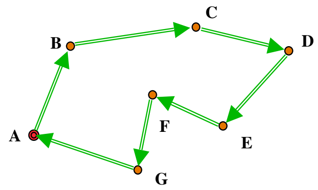
Udtømmende søgning

Udtømmende søgning
(kombinatorisk søgning)

Systematisk gennemsøgning af alle potentielle løsninger

Den rejsende sælgers problem (TSP):
En sælger skal besøge N byer
Find den korteste rundtur

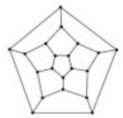
Eksempel med 7 byer



Problem med 4461 byer

Find den korteste rundtur

Udtømmende søgning i grafer



Mulig metode til løsning af TSP:

(1) Generer samtlige rundture (Hamilton-cykler) i grafen
(2) Vælg den korteste af disse

Samtlige rundture kan bestemmes således:

(1*a*) Generer systematisk samtlige simple veje i grafen
(1*b*) Vælg heraf dem, der omfatter samtlige grafens knuder, og hvor første og sidste knude er forbundet med en kant

Systematisk generering af alle simple veje i en graf

Kan opnås ved en lille ændring af metoden DFS til dybde-først-søgning i en graf

Algorithm *DFS*(*G, v*)
  setLabel(*v, VISITED*)
  for all $e \in$ *G.incidentEdges*(*v*)
    $w \leftarrow$ *opposite*(*v,e*)
    if *getLabel*(*w*) = *UNEXPLORED*
      DFS(*G, w*)
  setLabel(*v, UNEXPLORED*)

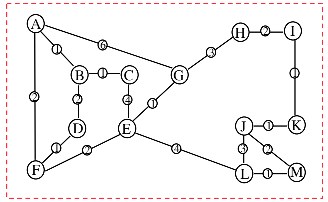
Kald:

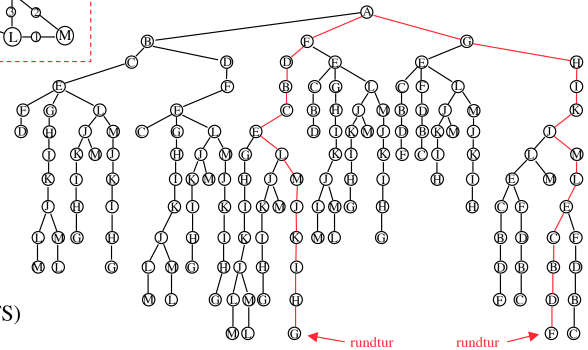
DFS(*G, v*)

hvor *v* er en vilkårlig knude i *G*

Metoden "rydder op efter sig selv", således at alternative veje kan undersøges

Eksempel på søgning





153 knuder (kald af DFS)

Generering af alle simple veje i en graf repræsenteret ved en nabomatrix

`w[k][t]` betegner vægten på kanten (`k`, `t`). Værdien 0 angiver, at kanten ikke findes

`level` angiver antallet af knuder på den aktuelle vej

`pos[k]` angiver positionen for knude `k` på denne vej

```
void dfs(int k) {
    pos[k] = ++level;
    for (int t = 1; t <= V; t++)
        if (w[k][t] != 0 && pos[t] == 0)
            dfs(t);
    level--; pos[k] = 0;
}
```

```
level = 0;
for (int k = 1; k <= V; k++)
    pos[k] = 0;
dfs(1);
```


Løsning af TSP

```
level = 0;
for (int k = 1; k <= V; k++)
    pos[k] = 0;
cost = 0;
best_cost = Integer.MAX_VALUE;
dfs(1);
```

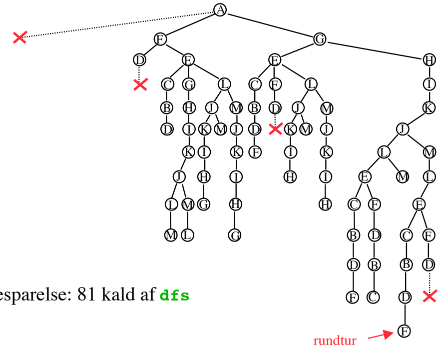

```
void dfs(int k) {
    pos[k] = ++level;
    if (level == V && w[k][1] != 0 &&
        cost < best_cost) {
        best_cost = cost;
        System.arraycopy(
          best_pos, 1, pos, 1, V);
    }
    for (int t = 1; t <= V; t++)
        if (w[k][t] != 0 &&
            pos[t] == 0) {
            cost += w[k][t];
            dfs(t);
            cost -= w[k][t];
        }
    level--; pos[k] = 0;
}
```

Beskæring af søgetræet

Beskæring: fjernelse af undertræer

Eksempel: fjernelse af symmetrier:
Forlang, at 3 knuder kommer i en bestemt rækkefølge

Kravet A -> ... -> C -> ... -> B -> ... beskærer træet



Besparelse: 81 kald af `dfs`

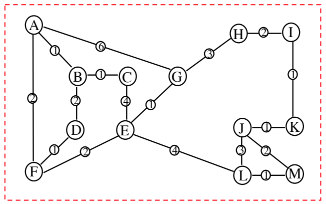
Implementering af beskæring

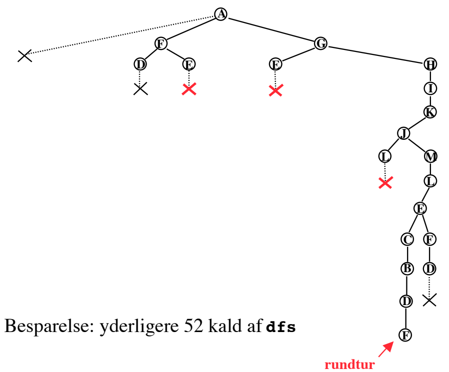
```
void dfs(int k) {
    pos[k] = ++level;
    for (int t = 1; t <= V; t++)
        if (w[k][t] != 0 && pos[t] == 0 &&
            (t != 3 || pos[2] != 0))
            dfs(t);
    level--; pos[k] = 0;
}
```

Besøg kun knude 3, hvis knude 2 er besøgt

Yderligere beskæring

Beskær en gren, hvis de ubesøgte knuder ikke er forbundne





Besparelse: yderligere 52 kald af `dfs`

Forgren-og-begræns
(branch-and-bound)

En metode til reduktion af søgningen ved løsning af optimeringsproblemer

De partielle løsningers omkostninger beregnes og benyttes til beskæring af søgetræet

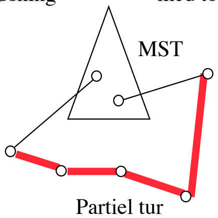
Eksempel: Hvis alle omkostninger for TSP er positive, kan en søgningen ad en vej afbrydes, hvis omkostningen for vejen er større end eller lig med den hidtil korteste rundtur

```
for (int t = 1; t <= V; t++) {
    if (w[k][t] != 0 && pos[t] == 0 &&
        cost + w[k][t] < best_cost) {
        cost += w[k][t];
        dfs(t);
        cost -= w[k][t];
    }
```

Generel beskæringsmetode

Hvis omkostningen for en partiel løsning, **plus** en *nedre* grænse for omkostningen for en færdiggørelse til en komplet løsning, er større end en *øvre* grænse for en løsning, så vil den partielle løsning ikke kunne færdiggøres til en *optimal* løsning

En nedre grænse for TSP udgør omkostningen af et MST (Minimal Spanning Tree) for de ubesøgte knuder, plus de to korteste kanter, der forbinder de to endeknuder i den partielle tur med to ubesøgte knuder



Baksporing
(backtracking)

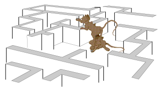
En problemløsningmetode til systematisk generering af alle mulige løsninger

Sålænge det er muligt, udvides en **partiel** løsning

Hvis en partiel løsning ikke kan udvides, "**bakspores**", d.v.s. vendes tilbage til en tidligere partiel løsning, for hvilken der findes en uprøvet udvidelsesmulighed

På denne måde bevæger algoritmen sig **forlæns** og **baglæns**, indtil en løsning er fundet, eller alle muligheder er udtømte.

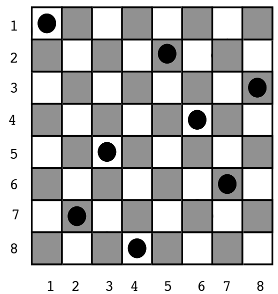
Skabelon til baksporing



```
void try(...) {
    for (alle_kandidater)
        if (kandidat_acceptabel) {
            registrer_kandidat;
            if (ufuldstændig_løsning)
                try(...);
            if (løsning_fundet)
                return;
            slet_registrering;
        }
}
```


8-dronningeproblemet

Placer 8 dronninger på et skakbræt, således at der ikke findes to dronninger, der kan slå hinanden (d.v.s. der er ikke to dronninger i samme række, søjle eller diagonal)

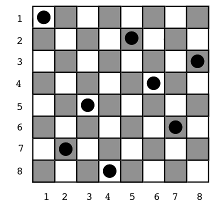


Ineffektiv og ufleksibel løsningsmetode

```
Search:
for (c1 = 1; c1 <= 8; c1++)
for (c2 = 1; c2 <= 8; c2++)
for (c3 = 1; c3 <= 8; c3++)
for (c4 = 1; c4 <= 8; c4++)
for (c5 = 1; c5 <= 8; c5++)
for (c6 = 1; c6 <= 8; c6++)
for (c7 = 1; c7 <= 8; c7++)
for (c8 = 1; c8 <= 8; c8++)
    if (isSolution(c1, c2, c3, c4, c5, c6, c7, c8)) {
        printSolution();
        break Search;
    }
```


Løsning med baksporing

```
void try(int row) {
    for (int col = 1; col <= 8; col++) {
        if (!underAttack(row, col)) {
            setQueen(row, col);
            if (row == 8)
                solutionFound = true;
            else
                try(row + 1);
            if (solutionFound)
                return;
            removeQueen(row, col);
        }
    }
}
```




```
solutionFound = false;
try(1);
if (solutionFound)
    printSolution();
```

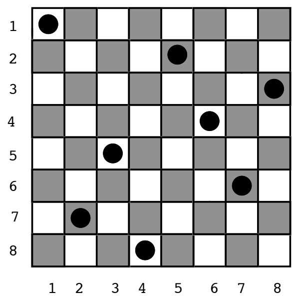

Datastrukturer

Repræsentation af aktuel stilling:
`int q[]; boolean up[], down[];`

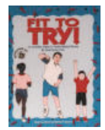
`q[col] == row`, hvis der er placeret en dronning i søjlen **`col`** med rækkenummeret **`row`**; ellers **`0`**

`up[col+row-2] == true`, hvis og kun hvis der er placeret en dronning på opad-diagonalen (↗) med nummer **`col+row-2`**

`down[col-row+7] == true`, hvis og kun hvis der er placeret en dronning på nedad-diagonalen (↘) med nummer **`col-row+7`**.



Færdig udgave af `try`




```
void try(int row) {
    for (int col = 1; col <= 8; col++) {
        if (q[col] == 0 && !up[row+col-2] && !down[col-row+7]) {
            q[col] = row;
            up[row+col-2] = down[col-row+7] = true;
            if (row == 8)
                solutionFound = true;
            else
                try(row+1);
            if (solutionFound)
                return;
            q[col] = 0;
            up[row+col-2] = down[col-row+7] = false;
        }
    }
}
```

Primitiver til baksporing

`choice(n)`
repræsenterer et valg imellem **n** alternativer
Metoden returnerer heltal imellem **1** og **n**

`backtrack()`
signalerer, at problemet ikke kan løses med de hidtil trufne valg

Er implementeret til C, C++, Pascal og Simula af Keld Helsgaun
Findes ikke til Java

Løsning ved brug af primitiverne

```
for (int row = 1; row <= 8; row++) {
    int col = choice(8);
    if (q[col] != 0 || up[row+col-2] || down[col-row+7])
        backtrack();
    q[col] = row;
    up[row+col-2] = down[col-row+7] = true;
}
printSolution();
```


Approksimative algoritmer

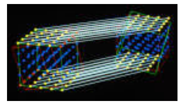
Køretiden for baksporingsalgoritmer er eksponentiel

Hvis hver knude i gennemsnit har α sønner, og længden af en løsningsvej er N, så er køretiden α^N

I nogle tilfælde behøves ikke en optimal løsning - en "rimelig god" løsning er tilstrækkelig

En **approksimativ algoritme** tilstræber at opnå en "rimelig god" løsning på kort tid - sædvanligvis med et polynomielt tidsforbrug

Permutationer



En algoritme til systematisk generering af samtlige permutationer af tallene fra 1 til N kan udledes direkte fra algoritmen til udtømmende søgning i en graf

Når udtømmende søgning anvendes på en **komplet** graf, så gennemløbes alle knuder i enhver mulig rækkefølge

```
void dfs(int k) {
    perm[k] = ++level;
    if (level == V)
        use(perm);
    for (int t = 1; t <= V; t++)
        if (perm[t] == 0)
            dfs(t);
    level--; perm[k] = 0;
}
```