não passa no tempo
  - Pesquisar todos os anos, incrementando 1 de cada vez
  - No entanto, solução “bruta” pode e deve ser implementada para... testar uma outra solução que tenham

- Vamos tentar pensar de maneira “ad-hoc” sobre como obter o **próximo** número:
  - Basta alterar apenas o último dígito?
  - Basta alterar apenas os 2 últimos dígitos?
  - E alterar N dígitos?

F – Twenty Thirteen

- Como saber se basta alterar N últimos dígitos?
  - Antes desse dígito são todos diferentes:
    - Ex: **1**23**1**XX não dá para mudar apenas XX
  - Dígitos por preencher entre min e max são no máximo N:
    - Ex: **48**XX não dá porque entre 4 e 8 faltam 5, 6 e 7
  - Começar a ver no dígito seguinte onde queremos mudar:
    - Ex: 47**5**390. Se queremos mudar 4 últimos, é necessário começar a ver 47**6**XXXX
  - A partir daí usar ordem crescente dos não usados
    - Ex: 375XXX fixo, preencher com 375**246**

F – Twenty Thirteen

- Vamos ver exemplo real com número 571923
  - Mudar apenas último? 57192X? Não porque faltam 3, 4, 6 e 8
  - Mudar 2 últimos? 5719XX? Não porque faltam 2, 3, 4, 6 e 8
  - Mudar 3 últimos? 571XXX? Não porque faltam 2, 3, 4 e 6
  - Mudar 4 últimos? 57XXXX? Sim, porque falta apenas o 6!
  - 571XXX: a seguir a 1, primeiro dígito usável é o 2!
    - Colocando 2, falta 3, 4 e 6 para três posições
  - 572XXX: é só colocar o 3, 4 e 6 por ordem crescente
  - 572346 É a resposta correcta.

F – Twenty Thirteen

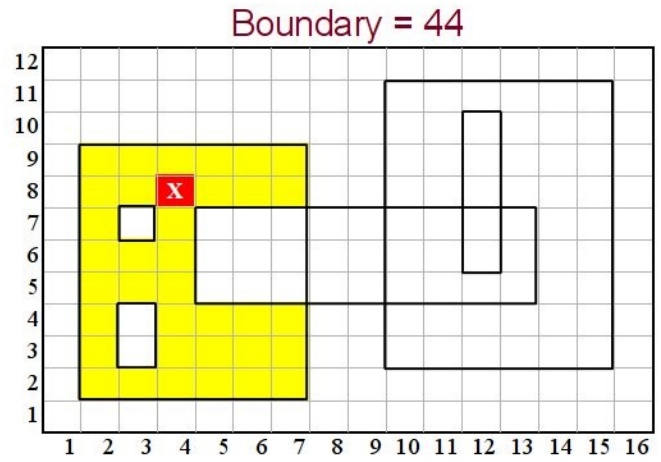
- Resolver o problema podia ser verificar se:
  - Posso mudar apenas dígito final?
  - Posso mudar 2 dígitos finais?
  - …
  - Posso mudar n dígitos?
  - Se nenhuma destas hipóteses der (ex: 99) é só escrever o um prefixo deste número: 1023456789
    - Ex: primeiro ano > 99 é 102

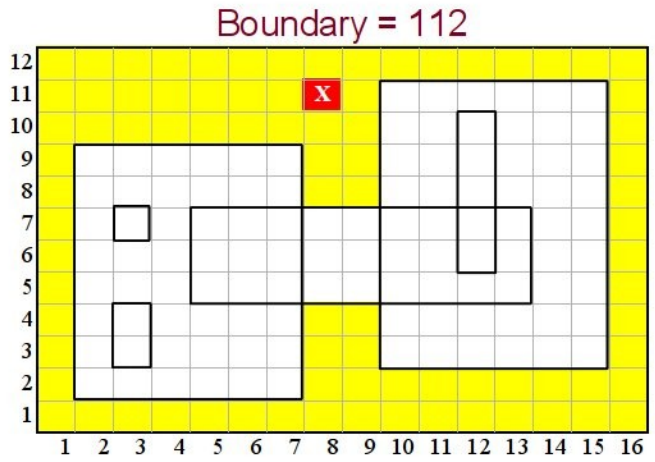
      primeiro ano > 9999 é 10234

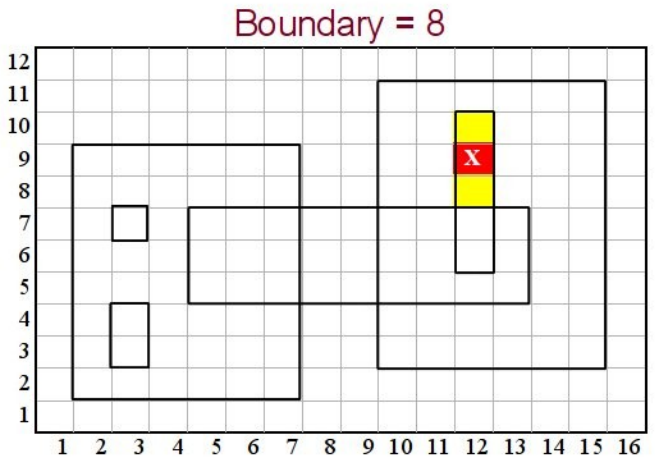
      primeiro ano > 999999 é 1023456

E – Boundaries

- **Autor:** Pedro Ribeiro (Fac. Ciências Universidade do Porto)
- **Input:** Um conjunto de rectângulos e de pontos
- **Output:** O "perímetro" da região que circunda cada ponto
- **Limites:** 500 rectângulos, 500 pontos, coordenadas até 1 milhão

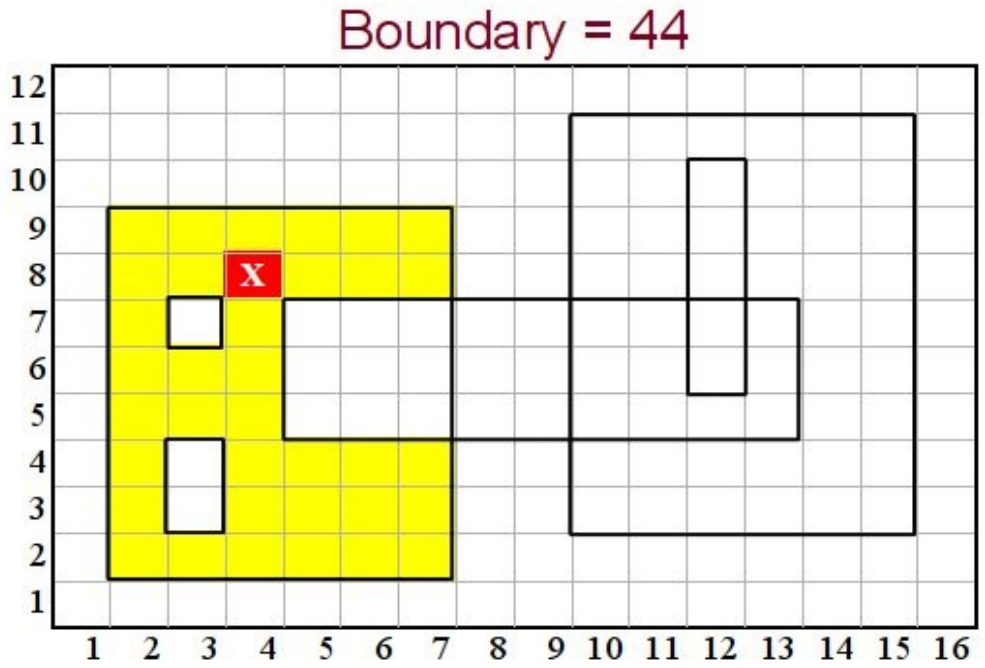






E – Boundaries

- Vamos imaginar que as coordenadas era mais pequenas e a grid cabia em memória (ex: 1000x1000)
  - Resolver o problema era usar um normal **flood fill**
  - Ao contrário de contar células era necessário contar tamanho das bordas (como as representar?)

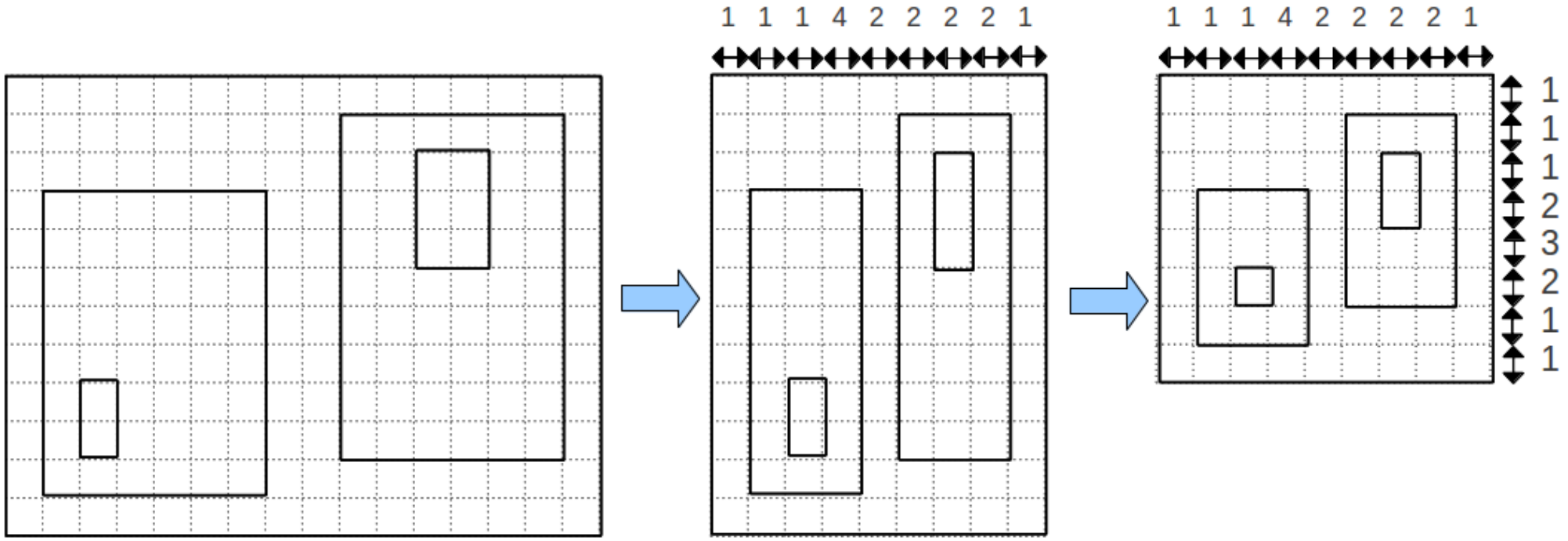


E – Boundaries

- Como lidar com coordenadas tão grandes?
- Nem todas as coordenadas interessam!
- Existem apenas 500 rectângulos...
  - Apenas interessa quando começa a termina um rectângulo
  - No máximo apenas em 1000 coordenadas X acontece algo!
  - No máximo apenas em 1000 coordenadas Y acontece algo!

E – Boundaries

- Podemos “**comprimir**” as coordenadas:
  - No máximo obtemos grid de 1000x1000



E – Boundaries

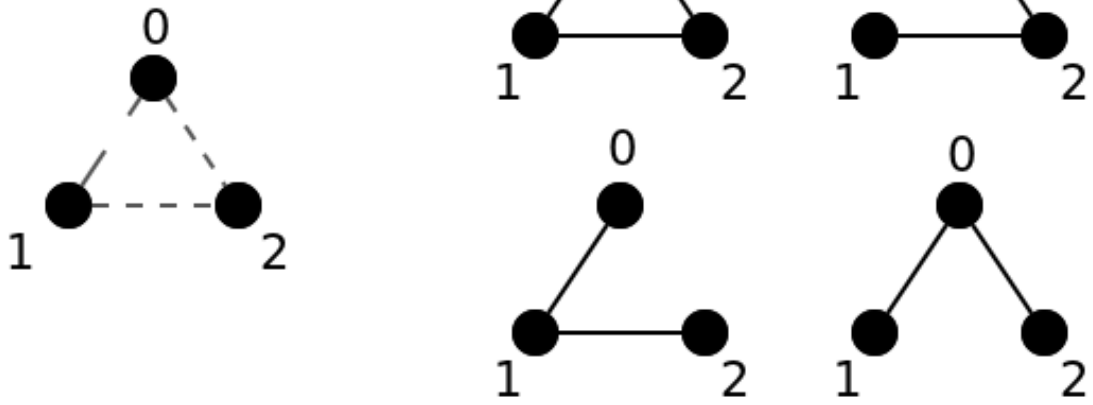
- Na prática:
  - Conjunto ordenado das coordenadas que interessam (X e Y)
  - Para “converter” uma coordenada na sua correspondente comprimida, podemos usar pesquisa binária
    - Cuidado com coordenadas de query que estão “no meio”!
  - Ter em conta o tamanho real de cada célula comprimida

E – Boundaries

- Tudo isto ainda não bastava!
  - Um flood fill normalmenten é implementado **recursivamente**
  - Sem algum cuidado, podemos ter um Stack Overflow”
    - Profundidade da pesquisa recursiva pode ser o nº de células de uma “área”!
  - Como resolver? Mudar para flood fill **iterativo**
    - Ex: usar BFS com filas
  - Para além disso era necessária uma “cache” de resultados
    - Não recalcular o perímetro de uma área já calculada

B – Say Geese

- **Autor:** Filipe Araújo (Universidade de Coimbra)
- **Input:** Grafo (nós/vértices + ligações/arestas)
- **Output:** Número total de “spanning subgraphs” (SS), aka *factor*
- **Limites:** 11 nós, 45 arestas



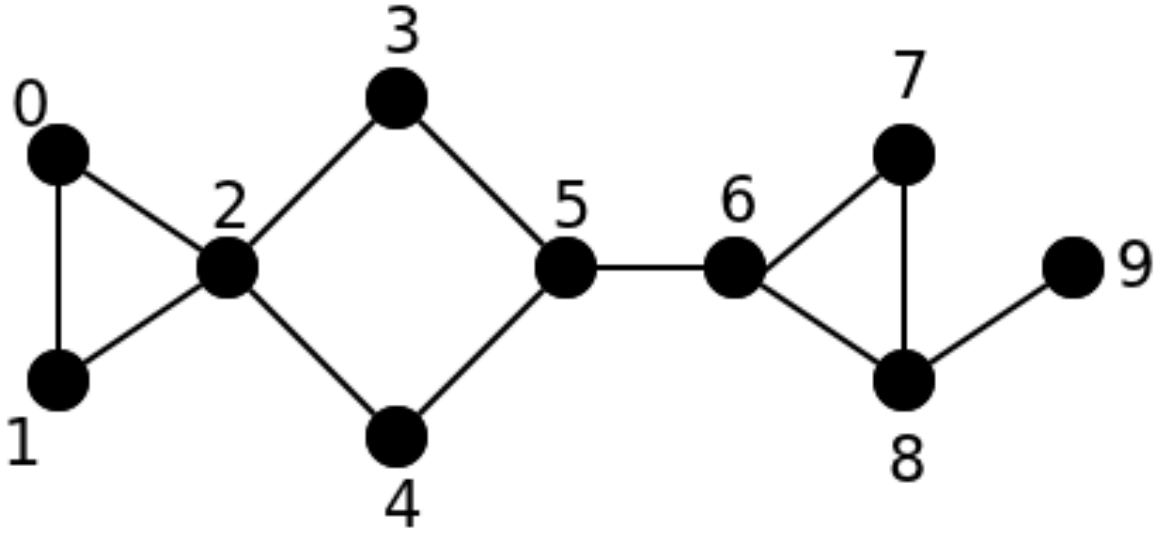
B – Say Geese

- O input parece pequeno... mas não é!
- Número de SS cresce exponencialmente
- Pesquisa exaustiva não chega para resolver
  - Bruto: 2^(nº arestas)

- Podemos pensar ao contrário
  - Retirar arestas enquanto não “desligarem” o grafo
- Mas só isto também não chega!

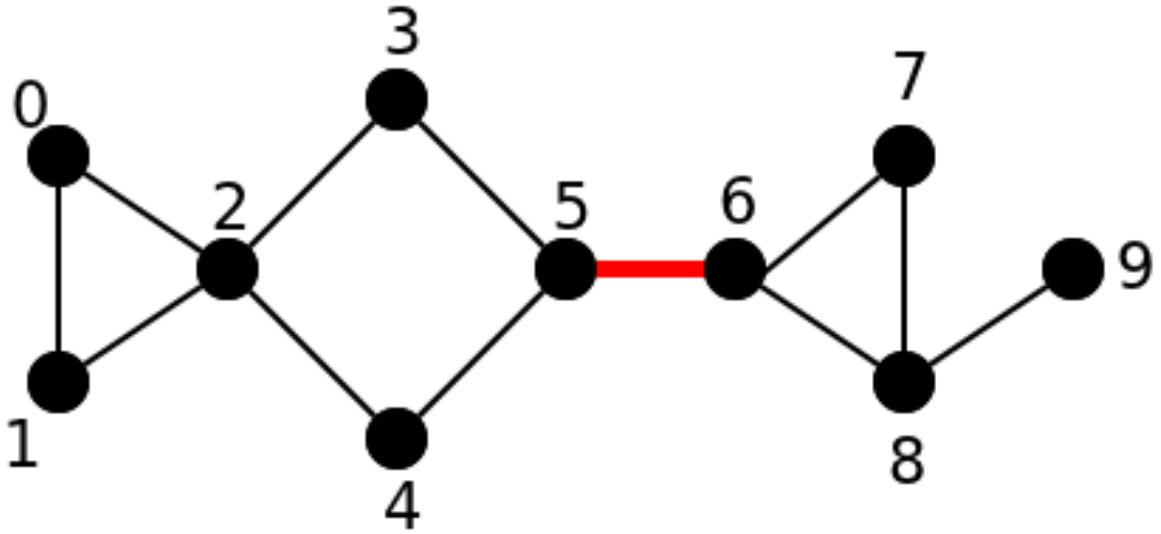
B – Say Geese

- Vamos pensar em termos de grafos...
  - Dado uma aresta, o que podemos fazer com ela?



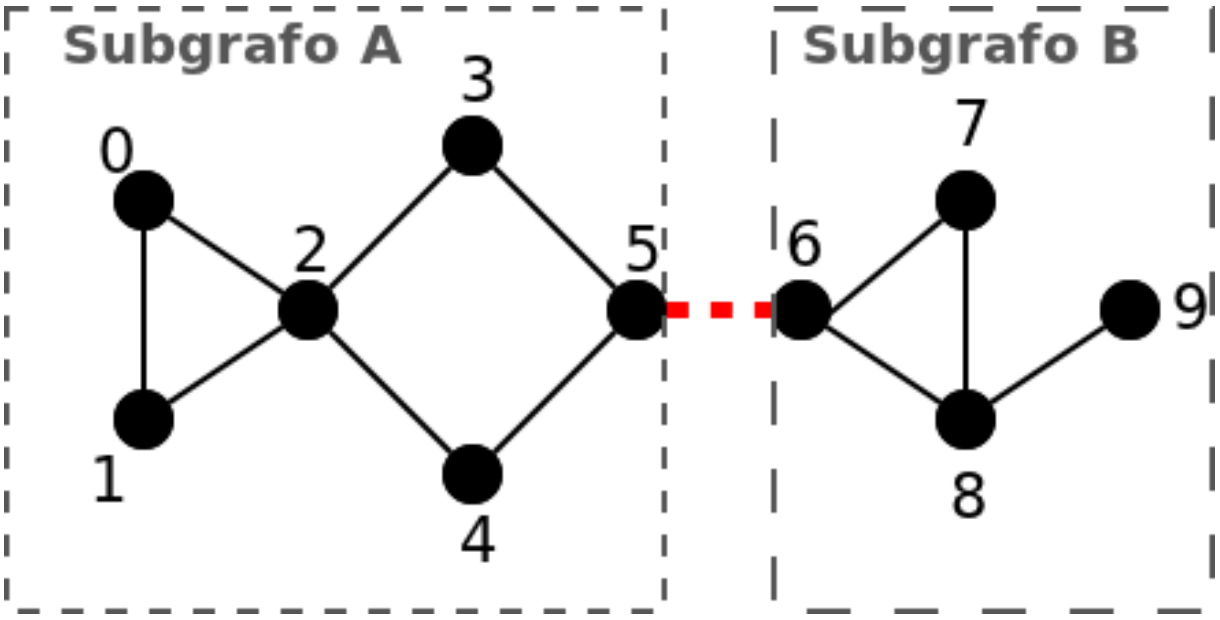
B – Say Geese

- A aresta 5-6 é uma “ponte” (*bridge*)
  - Se removida, “desliga” o grafo (deixa de ser conexo
    - Bastam 2 DFSs em O(n) para saber se é ponte
  - Todos os SS incluem esta aresta!
  - O que fazer com isso ?



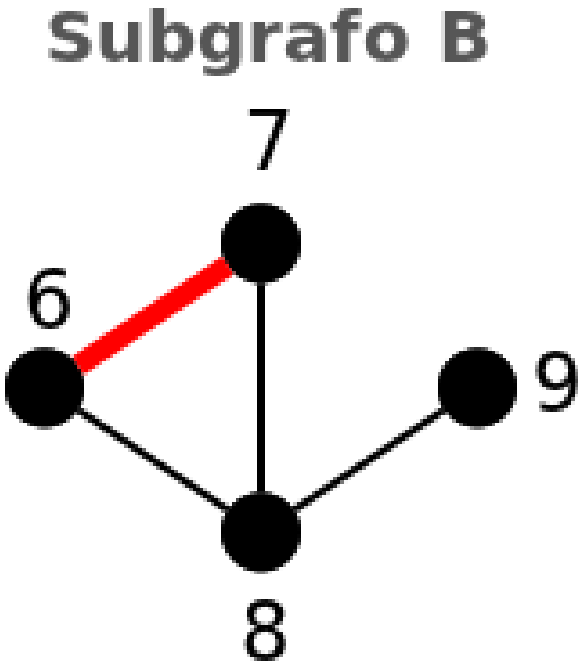
B – Say Geese

- A aresta 5-6 é uma “ponte” (*bridge*)
  - Número de SS total é igual a:
    - **Nº de SS de A x Nº de SS de B**
  - Temos agora dois subproblemas mais pequenos: A e B
    - **Dividir para conquistar!**



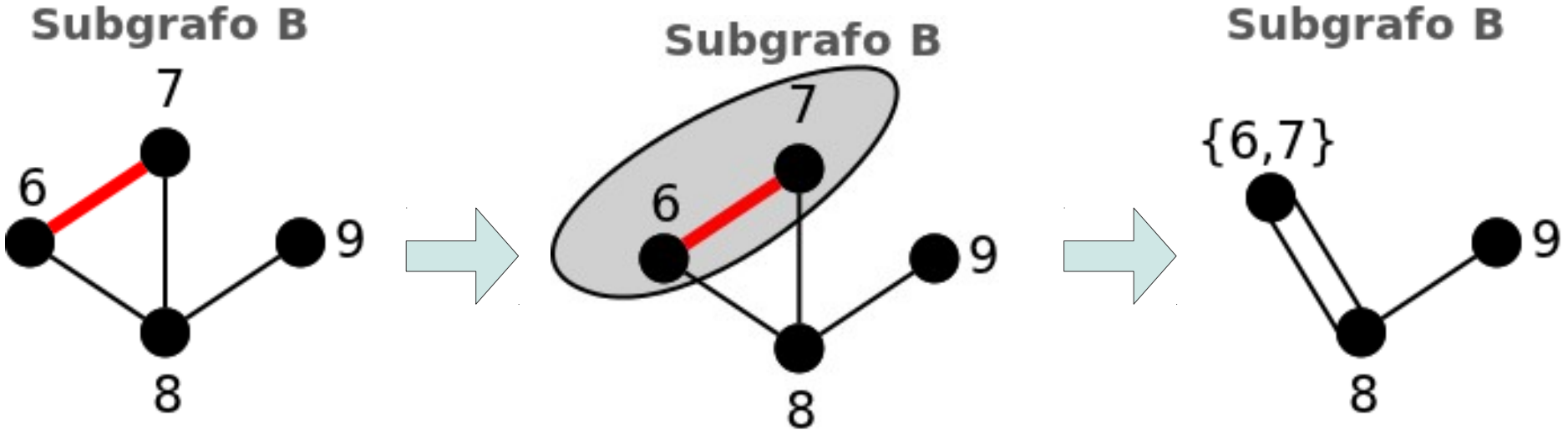
B – Say Geese

- Mas o que fazer se a aresta não é uma ponte? (ex: 6-7)
  - Podemos optar por incluir a aresta
    - Neste caso temos subproblema mais pequeno
  - Podemos optar por não incluir a aresta



B – Say Geese

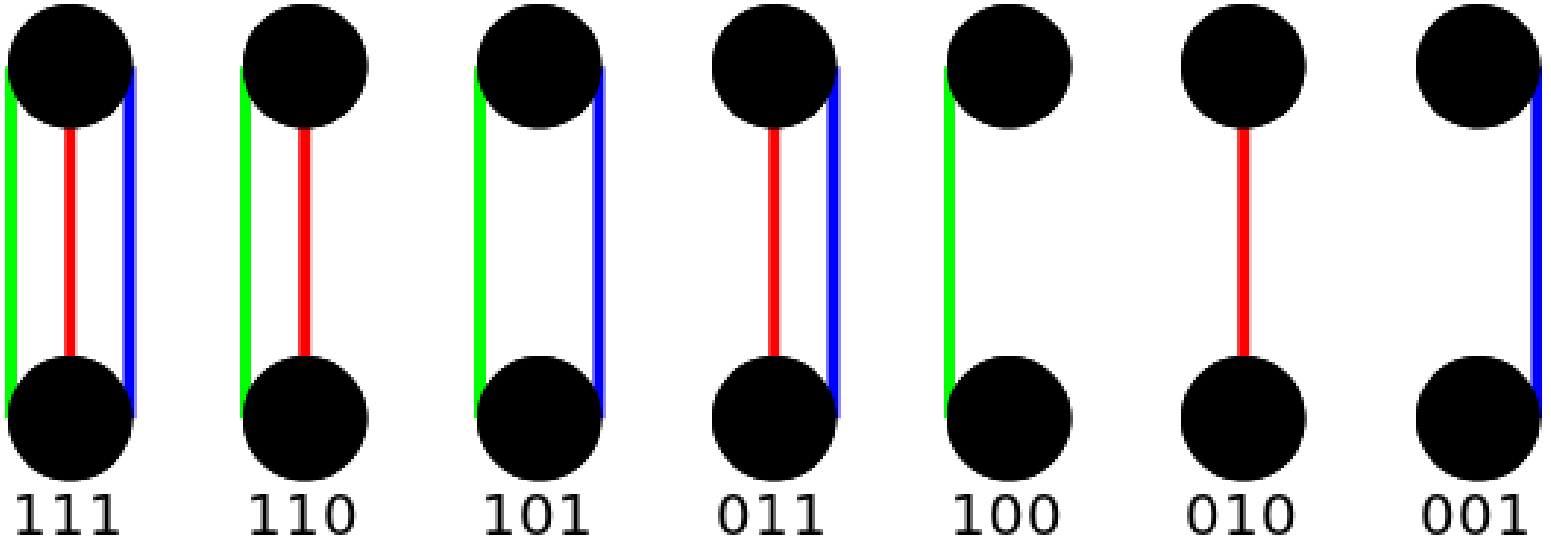
- Se optarmos por manter a aresta, podemos "**contrair**" o grafo
  - Unir os dois vértices da aresta num único vértice
  - Ter em conta quantas arestas existem antes
  - Temos novo subproblema mais pequeno
  - É um multigrafo (múltiplas arestas entre nós)



B – Say Geese

- O que altera o multigrafo?
  - Se existerem ***m*** ligações entre A e B então existem

    (**2^*m*)-1** maneiras de manter A e B ligados
    - Todas as 2^*m* combinações menos a de todas desligadas



B – Say Geese

- Agora tudo. Para um dado grafo G:
  - Escolher uma aresta do grafo (aridade *m*)
  - Se aresta é ponte:
    - Retirar aresta e criar dois novos subgrafos A e B
    - **#SS(G) = #SS(A) x #SS(B) x (2^m-1)**
  - Se aresta não é ponte:
    - Retirar aresta e criar novo subgrafo C
    - Comprimir aresta e criar novo subgrafo D
    - **#SS(G) = #SS(C) + #SS(D) * (2^m-1)**
  - Ordem das arestas influencia tempo
    - Pontes “cortam” mais o grafo

Conclusão

- Um conjunto de problemas **interessante** e **abrangente**

- Estes **slides** e mais um **conjunto de links** serão colocados no site oficial da MIUP

- A organização irá disponibilizar uma "**pós-prova**": tentem resolver os problemas!

- Contactem-me se precisarem de ajuda. Tal como vocês, gosto muito do mundo dos concursos de programação :)