729G78 Artificiell intelligens
Sökning – Heuristisk sökning

Arne Jönsson
HCS/IDA

LINKÖPINGS UNIVERSITET

Heuristisk sökning

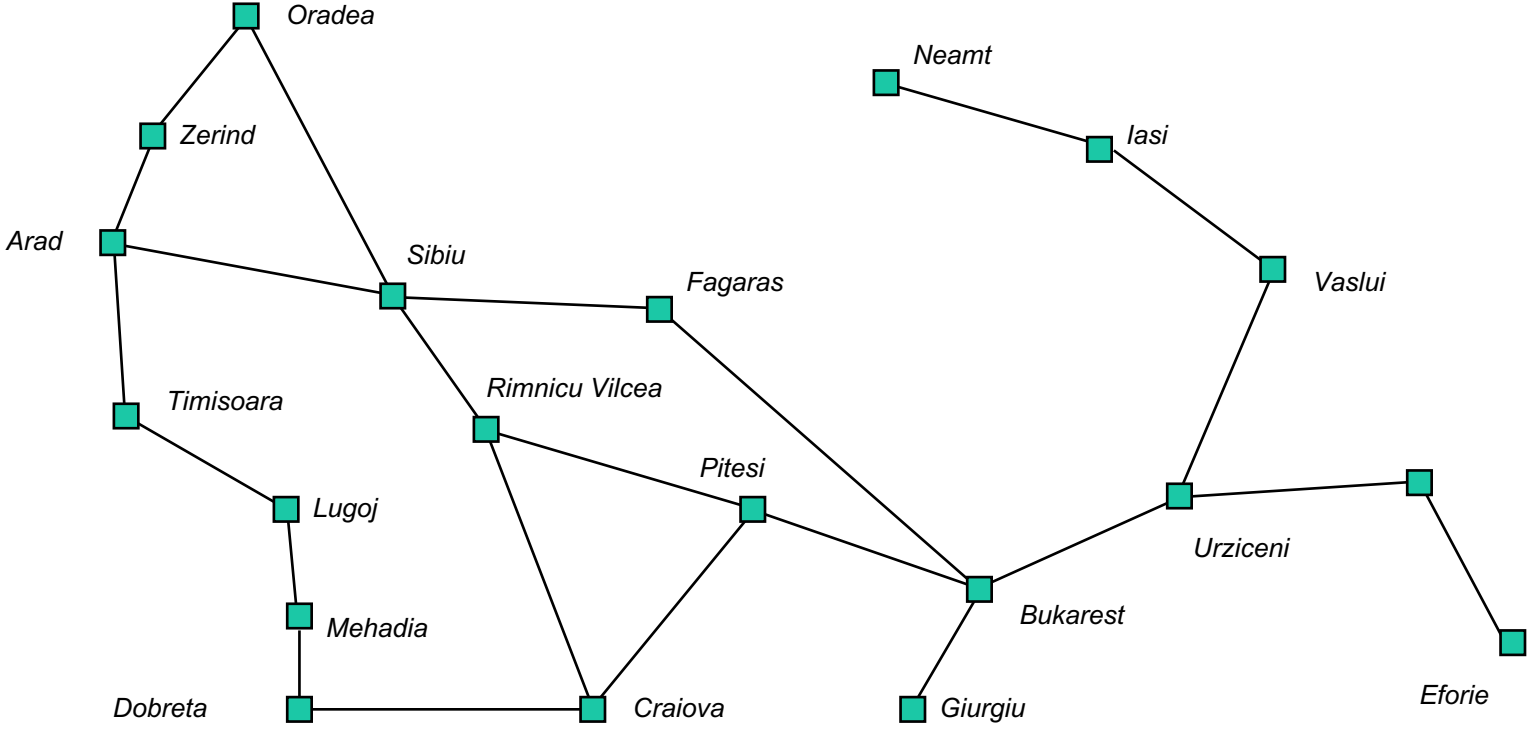
- Antar att vi har någon information om vilken nod som är bäst att expandera
- Greedy search
- A*
- Heuristik
- Hill Climbing

Greedy Search

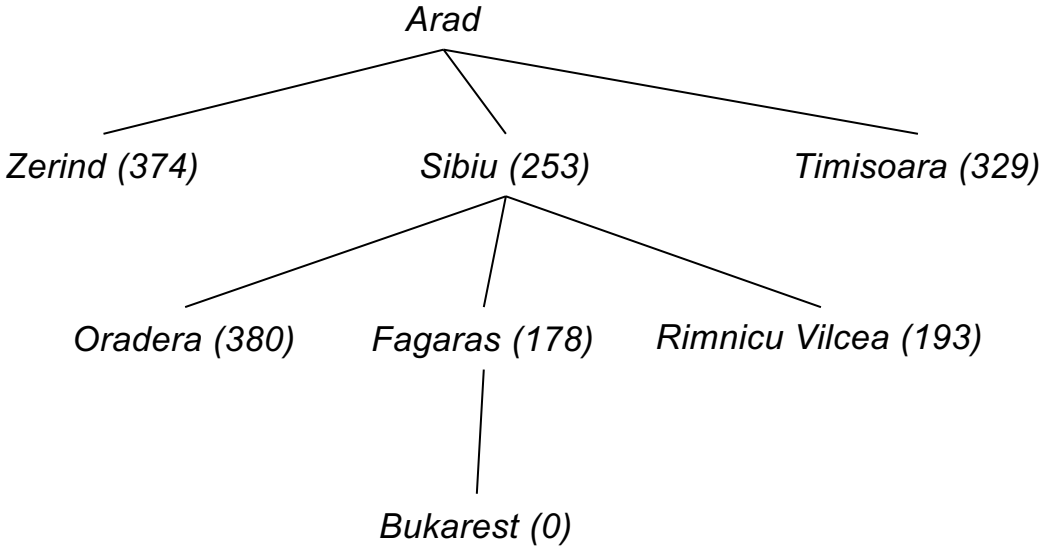
- Minimera uppskattad kostnad till målet, dvs expandra den nod som verkar vara närmast målet
- Inför h(n)= heuristisk funktion som uppskattar handlingskostnaden från nod n till målet
- h(n) = 0 betyder att n är målnod
- Ex Rumänienkartan
  - Fågelavståndet ett exempel på heuristik

h(n)=

Arad	366	Bukarest	0	Craiova	160	Dobreta	242	Eforie	161
Fagaras	178	Giurgiu	77	Hirsova	151	Iasi	226	Lugoj	244
Mehadia	241	Neamt	234	Oradea	380	Pitesi	98	Rimnicu V.	193
Sibiu	253	Tisioara	329	Urziceni	80	Vasliu	199	Zerind	374



Greedy Search, Arad → Bukarest



Greedy Search

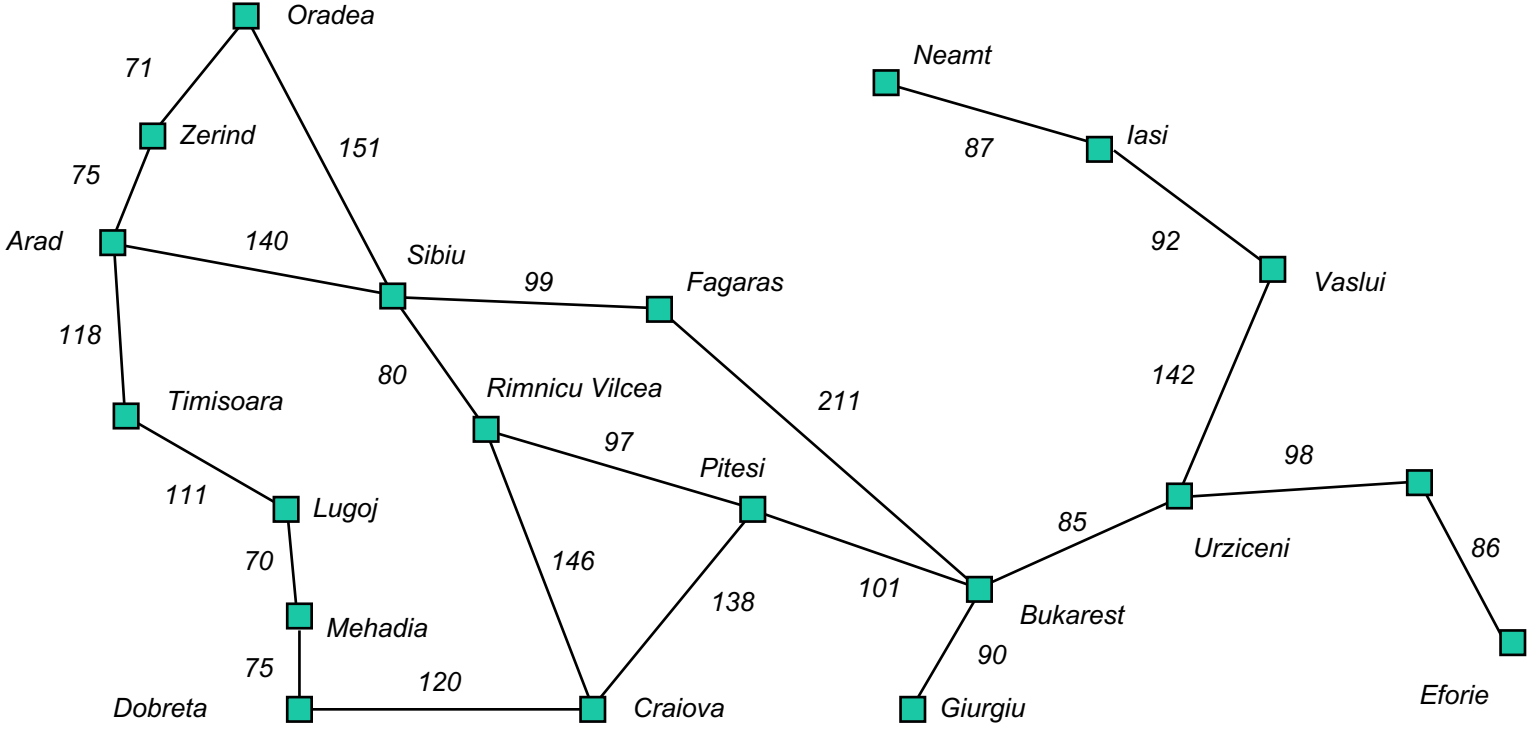
- Liknar djupet först och kan råka ut för samma problem
- Inte optimal
- Inte komplett
- Tids- och minneskomplexitet $O(b^d)$ men kan ofta bli bättre med god heuristik
- Jämför Uniform Cost
  - Komplett och optimal men ineffektiv
- Inför g(n) för kostnad från start för nod n

A*

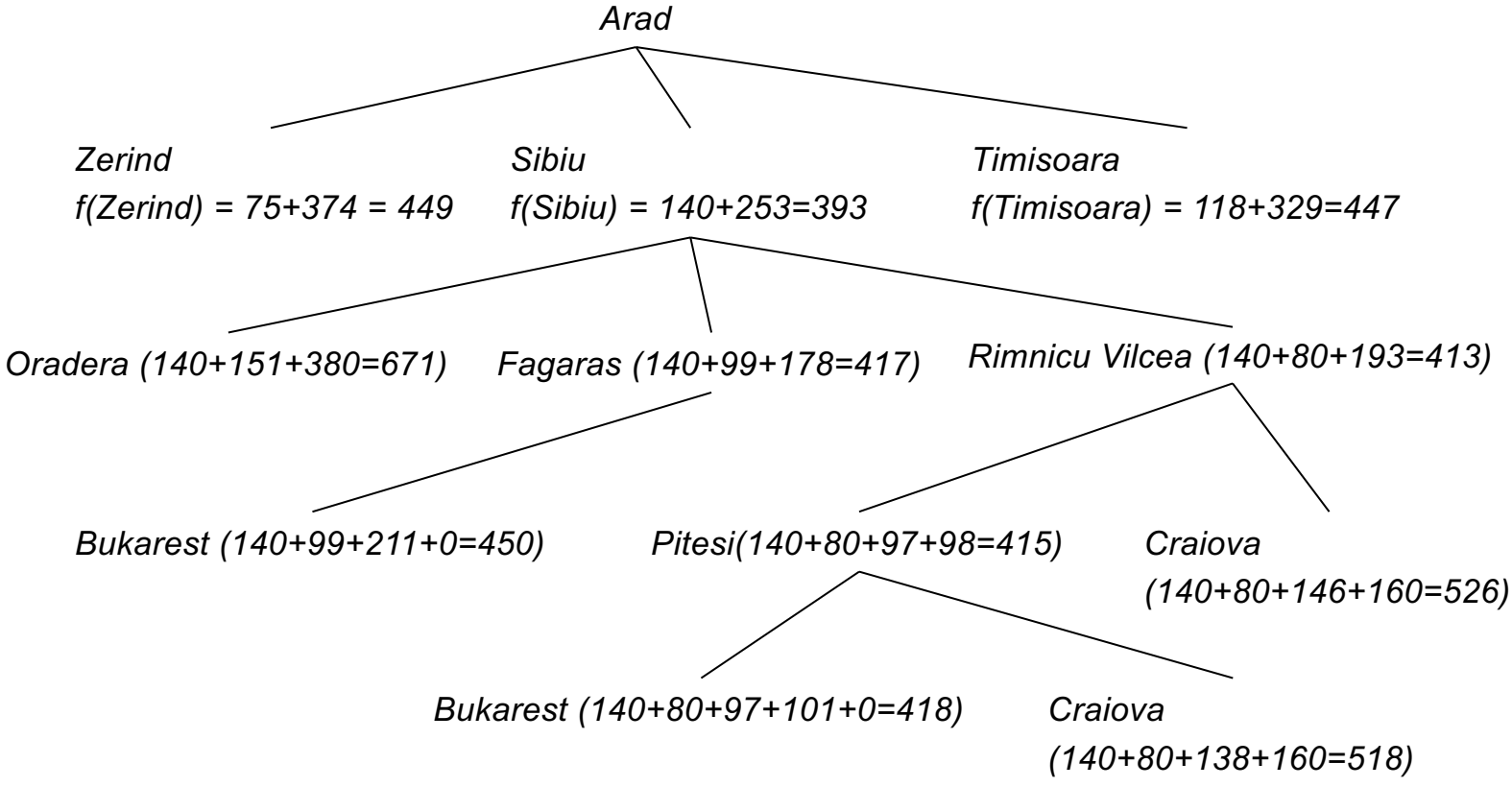
- Inför f(n) = g(n) + h(n), dvs tillryggalagd kostnad + uppskattad återstående kostnad
- Kräv också att h(n) aldrig överskattar den verkliga kostnaden
- h(n) kallas då tillåten (admissible), ex fågelavståndet

h(n)=

Arad	366	Bukarest	0	Craiova	160	Dobreta	242	Eforie	161
Fagaras	178	Giurgiu	77	Hirsova	151	Iasi	226	Lugoj	244
Mehadia	241	Neamt	234	Oradea	380	Pitesi	98	Rimnicu V.	193
Sibiu	253	Tisioara	329	Urziceni	80	Vasliu	199	Zerind	374

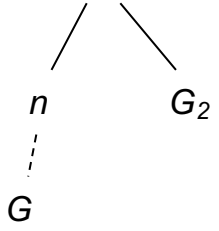


A*, Arad → Bukarest



A* är optimal

- Antag: lösningsnoden G optimum med kostnad C* och
  G2 lösning men med kostnad g(G2) > C*
- För att A* skall välja G2 måste det finnas en nod, n, på vägen till G som inte expanderats



- C* ≥ f(n) eftersom h aldrig överskattar (tillåten)
- f(n) ≥ f(G_2) eftersom A* expanderat till G_2, dvs C* ≥ f(G_2)
- Eftersom G_2 målnod så h(G_2) = 0 ⇒ f(G_2) = g(G_2)
  dvs C* ≥ g(G_2) vilket motsäger antagandet

Evalueringsfunktionen i A*

f(n) = g(n) + h(n)

där

f(n) = totalkostnad

g(n) = kostnad hittills

h(n) = uppskattad återstående kostnad (måste underskatta)

g(n) = 0 → Greedy search

h(n) = 0 → Uniform Cost

g(n) = 1 och *h(n)* = 0 → *f(n)* = 1, dvs Bredden först

Heuristik

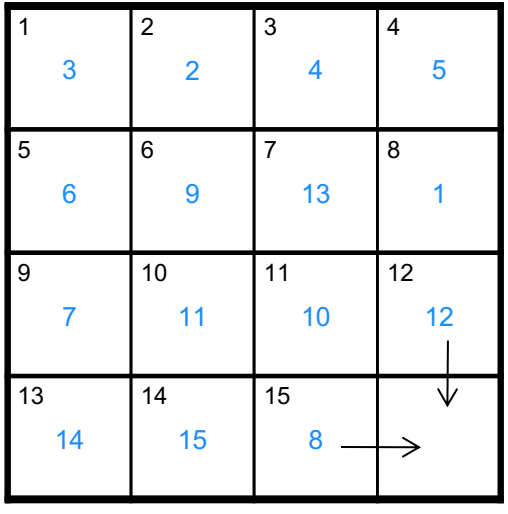
Heuristiska sökmetoder bättre med bättre heuristik, dvs bättre skattning av kostnaden

Ex. 15-spel

1 3	2 2	3 4	4 5
5 6	6 9	7 8	8 1
9 7	10 11	11 10	12 12
13 14	14 15	15 13	

Heuristik, 1

Antal brickor som ligger fel

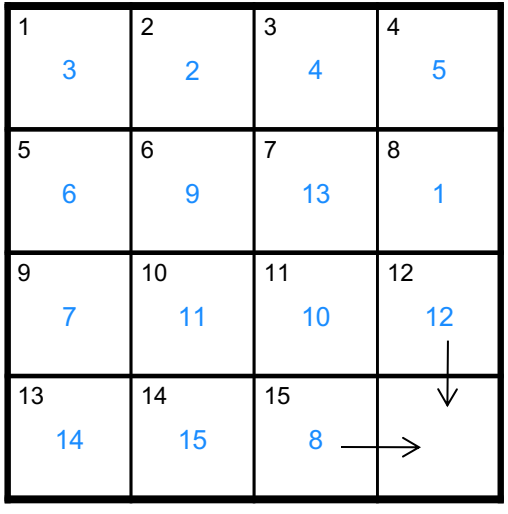


12: (1,3,4,5,6,7,8,9,10,11,12,13,14,15)=14

8: (1,3,4,5,6,7,8,9,10,11,13,14,15)=13

Heuristik, 2

Manhattanavståndet



12: 4+0+2+1+4+1+3+3+2+1+1+1+4+1+1=29

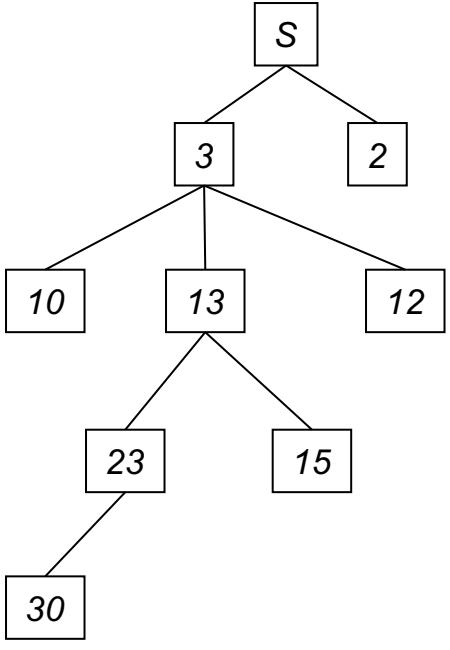
8: 4+0+2+1+4+1+3+2+2+1+1+0+4+1+1=27

Heuristik – konstruktion

- Förenklade problem kan användas för automatisk konstruktion av heuristik
- En optimal lösning för ett förenklat problem är en tillåten heuristik för orginalproblemet

Lokal sökning - Hill Climbing

- Om varje nytt tillstånd innehåller all nödvändig information för att gå vidare, dvs vi kan utvärdera en nod utan hänsyn till varifrån vi kom och vägen mindre viktig

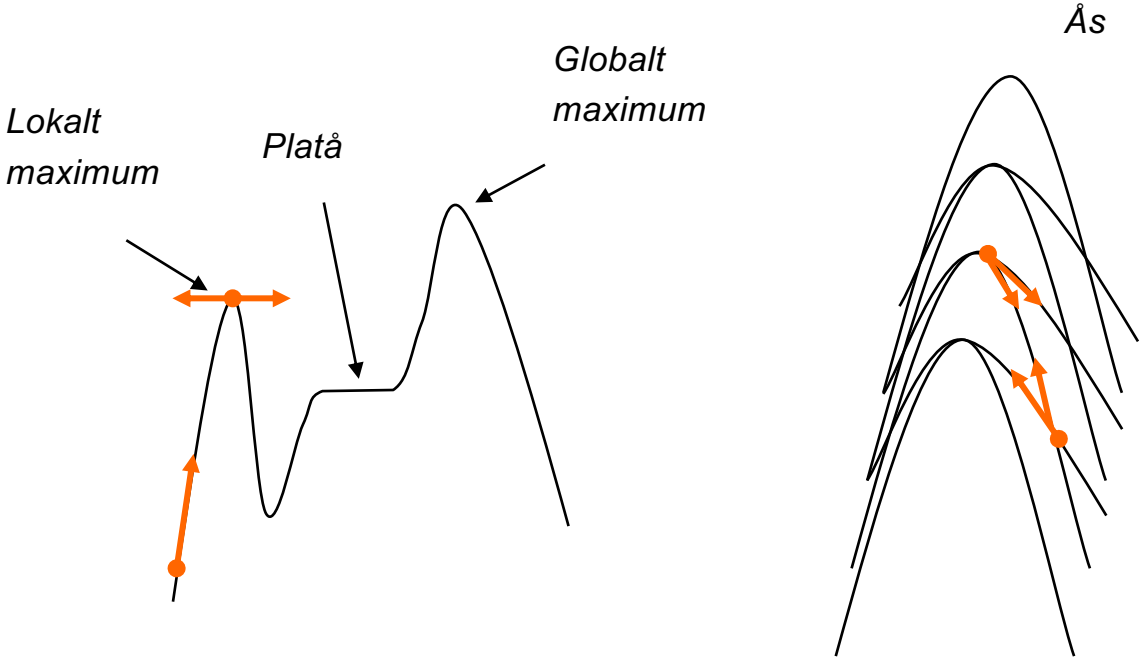


Hill Climbing

```
def hillClimbing (problem):
  current = makeNode(initialState(problem))
  while True:
     next = highestValuedSuccessor(current)
     if value(next) ≤ value(current):
        return current
     current = next
```

Problem för hill climbing

Lokal metod, hittar lokalt optimum



Simulated annealing

- Tillåter hopp till sämre nod ibland, om nya noden inte bättre än gamla
  - Om nya noden bättre så välj alltid att gå vidare med den
- Styrs av slumpen och hur nära ”mål” vi är
  - Sannolikheten att välja sämre nod större i början
- Styrs av en sk. avkylningsfunktion (annealing) $e^{\Delta E/T}$ med temperaturen T och $\triangle$E skillnaden mellan hur bra noderna är
  - T minskas enligt något schema

Simulated annealing

```
def simulatedAnnealing (problem, schedule):
   current = makeNode(initialState(problem))
   for t in range(1,∞):
       T = schedule(t)
       if T==0:
          return current
       next = makeRandomSuccesorNode(current)
       ΔE = value(next)- value(current)
       if ΔE > 0:
          current = next
       else:
          current = next
```

med sannolikheten $e^{-\Delta E/T}$

Genetiska algoritmer

- Iterativ metod som utgår från de bästa tillstånden och förbättrar dessa
- Naturligt urval
  - Operatorer inspirerade från naturlig genetisk variation
- Arbetar parallellt med flera sökrymder

Genetiska algoritmer

- Initialt ett antal slumpmässiga tillstånd, populationen
- Ett tillstånd, en individ, representeras som en sträng från ett ändligt alfabet (vanligtvis 0 och 1)
- En funktion, lämplighetsfunktionen, som ger högt värde för bra tillstånd och lågt för dåliga tillstånd
- Nya tillstånd generas genom att kombinera individer som har hög lämplighet, samt genom mutationer

Enkel genetisk algoritm

initialisera populationen

utvärdera populationen

while *inte tillfredställande resultat*:

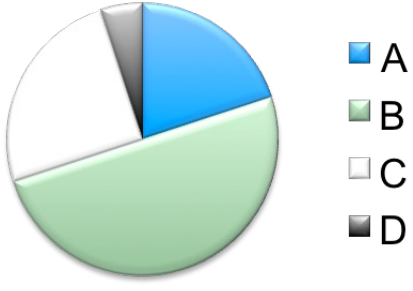
    välj ut föräldrar för reproduktion

    korsa och mutera

    utvärdera populationen

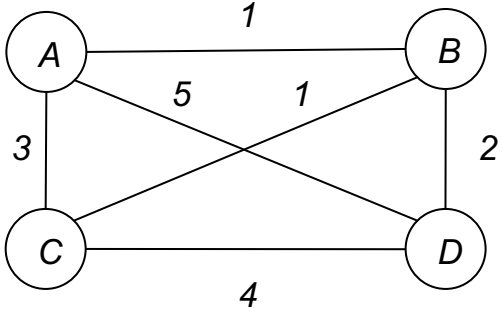
Rouletthjul

- Bättre fitnessvärde ger större chans att överleva men slump finns kvar
- Exempel
  Kromosom A, fitness 4
  Kromosom B, fitness 10
  Kromosom C, fitness 5
  Kromosom D, fitness 1



Exempel, 1

- Handelsresandeproblemet



Exempel, 2

- Korsning
  - Två gener. Ta första från första, andra från andra, tredje från första etc och bilda en ny gen. Om elementet redan finns ta nästa. Sluta med samma som start
- Mutation
  - Byt slumpmässigt två element
- Fitness
  - Totalavstånd
- Starttillstånd

  A B C D A

  B C D A B

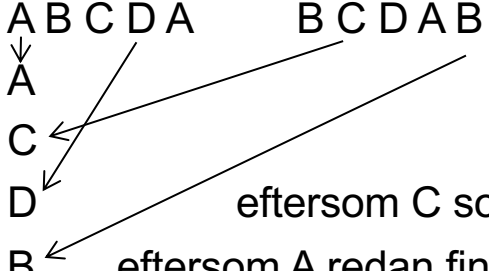
  C D A B C

  D A B C D

Exempel, 3

Korsa de två första:

A B C D A B C D A B



A

C

D eftersom C som är tredje från första genen redan finns

B eftersom A redan finns

Ny gen

A C D B A

Korsa de två sista:

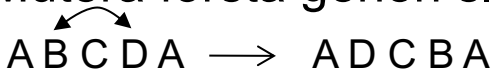
C D A B C D A B C D

Ny gen:

C A B D C

Exempel, 4

- Mutera första genen slumpmässigt

- Ny population

  A ^{1} B ^{2} C ^{4} D ^{5} A = 12
  B ^{2} C ^{4} D ^{5} A ^{1} B = 12
  C ^{4} D ^{5} A ^{1} B ^{2} C = 12
  D ^{5} A ^{1} B ^{2} C ^{4} D = 12

  A ^{3} C ^{4} D ^{2} B ^{1} A = 10
  C ^{3} A ^{1} B ^{2} D ^{4} C = 10
  A ^{5} D ^{4} C ^{1} B ^{1} A = 11

- Behåll de fyra bästa

Exempel, 5

- Ny population

  D ^{5} A ^{1} B ^{2} C ^{4} D = 12
  A ^{3} C ^{4} D ^{2} B ^{1} A = 10
  C ^{3} A ^{1} B ^{2} D ^{4} C = 10
  A ^{5} D ^{4} C ^{1} B ^{1} A = 11

- Korsa två första

  D C B A D

- Korsa två sista

  C D B A C

- Mutera första. Slumpen gav andra och tredje staden

  D B A C D

Exempel, 6

- Ny population

  D ^{5} A ^{1} B ^{2} C ^{4} D = 12
  A ^{3} C ^{4} D ^{2} B ^{1} A = 10
  C ^{3} A ^{1} B ^{2} D ^{4} C = 10
  A ^{5} D ^{4} C ^{1} B ^{1} A = 11
  D ^{4} C ^{2} B ^{1} A ^{5} D = 12
  C ^{4} D ^{2} B ^{1} A ^{3} C = 10
  D ^{2} B ^{1} A ^{3} C ^{4} D = 10

- Behåll de fyra bästa

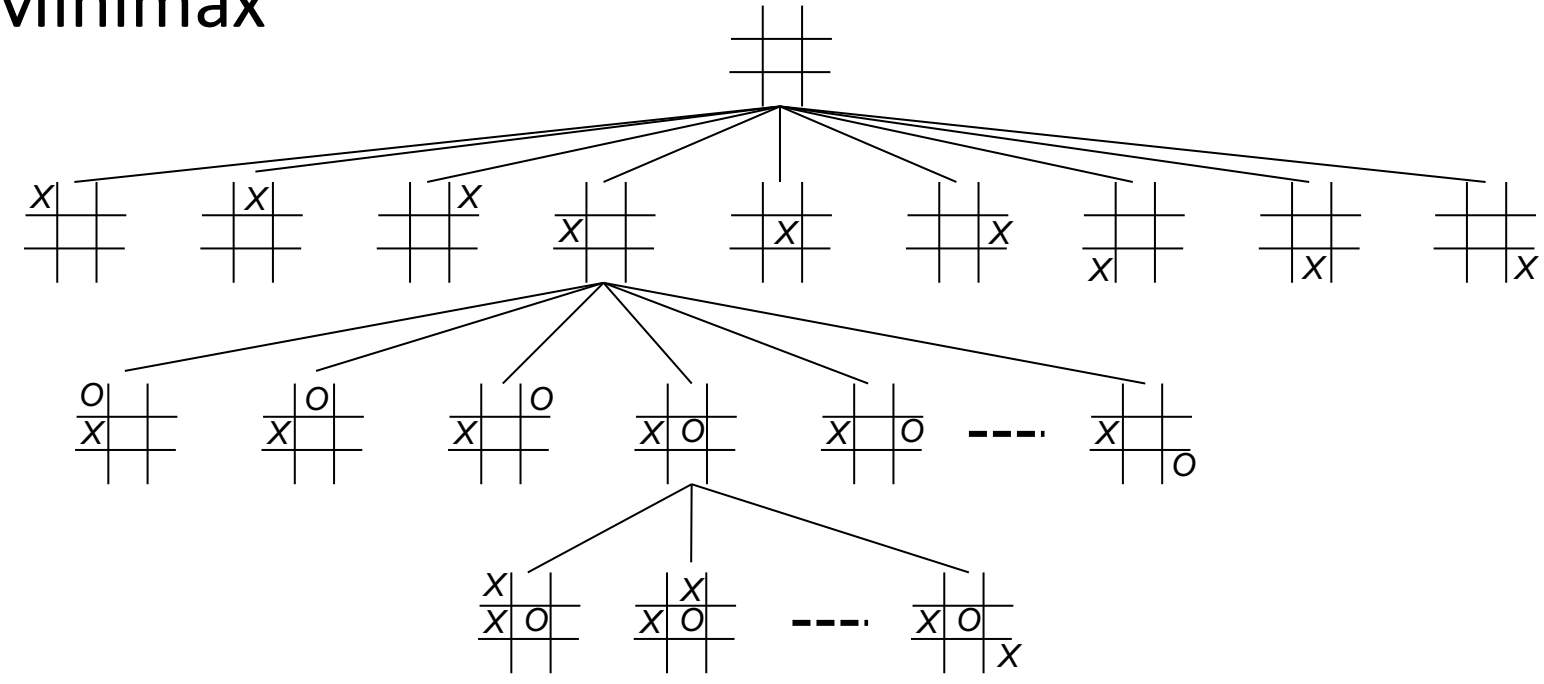
  A ^{3} C ^{4} D ^{2} B ^{1} A = 10
  C ^{3} A ^{1} B ^{2} D ^{4} C = 10
  C ^{4} D ^{2} B ^{1} A ^{3} C = 10
  D ^{2} B ^{1} A ^{3} C ^{4} D = 10

- etc

Spelförande program

- Minimax
- Evalueringsfunktioner
- alfa-beta cutoff

Minimax

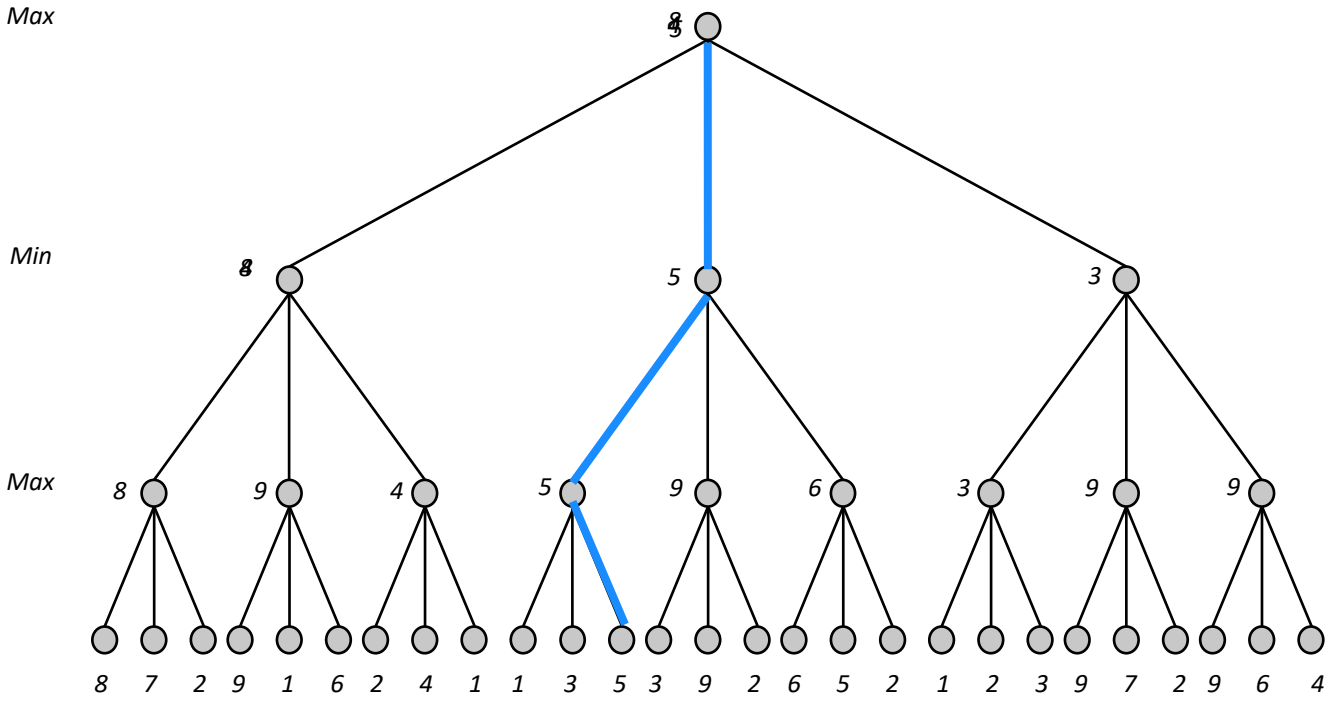


e(x, , ,x,o, , , ,)=4-4=0 *e(,x, ,x,o, , , , ,)=0* *e(, , ,x,o, , , ,x)=4-3=1*

Applicera evalueringsfunktion, t.ex.:

e(p)=n(vinstdrag för mig) – n(vinstdrag för motståndaren)

Ett sökträd



Evalueringsfunktion

- Linjär

$$e(p) = \sum_{i=0}^{n} w_i f_i$$

  - Ex: e(p)=1*n(vinst för mig)-1*n(motståndarvinst)
    e(p)=π*n(vinst för mig)-2,18*n(motståndarvinst)+1*n(diagonaler)
- Inlärning
  - Vikter och funktioner lärs in automatiskt utifrån gamla spel
  - Credit assignmentproblemet

alfa-beta cutoff, 1

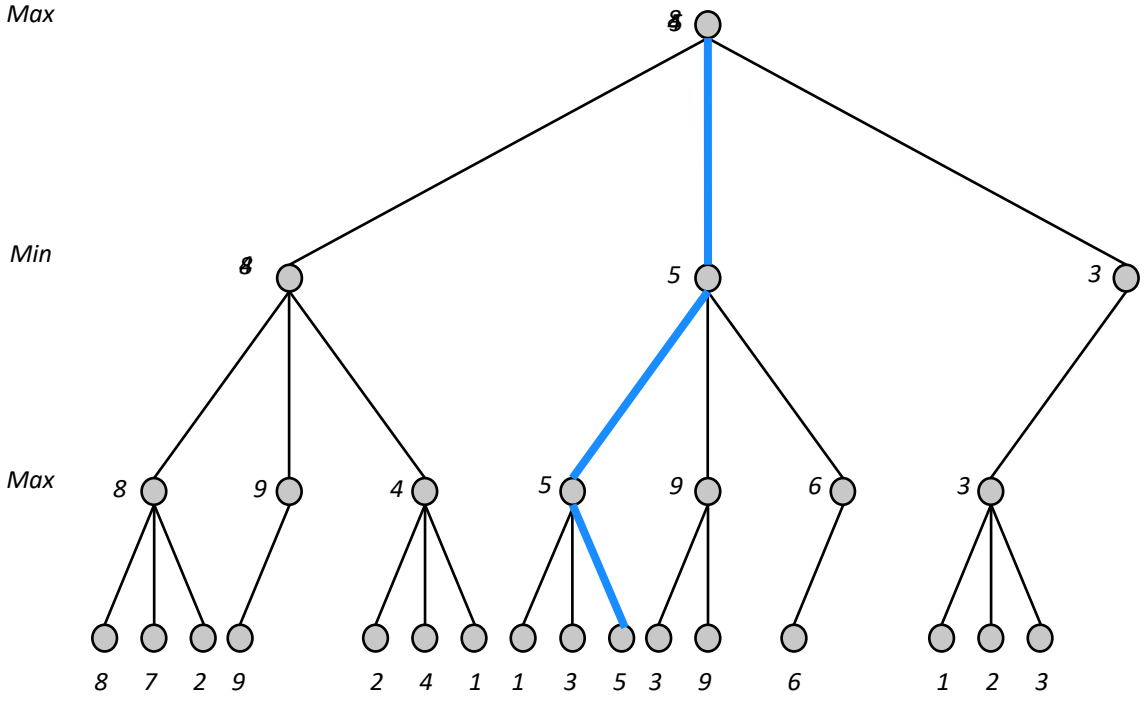
Två tröskelvärden α och β:

- α representerar det lägsta värdet en maxnod kan få, dvs en undre gräns för MAX (minst såhär mycket)
- β representerar det högsta värdet en minnod kan få, dvs en övre gräns för MIN (inte bättre än såhär)

alfa-beta cutoff, 2

- Initiering:
  - α = sämsta värdet
  - β = bästa värdet
- Tilldelning:
  - α = bästa efterföljare hitills, på MAX-nivå
  - β = sämsta efterföljare hitills, på MIN-nivå
- Regler:
  - Avbryt MIN-sökning vid nod med värde mindre än α
  - Avbryt MAX-sökning vid nod med värde större än β

Ett sökträd



Constraint satisfaction

Givet:

- En mängd variabler $X_1, X_2, \ldots X_n$
  - Med värden v_i till X_i ur domänen D_i
- En mängd begränsningar $C_1, C_2, \ldots C_n$
  - Som talar om vilka värden som är tillåtna

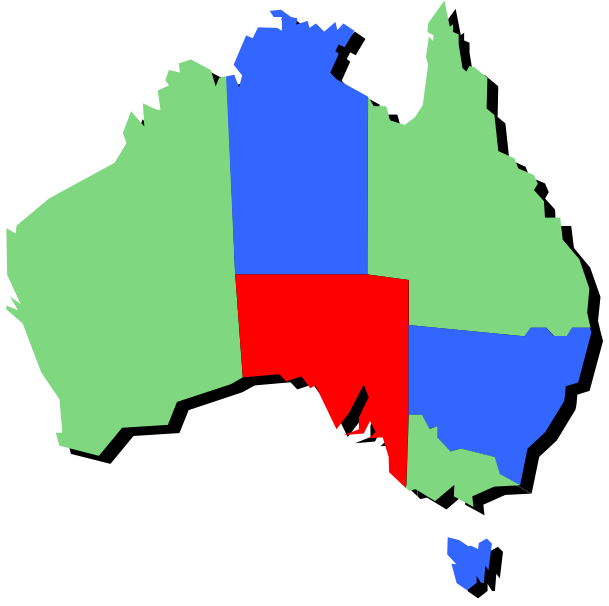
Mål:

- En tilldelning $\{X_i=v_i, X_j=v_j \ldots\}$ som är konsistent med begränsningarna

Ex. trefärgsproblemet

Färga varje stat:

- Tre färger
- Grannstater måste ha olika färg



Ex.

{(WA=grön),(NT=blå),
(Q=grön),(NSW=blå),
(SA=röd),(V=grön),(T=blå)}

Constraint satisfaction-sökning

- Bredden först skulle tilldela varje variabel ett värde och generera ett stort sökträd
- Eftersom ordningen saknar betydelse kan man istället tilldela en variabel ett värde och sen backa tillbaka om det inte blev bra

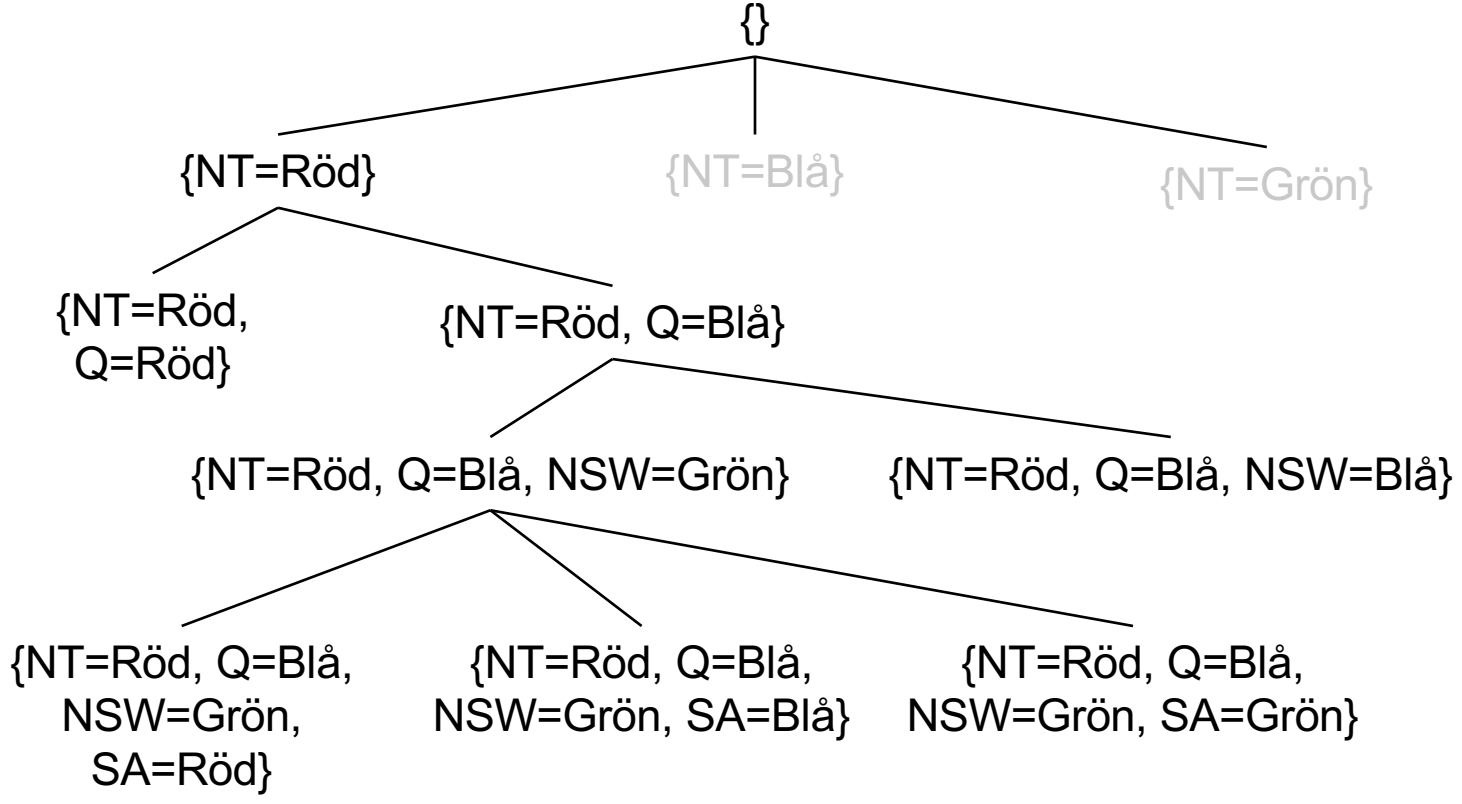
CSP - Sökproblem

- Tillståndsrymd
  - Initialtillstånd
    - Tom mängd {}
  - Funktion för tillståndsförändringar
    - Ett värde sätts för en variabel som saknar värde
- Måltest
  - En mängd där alla variabler har tilldelats värden
- Vägkostand
  - Konstant 1

CSP - Sökning

- Djupet först
- Lokal sökning
- “Backtracking”
  - Väljer en variabel och “expanderar”, dvs prövar olika värden
  - Om inget värde kan användas backar algoritmen

CSP Sökning – Exempel



Algoritm

```
def backtrackingSearch(csp):
   return recursiveBacktracking(csp, { })

def recursiveBacktracking(csp, assignment):
   if complete(assignment):
      return assignment
   var = selectUnassignedVariable(assignment,csp)
   for value in orderDomainValues(csp, var, assignment):
      if consistent(csp, var, value, assignment):
         add {var=value} to assignment
         result = recursiveBacktracking(csp, assignment)
         if result != failure:
            return result
         remove {var=value} from assignment
   return failure
```

Algoritm

- Variabler = `[NT, Q, NSW, SA, V, T, WA]`
- Värden = `[Röd, Blå, Grön]`
- Initialt assignment = `{ }`
- `selectUnassignedVariable` väljer variabel
  - Ex `var = NT`
- `orderDomainValues` väljer värden enligt någon heuristik
  - Ex `assignment = {NT = röd}`
- Nytt anrop `recursiveBacktracking({NT=röd}, csp})`
- `selectUnassignedVariable` väljer `var = Q`
- `orderDomainValues` väljer röd. Inte konsistent

Algoritm, forts

- väljer nytt värde i for-loopen `assignment = {Q=blå}`
- `recursiveBacktracking({NT=röd, Q=blå}, csp}`
- etc
- `{NT=röd, Q=blå, NSW=grön}`
- `var=SA` går inte utan det rekursiva anropet ger ett `failure`
- `remove {var=value} from assignment` tar bort något värde, t.ex. det sista `NSW=grön` och sen fortsätter algoritmen med ny variabel

Heuristik

- Val av variabel och värde
  - `selectUnassignedVariable` bör välja den variabel som är mest begränsad och först väljs den som har flest begränsningar, `SA`
  - `orderDomainValues` välj det minst begränsande värdet
- Constraint propagation
  - Propagera effekten av en variabels begränsningar till de andra variablerna
- Backtrackingstrategier
  - Ta bort det värde, eller den variabel, som orsakade konflikten

Sammanfattning, 1

- Problemformulering
  - Tillståndsrymd
- Blind sökning
  - Värdering av sökstrategier, komplett, optimal, tids- och minnesåtgång
  - Bredden först, djupet först, Uniform Cost, djupbegränsad sökning, iterativ fördjupning, dubbeliriktad sökning
  - Loophantering
- Heuristisk sökning
  - Greedy Search, A*
  - Heuristik
  - Hill Climbing, simulated annealing

Sammanfattning, 2

- Genetiska algoritmer
- Spelförande program
  - Minimax
  - Evalueringsfunktioner
  - Alpha-beta cutoff
- Constraint satisfaction
  - Heuristik