Kapitel 13

Sökning

13.1 Introduktion till sökning

Om vi tycker att sortering är en viktig disciplin inom datalogin, så måste sökning vara en ännu viktigare, eftersom sorteringens uppgift är att ordna data och information på ett sätt, som ska göra det enklare att finna det man söker!

I det här kapitlet ska vi diskutera olika tekniker och problem i samband med sökning. Vi delar in metoderna i tre grupper allt efter storleken på den datamängd vi ska söka i

- **Tabell** kallar vi en datamängd som vi enkelt kan lagra i RAM-minnet. Ofta handlar det här om en array. Vi kallar detta *internsökning*.
- När datamängden inte får plats i RAM-minnet måste vi kanske söka direkt i en **fil**. Att läsa en post i en binärfil kostar mera (tar längre tid) än att hämta samma uppgift från en array och därför vill vi hålla nere antalet filaccesser. En **databas** innehåller ofta flera filer, vilket förstås leder till än mer komplicerade sökmetoder. Vi kallar dessa metoder *externsökning*

13.2 Intern sökning

13.2.1 Linjärsökning

I en array i minnet finns n *poster*. Varje post har en *nyckel* a_i $1 \leq i \leq n$. För en given nyckel b vill vi nu finna motsvarande post i arrayen. Genom att från arrayens början jämföra nyckel för nyckel $a_1, \ldots a_n$ med den givna nyckeln b kommer vi senast efter n steg fram till antingen

- *träff*. Vilket betyder att vi funnit det vi söker. Detta kan ske när som helst under de n möjliga stegen.
- *bom*. Nyckeln kunde inte återfinnas i arrayen. Detta besked kan vi inte ge förrän vi sökt igenom alla n nycklarna i arrayen.

Antal jämförelser. Antag att *söknyckeln* alltid är någon av nycklarna a_i $1 \leq i \leq n$ (som ju alltid kommer att leda till träff) och att varje nyckel är lika trolig. För att finna den första nyckeln behöver man endast göra 1 jämförelse och för att finna nyckeln på plats k i arrayen görs k jämförelser.

Det förväntade värdet av jämförelser *för en söknyckel* blir

$$\frac{\sum_{k=1}^{n} k}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

För att söka upp 1000 nycklar i en tabell med 10 000 kan man alltså förvänta sig $5000 \cdot 1000 = 5\,000\,000$ jämförelser.

Om man däremot vet att $1/4$ av söknycklarna inte återfinns i tabellen kommer det förväntade antalet jämförelser för en söknyckel att bli

$$\frac{3}{4} \cdot \frac{n+1}{2} + \frac{1}{4} \cdot n = \frac{5n+3}{8}$$

Med de 1000 söknycklarna i en tabell med 10 000 får vi nu 6 250 000 jämförelser.

Algoritmen för *linjärsökning* är förstås inte svår att implementera i C.

Algorithm 13.2.1: LINJÄRSÖKNING(tal, tabell, n)

```
k ← 1
while k ≤ n and tabell[k] ≠ tal
  do k ← k + 1
return (k)
```

Algoritmen söker i arrayen *tabell* efter nyckeln *tal*. Om nyckeln påträffas returneras *k*, index till den plats där nyckeln finns. Om sökningen ledde till bom returneras talet $n + 1$.

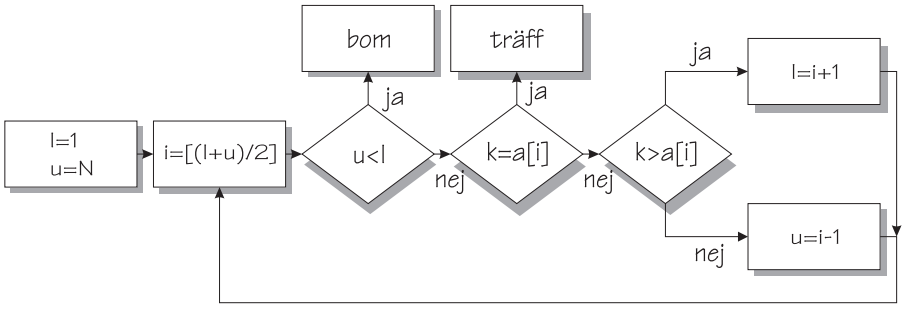
13.2.2 Binärsökning

Det här har ni hört många gånger nu! *Jag tänker på ett tal mellan* 1 *och* 1023. *Du ska gissa detta tal med så få gissningar som möjligt. Efter varje gissning kommer jag att avge ett av dessa tre svar: Rätt tal, För stort, För litet.*

Det är inte svårt att komma på att man bör starta med talet 512 och om svaret blir *För litet*, så blir nästa gissning $(513 + 1023)/2 = 768$. Med denna metod når vi det *hemliga talet*

senast efter 10 gissningar. Talet 1023 är förstås inte slumpmässigt valt. Det tal $2^{10}-1$ passar lika bra som 3, 7, 15, 31 och alla andra tal som kan skrivas på formen 2^n-1

Denna teknik kan vi använda för att söka en nyckel i en array och kallar den då för *binärsökning*. Efter att ha sorterat arrayens nycklar i stigande ordning använder vi följande algoritm.



Figur 13.1: *Flödesschema över binärsökning*

Uttryckt i C kan binärsökning se ut så här

```
1  int binarsokning(int tal,int a[],int n1){
2     int u=0,n=n1-1,k;
3     do{
4        k=(u+n)/2;
5        if(tal<a[k])
6           n=k-1;
7        else
8           u=k+1;
9     }while(tal!=a[k] && u<=n);
10    if(tal==a[k])
11       return k;
12    else
13       return n1+1;
14 }
```

Då arrayen a innehåller $2^{k-1} \leq N \leq 2^k$ tal behövs som mest k jämförelser för att avgöra *träff* eller *bom*.

13.3 Extern sökning

När RAM-minnet inte räcker till för att lagra hela datamängden (binärfilen) får man finna andra vägar. Vi delar upp problemet på följande typer

- Nycklarna läses in till en array i minnet och lagras tillsammans med *postnumren* (de nummer som anger vilken plats posten har i filen). Varefter arrayen sorteras. När information för en given nyckel efterfrågas söker vi upp nyckeln i arrayen med hjälp av binärsökning. Vidhängande postnummer ger oss sedan möjlighet att söka upp och läsa in posten från filen.
- Kanske vill vi inte heller lagra nycklarna i RAM-minnet och väljer då en metod som kallas *hashing*.

13.3.1 Indexering

Indexsering är en vanlig metod, speciellt då posternas storlek är betydande. I en speciell *indexfil* lagrar vi nycklarna tillsammans med *postnummer*. Denna fil kan sorteras utan att posterna i sin helhet flyttas. Då hela indexfilen kan läsas in till RAM-minnet och sorteras blir sökning speciellt snabb. *Borttagning* och *Tillägg* av poster kan enkelt göras till binärfilen, men kräver uppdatering av indexfilen.

Det är inte ovanligt att en datafil har flera indexfiler, vilket betyder att posten har flera nycklar, man vill kunna söka på flera olika fält i posten.

Exemplet nedan arbetar med ett register, en binärfil med poster som vi kallar *datafilen*. Posterna är av typen *persontyp* med följande utseende:

namn	**typ**	**storlek**
Personnummer	char	12
Förnamn	char	10
Efternamn	char	12

Vi behöver dessutom en post för indexet, som vi kallar för *nyckeltyp*. Vårt index kommer att bli en array som vi kallar rätt och slätt, *index*.

namn	**typ**	**storlek**
Personnummer	char	12
Postnummer	int	

Vi behöver nu en rutin för att från *datafilen* bygga upp *index*. Indata till denna rutin är arrayen *index* som har plats för lika många poster av typen *nyckeltyp* som det finns poster i datafilen. Vi vet att det finns *antal* poster på *datafilen*.

Algorithm 13.3.1: INIT(index)

öppna *datafilen*
for k ← 1 **to** antal
do {
  Läs en post från *datafilen*
  index[k].personnummer = datafil[k].personnummer
  index[k].postnummer = k
}
Sortera index efter personnummer

Vi använder den inbyggda QuickSort för sorteringen. Eftersom *qsort* är en ganska ny bekantskap för oss visar vi här funktionen, *ordning*, som ska avgöra ordningen mellan två poster och själva anropet

```
1 int ordning(const void  *a,const void *b) {
2    return strcmp(((element *)a)->persnr,((element *)b)->persnr);
3
4 }
5   qsort(index,antal,sizeof(nyckeltyp),ordning);
```

Rutinen som startar en sökning kan se ut så här. Indata är *index* och personnumret till den person man söker.

Algorithm 13.3.2: SÖKNING(index, personnummer)

r ← BINÄRSÖKNING(index, personnummer)
if r ≤ antal
then {
  Läs datatfil[index[r]].postnummer
  Behandla data i den inlästa posten
}

Funktionen BINÄRSÖKNING har anpassats för att kunna hantera poster av *nyckeltyp*. Funktionen returnerar platsen där det eftersökta personnumret finns i *index*.

UPPGIFT 13.1
Program med indexering. Implementera dessa idéer med hjälp av datafilen *register.dat*, vars post överensstämmer med exemplet ovan.

Välj slumpmässigt 10 000 gånger, och mät samtidigt tiden, ett personnummer från *index*. Sök sedan personens namn med rutinen du skrivit.

Jämför tiden med att lika många gånger direkt söka personens namn i filen! Om ingen skillnad uppstår så får vi väl överge den här metoden.

UPPGIFT 13.2
Hur ska borttagning och tillägg utföras? Tänk igenom hur en förändring (borttagning eller tillägg av post) i registret kommer att påverka dina rutiner.

13.3.2 Hashkodning

Hashkodning går ut på att finna en funktion som delar upp nycklarna i ett lämpligt antal *grupper*. I det exempel vi studerar är nycklarna personnummer från filen *register.dat*. Genom att endast betrakta *födelsenumret*, de tre siffrorna på slutet (ej kontrollsiffran) sprider vi ut personnumren på cirka 1000 olika grupper. Har vi tur kommer dessa grupper att få ungefär 50 medlemmar var.

I varje grupp länkar vi sedan samman medlemmarna i en lista. När frågan efter ett personnummer anländer, har vi bara att gå till rätt grupp och där göra en linjär sökning bland förhoppningsvis 50 poster. När vi hittar posten får vi uppgifter var på filen resten av posten ligger och kan läsa in den för vidare bearbetning. En sökmetod som förstås går betydligt fortare än att ”tröska” igenom hela filen.

13.3.3 Några olika hashfunktioner

Först ska vi betrakta några olika förslag till hashfunktioner. Det är dessa funktioner som tar emot en nyckel, i form av en sträng eller ett heltal, och som transformerar om den till en hashkod.

Vi börjar med en funktion som liknar den vi nämnde ovan,

```
1 int raktav(char s[],int b){
2   char s1[5];
3   int k,j=0;
4
5   for(k=7;k<11;k++)
6     s1[j++]=s[k];
7   return atoi(s1)%b;
8 }
```

Här plockar vi ut de fyra sista siffrorna i personnummer-strängen och med hjälp av *atoi* översätter vi dem till ett heltal. Parametern `b` anger antalet grupper och därför tar vi `s1` mod `b` för att få ett tal mellan 0 och $b-1$.

Under förutsättning att *födelsenumren* fördelar sig jämnt över intervallet kommer denna metod att fungera. En nackdel är dock att om vi vill ha fler än 10000 grupper så kan den här metoden inte användas med framgång. En annan nackdel är att metoden är specifik för en viss tillämpning.

```
1 int hashnaiv(char s[],int b){
2   int i,j=0;
3   for(i=0;s[i]!='\0';i++)
4     j+=s[i];
5   return j%b;
6 }
```

Detta är ett försök till en hashfunktion, som ska fungera för alla strängar. Man adderar helt enkelt ascii-värdena för de ingående tecknen. Detta är en helt oanvändbar metod, där gruppernas storlekar kommer att variera mycket. Den är oanvändbar därför att då vi tar en siffer-sträng med till exempel 10 tecken, så kommer väntevärdet för summan att ligga runt 530. Det är bara en sträng, "0000000000", som ger värdet 480.

```
1 int rcshash(char s[],int b){
2   int n,h,i;
3   n=strlen(s);
4   h=0;
5   for(i=0;i<n;i++)
6     h=(h<<2)^s[i];
7   h=h&0x7FFFFFFF;
8   if(b>0)
9     return h%b;
10  else
11    return h;
12 }
```

Denna funktion är betydligt mer komplicerad. Den använder sig av inte mindre än tre olika *bitoperation*. `<<` står för *vänsterskift*, `^` för bitvis *xor* och `&` står för bitvis *and*. Funktionen, som är upphittad på Internet, kan vara empiriskt framtagen. För gjorda tester verkar den fungera bra. Det är ingen större idé att ge sig på att analysera den och försöka förstå varför de olika operationerna används i just den ordningen. Från vänsterskiftningen förstår vi att det dröjer 16 tecken (varv i loopen) innan första tecknet slutar att inverka på slutresultatet. Den bitvisa *and*-operationen med *maxint* är bara till för slutligen göra `h` positiv.

```
1 int hashU(char s[], int b){
2   int h,a1=31415,a2=27183,i;
3   h=0;
4   for(i=0;s[i]!='\0';i++){
5     a1=a1*a2%(b-1);
6     h=(a1*h+s[i])%b;
7   }
8   return h;
9 }
```

Den här hashfunktionen kan också med framgång användas för godtyckliga sträng-nycklar. Istället för bitoperationer används här två heltal som säkert inte är godtyckligt valda. Det är viktigt att försöka ge varje tecken i nyckel-strängen möjlighet att påverka h.

Alla fyra funktionerna har används för att dela upp de 50 000 personnumren från *register.dat* i grupper. Tabellen visar resultatet.

Grupper	**HashNaiv**				**RaktAv**			
	Avvik	Max	Min	0-grp	Avvik	Max	Min	0-grp
51	46115	2840	2	0	46	1046	905	0
1001	1871757	2840	0	950	2510	84	23	0
10001	23493284	2840	0	9950	20944	19	0	180
50001	∞	2840	0	49950	∞	19	0	40180
Grupper	**RCSHash**				**HashU**			
	Avvik	Max	Min	0-grp	Avvik	Max	Min	0-grp
51	51	1039	894	0	39	1037	925	0
1001	1093	76	28	0	1078	79	30	0
10001	15773	16	0	71	15149	17	0	71
50001	∞	9	0	18937	∞	7	0	19527

Försök har här gjorts (alltid på samma material) med fyra olika antal grupper. För varje försök och metod har max- och min-antalet i grupperna uppmätts. Dessutom har antalet tomma grupper räknats. Avvikelsen är ett försök till mått på hur bra metoden fungerar.

Att *HashNaiv* är värdelös torde härmed vara bevisat. Tydligen kan den inte ge mer än 51 olika resultat. När antalet grupper går över denna gräns ökas bara antalet 0-grupper. *RaktAv* hänger med ganska bra upp till 10000 grupper, som är den övre gränsen för olika resultat som funktionen kan leverera. De andra två metoderna, *RCSHash* och *HashU* är ganska likvärdiga, med en liten fördel för den senare.

Den ideala funktionen är förstås en som kan sprida 50 000 nycklar i 50 000 grupper med exakt en nyckel i varje. Detta är dock omöjligt här eftersom det finns dubbletter bland de framslumpade ”personnumren”.

Snabbheten med vilken man kan räkna ut hashkoden är förstås av betydelse. Det är den tiden tillsammans med tiden för den linjära sökningen i en, förhoppningsvis kort, länkad lista som är den totala tiden för sökningen.

13.3.4 Ett exempel

Med samma register som i exemplet om *indexering* ser vi här ett program som arbetar efter de ovan givna riktlinjerna. Förutom typen *persontyp* behöver vi en förändrad nyckeltyp här kallad *element*

```
1 struct element{
2    char persnr[12];
3    long postnr;
4    struct element *naesta;
5 };
```

Posten innehåller som tidigare två fält för personnummer och postnummer, men dessutom en pekare för att vi ska kunna konstruera listor. Antalet poster på filen ANTAL är 50 1000 och antalet grupper NGR är valt till 1000.

Deklarationen `element *tab[NGR]` ger oss en array för pekare till *element*. Här ska adressen till första elementet i varje lista sparas.

Initiera bygger upp hela datastrukturen. När det jobbet är gjort är det bara att sätta igång att söka.

Algorithm 13.3.3: INIT()

for i ← 1 **to** NGR
  do tab[i] ← NULL
Öppna datafilen
for i ← 1 **to** ANTAL
  do { Läs en post **p** från filen
         nr ← HASHU(p.persnr, NGR)
         Sammanställ element **e** med hjälp av data från **p**
         LÄGGUPP(nr, **e**)

Funktionen *LäggUpp* placerar **e** sist i den länkade listan vars startadress finns i *tab[nr]*.

Algorithm 13.3.4: LÄGGUPP(nr, *e*)

ny ← pekare till plats på heapen
e kopieras till platsen på heapen
if tab[nr] = NULL
  then tab[nr] ← ny
  else { p ← tab[nr]
           while p.naesta ≠ NULL
             do p ← p.naesta
           p.naesta ← ny

Själva sökandet går sedan till på följande sätt. En nyckel i form av ett personnummer presenteras. Hashkoden för personnumret bestäms med hjälp av hashfunktionen. Nu vet man i vilken av de 1000 länkade listorna man ska söka. Dessa listor är i genomsnitt 50 poster långa. Efter att loopen avslutats vet vi om personnumret finns i datastrukturen. Om så är fallet har vi nu tillgång till postnumret och kan efter en *seek* och en *fread* läsa in tillhörande post.

Algorithm 13.3.5: SÖK()

```
pnr ← sökt personnummer
nr ← HASHU(pnr, NGR)
pek ← tab[nr]
repeat
 r ← pnr = pek.persnr
 if not r
   then pek ← pek.naesta
until pek = NULL or r
if r
  then { p ← post pek.postnr från filen
       { Behandla data i den inlästa posten
```

UPPGIFT 13.3

Variant på linjärsökning. I funktionen för *linjärsökning* utförs i while-loopens villkor, två jämförelser, dels om en *träff* och dels att index inte har blivit för stort.

Den sista jämförelsen kan vi slippa genom att utöka arrayen med en cell och i denna cell, innan sökningen påbörjas, lägga söknyckeln. Blir det ingen *träff* förrän vi når *a[n+1]* så vet vi att det är en *bom*!

Genomför en test med de två metoderna och uppskatta hur många procent förbättring denna metod ger i jämförelse med den ursprungliga.

UPPGIFT 13.4

Sökning med wildcards. I MSDOS kan man använda så kallade ”wildcards” för att finna namn på filer. *DIR S*.DAT* betyder att listan kommer att innehålla alla filnamn som börjar på *S* och som har filtillägget *DAT*.

Asterisken (*) är ett så kallat *wildcard*, som står för noll och upp till 7 tecken (eftersom ett filnamn i DOS bara kan bestå av 8 tecken).

Ett annat wildcard är frågetecknet (?), som står för precis ett tecken. *DIR S??.DAT* ger en lista över namnen på de filer som innehåller exakt tre tecken och som inleds med *S* och

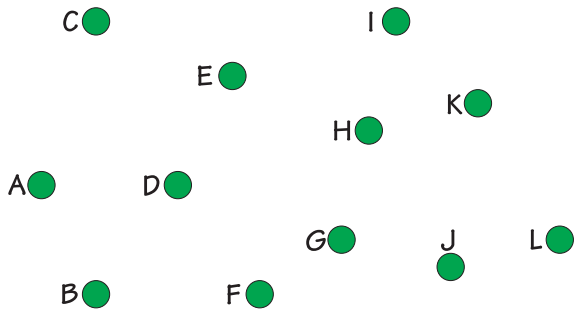
som dessutom förstås har filtillägget *DAT*. Att kombinera de två tecknen (*) och (?) går inget vidare i MSDOS. Du ska nu därför skriva ett program som fungerar enligt följande och som arbetar med, namnen på kommuner i *kommun.dat*.

Reglerna för hur programmet ska fungera ges av följande tabell

*	Listar samtliga kommuner
S*	Listar samtliga kommuner som börjar på S
*A	Listar samtliga kommuner som slutar på A
B	Listar samtliga kommuner som innehåller bokstaven B
FF	Listar samtliga kommuner som innehåller två F intill varandra
*S*T*	Listar samtliga kommuner där bokstäverna S och T förekommer i denna ordning. STOCKHOLM men också SIGTUNA
?J?	Kommuner på tre bokstäver med J i mitten. Finns bara en – HJO.
Å?Å?	Även här finns det bara en kommun – ÅMÅL.
*A?	Listar alla kommuner där *näst* sista bokstaven är A
A?A	Någonstans i namnet ska finnas två A med exakt en bokstav emellan.
?*A*	Listar alla kommuner som innehåller A från och med andra bokstaven. ARBOGA går bra, men inte ALINGSÅS.
??*A??*	Här måste bokstaven A vara väl inbäddad

13.4 Handelsresandeproblemet

13.4.1 Ett mindre problem



Figur 13.2: *Karta över de 12 städerna A till L*

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	22	31	25	40	44	55	60	71	76	81	95
B	22	0	50	25	47	30	46	58	74	65	78	85
C	31	50	0	33	26	58	60	53	55	79	71	93
D	25	25	33	0	22	25	31	36	50	52	57	70
E	40	47	26	22	0	40	36	26	31	53	45	67
F	44	30	58	25	40	0	18	36	55	35	53	55
G	55	46	60	31	36	18	0	20	41	20	35	40
H	60	58	53	36	26	36	20	0	20	29	20	40
I	71	74	55	50	31	55	41	20	0	46	21	50
J	76	65	79	52	53	35	20	29	46	0	30	20
K	81	78	71	57	45	53	35	20	21	30	0	29
L	95	85	93	70	67	55	40	40	50	20	29	0

Vi startar med en glupsk algoritm som i varje steg väljer den stad som ligger närmast den vi befinner oss i. Vi kan inte räkna med ett optimalt resultat. När vi startar i A får vi följande resultat

```
Var ska resan starta (A-L): A
A - B - D - E - C - H - G - F - J - L - K - I - A
Total reslängd 362
```

Men när vi startar i D blir resultatet

```
Var ska resan starta (A-L): D
D - E - C - A - B - F - G - H - I - K - L - J - D
Total reslängd 311
```

Hur bra detta resultat är ser vi när vi skriver en algoritm med backtracking. Då antalet städer är 12 har vi att gå igenom $(12-1)! = 11! = 39\,916\,800$ möjligheter, ett rimligt antal. Vi kommer att få ett optimalt resultat och det spelar ingen roll i vilken stad vi startar resan. Resultatet blir

```
Var ska resan starta (A-L): D
A - B - D - F - G - J - L - K - I - H - E - C - A
Total reslängd 283
```

Studerar vi resvägen tillsammans med kartan förstår vi att den glupska algoritmen saknar möjligheten att lägg upp resan som en verklig rundtur och ofta blir då sista resan onödigt lång.

Datastrukturerna som behövs är en *avståndstabell* `int avst[N][N]` och en array som håller reda på vilka städer vi besökt `int v[N]`. I `resa[N]` lagrar vi namnen (bokstaven) på staden vi besöker och får på sätt till slut hela resan.

Innan vi anropar funktionen `res(a,n,tot)`, där `a` är aktuell stad, `n` antalet städer vi besökt, och `tot`, resvägen så här långt, initierar vi problemet med:

```
1 varit[0]=1;
2 resa[0]='A';
3 res(0,0,0);
```

Funktionen får följande utseende:

```
1  void res(int a, int n,int tot){
2    int i,j;
3    if(n==antal-1){
4      tot=tot+avst[a][0];
5      if (tot<min){
6        min=tot;
7        for(i=0;i<antal;i++)
8          minresa[i]=resa[i];
9      }
10   }
11   else
12     for(i=0;i<antal;i++){
13       if (varit[i]==0){
14         varit[i]=1;
15         resa[n+1]='A'+i;
16         res(i,n+1,tot+avst[a][i]);
17         varit[i]=0;
18       }
19     }
20 }
21
```

En typisk backtracking-funktion, som når resultatet efter 108505112 anrop och som vi sett många gånger nu.

4 Vi får inte glömma att lägga till den sista delen av resan. Den som leder hem till staden där vi startade.

8 `minresa` är en sträng till vilken vi för över den bästa resan vart efter vi finner det.

Genom att ändra rad 13 i funktionen till

```
1       if (varit[i]==0 && tot+avst[a][i]<min){
```

kommer vi att klippa i rekursionsträdet och klarar oss nu med 1116973 anrop. En rejäl förbättring.

I samband med dynamiska programmering presenterades tekniken kallad memoization, det vill säga att lagra resultat i rekursionsträdet i en tabell. Vi ska se vad denna teknik kan leda till här.

Först behöver vi en ofta stor tabell, `tab`, som vi skapar dynamiskt när vi vet hur stort problemet är. Arrayen `varit` fungerar som en mängd som håller reda på vilka städer vi besökt. `a` talar om vilken stad vi just nu befinner oss i.

```
1  int tilltal(int a){
2    int i,tal;
3    tal=0;
4    for(i=1;i<antal;i++)
5      tal=2*tal+varit[i];
6    tal=tal*32+a;
7    return tal;
8  }
```

H Tillsammans bildar `varit` och `a` en situation som eventuellt kan dyka upp flera gånger i rekursionsträdet. Genom att se `varit` som ett binärt tal, `tal`, med sina 1:or och 0:or, kan vi översätta det till ett tal. Om vi begränsar antalet städer till 31, räcker det med 5 bitar för att lagra detta tal. Genom `tal*32+a` har vi bildat ett tal som unikt beskriver situationen. `tal` fungerar nu som index till vår tabell.

```
1  void res(int a, int n,int tot){
2    int i,j,t,avstand;
3    t=tilltal(a);
4    if(tab[t]<=tot)
5      return;
6    if(n==antal-1){
7      tot=tot+avst[a][0];
8      if (tot<min)
9        min=tot;
10   }
11   else
12     for(i=0;i<antal;i++){
13       if (varit[i]==0 && tot+avst[a][i]<min){
14         varit[i]=1;
15         avstand=tot+avst[a][i];
16         res(i,n+1,avstand);
17         if(avstand<tab[t])
18           tab[t]=avstand;
19         varit[i]=0;
20       }
21     }
22 }
```

H Funktionen `res` består av en vanlig backtracking-algoritm.

3 Först tar vi reda på vilket index `t` situationen har i `tab`

4-5 Om det reda finns ett bättre resultat i tabellen avbryter vi anropet. Det är här vi kommer att vinna tid.

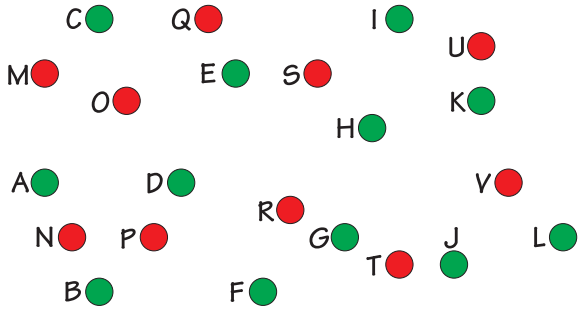
6-10 Om `n`, som håller reda på antalet städer vi besökt är ett mindre än antalet städer i problemet är vi nästan framme. Återstår bara att addera resan tillbaka till stad `1` där vi startade. Sedan kollar vi om vi funnit ett bättre minimum än tidigare.

12-21 För att lägga till en ny stad får vi inte ha varit där tidigare, samt om vi lägger till avståndet till denna stad får det inte överskrida gällande minimum.

17-18 Om vi kommit så här långt och har ett resultat som är bättre än tidigare lagra vi det i `tab`.

Genom denna förbättring kommer krävs det nu endast 281 239 anrop. Det vi vinner i tid förlorar vi dock i utrymme.

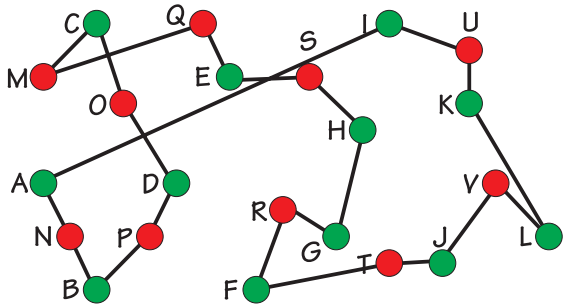
13.4.2 Ett lite större problem



Figur 13.3: *Karta över de 22 städerna A till V*

När vi nu utökar problemet till 22 städer kan vi fortfarande använda oss av den glupska algoritmen och få svaret 403 om vi startar i A. Att det inte kan vara den optimala lösningen ser vi direkt från figur 13.7

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
A	0	22	31	25	40	44	55	60	71	76	81	95	20	11	21	22	42	45	53	66	83	85
B	22	0	50	25	47	30	46	58	74	65	78	85	41	11	35	14	53	38	56	55	83	77
C	31	50	0	33	26	58	60	53	55	79	71	93	14	40	15	41	20	49	41	71	70	80
D	25	25	33	0	22	25	31	36	50	52	57	70	32	22	18	11	30	20	32	42	60	60
E	40	47	26	22	0	40	36	26	31	53	45	67	35	42	20	33	11	26	15	46	45	53
F	44	30	58	25	40	0	18	36	55	35	53	55	56	36	43	22	50	15	41	25	60	49
G	55	46	60	31	36	18	0	20	41	20	35	40	62	50	47	35	47	11	30	11	43	31
H	60	58	53	36	26	36	20	0	20	29	20	40	60	58	45	44	36	21	14	25	25	26
I	71	74	55	50	31	55	41	20	0	46	21	50	65	72	52	60	35	40	18	45	15	36
J	76	65	79	52	53	35	20	29	46	0	30	20	82	70	67	55	63	31	43	10	40	18
K	81	78	71	57	45	53	35	20	21	30	0	29	80	79	65	65	52	40	30	33	10	15
L	95	85	93	70	67	55	40	40	50	20	29	0	99	90	83	75	76	50	54	30	38	14
M	20	41	14	32	35	56	62	60	65	82	80	99	0	30	15	36	31	51	50	73	80	87
N	11	11	40	22	42	36	50	58	72	70	79	90	30	0	26	15	47	40	54	60	82	80
O	21	35	15	18	20	43	47	45	52	67	65	83	15	26	0	25	21	36	35	58	65	71
P	22	14	41	11	33	22	35	44	60	55	65	75	36	15	25	0	41	25	42	45	69	65
Q	42	53	20	30	11	50	47	36	35	63	52	76	31	47	21	41	0	38	22	57	50	62
R	45	38	49	20	26	15	11	21	40	31	40	50	51	40	36	25	38	0	25	22	46	40
S	53	56	41	32	15	41	30	14	18	43	30	54	50	54	35	42	22	25	0	38	30	40
T	66	55	71	42	46	25	11	25	45	10	33	30	73	60	58	45	57	22	38	0	42	25
U	83	83	70	60	45	60	43	25	15	40	10	38	80	82	65	69	50	46	30	42	0	25
V	85	77	80	60	53	49	31	26	36	18	15	14	87	80	71	65	62	40	40	25	25	0



Figur 13.4: *En del val blir helt tokiga när man är för "glupsk"*

Att köra en fullständig backtracking är det inte tal om eftersom $21! = 51\,090\,942\,171\,709\,440\,000$. Det är troligtvis heller ingen idé att testa backtracking med det extra villkoret som klipper i trädet. Vi skulle kunna använda 403, som den glupska algoritmen ger. Det är inte ovanligt att man använder ett resultat från en glupsk algoritm för att ge ett hyfsat startvärde.

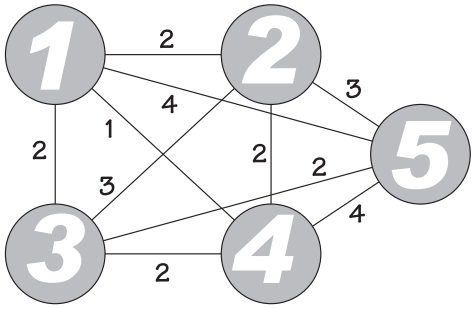
Men i det här fallet duger inte heller det. Frågan är nu om vi över huvud taget kan klara det här problemet med rimliga resurser.

Med hjälp av dynamisk programmering kan man sänka komplexiteten från $O(n!)$ till $O(n^2 2^n)$ för vårt exempel betyder det 42 030 043 136 ett litet antal i jämförelse med det enorma talet ovan.

Här följer en skiss över hur man tänker då man konstruerar en algoritm för TSP med hjälp av dynamisk programmering. Först måste man definiera vad man menar med ett delprob-

lem och sedan hur man använder lösningen av detta problem för att utöka delproblemet till fler städer.

Antag att vi startade i 1 och att vi därefter besökt ett antal städer och nu befinner oss i stad i. Dessutom känner vi till den kortaste resa som startat i 1 och slutat i och gått genom de städer vi besökt. För att nu lägga till ytterligare en stad j, studerar vi alla delresor (som startar i 1 och slutar i och som inte har gått genom j.



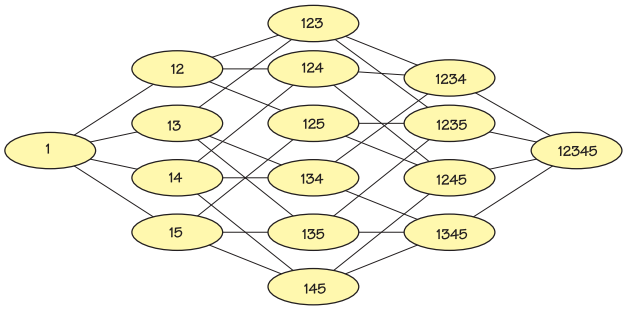
Figur 13.5:

```
1 int avst[100][100],varit[100],antal;
2 short int *tab;
3 char resa[100];
```

H Globala variabler. I `avst` lagras avståndsmatrisen. `varit` är en mängd som håller reda på i vilka städer vi varit. `antal` anger antalet städer. `tab` är en pekare till en, ibland gigantisk array, som ska hålla reda på alla mellanresultat. I `Resa` lagras namnen på städerna i den ordning de besöks.

```
1 int tilltal(int varit[],int a){
2   int i,tal;
3   tal=0;
4   for(i=1;i<antal;i++)
5     tal=2*tal+varit[i];
6   tal=tal*32+a;
7   return tal;
8 }
```

H Här översätts mängden `varit` till ett index som ska användas för i `tab`. Det först framräknadet talet multipliceras med 32 för att man sedan ska kunna addera ett tal `a`, upp till 31, som står för den stad man just nu befinner sig i.



Figur 13.6:

```
1  void kontroll(int varit[]){
2    int i,j,k,index1,index2,av;
3    for(i=1;i<antal;i++)
4      if(varit[i]==1){
5        index1=tilltal(varit,i);
6        varit[i]=0;
7        for(j=0;j<antal;j++)
8          if(varit[j]){
9            index2=tilltal(varit,j);
10           av=tab[index2]+avst[i][j];
11           if(av<tab[index1])
12             tab[index1]=av;
13         }
14       varit[i]=1;
15     }
16 }
```

H Programmets hjärta. Indata är en mängd `varit`

3-4 Vi söker i tur och ordning upp alla städer som ingår i mängden.

5 Vi bestämmer oss för att vi befinner oss i staden `i` och beräknar motsvarande index, där vi senare ska lagra ett resultat.

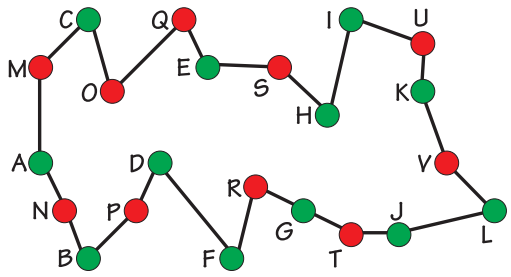
6 Vi plockar tillfälligt bort staden ur mängden.

7-9 Nu har vi en mängd som innehåller en stad mindre. Vi söker upp alla städer i denna mängd och beräknar index.

10-12 I `tab[index2]` finns ett avstånd lagrat, den kortaste resan mellan städerna i mängden. Vi lägger så till avståndet mellan denna stad och den vi just plockade bort. Om summan är mindre än det som tidigare finns i `tab[index1]` har vi funnit en bättre resa och uppdaterar `tab[index1]`.

```
1  void generera(int m[],int n,int s,int k){
2    int i;
3    if(n==k)
4      kontroll(m);
5    else
6      for(i=s;i<antal;i++){
7        m[i]=1;
8        generera(m,n+1,i+1,k);
9        m[i]=0;
10     }
11 }
```

H En funktion vi sett tidigare, som genererar alla mängder med k element.



Figur 13.7: *I det slutliga resultatet finns inget som överraskar*

```
1  int main(void){
2    int i,j,langd=1,min=INT_MAX,a;
3    FILE *fil;
4    fil=fopen("TSP3.dat","rt");
5    fscanf(fil,"%d",&antal);
6    for(i=0;i<antal;i++)
7      for(j=0;j<antal;j++)
8        fscanf(fil,"%d",&avst[i][j]);
9    fclose(fil);
```

```
1   langd=(int)pow(2,antal)*32/2;
2   tab=(short int *)malloc(langd*sizeof(short int));
3   if(tab==NULL){
4     printf("*** FEL ***");
5     getch();
6     return;
7   }
8   for(i=0;i<langd;i++)
9     tab[i]=SHRT_MAX;
10  tab[0]=0;
11  varit[0]=1;
12  for(i=0;i<=antal-1;i++)
13    generera(varit,0,1,i);
14  for(i=1;i<antal;i++){
15    a=tab[(int)(pow(2,antal-1)-1)*32+i]+avst[0][i];
16    if(a<min)
17      min=a;
18  }
19  free(tab);
20  printf("Kortaste resan : %d\n",min);
21 }
```

11-19 Vi bestämmer hur stor den dynamiska arrayen `tab` ska vara. Genererar den. Håller koll på att den verkligen får plats på heapen. Tabellen är skapad med `short int` för spara plats och för att vi tror `SHRT_MAX` ska räcka för våra relativt små problem.

20 `tab[0]=0` indikerar att då resan startar i stad 1 är tillryggalagd sträcka 0

21 `varit[0]=1` betyder att stad 1 kommer att ingå i alla mängder.

24-28 Det sista steget då vi tar reda på det optimala värdet genom att resa hem till stad 1 igen.