729G78 Artificiell intelligens
Sökning – Heuristisk sökning

Robin Keskisärkkä

HCS/IDA

Heuristisk sökning

- Antar att vi har någon information om vilken nod som är bäst att expandera

- Greedy search
- A*
- Heuristik
- Hill Climbing

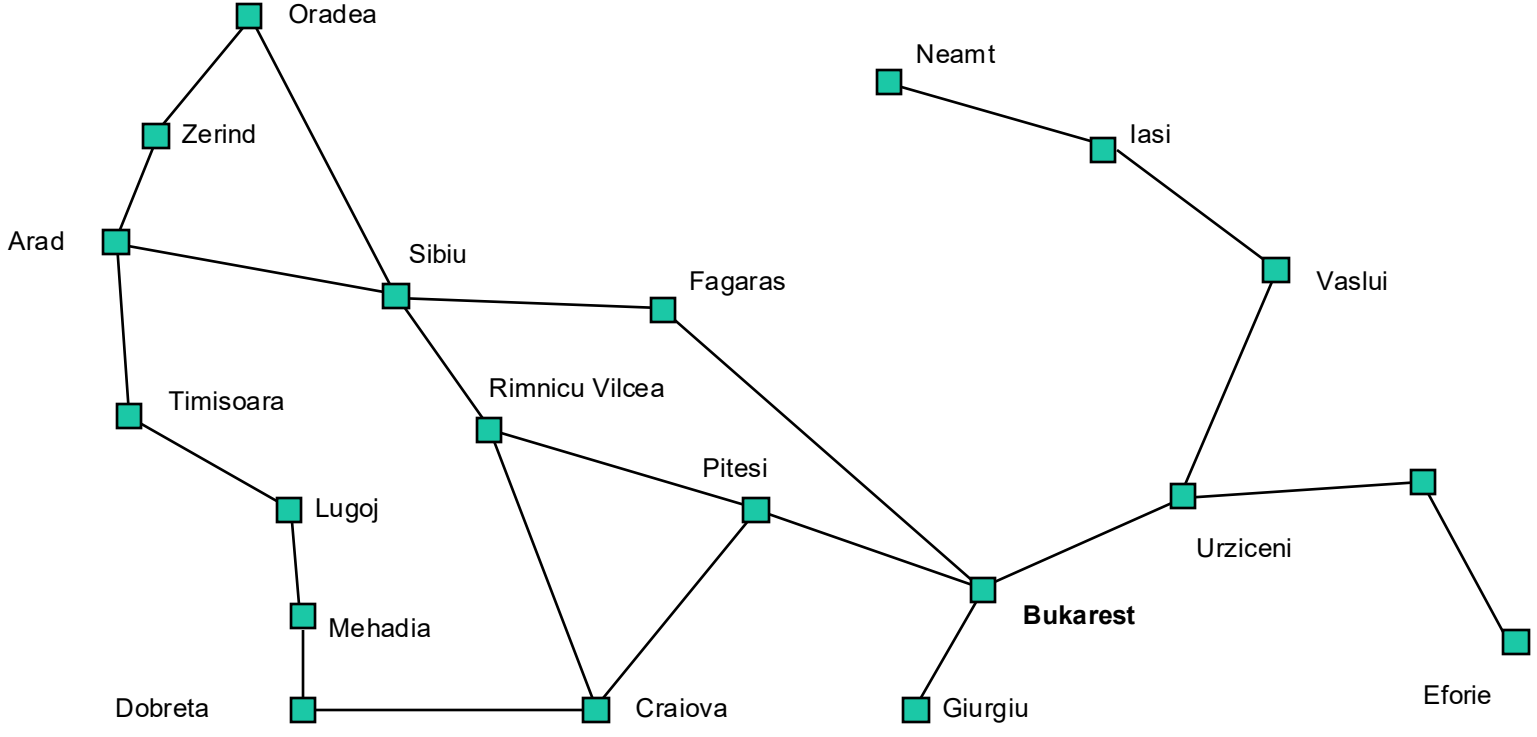
Greedy Search

- Minimera uppskattad kostnad till målet, dvs expandera 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 uppskattat avstånd till målet är 0
- Ex Rumänienkartan
  - Fågelavståndet ett exempel på heuristik

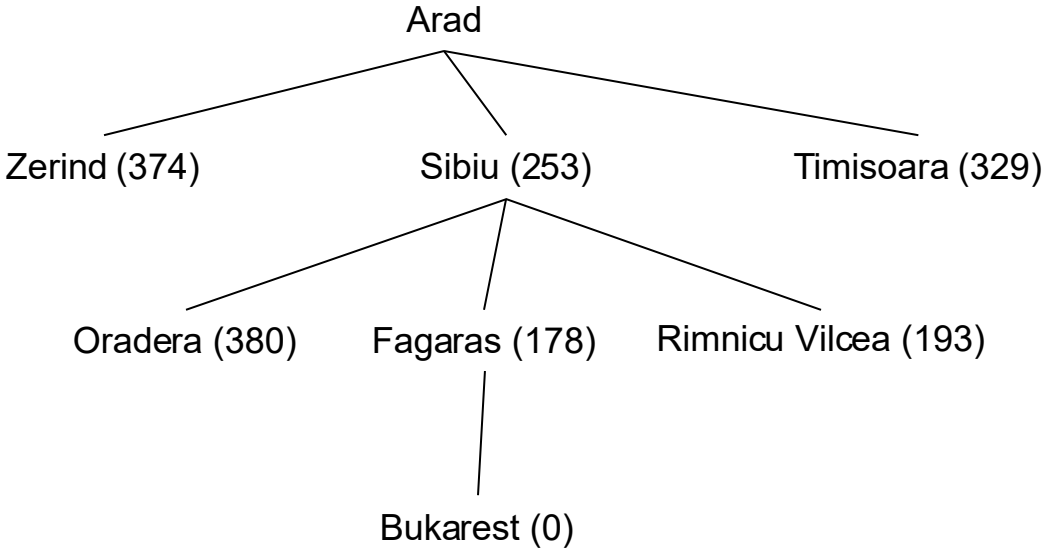
Uppskattningar för avstånd till Bukarest

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	Timisoara	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*

A*

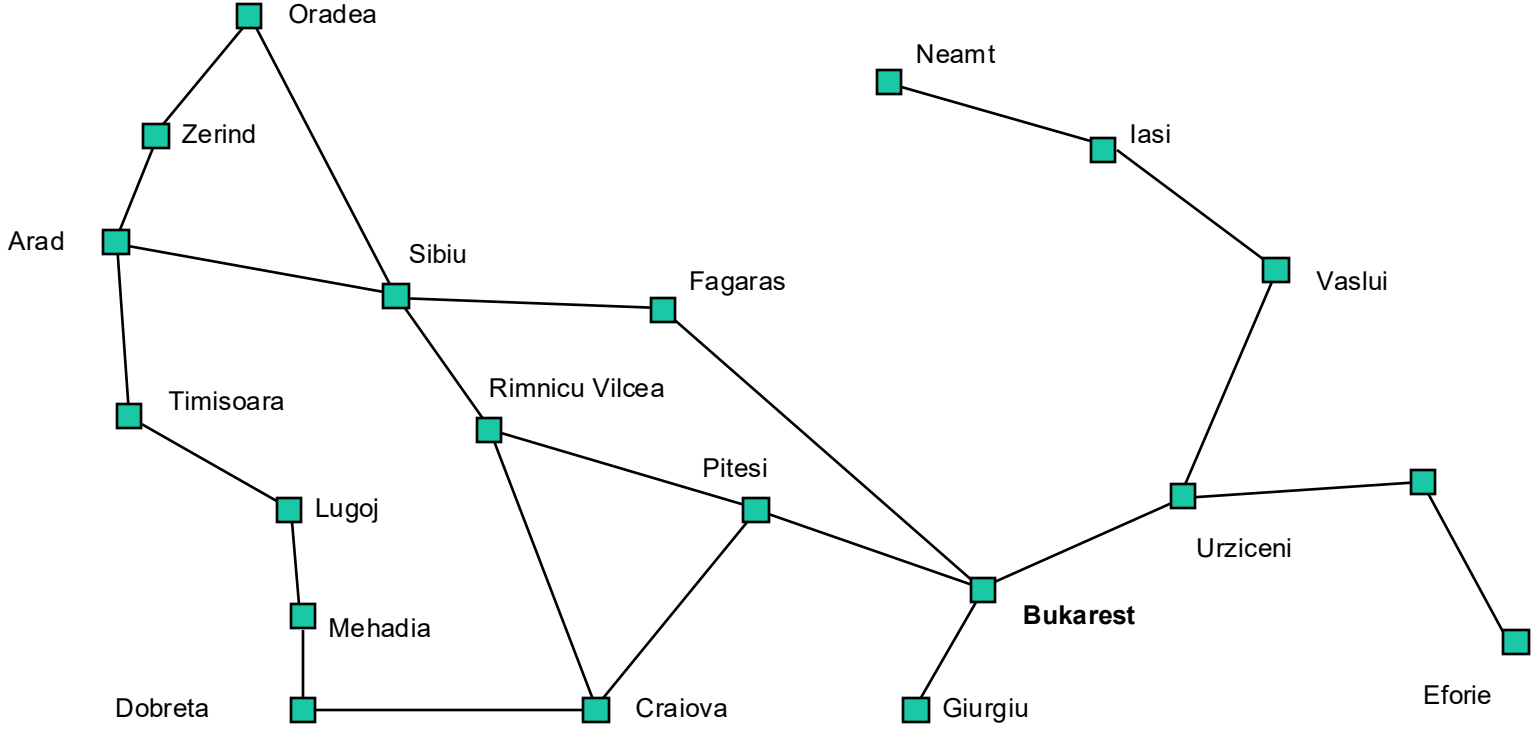
- Expandera noderna i ordning baserat på tillryggalagd kostnad + uppskattad återstående kostnad
- f(n) = g(n) + h(n), där g(n) representerar tillryggalagd kostnad från startnoden

- Heuristiken måste vara tillåten (*admissible*), dvs. h(n) får aldrig överskatta den verkliga kostnaden att ta sig till målet
  - Om den överskattar kan vi missa att expandera potentiellt bättre vägar!

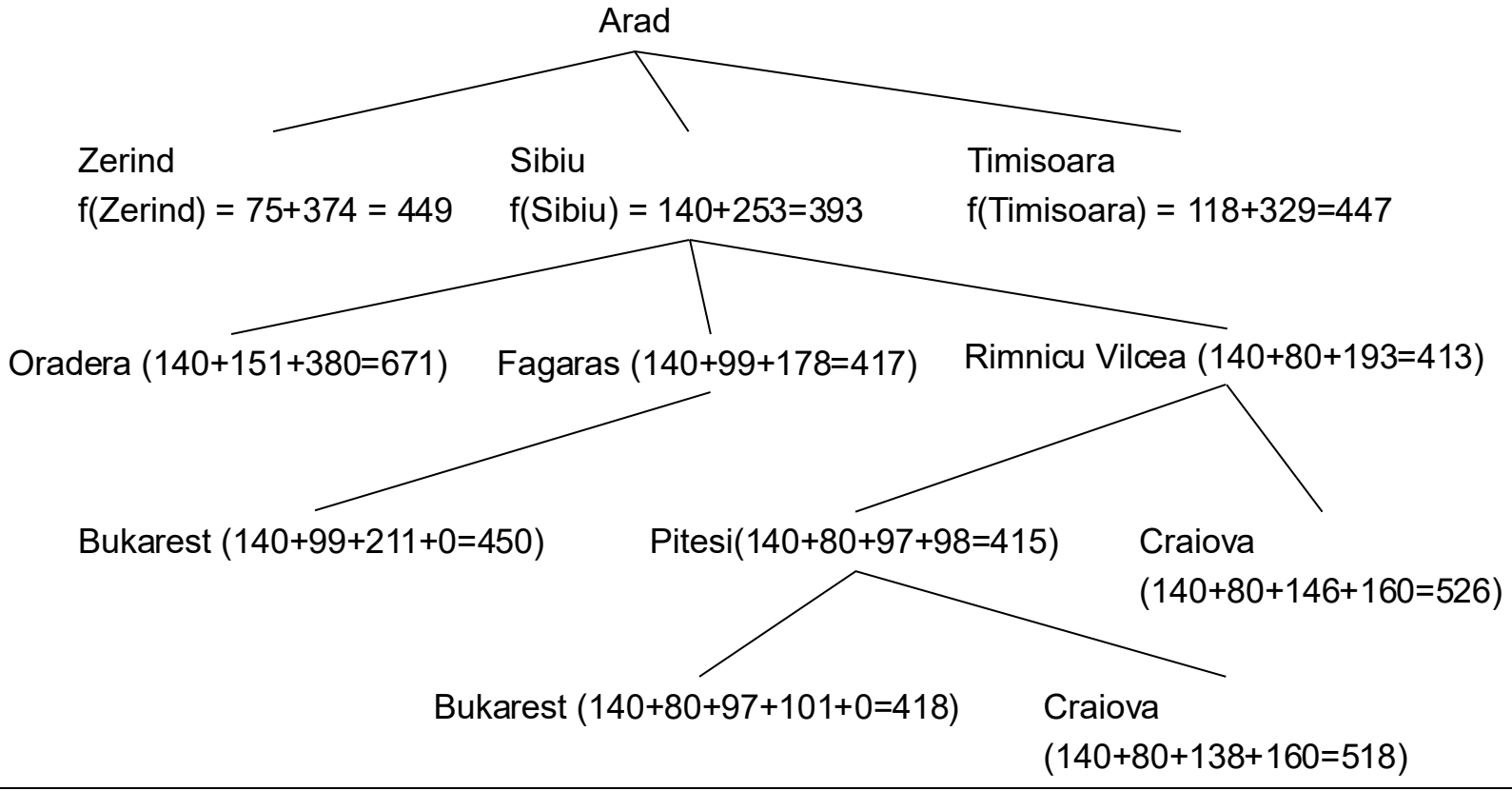
Uppskattningar för avstånd till Bukarest

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	Timisoara	329	Urziceni	80	Vasliu	199	Zerind	374



A*, Arad → Bukarest



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

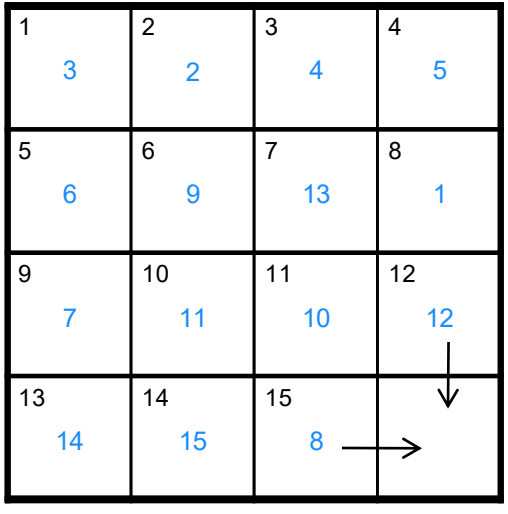
Heuristiska sökmetoder bättre med bättre heuristik, dvs bättre skattning av kostnaden

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

Antal brickor som ligger fel

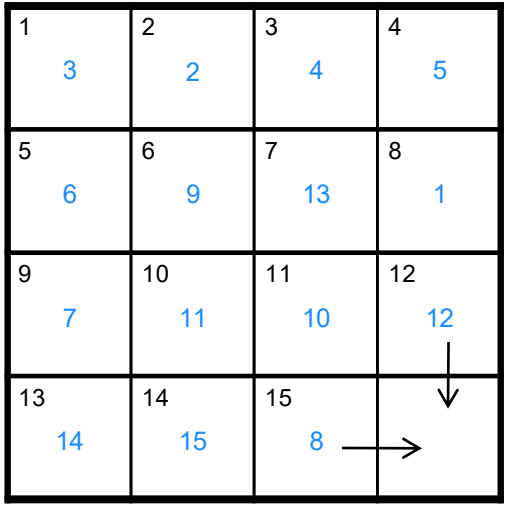


12: (1,3,4,5,6,7,8,9,10,11,12,13,14,15)=14

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

Heuristik

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

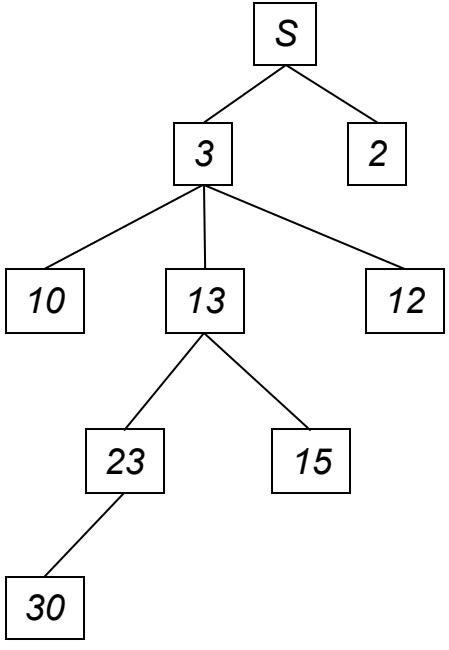
- Förenklade problem används ofta för konstruktion av heuristik
- En optimal lösning för ett förenklat problem är en tillåten heuristik för orginalproblemet

A*

```
def astar(problem):
  n = initialNode(problem)
  f = g(n) + h(n, problem)
  frontier = []
  frontier = insertSorted((f, n), frontier)
  while True:
    if isEmpty(frontier):
      return failure
    node = pop(frontier)
    if goalTest(problem, state(node)):
      return node
    nodes = expand(node, problem)
    for n in nodes:
      f = g(n) + h(n, problem)
      frontier = insertSorted((f, n), frontier)
```

Lokal sökning – Hill Climbing

- Inget målvärde finns
- Hitta det lokalt bästa värdet
- 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 är mindre viktig

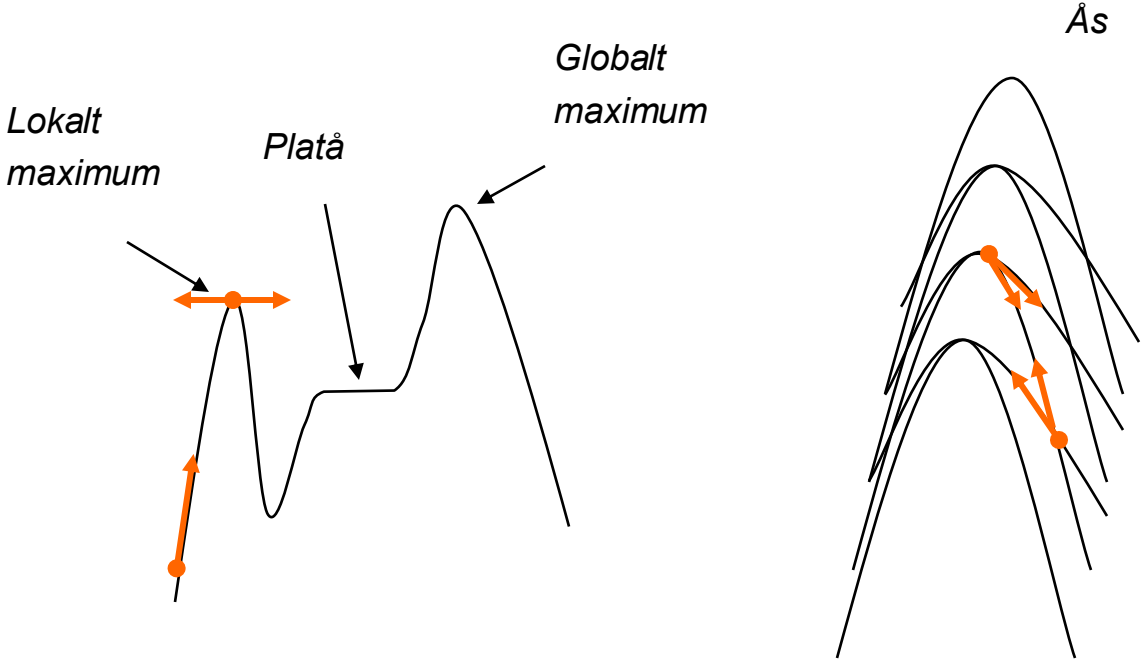


Hill Climbing

```
def hillClimbing (problem):
  current = initialNode(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 så kallad 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 = initialNode(problem)
  for t in range(1,∞):
    T = schedule(t)
    if T == 0:
      return current
    next = makeRandomSuccesorNode(current)
    △E = value(next) - value(current)
    if △E > 0:
      current = next
    else:
      current = next med sannolikheten
```

$e^{\Delta E/T}$

$e^{\Delta E/T}$

ΔE = -1

T=1: $e^{-1/1} = \frac{1}{e} \approx 0{,}368$

T=2: $e^{-2/1} = \frac{1}{e^2} \approx 0{,}135$

T=3: $e^{-3/1} = \frac{1}{e^3} \approx 0{,}050$

...

T=10: $e^{-10/1} = \frac{1}{e^{10}} \approx 0{,}000$

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*:

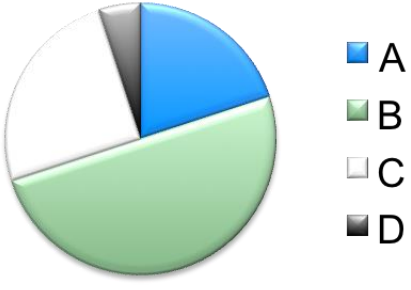
    välj ut föräldrar för reproduktion

    korsa och mutera

    utvärdera populationen

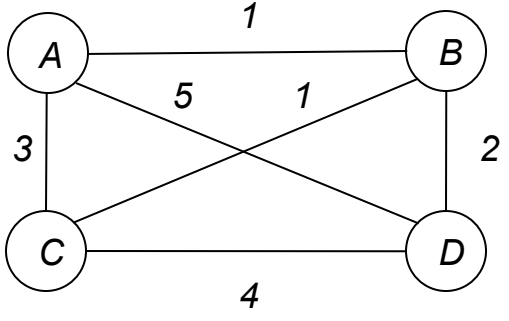
Rouletthjul

- 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
  - Hitta kortaste sträckan för att besöka alla städer och sluta i samma stad som du startade i



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

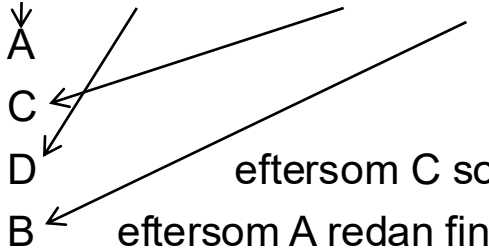
  C D A B C

  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:

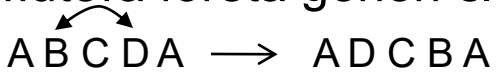
C D A B C D A B C D

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

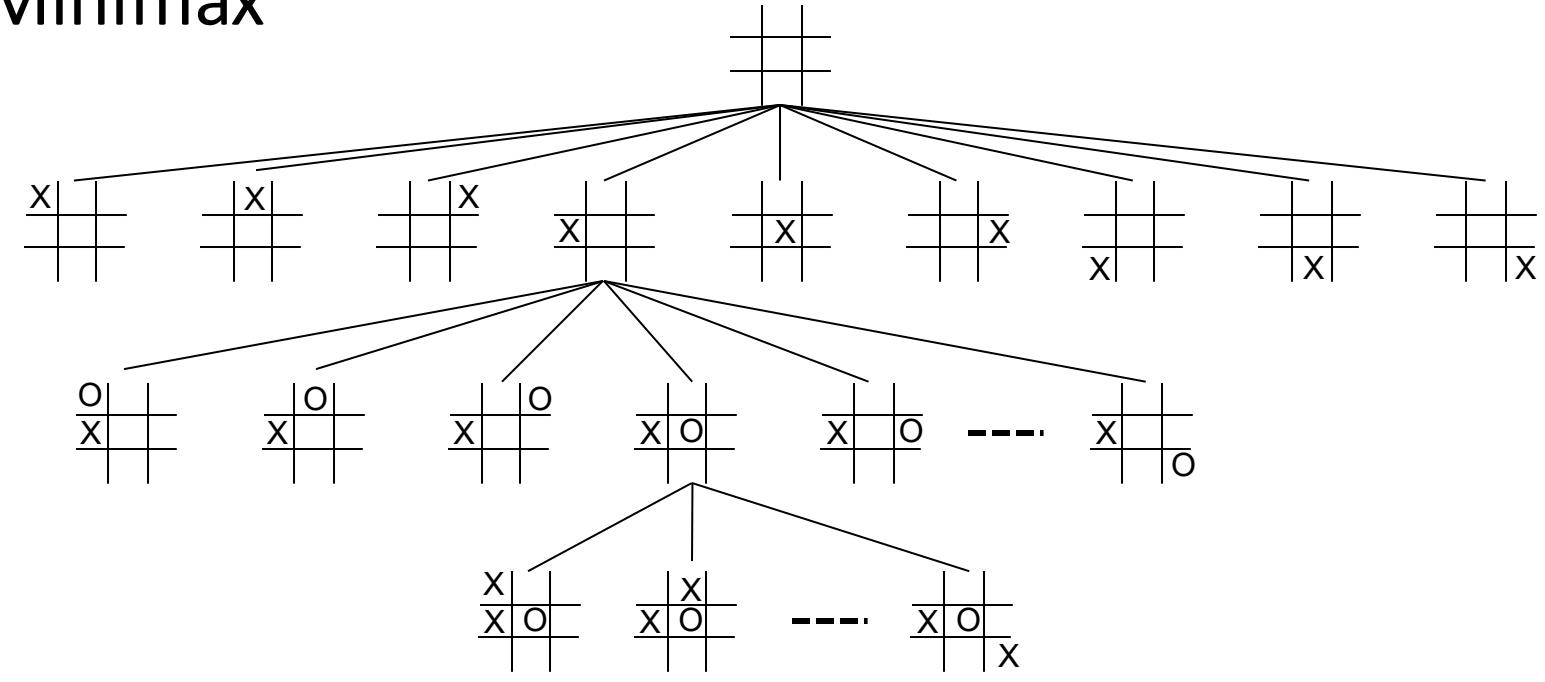
  A ^{3} C ^{4} D ^{2} B ^{1} A = 10
  C ^{3} A ^{1} B ^{2} D ^{4} C = 10
  C ^{4} D ^{2} B ^{1} A ^{3} C = 10
  D ^{2} B ^{1} A ^{3} C ^{4} D = 10

- etc

Spelförande program

- Minimax
- Evalueringsfunktioner
- alfa-beta cutoff

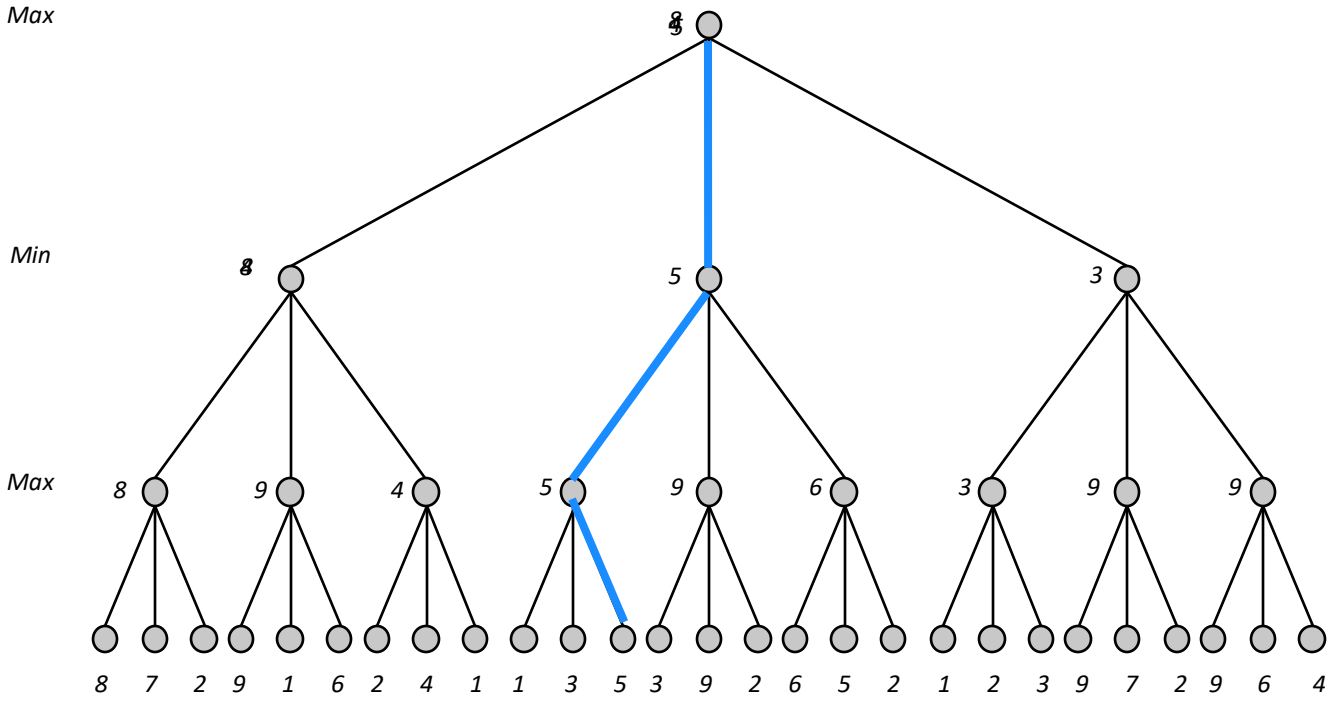
Minimax



Applicera evalueringsfunktion, t.ex.:

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

Ett sökträd



alfa-beta cutoff, 1

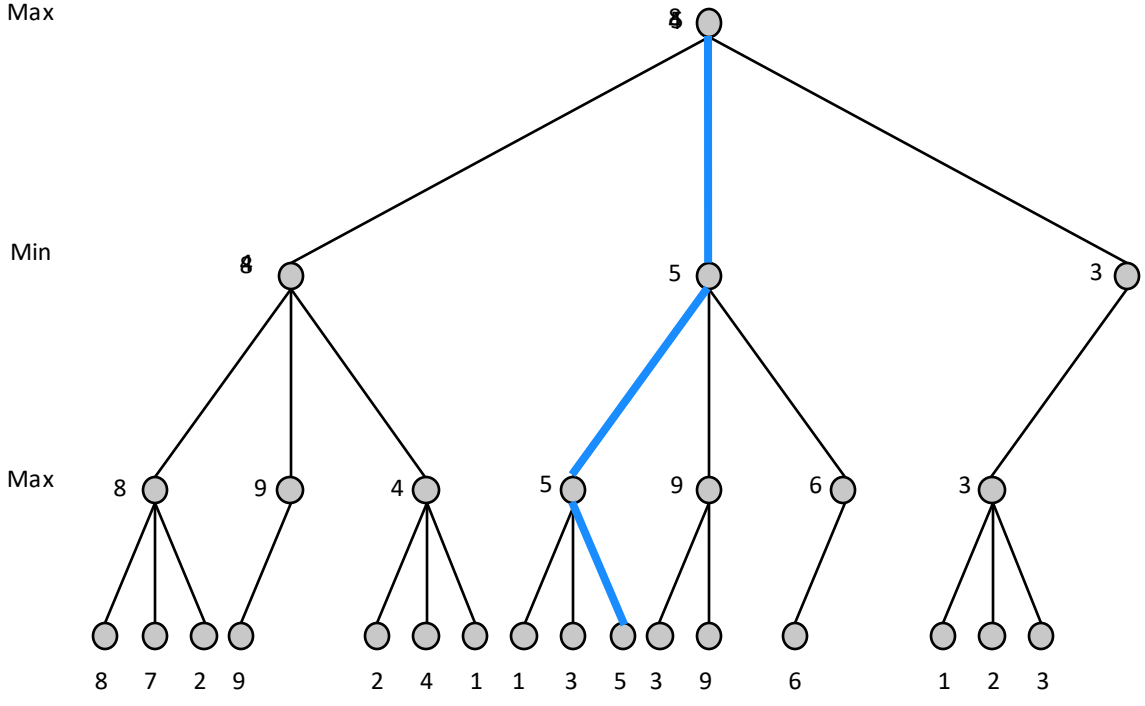
Två tröskelvärden α och β:

- α representerar det lägsta värdet en max-nod kan få, dvs. en undre gräns för MAX (minst så här mycket)
- β representerar det högsta värdet en min-nod kan få, dvs. en övre gräns för MIN (inte bättre än så här)

alfa-beta cutoff, 2

- Initiering:
  - α = sämsta värdet
  - β = bästa värdet
- Tilldelning:
  - α = bästa efterföljare hittills, på MAX-nivå
  - β = sämsta efterföljare hittills, 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, ..., X_{n-1}, X_n
  - Med värden v_i till X_i ur domänen D_i
- En mängd begränsningar C_1, C_2, ..., C_{n-1}, C_n
  - Som talar om vilka värden som är tillåtna

Mål:

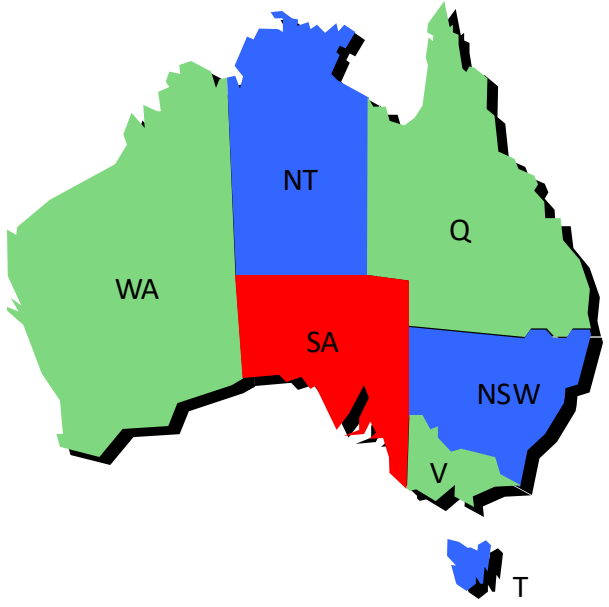
- En tilldelning { $X_i=v_i$, $X_j=v_j$, ... } som är konsistent med begränsningarna

Ex. trefärgsproblemet

Färga varje delstat:

- Tre färger
- Grannstater måste ha olika färg

{ (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

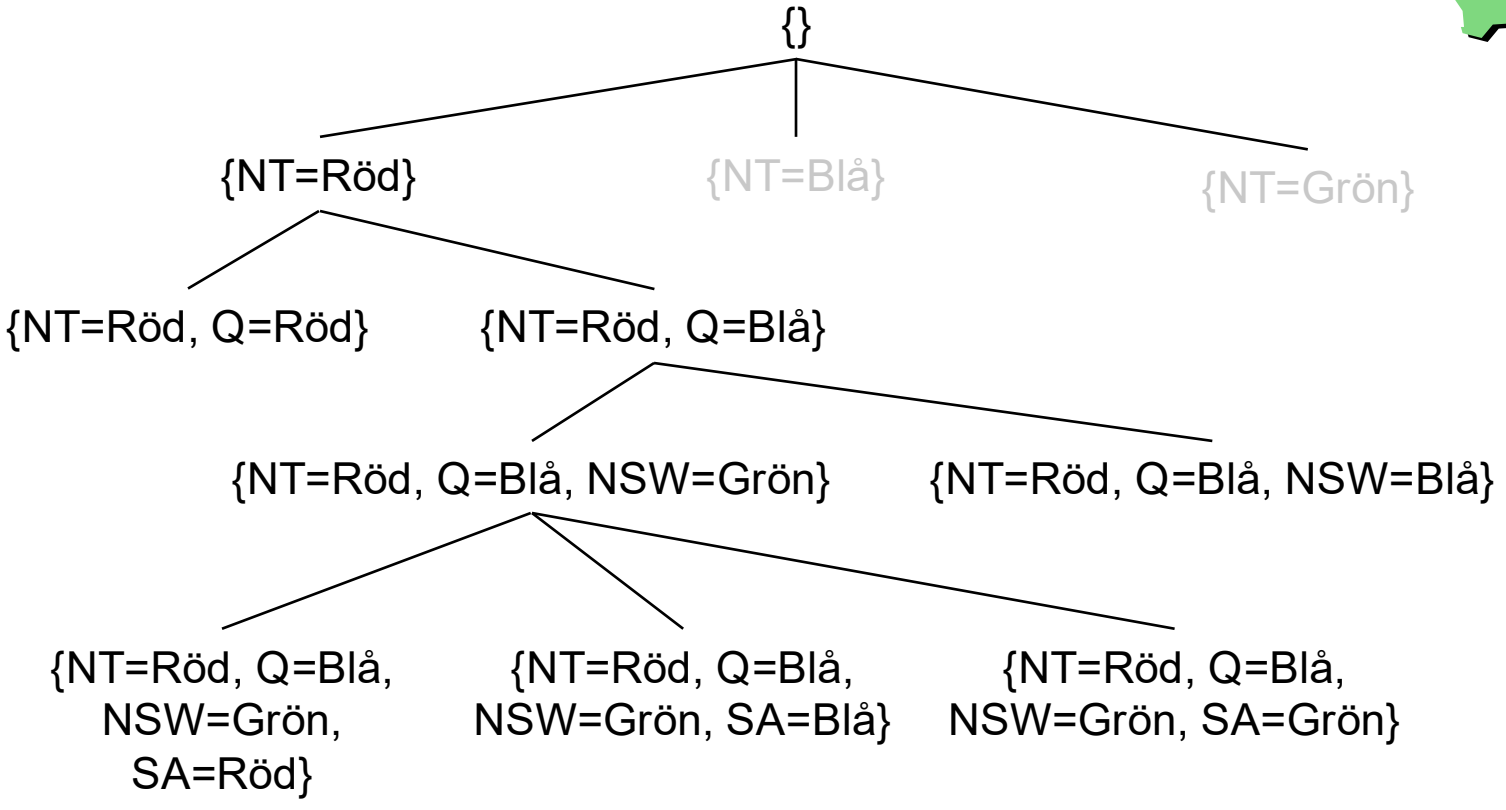
Constraint satisfaction-problem (CSP)

- 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]`
- Initialtillstånd = `{ }`
- `selectUnassignedVariable` väljer variabel
  - Ex `var = NT`
- `orderDomainValues` väljer värden enligt någon heuristik
- Ex `assignment = {NT = röd}`
- Nytt anrop `recursiveBacktracking(csp, {NT=röd})`
- `selectUnassignedVariable` väljer `var = Q`
- `orderDomainValues` sorterar värden

Algoritm, forts

- Väljer nytt värde i for-loopen `assignment={Q=blå}`
- `recursiveBacktracking({NT=röd, Q=blå}, csp}`
- …
- `{NT=röd, Q=blå, NSW=grön}`
- `var=SA` går inte utan det rekursiva anropet ger 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
  - `orderDomainValues`
    - Välj det minst begränsande värdet
- Constraint propagation
  - Propagera effekten av en variabels begränsningar till de andra variablerna
- Backtracking-strategier
  - Ta bort tilldelningen som orsakade konflikten

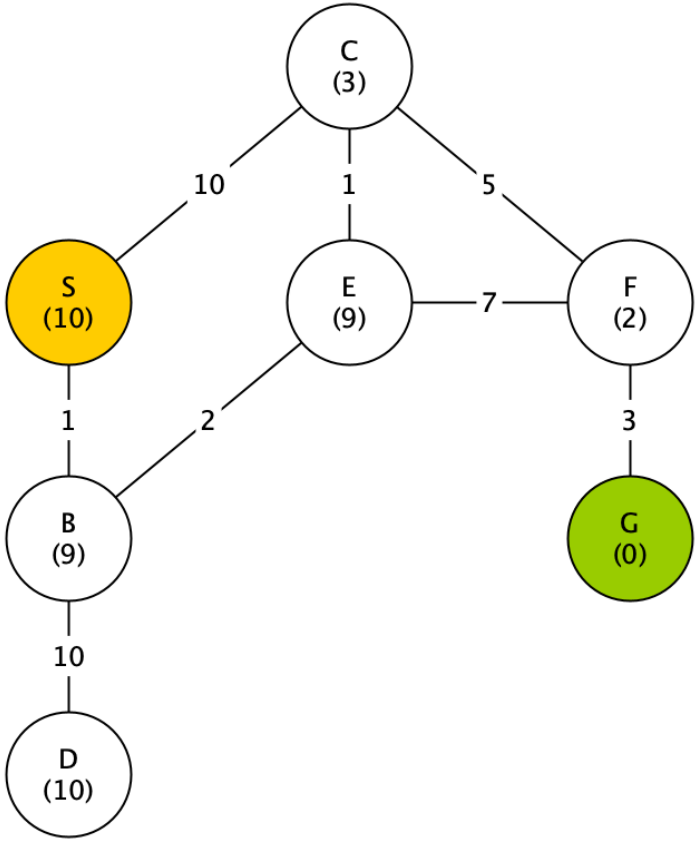
Sammanfattning, 1

- Problemformulering
  - Tillståndsrymd
- Blind sökning
  - Värdering av sökstrategier, komplett, optimal, tids- och minnesåtgång
  - Bredden först, djupet först, Uniform Cost, djupbegränsad sökning, iterativ fördjupning, dubbelriktad 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

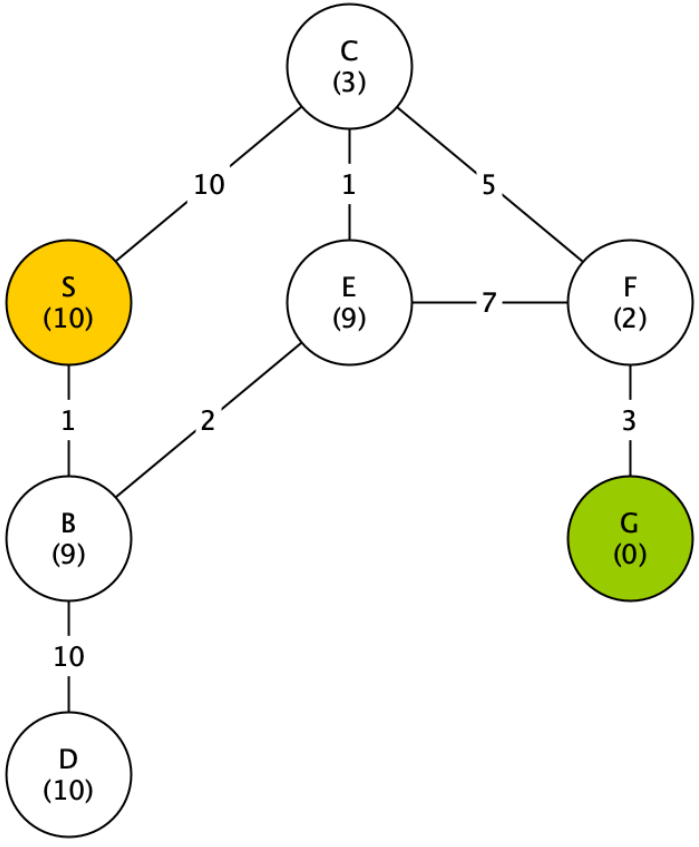
Exempel

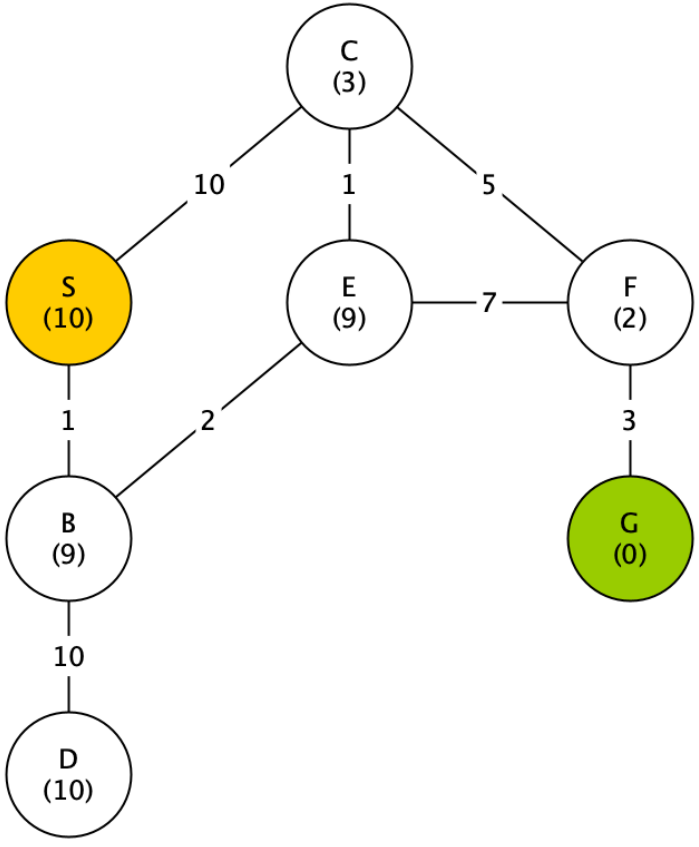


Heuristiskt värde inom parenteser, vägkostnad på respektive båge.

- I vilken ordning skulle Greedy search söka igenom trädet?
- I vilken ordning skulle A* söka igenom trädet?

Greedy search

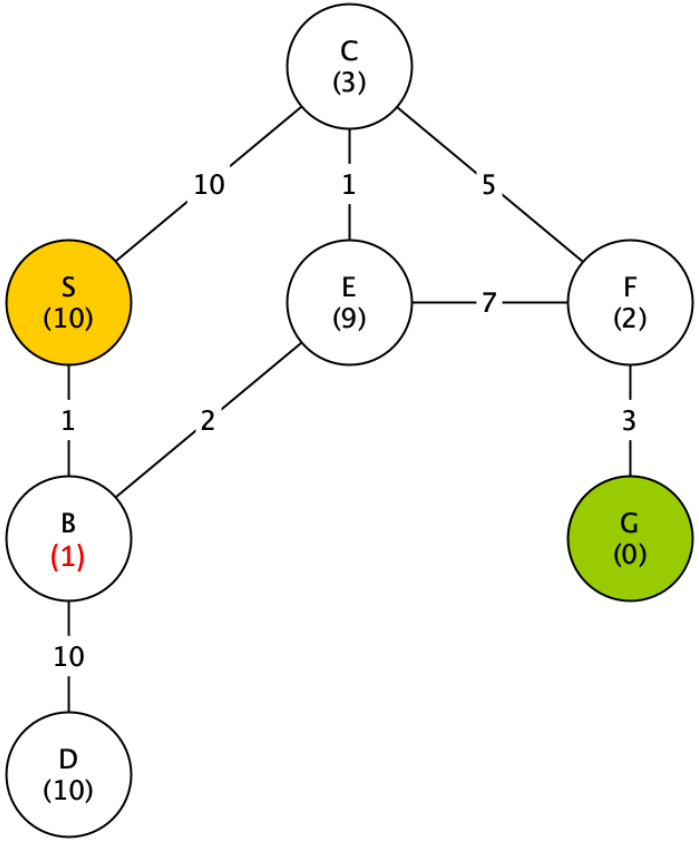




Greedy search

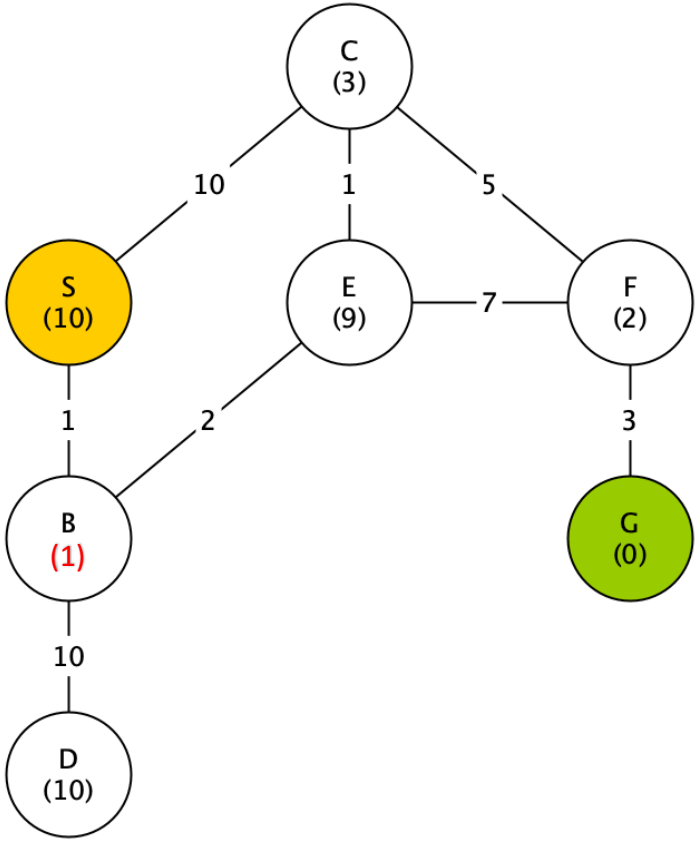
frontier = [**S**]
frontier = [B, **C**]
frontier = [B, S, E, **F**]
frontier = [B, S, E, C, E, **G**]

S => C => F => G, vägkostnad 18



Greedy search

Vad händer om B har lägre heuristiskt värde än C?



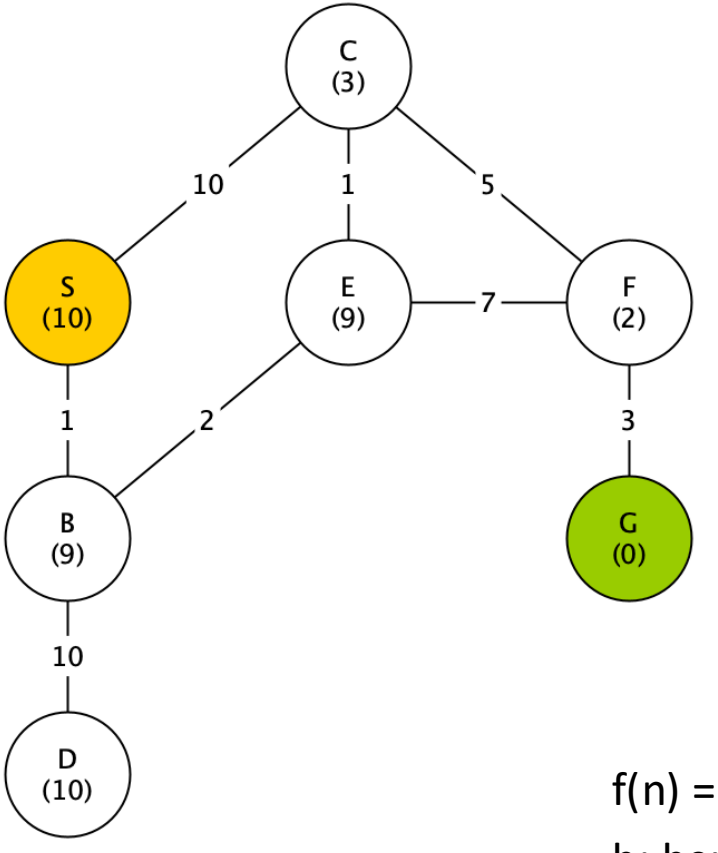
Greedy search

Vad händer om B har lägre heuristiskt värde än C?

frontier = [S]
frontier = [**B**, C]
frontier = [S, D, **E**]
frontier = [S, D, **B**, C, F]
frontier = [S, D, C, F, S, D, **E**]
frontier = [S, D, C, F, S, D, **B**, C, F]
...

Ingen lösning!

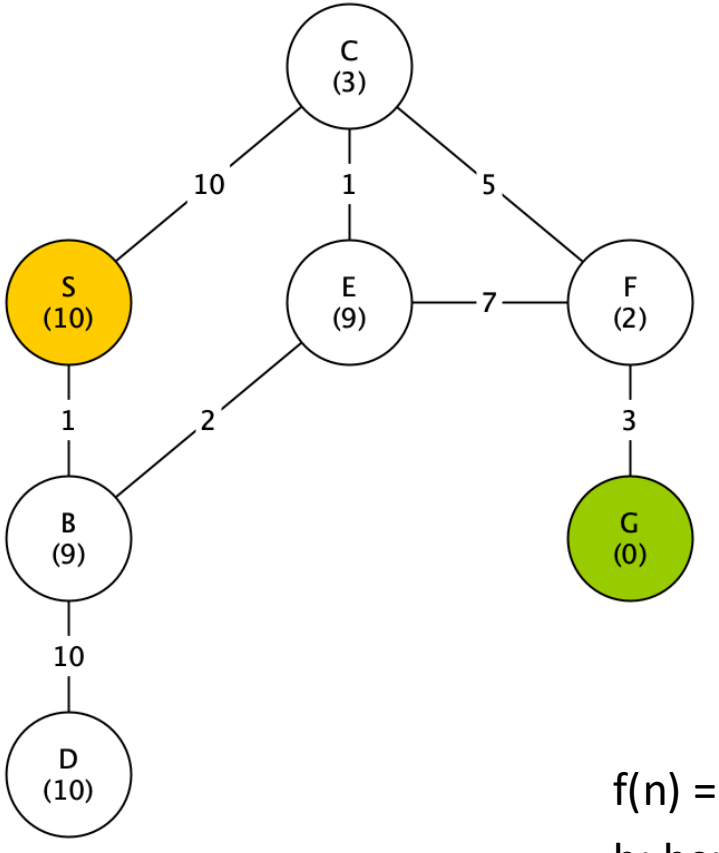
A*-sökning



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

h: heuristiskt värde (gissat avstånd kvar till målet)

g: total vägkostnad



A*-sökning

frontier = [**(10, S)**]

frontier = [**(10, B)**, (13, C)]

frontier = [(13, C), **(12, S)**, (21, D), (12, E)]

frontier = [(13, C), (21, D), **(12, E)**, (12, B), (15, C)]

frontier = [(13, C), (21, D), (12, B), (15, C), [14, B], **(7, C)**, (12, F)]

frontier = [(13, C), (21, D), (12, B), (15, C), [14, B], (12, F), (24, S), (14, E), **(11, F)**]

frontier = [(13, C), (21, D), (12, B), (15, C), [14, B], (12, F), (24, S), (14, E), (16, E), (17, C), **(12, G)**]

S => B => E => C => F => G, vägkostnad 12

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

h: heuristiskt värde (gissat avstånd kvar till målet)

g: total vägkostnad