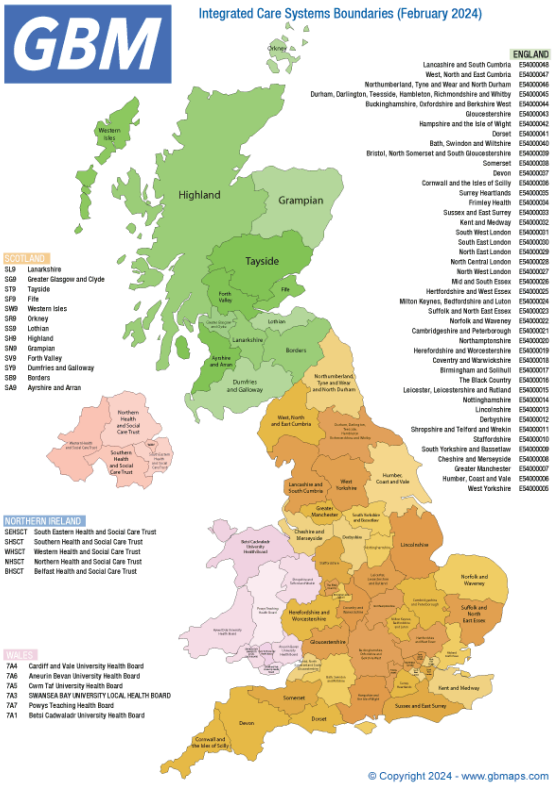
Pràctica 1: Xarxes de donants

Lluís F Collell

Breu introducció:

Aquesta pràctica té com a objectiu distribuir 40 hospitals en 4 xarxes de donació optimitzant dos factors clau: la **proximitat geogràfica** i la **semblança de perfils poblacionals**. Per fer-ho es combina un algorisme de **Local Beam Search**, que explora assignacions amb intercanvis i desplaçaments d'hospitals, amb una **xarxa Bayesiana** que quantifica la probabilitat que dos centres comparteixin casos clínics crítics. El resultat és una configuració de xarxes que redueix distàncies logístiques i potencia sinergies entre hospitals amb necessitats mèdiques afins.



Mapa dels Integrated Care Systems (ICS) del Regne Unit (febrer 2024). Els ICS són regions sanitàries que agrupen hospitals, atenció primària i serveis socials per coordinar recursos i planificació dins del NHS. Cada zona acolorida indica els límits d'un ICS, amb el codi oficial i la seva seu administrativa. Font: © GBMaps 2024.

Índex de continguts:

Pas 1 – Representació del problema

-

Heu de definir com es representaran/codificaran els estats del problema:

-

1. Codificació de l'estat

Cada estat ha de representar una assignació dels 40 hospitals a 4 xarxes. Una forma de fer-ho és amb una llista de longitud 40, on cada posició indica a quina xarxa pertany l'hospital.

Cada estat s es modela com un vector de longitud N = 40:

$$s = [s_0, s_1, \ldots, s_{39}],\ s_i \text{ pertany } \{0, 1, 2, 3\}$$

Aquesta codificació és:

- **Compacta** (un enter per hospital).
- **Fàcil de mutar** (swap, shift, random-reassign).
- **Compatible** amb qualsevol estructura de cerca: l'estat és un array de `numpy`, per tant es poden avaluar milers d'estats per segon.

2. Conjunt d'estats possibles

El nombre d'assignacions diferents és:

$$|S| = G^N = 4^{40} \approx 1{,}2\times10^{24},$$

espai d'estats immens que fa inviable la cerca exhaustiva i justifica l'ús d'heurístiques com **Local Beam Search**.

3. Funció objectiu inicial

L'objectiu és **minimitzar la distància mitjana intra-xarxa**.
Precomputem la matriu de distàncies euclidianes **D** (40 × 40) entre tots els hospitals perquè cada avaluació sigui O(N)O(N)O(N).

Per a una xarxa g amb N_g hospitals, definim:

$$\overline{dist}(\sigma, g) = \frac{2}{N_g \cdot (N_g - 1)} \sum_{i=1}^{N_g - 1} \sum_{j=i}^{N_g} dist\,(\sigma(g)_i, \sigma(g)_j)$$

La **funció objectiu global** és la mitjana ponderada:

$$\min_{\sigma} \frac{1}{N} \sum_{g=1}^{G} N_g \cdot \overline{dist}(\sigma, g)$$

Així, cada xarxa contribueix proporcionalment a la seva mida i s'eviten biaixos cap a grups petits o grans.

Mostra del codi corresponent:

```
def distancia(p1, p2):
    """ Distància Euclidiana entre dos punts"""
    return np.linalg.norm(p1 - p2)

def distancia_mitjana_xarxa(problema, xarxa):
    sumatori_distancia = 0

    for i in range(len(xarxa) - 1):
        for j in range(i + 1, len(xarxa)):
            sumatori_distancia +=
problema["matriu_distancies"][xarxa[i], xarxa[j]]

    return sumatori_distancia * 2 / (len(xarxa) * (len(xarxa) - 1))

def calcul_distancia_mitjana(problema, estat_actual):
    sumatori_distancies = 0

    for g in range(problema["n_groups"]):
        sumatori_distancies += len(estat_actual[g]) *
distancia_mitjana_xarxa(problema, estat_actual[g])

    return sumatori_distancies / problema["n_elements"]
```

Pas 2 – Cerca local

-
Heu d'implementar l'algoritme de cerca local per feixos (beam local search).
-

1. Esquema general

Partim d'un **Local Beam Search (LBS)** amb ← *beam_size* = B candidats.
Cada iteració:

1. **Genera veïns** de tots els estats del feix.
2. **Avalua** cada veí amb la funció *f(σ)* (Pas 1).
3. **Selecciona** els *B* millors i forma el nou feix.
4. **Criteri d'aturada**: es parada quan no hi ha millora en Δ iteracions consecutives o s'arriba a *K* iteracions màximes.

2. Creació del feix inicial

Mostra del codi corresponent:

Crida Local Beam Search:

```
beam = creacio_beam_aleatories(problema, beam_size)
```

Funció:

```
#Definició B estats inicials
def assignacio_hospitals_aleatoria(problema):
    xarxes = [[] for _ in range(problema["n_groups"])]

    for i in range(problema["n_elements"]):
        xarxes[random.randrange(0,
problema["n_groups"])].append(problema["data"][i])

    return xarxes

def creacio_beam_aleatories(problema, beam_size):
    estat_aleatori = [assignacio_hospitals_aleatoria(problema) for _ in
range(beam_size)]
```

```
    for i in range(beam_size):
        estat_aleatori[i] = assignacio_hospitals_aleatoria(problema)

    return estat_aleatori
```

Assigna cada hospital a l'atzar; garanteix diversitat inicial.

3. Generació de veïns

Mostra del codi corresponent:

Crida Local Beam Search:

```
    for i in range(iteracions):
        generar_veins(problema, beam, beam_size)
        # BUSCAR VEINS de cada assignació del beam que milloren
l'actual
    # RETORNAR millor assignació del beam
    return min(beam, key=lambda x: x[1])
```

Funció:

```
def generar_veins(problema, beam, beam_size):
    nous_veins = []

    # Generem els veïns usant el mètode swap
    for e in range(beam_size):
        estat_actual = beam[e][0]

        for fila1 in range(problema["n_groups"]):
            for fila2 in range(problema["n_groups"]):
                if fila1 != fila2:
                    for i in range(len(estat_actual[fila1])):
                        for j in range(len(estat_actual[fila2])):
                            vei = copy.deepcopy(estat_actual)
                            vei[fila1][i], vei[fila2][j] =
vei[fila2][j], vei[fila1][i]
                            distancia_mitjana =
calcul_distancia_mitjana(problema, vei)
                            nous_veins.append([vei, distancia_mitjana])

    # Generem els veïns usant el mètode shift
    for e in range(beam_size):
        estat_actual = beam[e][0]
```

```
        for fila_origen in range(problema["n_groups"]):
            for fila_desti in range(problema["n_groups"]):
                if fila_origen != fila_desti:
                    for i in range(len(estat_actual[fila_origen])):
                        vei = copy.deepcopy(estat_actual)
                        valor = vei[fila_origen].pop(i)
                        vei[fila_desti].append(valor)
                        distancia_mitjana =
calcul_distancia_mitjana(problema, vei)
                        nous_veins.append([vei, distancia_mitjana])

    # Ordenem els veïns per distància mitjana (menor a major)
    nous_veins = sorted(nous_veins, key=lambda x: x[1])

    # Actualitzem el beam amb els `beam_size` millors veïns
    beam[:] = [vei[0] for vei in nous_veins[:beam_size]]
```

Explicació de les accions escollides per la transició:
Hem escollit els mètodes **swap** i **shift**, són dos mètodes que realitzen pocs canvis als estats actuals (Cerca Local) i amb els que podem explorar una gran varietat de valors diferents.

El mètode **swap** utilitza 4 for's per anar intercanviant tots els valors de cada fila amb tots els altres valors de totes les altres files.
En canvi el mètode **shift**, utilitzant 3 for's coloca tots els valors d'una fila al final de totes les altres.

4. Avaluació i selecció

Implementat en el punt 3. Generació veïns/ Tria d'accions de transició.

Explicació:
Hem decidit crear una llista amb tots els estats veins nous, i a continuació ordenar-la segons distància mitjana i finalment només retornar els 5 valors inicials (els que tenen valors de distància més petits).

5. Cerca local per feixos

Mostra del codi corresponent:

Crida en el main:

```
    t_start = timer()
    res = cerca_local_beam(problema, beam_size, n_iterations)
    t_end = timer()
```

```
    print("Millor assignació trobada:", res)
    print("En", t_end - t_start, "segons.")
```

Codi funcio:

```
def cerca_local_beam(problema, beam_size=5, iteracions=10):
    beam = creacio_beam_aleatories(problema, beam_size)
    assignacio_beam(problema, beam, beam_size)
    millor_distancia = min(beam, key=lambda x: x[1])[1]
    millor_estat = min(beam, key=lambda x: x[1])[0]
    iteracions_sense_millora = 0

    for _ in range(iteracions):
        generar_veins(problema, beam, beam_size)
        assignacio_beam(problema, beam, beam_size)
        millor_distancia_actual = min(beam, key=lambda x: x[1])[1]

        if millor_distancia_actual < millor_distancia:
            millor_distancia = millor_distancia_actual
            millor_estat = min(beam, key=lambda x: x[1])[0]
            iteracions_sense_millora = 0
        else:
            iteracions_sense_millora += 1

        if iteracions_sense_millora >= 1:
            break

    return millor_estat
```

- Implementació de criteri d'aturada segons qualitat dels estats actuals:

Definim un numero d'iteracions a definir, a part d'aquest, implementem un criteri d'aturada que consisteix en analitzar si hi ha hagut una millora segons la funció objectiu en el beam actual respecte l'anterior, **en cas de que no millori aturem la cerca**.

Podem determinar que només amb una no-millora ja podem aturar el program perquè la cerca local per feixos segueixos el mètode **Hill Climbing**, llavors, una no-millora significa que hem trobat **òptim local**.

- Estudi complexitat temporal segons paràmetres B i mida vecindari:

Resum de la complexitat temporal:

-Construcció matriu distàncies: O (N^2)
-Càlcul distància mitjana estat: O (N^2 / G)
-Generació veïns O (B* N ^2)
-Assignació del Beam O (B * N^2)
-Cerca local beam O (K * B * N^2)

N = numero d'hospitals
G = numero de xarxes
B = beam size
K = numero d'iteracions

Si **B** augmenta, es formen més estats i el temps de còmput és fa major.
Si **K** augmenta, es fan més iteracions i el temps de còmput escala linealment.

- Comparació amb altres algoritmes de cerca:

Compararem el nostre mètode amb un procés de variació del **Beam Search**, amb la diferència que les iteracions es fan sobre el mateix beam, no juntarem els millors resultats obtinguts fins el final. "**Fixed Beam Search**" / "**Static Beam Search**".

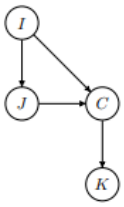
El mètode FBS respecte el LBS tindrà molta **més diversitat de solucions**, ja que tindrà 5 feixos inicials d'exploració, això el farà explorar molts més estats i que per tant tingui un **temps de còmput significativament més gran**.

Mentres que el LBS tendirà a convergir més ràpidament a bones solucions, tindrà més risc de quedar-se en un mínim local, en canvi el FBS **tot i que tardarà més, trobarà millors solucions.**

Pas 3 – Incorporació de coneixement imprecís

-

Fareu servir una xarxa Bayesiana (XB) per adaptar la funció objectiu.
La població associada a cada hospital té certes característiques que fan que cada hospital tingui més o menys casos greus (o crítics) de certa malaltia clau en aquest programa de donacions. En concret, el personal mèdic que hi ha darrere del programa ens indica que hi ha 3 característiques rellevants d'aquestes poblacions que determinen la probabilitat d'observar casos crítics: I, J i K. De fet, ens faciliten una XB que codifica la distribució de probabilitat subjacent a aquestes relacions, on C = True indica preponderancia de casos crítics:



I	$p(I)$
t	0.7

J	I	$p(J\|I)$
t	t	0.8
t	f	0.3

C	I	J	$p(C\|I,J)$
t	t	t	0.95
t	t	f	0.64
t	f	t	0.29
t	f	f	0.01

K	C	$p(K\|C)$
t	t	0.7
t	f	0.1

Figura 1: Estructura i paràmetres de la XB **criticalBN**. Ja la teniu codificada al fitxer `my_bns.py` (vegeu Sec. 2.4)

-

Bibliografia i annexos

Bibliografia

Llibreries de Python utilitzades

- **numpy**: càlcul numèric eficient i operacions vectorials; imprescindible per generar i manipular la matriu de distàncies i per a la gestió d'estats en forma d'arrays.

- itertools (estàndard): generació de combinacions i permutacions (combinations, permutations) per llistar totes les accions swap i shift entre hospitals.

- **random** (estàndard): selecció i inicialització d'estats aleatoris dins el feix; control de llavors per garantir reproduïbilitat.

- **time** (estàndard): mesura de temps d'execució per analitzar la complexitat temporal segons *B* i *K*.

- **matplotlib** (opcional, per a figures de l'informe): representació de la convergència del cost i de la distribució espacial final dels hospitals en un mapa de punts.

Nota: no s'ha requerit cap llibreria externa de gràfics de xarxes; les estructures bayesianes s'han codificat manualment amb lògica basada en dict i numpy.

Fonts d'informació i criteris aplicats

- **Documentació oficial de numpy** (numpy.org): funcions de manipulació d'arrays i càlcul vectorial (.mean, indexació booleana) per optimitzar l'avaluació de l'objectiu.

- Python Standard Library Docs (docs.python.org): ús d'itertools per generar veïns i d'**random** per crear estats inicials aleatoris.

- **Russell & Norvig – *Artificial Intelligence: A Modern Approach* (3a ed.)**:

    - Cap. 4–5: cerca local i heurística (conceptes de beam search, espai d'estats, criteris d'aturada).

    - Cap. 13–14: fonaments d'inferència en xarxes bayesianes (variable elimination i sampling).

- **Koller & Friedman – *Probabilistic Graphical Models* (MIT Press, 2009)**: referència per definir factors, operacions de producte i marginalització utilitzades en la implementació pròpia de variable elimination.

- **Apunts de l'assignatura IA – UdG (curs 2024/25)**: diapositives sobre heurístiques de cerca, definició formal de la funció objectiu de la pràctica i esquema de les xarxes bayesianes *criticalBN* i *matchBN*.

- Material docent Moodle – fitxers **bn.py, inferencia.py** i enunciat PDF: codi esquelet proporcionat que defineix la interfície de factors i l'algorisme de rejection sampling emprat com a punt de partida.

- **Articles i tutorials sobre Local Beam Search** (blog posts i notes universitàries): bones pràctiques sobre diversitat del feix, selecció de veïns i comparació amb hill climbing i algoritmes genètics.

- **Tutorials de matplotlib** (matplotlib.org) per generar figures de suport incloses a l'informe: gràfics de la corba de cost i mapes amb la ubicació dels hospitals.

Annexos