Pretraživanje sa vraćanjem (backtracking), gramzivi algoritmi i algoritam grana i granica

Strukture podataka i algoritmi 2

Algoritamski postupak

- Algoritamski postupak – ideja ili šablon primenljiv na širi spektar algoritamskih problema
- Pretraživanje sa vraćanjem (engl. ***backtracking***) je algoritamski postupak primenjiv na široku klasu problema kombinatorne enumeracije i optimizacije
- Kombinatorna enumeracija: naći sve konfiguracije nekog sistema koje zadovoljavaju određen kriterijum
  - Naći sve podskupove nekog skupa pozitivnih brojeva čiji je zbir jednak nekom broju T
  - Naći sve rasporede 8 kraljica na šahovskoj tabli tako da se kraljice međusobno ne napadaju
- Kombinatorna optimizacija: naći konfiguraciju nekog sistema koja zadovoljava određen **optimizacioni** kriterijum
  - Problem trgovačkog putnika: naći najkraću kompletnu turu na mapi gradova

Konfiguracije kombinatornog sistema

- Konfiguraciju nekog kombinatornog sistema možemo uvek predstaviti vektorom/nizom C = (c_0, c_1, c_2, c_3, c_4, ..., c_t) čije elemente nazivamo komponentama.

- Konfiguracija dama na šahovskoj tabli – vektor od 8 komponenti, komponente su koordinate polja na koje su postavljene kraljice

- Konfiguracija podskupa skupa S – logički vektor C od k komponenti, gde je k kardinalnost skupa S.
  - C[i] = true → S[i] je u podskupu
  - C[i] = false → S[i] nije u podskupu

- **Parcijalna konfiguracija kombinatornog sistema** – vektor C_k, $k < t$, kod koga su prvih k komponenti poznati (određeni), dok su ostale komponente nepoznate (neodređene)

Backtracking

- **Problem: naći sve konfiguracije C koje zadovoljavaju logički predikat P**
- Osnovna ideja backtrackinga:
  - Na osnovu predikata P odredimo predikat Q koji je primenljiv na parcijalne konfiguracije sistema kojim definišemo da li neka parcijalna konfiguracija C_k **proširiva**
    - Kraljice: Q = postoji C_{k+1} u kojoj se kraljice ne napadaju
    - Subset sum: Q = postoji C_{k+1} u kojoj je zbir elemenata <= T
  - Parcijalne konfiguracije koje ne zadovoljavaju Q zovemo **konfiguracijama neuspeha**.
  - Parcijalne konfiguracije koje zadovoljavaju Q zovemo još i **potencijalnim konfiguracijama (konfiguracije koje mogu voditi do uspeha)**.
  - **Backtrakcing je sistematično (iscrpno, sveobuhvatno) proširivanje potencijalnih konfiguracija**

Stablo bektreka

- Iscrpno proširivanje potencijalnih konfiguracija možemo predstaviti *backtracking stablom*

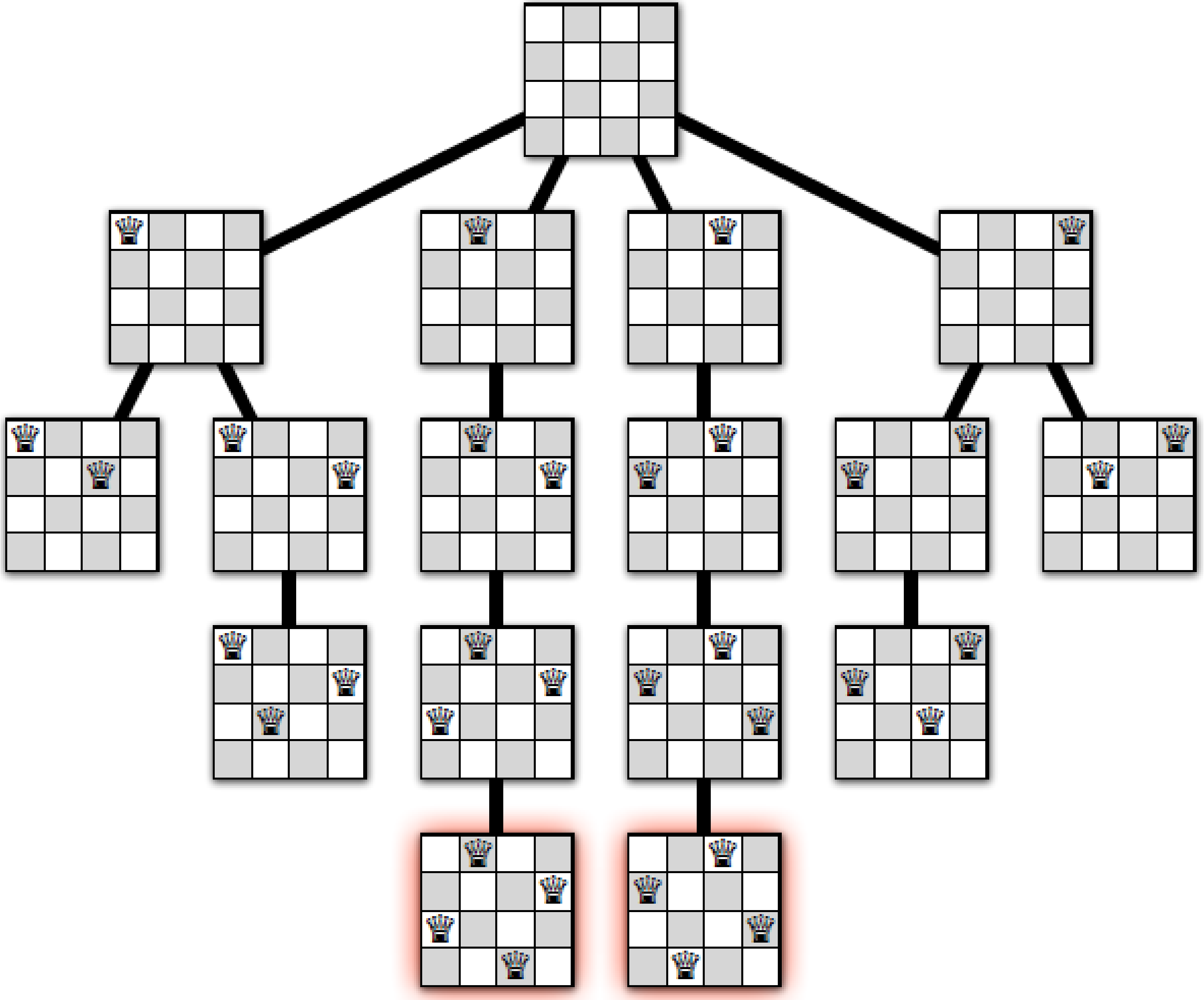
- Koren stabla: prazna konfiguracija

- Unutrašnji čvorovi stabla: potencijalne konfiguracije

- Listovi stabla: ciljne konfiguracije (konfiguracije koje zadovoljavaju P) ili konfiguracije neuspeha

- Čvor A je sin čvora B: konfiguracija A se dobija proširivanjem potencijalne konfiguracije B

Stablo bektreka



Backtracking

- Backtracking je “obilazak” backtracking stabla **u dubinu**
  - Ne konstruišemo stablo bektreka – imamo jednu parcijalnu konfiguraciju koja se menja **proširivanjem i sažimanjem**
  - Dinamika promene te parcijalne konfiguracije korespondira obilasku stabla bektreka u dubinu
  - Parcijalna konfiguracija se proširuje u ciljnu konfiguraciju (konfiguracija koja zadovoljava *P*) → imamo rešenje
  - Ako se parcijalna konfiguracija ne može proširiti tada je **sažimamo i nastavljamo sa proširenjima sažete parcijalne konfiguracije**
    - Sažimanje – izbacivanje poslednje komponente iz parcijalne konfiguracije
  - Proširenje parcijalne konfiguracije ne uspeva iz dva razloga
    - Već smo ispitali sve proširenja parcijalne konfiguracije
    - Proširenje vodi u konfiguraciju neuspeha
  - **Sažimanje prve parcijalne konfiguracije → kraj backtrackinga**

Backtracking

- **Moramo uvesti neki red prilikom proširivanja trenutne parcijalne konfiguracije**
  - Da ne bi proširivali na neku konfiguraciju koja je već bila ispitivana

- Za trenutnu parcijalnu konfiguraciju dužine k – 1 formiramo D_k: listu dopustivih elemenata za k-tu komponentu
  - Kako imamo *t* komponenti to ćemo imati *t* listi dopustivih elemenata.

- Trenutnu parcijalnu konfiguraciju C_{k-1} proširujemo tako što
  - f = prvi element liste D_k
  - Obrišemo f iz D_k
  - Formiramo C_k proširenjem C_{k-1} sa f

```
Vector C;                          // konfiguracija sistema
void backtracking() {
    D[] = new D[C.length];         // liste dopustivih elemenata
    int k = 0;                     // trenutna komponenta (dužina parcijalne konfiguracije)

    compute D[0];                  // odredi dopustive elemente za prvu komponentu
    while (k >= 0) {
        while (!D[k].empty) {
            f = remove first element in D[k];
            C[k] = f;
            if (k == C.length – 1)
                printConfiguration();        // imamo rešenje
            else {
                k++;                         // idemo na sledeću komponentu
                compute D[k];                // odredi dopustive elemente za sledeću komp.
            }
        }
        k--;                                 // vraćamo se nazad
    }
}
```

Određivanje dopustivih elemenata se vrši na osnovu predikata Q primenjenog na trenutnu parcijalnu konfiguraciju

```
Vector C;                         // trenutna konfiguracija sistema

void backtracking() {
    backtracking(0);
}

// k – trenutna komponenta
void backtracking(int k) {
    compute D;                    // odredi listu dopustivih elemenata za k-tu komponentu
    while (!D.empty()) {
        f = remove first element in D;
        C[k] = f;
        if (k == C.length – 1)
            printConfiguration();
        else
            backtracking(k + 1)
    }
}
```

Lista dopustivih elemenata

- Ukoliko za komponente C znamo da su u nekom intervalu **[min, max]** tada umesto liste dopustivih elemenata možemo čuvati samo jedan dopustiv element

- Prvi dopustiv element – najmanji dopustiv element

- Sledeći dopustiv element se određuje na osnovu prethodnog dopustivog elementa

- Kraljice: vrste/kolone su u intervalu [0 .. 7]

```
Vector C;

void backtracking() {
    Vector D;
    int k = 0;

    D[0] = first available element >= min
    while (k >= 0) {
        while (D[k] <= max) {
            C[k]  = D[k]
            D[k]  = first available element > D[k] or max + 1 (no available elements)
            if (k == C.length – 1) printConfiguration();
            else {
                k++;
                D[k] = first available element >= min or max + 1 (no available elements)
            }
        }
        k--;
    }
}
```

Problem *N* kraljica

- N kraljica je potrebno rasporediti na šahovsku tablu dimenzija N x N tako da nema kraljica koje se međusobno napadaju. **Koliko ima takvih rasporeda i koji su?**

- Dve kraljice na pozicijama (x_1, y_1), (x_2, y_2) se napadaju ako je zadovoljen jedan od sledećih uslova
  - $x_1 = x_2$
  - $y_1 = y_2$
  - $| x_1 - x_2 | = | y_1 - y_2 |$

- Bektrekovaćemo po vrstama
  - Može i po kolonama, potpuno svejedno

Raspored kraljica na tabli

- Predstavićemo nizom, ne matricom *boolean*-a

  int[] table = new int[N];

- Kraljice se nalaze na pozicijama
  - **(0, table[0])**
  - **(1, table[1])**
  - **(2, table[2])**
  - **…**
  - **(N – 1, table[N – 1])**

Verzija 1 – liste dopustivih polja

```java
public class NQueens {
	private int n;             // velicina table
	private int slCounter;     // brojac resenja
	private int[] table;       // raspored kraljica

	// lista raspolozivih polja za neku vrstu
	private class AvailableCell {
		int y;
		AvailableCell next;

		public AvailableCell(int y) {
			this.y = y;
		}
	}

	// availableCells[i] - lista raspolozivih polja za i.-tu vrstu
	private AvailableCell[] availableCells;

	public NQueens(int n) {
		this.n = n;
		table = new int[n];
		availableCells = new AvailableCell[n];
	}
	//... to be continued
}
```

Štampanje rešenja

```
private void printSolution() {
	++slCounter;
	System.out.println("Solution " + slCounter);
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			if (j == table[i]) {
				System.out.print("X ");
			} else {
				System.out.print("- ");
			}
		}
		System.out.println();
	}
	System.out.println();
}
```

Bektrek

```java
public void find() {
    // formiraj listu raspolozivih polja za prvu vrstu
    for (int i = n - 1; i >= 0; i--) {
        AvailableCell c = new AvailableCell(i);
        c.next = availableCells[0];
        availableCells[0] = c;
    }

    int currentRow = 0;
    while (currentRow >= 0) {
        while (availableCells[currentRow] != null) {
            table[currentRow] = availableCells[currentRow].y;
            availableCells[currentRow] = availableCells[currentRow].next;

            // proveri da li imamo resenje
            if (currentRow == n - 1) {
                printSolution();
            } else {
                // idi napred
                currentRow++;
                findAvailableCells(currentRow);
            }
        }

        // vrati se unazad
        currentRow--;
    }
}
```

```java
private void findAvailableCells(int currentRow) {
	availableCells[currentRow] = null;

	for (int j = n - 1; j >= 0; j--) {
		// proveravamo da li je polje (currentRow, j) dostupno
		// tako sto za sve prethodne vrste i, 0 <= i < currentRow
		// proveravamo da li je polje (i, table[i]) u koliziji sa
		// poljem (currentRow, j). Kolizija se desava ako je
		// (1) table[i] = j - kraljice se napadaju "vertikalno"
		// (2) |i - currentRow| = |table[i] - j|
		//      kraljice se napadaju "dijagonalno"
		boolean available = true;
		for (int i = 0; i < currentRow; i++) {
			if (table[i] == j ||
			    Math.abs(i - currentRow) == Math.abs(table[i] - j))
			{
				available = false;
				break;
			}
		}

		if (available) {
			AvailableCell c = new AvailableCell(j);
			c.next = availableCells[currentRow];
			availableCells[currentRow] = c;
		}
	}
}
```

92

```java
public static void main(String[] args) {
    NQueens nq = new NQueens(8);
    nq.find();
}
```

```
Solution 92
- - - - - - - X
- - - X - - - -
X - - - - - - -
- - X - - - - -
- - - - - X - -
- X - - - - - -
- - - - - - X -
- - - - X - - -
```

Verzija 2 – čuvanje jednog dopustivnog elementa

```
public class NQueensMin {

	private int n;
	private int slCounter;
	private int[] table;

	// firstAvailableCell[i] - prvo dostupno polje za vrstu i
	private int[] firstAvailableCell;

	public NQueensMin(int n) {
		this.n = n;
		table = new int[n];
		firstAvailableCell = new int[n];
	}

	//... to be continued
}
```

Bektrek

```
public void find() {
	firstAvailableCell[0] = 0;
	int currentRow = 0;

	while (currentRow >= 0) {
		while (firstAvailableCell[currentRow] < n) {
			table[currentRow] = firstAvailableCell[currentRow];
			firstAvailableCell[currentRow]++;
			updateFirstAvailableCell(currentRow);

			if (currentRow == n - 1) {
				printSolution();
			} else {
				currentRow++;
				firstAvailableCell[currentRow] = 0;
				updateFirstAvailableCell(currentRow);
			}
		}

		currentRow--;
	}
}
```

Ažuriranje dopustivog polja

```
private void updateFirstAvailableCell(int currentRow) {
	for (int j = firstAvailableCell[currentRow]; j < n; j++) {
		// proveravamo da li je polje (currentRow, j) u koliziji
		// sa ostalim poljima
		boolean available = true;
		for (int i = 0; i < currentRow; i++) {
			if (table[i] == j ||
			    Math.abs(i - currentRow) == Math.abs(table[i] - j))
			{
				available = false;
				break;
			}
		}

		if (available) {
			firstAvailableCell[currentRow] = j;
			return;
		}
	}

	// nema dostupne celije
	firstAvailableCell[currentRow] = n;
}
```

Rekurzivno rešenje

```
public void find() {
      find(0);
}

private void find(int currentRow) {
      if (currentRow == n) {
            printSolution();
      } else {
            for (int j = 0; j < n; j++) {
                  if (availableCell(currentRow, j)) {
                        table[currentRow] = j;
                        find(currentRow + 1);
                  }
            }
      }
}

private boolean availableCell(int currentRow, int currentColumn) {
    for (int i = 0; i < currentRow; i++) {
        if (table[i] == currentColumn ||
            Math.abs(i - currentRow) == Math.abs(table[i] - currentColumn)
         ) {
            return false;
         }
    }

    return true;
}
```

Problemi kombinatorne optimizacije

- Probleme kombinatorne optimizacije je često moguće formulisati na sledeći opšti način:
  - Dat je skup *S* od *n* objekata.
  - Svaki objekat ima neku vrednost, težinu, cenu ili profit koja je predstavljena funkcijom *f*: *S* → Integer
  - Funkcija *f* je definisana i na partitivnom skupu skupa S i aditivna je.
    - *f*(T) = zbir *f*(x) za svako *x* iz *T*, gde je *T* podskup *S*
  - Potrebno je naći podskup skupa S koji zadovoljava određena ograničenja definisana logičkim predikatom *P* tako da se *f* maksimizuje (ili minimizuje).

Problem ruksaka

- Problem ruksaka je jedna od instanci problema sa prethodnog slajda.

- Dat je skup *S* od *n* predmeta pri čemu svaki predmet ima težinu (*weight*) i tržišnu cenu (*profit*).
- Dat je ranac u koji je moguće smestiti predmete čija težina ne prevazilazi neku maksimalnu dozvoljenu težinu *knapsackWeight*.
- Problem: napuniti ranac predmetima iz *S* tako da se maksimizuje profit.

- **U čemu je lepota problema ruksaka: gramzivo punjenje ranca ne maksimizuje profit u opštem slučaju.**

Gramzivi algoritmi

- Posmatramo opštu formulaciju problema kombinatorne optimizacije

```
Set greedy(Set S) {
    Sort S in non-increasing order by f
        or some function derived from f
    Set solution = empty set
    for (int i = 0; i < S.length; i++) {
        if (solution U {S[i]} satisfies P) {
            solution = solution U {S[i]}
        }
    }
    return solution
}
```

Gramzivi algoritmi

- Ideja gramzivih algoritama je da
  - Lokalno optimalne odluke vode globalno optimalnom rešenju
  - Jednom donešena odluka se ne može opozvati (nema vraćanja nazad)

- Gramzive strategije za punjenje ranca
  - Sortiraj elemente po profitu
  - Sortiraj elemente po težini od najmanje ka najvećoj
  - Sortiraj elemente po “gustini profita” (profit / weight)

Konkretan primer

- Maksimalna dozvoljena težina ranca je 11

#	W	P	P/W
1	7	28	4
2	6	22	3.66
3	5	18	3.6
4	2	6	3
5	1	1	1

- Gramziv po profitu = gramziv po gustini profita
  - Maks profit = 35, R = {1, 4, 5}
- Gramziv po težini
  - Maks profit = 25 , R = {3, 4, 5}
- Optimalno rešenje
  - Maks profit = 40, R = {2, 3}

Bektrek rešenje problema ruksaka

- Bektrekom tražimo sve podskupove T skupa predmeta S koji zadovoljavaju kriterijum maksimalne težine ruksaka...
- ... i trenutni podskup T sačuvamo kao optimalno rešenje ukoliko je profit ostvaren sa T veći od do tada maksimalnog profita
- Konfiguraciju sistema predstavimo logičkim nizom
  in[k] = true → k-ti predmet u ruksaku
  in[k] = false → k-ti predmet nije u ruksaku
- Dopustivi elementi za k-tu komponentu su:
  - **true** ako se k-ti element može smestiti u ranac
    - zbir težina prethodno ubačenih predmeta + težina k-tog predmeta manja ili jednaka od maksimalne dozvoljene težine
  - **false** uvek

Item – klasa koja predstavlja jedan predmet

```java
public class Item {

	private int weight;
	private int profit;

	public Item(int weight, int profit) {
		this.weight = weight;
		this.profit = profit;
	}

	public int getWeight() {
		return weight;
	}

	public int getProfit() {
		return profit;
	}

	public String toString() {
		return "(" + weight + ", " + profit + ")";
	}
}
```

Rekurzivno rešenje...

```
public class KnapsackRec {

	private int knapsackWeight;
	private Item[] items;
	private boolean in[];

	private int maxProfit;
	private boolean maxProfitConfiguration[];

	public KnapsackRec(int knapsackWeight, Item[] items) {
		this.items = items;
		this.knapsackWeight = knapsackWeight;
		in = new boolean[items.length];
		maxProfitConfiguration = new boolean[items.length];
	}

	//... to be continued
}
```

```java
public void find() {
	find(0, 0, 0);
}

private void find(int k, int currentWeight, int currentProfit) {
	if (k == items.length) {
		if (currentProfit > maxProfit) {
			// sacuvaj resenje
			maxProfit = currentProfit;
			for (int i = 0; i < items.length; i++) {
				maxProfitConfiguration[i] = in[i];
			}
		}
	} else {
		if (items[k].getWeight() + currentWeight <= knapsackWeight) {
			in[k] = true;
			find(
				k + 1,
				currentWeight + items[k].getWeight(),
				currentProfit + items[k].getProfit()
			);
		}

		in[k] = false;
		find(k + 1, currentWeight, currentProfit);
	}
}
```

Metoda koja štampa rešenje

```
public void printSolution() {
     System.out.println("Max profit: " + maxProfit);
     System.out.println("Solution... ");
     for (int i = 0; i < items.length; i++) {
          if (maxProfitConfiguration[i]) {
               System.out.println(items[i]);
          }
     }
}
```

Primer upotrebe klase

```
public static void main(String[] args) {
	int[] weight = {1, 2, 5, 6, 7};
	int[] profit = {1, 6, 18, 22, 28};
	Item[] items = new Item[weight.length];
	for (int i = 0; i < items.length; i++) {
		items[i] = new Item(weight[i], profit[i]);
	}

	KnapsackRec kr = new KnapsackRec(11, items);
	kr.find();
	kr.printSolution();
}
```

```
Max profit: 40
Solution...
(5, 18)
(6, 22)
```

Algoritam grana i granica (engl. branch and bound)

- Algoritam grana i granica je modifikacija *backtracking* postupka za probleme kombinatorne optimizacije

- Ideja:
  - $C_{k\text{-}1}$: trenutna parcijalna konfiguracija sistema
  - D_k: skup dopustivih elemenata za *k*-tu komponentu
  - Za svako *d* iz D_k **ocenimo gornje ograničenje (bound)** funkcije *f* u ciljnjim konfiguracijama koje se dobijaju iz $C_{k\text{-}1}$ proširenim sa *d*.
  - Da li se isplati proširiti $C_{k\text{-}1}$ sa *d?*
    - Ne, ako je **bound(d) <= trenutnog maksimalnog *f***.

Branch and bound rešenje problema ruksaka

- Neka je *M* trenutno maksimalno rešenje
- Za *k*-ti predmet i trenutnu parcijalnu konfiguraciju je potrebno oceniti
  - Koliko je gornje ograničenje maksimalnog profita (PA) ubacivanjem predmeta u ruksak (ako se predmet može dodati u ruksak)?
  - Koliki je gornje ograničenje maksimalnog profita (PS) neubacivanjem predmeta u ruksak?
- Ako je PA > M – ubacujem predmet u ruksak i nastavljam dalje sa bektrekom
- Ako je PS > M – ne ubacujem predmet u ruksak i nastavljam dalje sa bektrekom
- **Gornje ograničenje maksimalnog profita je moguće ustanoviti koristeći gramzivi algoritam.**

Ocena gornjeg ograničenja maksimalnog profita

- n – broj predmeta
- $S = \{s_1, s_2, \ldots, s_n\}$ – skup predmeta
- C_{k-1} - trenutna parcijalna konfiguracija ruksaka
- P - profit ostvaren trenutnom parcijalnom konfiguracijom
- W' – preostalo mesto u ruksaku u trenutnoj parcijalnoj konf.
- Za k-ti predmet $s_k = (p_k, w_k)$ imamo dve opcije
  - s_k dodajemo u ruksak

    $bound(s_k) = P + p_k + bound(\{s_{k+1} .. s_{n-1}\}, W' - w_k)$
  - s_k ne dodajemo u ruksak

    $bound(s_k) = P + bound(\{s_{k+1} .. s_{n-1}\}, W')$

gde je **bound(S, W)** gornje ograničenje maksimalnog profita za skup predmeta S i ruksak kapaciteta W

bound(S, W)

- Problem ruksaka podrazumeva da su predmeti nedeljivi
- Ako bi predmeti bili deljivi (možemo uzeti parče predmeta) tada bi maksimalni profit mogli dobiti gramzivim algoritmom po gustini profita.
- Profit ostvaren selekcijom nedeljivih predmeta je uvek manji ili jednak od profita ostvarenog selekcijom deljivih predmeta.

#	W	P	P/W
1	7	28	4
2	6	22	3.66
3	5	18	3.6
4	2	6	3
5	1	1	1.0

Kapacitet ruksaka W = 11

Maksimalan profit = 40

uzmemo drugi i treći predmet

Bound = 28 + 4/6 * 22 = 42.66

uzmemo prvi i 4/6 drugog predmeta

```java
public class Item implements Comparable<Item> {
	private int weight, profit;

	public Item(int weight, int profit) {
		this.weight = weight;
		this.profit = profit;
	}

	public int getWeight() { return weight; }
	public int getProfit() { return profit; }

	public String toString() {
		return "(" + weight + ", " + profit + ")";
	}

	public int compareTo(Item other) {
		double pd = (double) profit / (double) weight;
		double opd = (double) other.profit / (double) other.weight;
		if (pd > opd)
			return -1;
		else if (pd < opd)
			return 1;
		else
			return 0;
	}
}
```

```java
public class KnapsackBB {

	private int knapsackWeight;
	private Item[] items;
	private boolean in[];

	private int maxProfit;
	private boolean maxProfitConfiguration[];

	public KnapsackBB(int knapsackWeight, Item[] items) {
		this.items = items;
		this.knapsackWeight = knapsackWeight;
		in = new boolean[items.length];
		maxProfitConfiguration = new boolean[items.length];
	}

	public void find() {
		Arrays.sort(items);
		find(0, 0, 0);
	}

	//... to be continued
}
```

Branch and bound

```
public void find(int k, int currentWeight, int currentProfit) {
        if (k == items.length) {
                if (currentProfit > maxProfit) {
                        updateBestSolution(currentProfit);
                }
        } else {
                if (items[k].getWeight() + currentWeight <= knapsackWeight) {
                        int nextWeight = items[k].getWeight() + currentWeight;
                        int nextProfit = items[k].getProfit() + currentProfit;
                        double boundAdd = nextProfit +
                                        bound(k + 1, knapsackWeight - nextWeight);
                        if (boundAdd > maxProfit) {
                                in[k] = true;
                                find(k + 1, nextWeight, nextProfit);
                        }
                }

                double boundSkip = currentProfit +
                                bound(k + 1, knapsackWeight - currentWeight);
                if (boundSkip > maxProfit) {
                        in[k] = false;
                        find(k + 1, currentWeight, currentProfit);
                }
        }
}
```

bound

```
private double bound(int k, int weight) {
        int sumw = 0;
        double profitBound = 0.0;
        while (k < items.length && sumw + items[k].getWeight() <= weight) {
                sumw += items[k].getWeight();
                profitBound += items[k].getProfit();
                k++;
        }

        if (k < items.length) {
                double fraction = (double) (weight - sumw) /
                                  (double) items[k].getWeight();

                profitBound += items[k].getProfit() * fraction;
        }

        return profitBound;
}
```

Generisanje skupova bektrekom

- Pretraživanjem sa vraćanjem možemo generisati kompleksne skupove koji se formiraju na osnovu nekog skupa *A*

- Primeri kompleksnih skupova izvedenih iz skupa *A*
  - *P*(*A*) - partitivni skup skupa *A*
  - Podskup *P*(*A*) koji sadrži skupove kardinalnosti *k* - kombinacije bez ponavljanja skupa *A* *k*-te klase
  - Skup A^k – varijacije sa ponavljanjem skupa *A* *k*-te klase
  - Skup svih permutacija elemenata skupa *A*

- Pretpostavićemo da je skup A reprezentovan nizom

Partitivni skup skupa A

- Partitivni skup skupa A je skup svih podskupova skupa A (uključivši prazan skup i sam skup A).
- **Primer.** A = {2, 7, 9}
  P(A) = { Ø, {2}, {7}, {9}, {2, 7}, {2, 9}, {7, 9}, {2, 7, 9}}
- Konfiguracija kombinatornog sistema – logički niz od *n* elemenata, gde je *n* kardinalnost skupa A
  - in[k] = true → k-ti element skupa A je u podskupu
  - in[k] = false → k-ti element skupa A nije u podskupu
- **Skup dopustivih elemenata za svaku komponentu konfiguracije je {true, false}**

```java
public class PowerSet {
	private Object[] set;
	private boolean[] in;

	public PowerSet(Object[] set) {
		this.set = set;
		in = new boolean[set.length];
	}

	private void printSubset() {
		boolean empty = true;
		for (int i = 0; i < in.length; i++) {
			if (in[i]) {
				empty = false;
				System.out.print(set[i] + " ");
			}
		}

		if (empty)
			System.out.println("Empty set");
		else
			System.out.println();
	}

	// ... to be continued
}
```

```java
public void generate() {
    backtrack(0);
}

private void backtrack(int i) {
    if (i == in.length) {
        printSubset();
    } else {
        in[i] = false;
        backtrack(i + 1);
        in[i] = true;
        backtrack(i + 1);
    }
}
```

Kombinacije bez ponavljanja

- Dat je skup *A* od *n* elemenata
- Neka je *P* partitivni skup skupa A
- C = Kombinacije bez ponavljanja *k*-te klase skupa A
  = svi podskupovi skupa P čija je kardinalnost *k*.
- **Primer.** A = {1, 4, 7, 9}, k = 3
  C = {{1, 4, 7}, {1, 4, 9}, {1, 7, 9}, {4, 7, 9}}
- Očigledno rešenje:
  - Generisati bektrekom sve podskupove skupa A
  - Štampati samo one podskupove koji imaju k elemenata

```java
public void generate() {
	backtrack(0, 0);
}

private void backtrack(int i, int numElements) {
	if (i == n) {
		if (numElements == k)
			printSubset();
	} else {
		if (numElements < k) {
			in[i] = true;
			backtrack(i + 1, numElements + 1);
		}

		in[i] = false;
		backtrack(i + 1, numElements);
	}
}
```

Kombinacije bez ponavljanja

- **Efikasnije rešenje**
  - Konfiguraciju kombinatornog sistema predstaviti celobrojnim nizom c od k elemenata koji predstavlja **indekse selektovanih elemenata** skupa A
  - $c[i] = z$, z-ti element skupa A je i-ti element kombinacije
  - **Npr**. za $A = \{1, 4, 7, 9\}$, $k = 3$, $c = [0, 2, 3]$ imamo kombinaciju

    $K = \{A[0], A[2], A[3]\} = \{1, 7, 9\}$
  - Za $n = 4$, $k = 3$ niz c uzima vrednosti redom

    [0, 1, 2]
    [0, 1, 3]
    [0, 2, 3]
    [1, 2, 3]
  - U opštem slučaju, za i-tu komponentu niza c
    - Minimalni dopustivi element je $c[i - 1] + 1$ (0 za $i = 0$)
    - Maksimalni dopustivi element je $n - k + i$

      Poslednja konfiguracija u opštem slučaju:

      $[n - k, n - k + 1, n - k + 2, \ldots, n - 3, n - 2, n - 1]$

```java
public class Combinations {
	private Object[] set;
	private int n, k;
	private int[] combination;

	public Combinations(Object[] set, int k) {
		if (set == null || set.length == 0)
			throw new IllegalArgumentException("empty set");

		if (k <= 0 || k > set.length)
			throw new IllegalArgumentException("invalid k");

		this.set = set;
		this.n = set.length;
		this.k = k;
		combination = new int[k];
	}

	private void printCombination() {
		for (int i = 0; i < k; i++) {
			System.out.print(set[combination[i]] + " ");
		}
		System.out.println();
	}

	//... to be continued
}
```

```java
public void generate() {
    backtrack(0);
}

private void backtrack(int i) {
    if (i == k) {
        printCombination();
    } else {
        int j = 0;
        if (i > 0)
            j = combination[i - 1] + 1;

        while (j <= n - k + i) {
            combination[i] = j;
            j++;
            backtrack(i + 1);
        }
    }
}
```

Varijacije sa ponavljanjem

- Dat je skup *A* od *n* elemenata
- Potrebno je izgenerisati sve *k*-torke koje pripadaju skupu
  $S = A^k$, $k > 0$
- **Primena.** *k* ugenježdenih petlji, pri čemu je svaka indeksirana po elementima skupa A.
- **Primer.** $A = [1, 4, 7]$, $k = 2$

S = A x A = {(1, 1), (1, 4), (1, 7), (4, 1), (4, 4), (4, 7), (7, 1), (7, 4), (7, 7) }

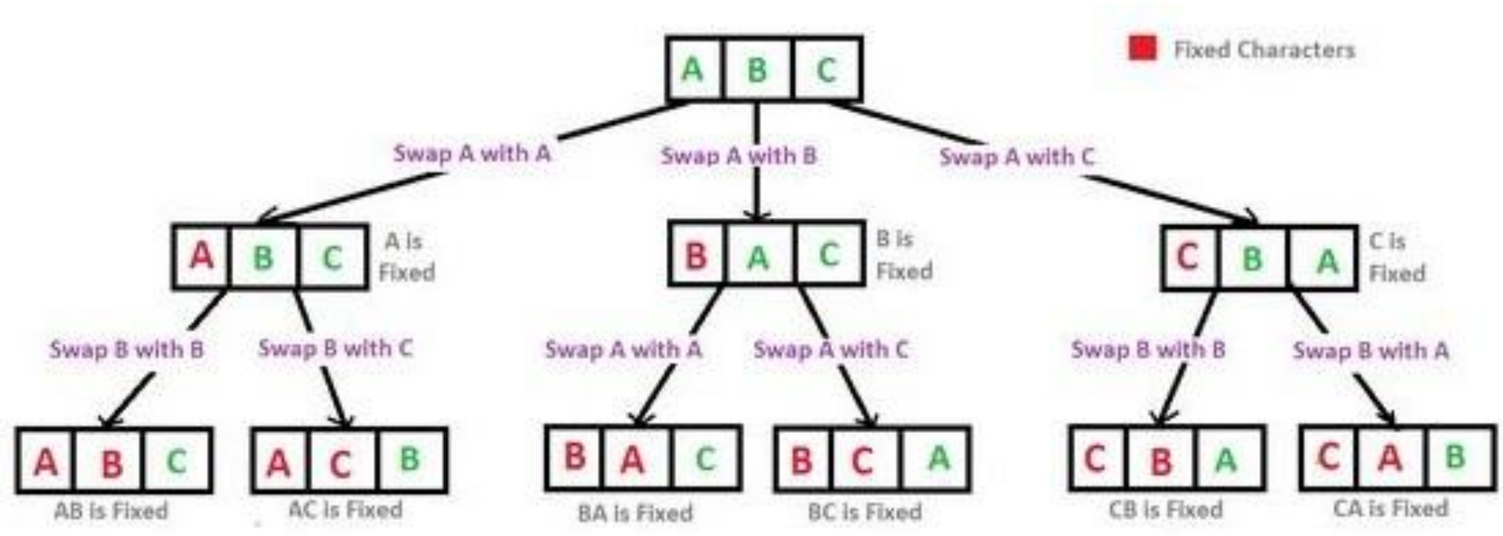
- **Konfiguracija kombinatornog sistema – jedna varijacija skupa *A* (niz *v* od *k* elemenata)**
- **Skup dopustivih elemenata za svaku komponentu konfiguracije je skup *A***

```java
public class Variations {
	private Object[] set;
	private Object[] variation;

	public Variations(Object[] set, int k) {
		if (set == null || set.length == 0)
			throw new IllegalArgumentException("empty set");

		if (k <= 0)
			throw new IllegalArgumentException("k must be > 0");

		this.set = set;
		variation = new Object[k];
	}

	private void printVariation() {
		for (int i = 0; i < variation.length; i++) {
			System.out.print(variation[i] + " ");
		}
		System.out.println();
	}

	//... to be continued
}
```

```java
public void generate() {
    backtrack(0);
}

private void backtrack(int i) {
    if (i == variation.length) {
        printVariation();
    } else {
        for (int j = 0; j < set.length; j++) {
            variation[i] = set[j];
            backtrack(i + 1);
        }
    }
}
```

Permutacije skupa

- Permutacija skupa A je neki redosled elemenata skupa A.
- Npr. za skup A = {1, 2, 3} skup svih permutacija je
  Perm = {[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 2, 1], [3, 1, 2]}
- Permutacije generišemo razmenama elemenata počevši od neke proizvoljne početne permutacije P (npr. P = A)
  - Konfiguracija kombinatornog sistema – jedna permutacija skupa
  - Neka je P' trenutna parcijalna konfiguracija – fiksirani deo permutacije
  - Za svaki element E iz nefiksiranog dela permutacije
    - Zamenimo E sa prvim elementom iz nefiksiranog dela permutacije F
    - Povećamo fiksirani deo permutacije za 1 (idemo napred)
    - Zamenimo E sa F (vraćanje u nazad, poništavanje efekta prve razmene)



```java
public class Permutations {
	private Object[] permutation;
	private int n;

	public Permutations(Object[] set) {
		if (set == null || set.length == 0) {
			throw new IllegalArgumentException("Empty set");
		}

		n = set.length;
		permutation = new Object[n];
		for (int i = 0; i < n; i++) {
			permutation[i] = set[i];
		}
	}

	private void printPermutation() {
		for (int i = 0; i < n; i++)
			System.out.print(permutation[i] + " ");
		System.out.println();
	}

	\\ ... to be continued
}
```

```java
public void generate() { backtrack(0); }

private void backtrack(int k) {
    if (k == n) printPermutation();
    else {
        for (int i = k; i < n; i++) {
            swap(k, i);
            backtrack(k + 1);
            swap(i, k);
        }
    }
}

private void swap(int x, int y) {
    Object tmp = permutation[x];
    permutation[x] = permutation[y];
    permutation[y] = tmp;
}
```