Algoritmi in podatkovne strukture 2

Računska geometrija

Luka Fürst

Računska geometrija

- Reševanje problemov, ki jih lahko opišemo z geometrijskimi objekti
- Široko področje uporabe
- Omejili se bomo na dve dimenziji

Izzivi

- Pogosto smo prisiljeni delati z realnimi števili
- Že izračun kota med dvema premicama zahteva uporabo transcendentnih funkcij ($\arcsin$ ali $\arccos$)

Izzivi

- Numerične zagate
- Sta točki, ki sta si »zelo« blizu, v resnici ena in ista točka?
- Sta »skoraj« vzporedni premici dejansko vzporedni?
- Test $a = b$ izvedemo kot $|a - b| < \varepsilon$

Izzivi

- Zoprni posebni primeri
- Kolinearne točke pri računanju konveksne ovojnice
- Presečišča treh ali več daljic pri iskanju presečišč med daljicami

Točka

- Ponavadi jo predstavimo kot par (x, y)
  - x: vodoravna razdalja od koordinatnega izhodišča
  - y: navpična razdalja od koordinatnega izhodišča
- Alternativa: polarne koordinate
  - r: razdalja od koordinatnega izhodišča $O(0, 0)$
  - φ: kot, ki ga oklepata vektor $(1, 0)$ in krajevni vektor točke

Vektor

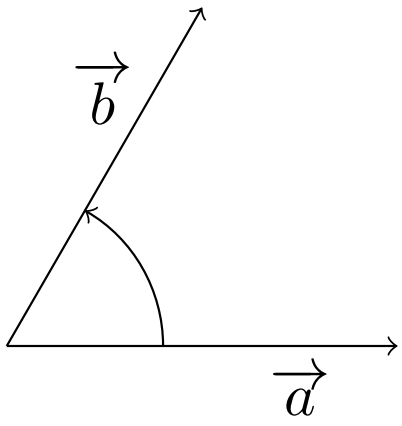
- Ekvivalenčni razred usmerjenih daljic
- Vektor (x, y) predstavlja vse daljice $\overrightarrow{AB}$ z lastnostjo $x = x_B - x_A$ in $y = y_B - y_A$
- $\overrightarrow{OA} = (x, y)$ je krajevni vektor točke $A(x, y)$

Vektorski produkt

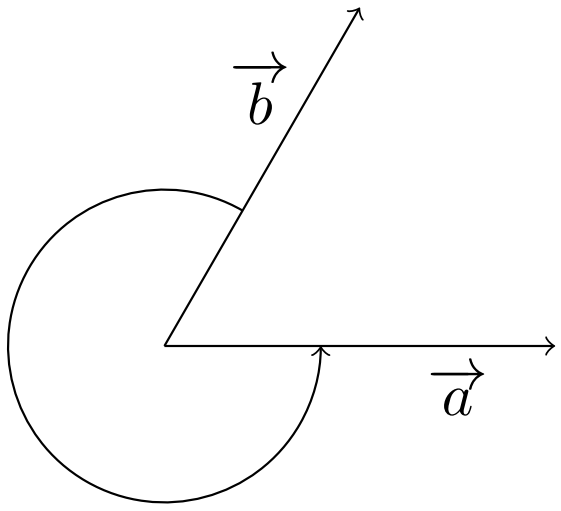
- Definiran v treh dimenzijah
- $\overrightarrow{a} \times \overrightarrow{b}$ je vektor, pravokoten na ravnino, ki jo določata vektorja $\overrightarrow{a}$ in $\overrightarrow{b}$
- $|\overrightarrow{a} \times \overrightarrow{b}| = |\overrightarrow{a}| \cdot |\overrightarrow{b}| \cdot |\sin \angle(\overrightarrow{a}, \overrightarrow{b})|$
- Smer vektorja $\overrightarrow{a} \times \overrightarrow{b}$ je določena po pravilu desne roke
  - Kam kaže palec, če s preostalimi prsti »objamemo« najkrajšo pot od $\overrightarrow{a}$ do $\overrightarrow{b}$?

Merjenje kotov

- Kot med vektorjema merimo v nasprotni smeri urinega kazalca
- $\angle(\overrightarrow{b}, \overrightarrow{a}) = 360° - \angle(\overrightarrow{a}, \overrightarrow{b})$



$\angle(\overrightarrow{a}, \overrightarrow{b}) = 60°$



$\angle(\overrightarrow{b}, \overrightarrow{a}) = 300°$

Vektorski produkt

- Če se omejimo na koordinatno ravnino, je vektorski produkt vzporeden z osjo z
- Za nas bo »vektorski produkt« zgolj koordinata z vektorskega produkta, kot je definiran v matematiki

$$(x_1, y_1) \times (x_2, y_2) = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1 y_2 - x_2 y_1$$

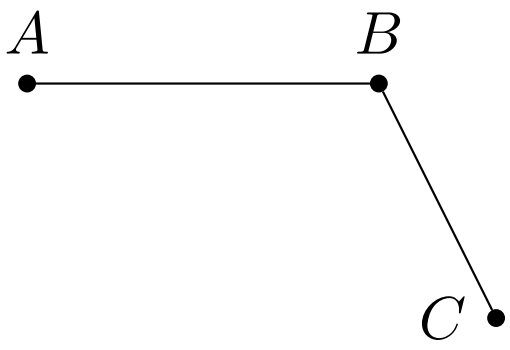
Levi in desni zasuk

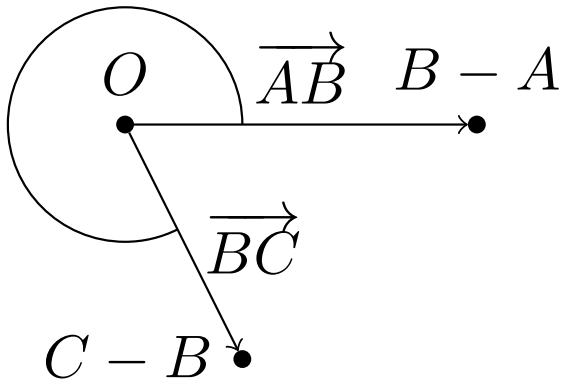
- Točke A, B in C tvorijo levi zasuk natanko tedaj, ko je $\overrightarrow{AB} \times \overrightarrow{BC} > 0$

- Če se »peljemo« od A do B, zavijemo levo, da gremo proti C

Levi in desni zasuk

- Točke A, B in C tvorijo desni zasuk natanko tedaj, ko je $\overrightarrow{AB} \times \overrightarrow{BC} < 0$

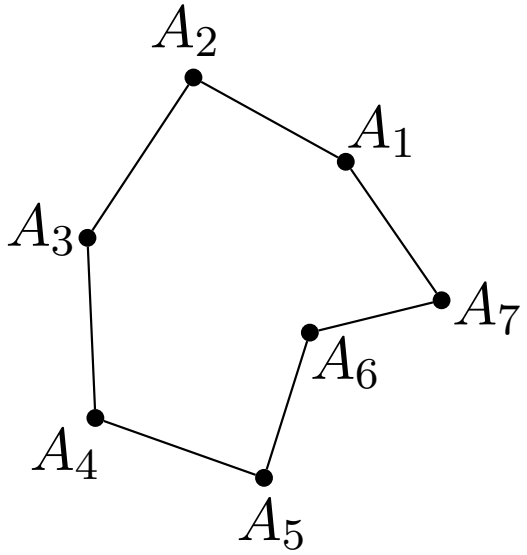




- Če se »peljemo« od A do B, zavijemo desno, da gremo proti C

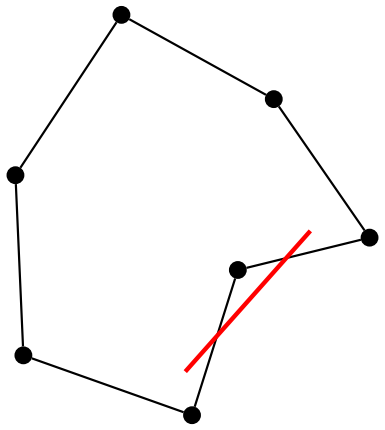
Večkotnik

- Podamo ga z zaporedjem oglišč A_1, A_2, ..., A_n v nasprotni smeri urinega kazalca
- V izogib sitnostim z računanjem po modulu bomo predpostavili $A_{n+1} = A_1$, $A_{n+2} = A_2$ itd.
- Omejili se bomo na enostavne večkotnike
  - se ne sekajo, nimajo lukenj



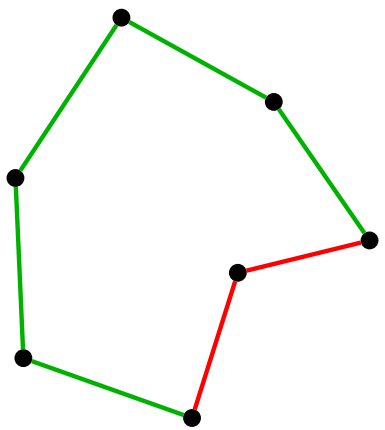
Konveksen in konkaven večkotnik

- Večkotnik je konveksen natanko tedaj, ko za vsak par točk A in B v njegovi notranjosti oz. na robu velja, da se celotna daljica AB nahaja v njegovi notranjosti oz. na robu
- Sicer je konkaven
- Sledeči večkotnik je konkaven:



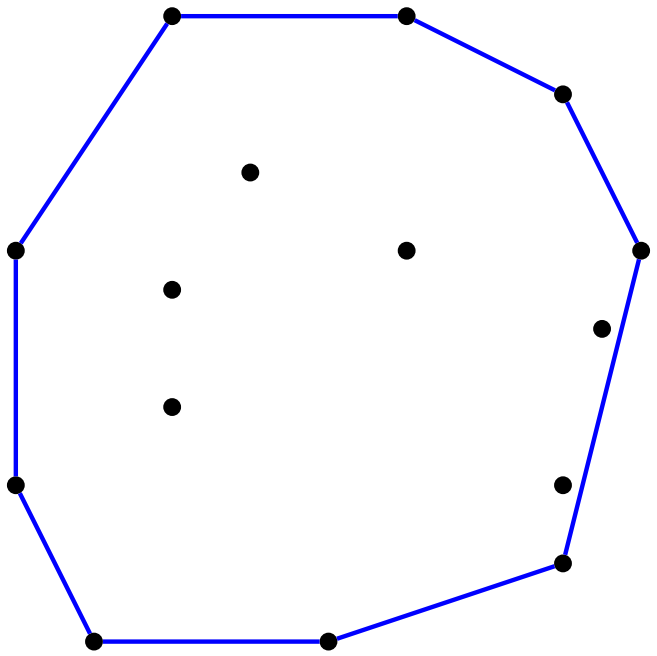
Konveksen in konkaven večkotnik

- Uporabnejša definicija: večkotnik z oglišči A_1, A_2, $\ldots$, A_n v nasprotni smeri urinega kazalca je konveksen natanko tedaj, ko za vsak $i \in \{1, \ldots, n\}$ oglišča A_i, A_{i+1} in A_{i+2} tvorijo levi zasuk



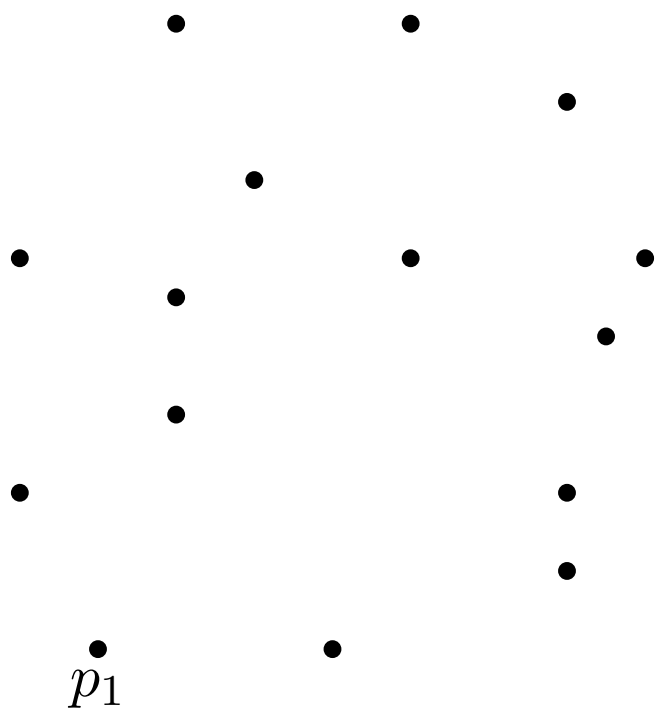
Konveksna ovojnica

- Najmanjši konveksni večkotnik, ki v celoti vsebuje podano množico $n \geq 3$ točk
- Večkotnik vsebuje točko, če se ta nahaja v njegovi notranjosti ali na robu
- Predpostavka: obstajajo vsaj 3 točke, ki ne ležijo na isti premici



Grahamov algoritem

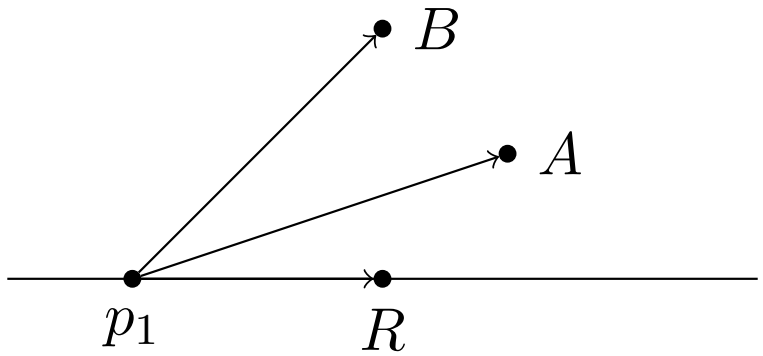
- Pričnemo s točko z najmanjšo koordinato y
- Če je takih točk več, vzamemo najbolj levo med njimi
- To naj bo točka $p_1 = (x_1, y_1)$



Grahamov algoritem

- Preostale točke uredimo po naraščajočem kotu glede na premico $y = y_1$
- Kotov nam ni treba neposredno računati
- Zadošča, da med seboj primerjamo vektorske produkte

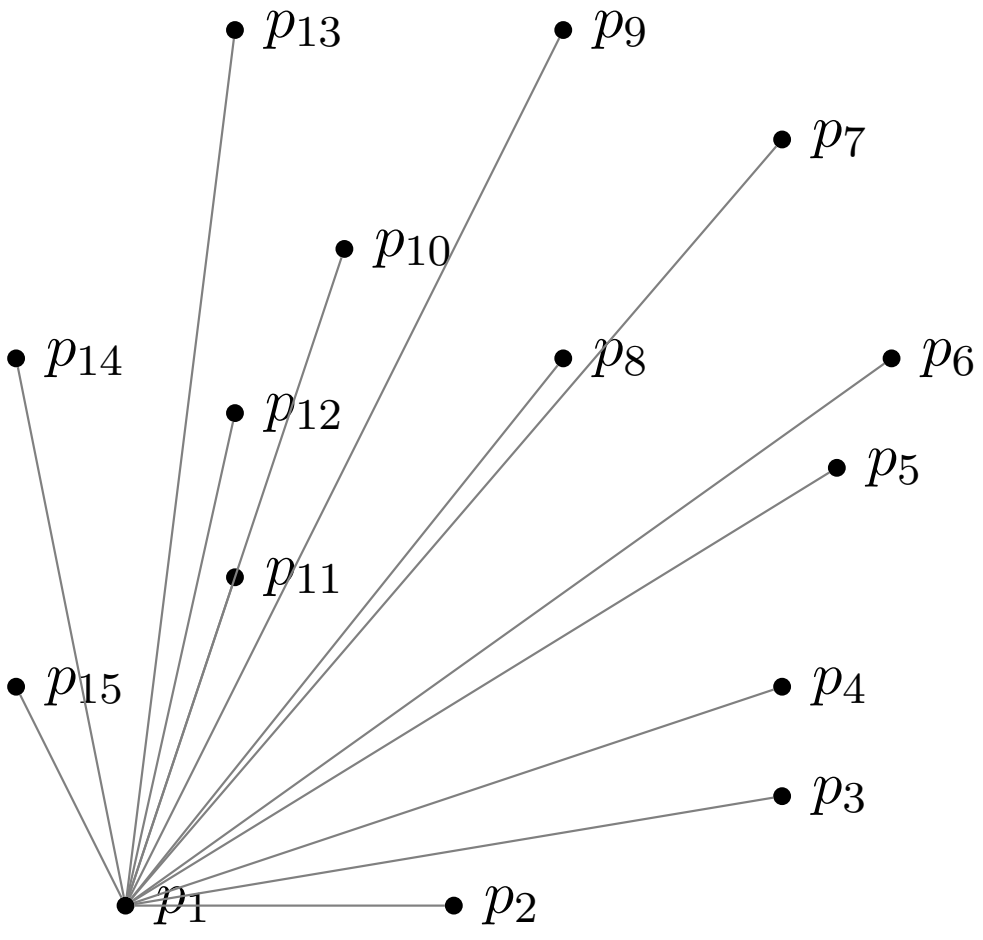
$$\overrightarrow{p_1A} \times \overrightarrow{p_1B} > 0 \iff \angle(\overrightarrow{p_1R}, \overrightarrow{p_1A}) < \angle(\overrightarrow{p_1R}, \overrightarrow{p_1B})$$



- Ker smo točko p_1 izbrali kot skrajno levo med najnižje ležečimi točkami, za vsako točko T od preostalih vhodnih točk velja $\angle(\overrightarrow{p_1R}, \overrightarrow{p_1T}) > 0$

Grahamov algoritem

- Dobimo vrstni red točk $p_1, p_2, \ldots, p_n$



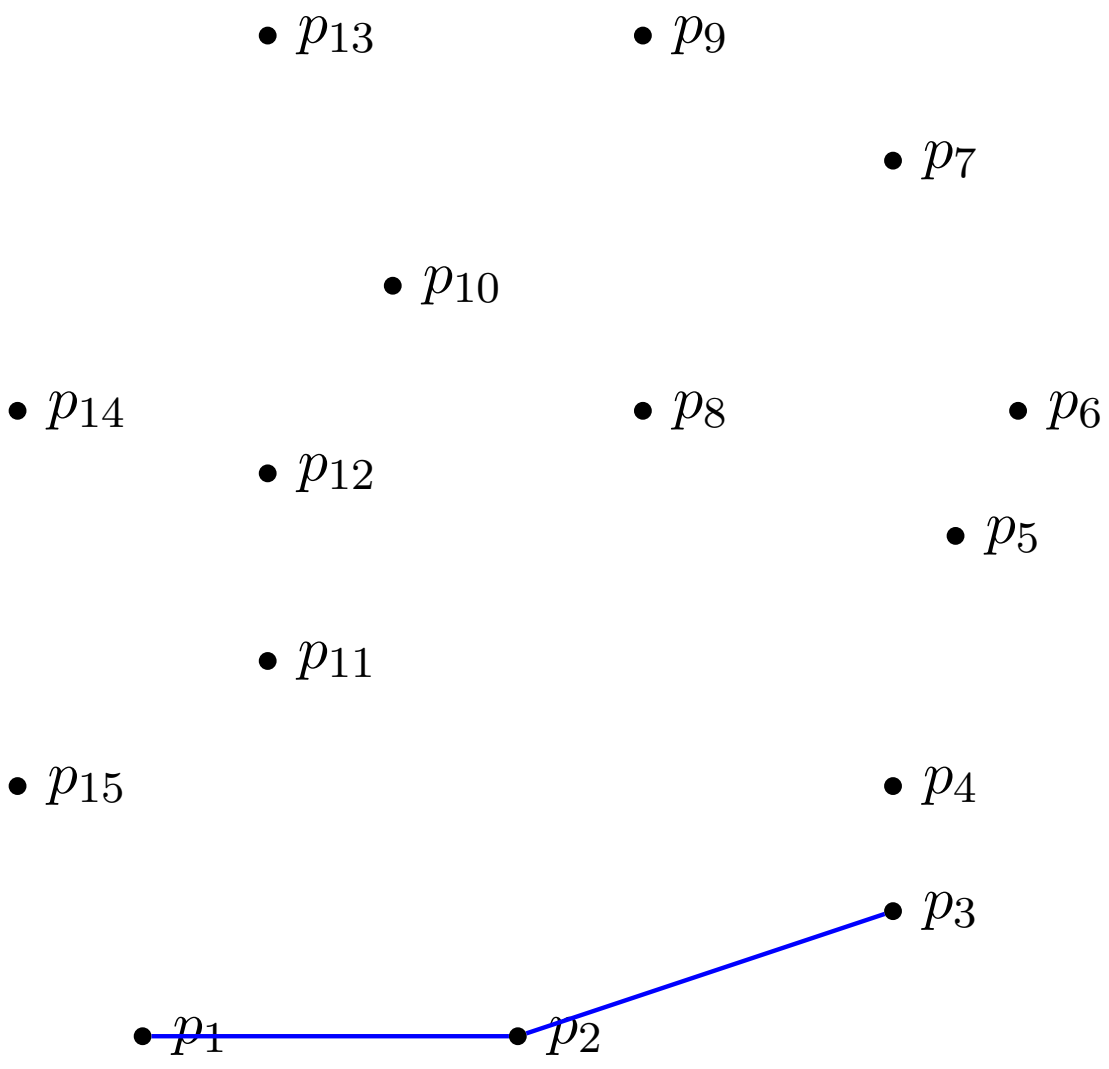
Grahamov algoritem

- Grahamov algoritem iterativno vzdržuje sklad
- Po obravnavi točke p_i $(i \geq 3)$ so na skladu točke, ki tvorijo konveksno ovojnico točk $p_1, p_2, \ldots, p_i$
- Na dnu sklada je točka p_1, na vrhu pa točka p_i
- Točke so od dna do vrha razvrščene v nasprotni smeri urinega kazalca

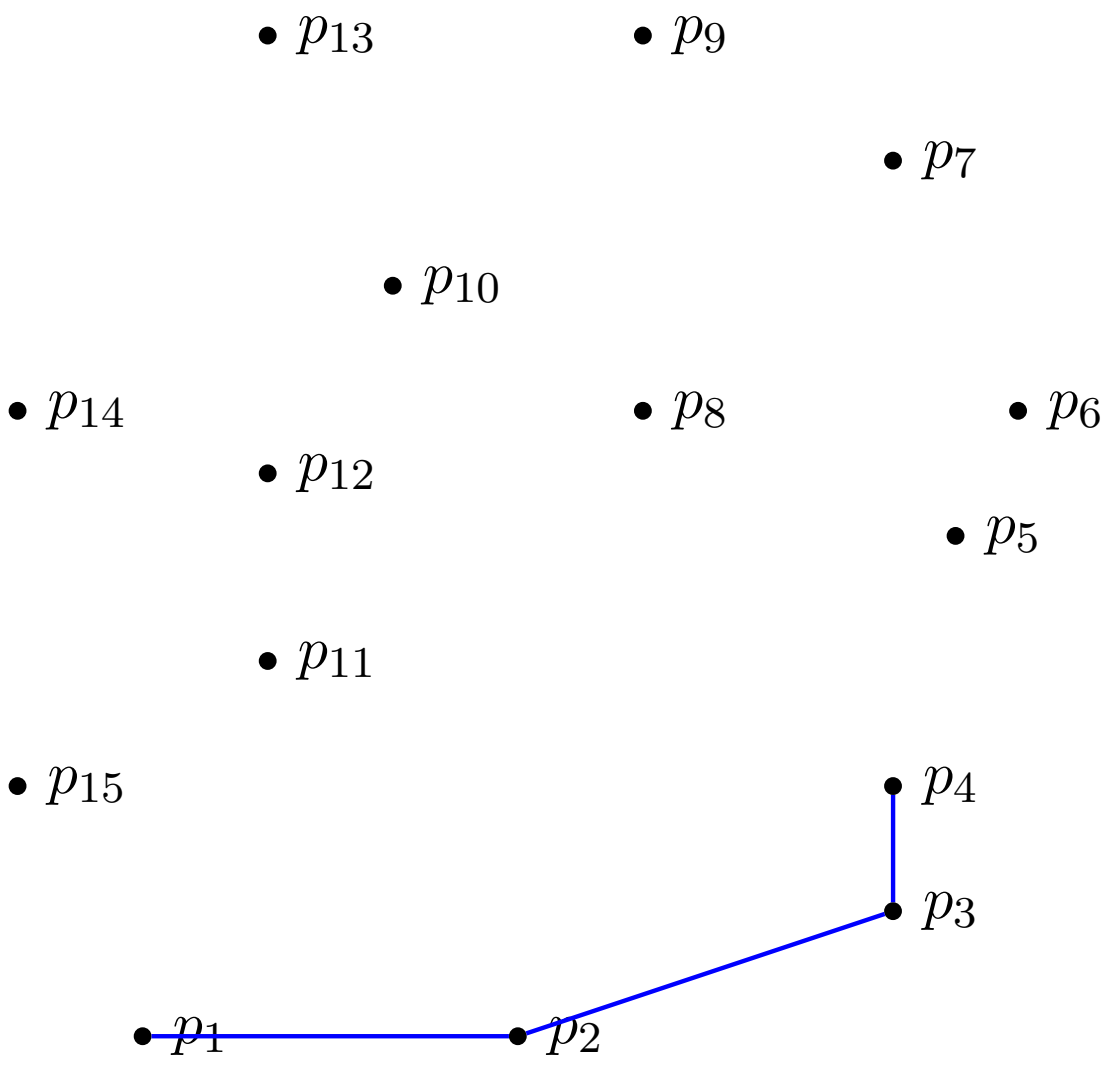
Grahamov algoritem

function GRAHAM(T)
    Pridobi vrstni red točk p_1, p_2, ..., p_n
    Inicializiraj sklad S
    DODAJ(S, p_1)
    DODAJ(S, p_2)
    DODAJ(S, p_3)
    for $i \leftarrow 4$ **to** n **do**
        while $S.podvrh$, $S.vrh$ in p_i ne tvorijo levega zasuka **do**
            ODVZEMI(S)
        DODAJ(S, p_i)
    return S

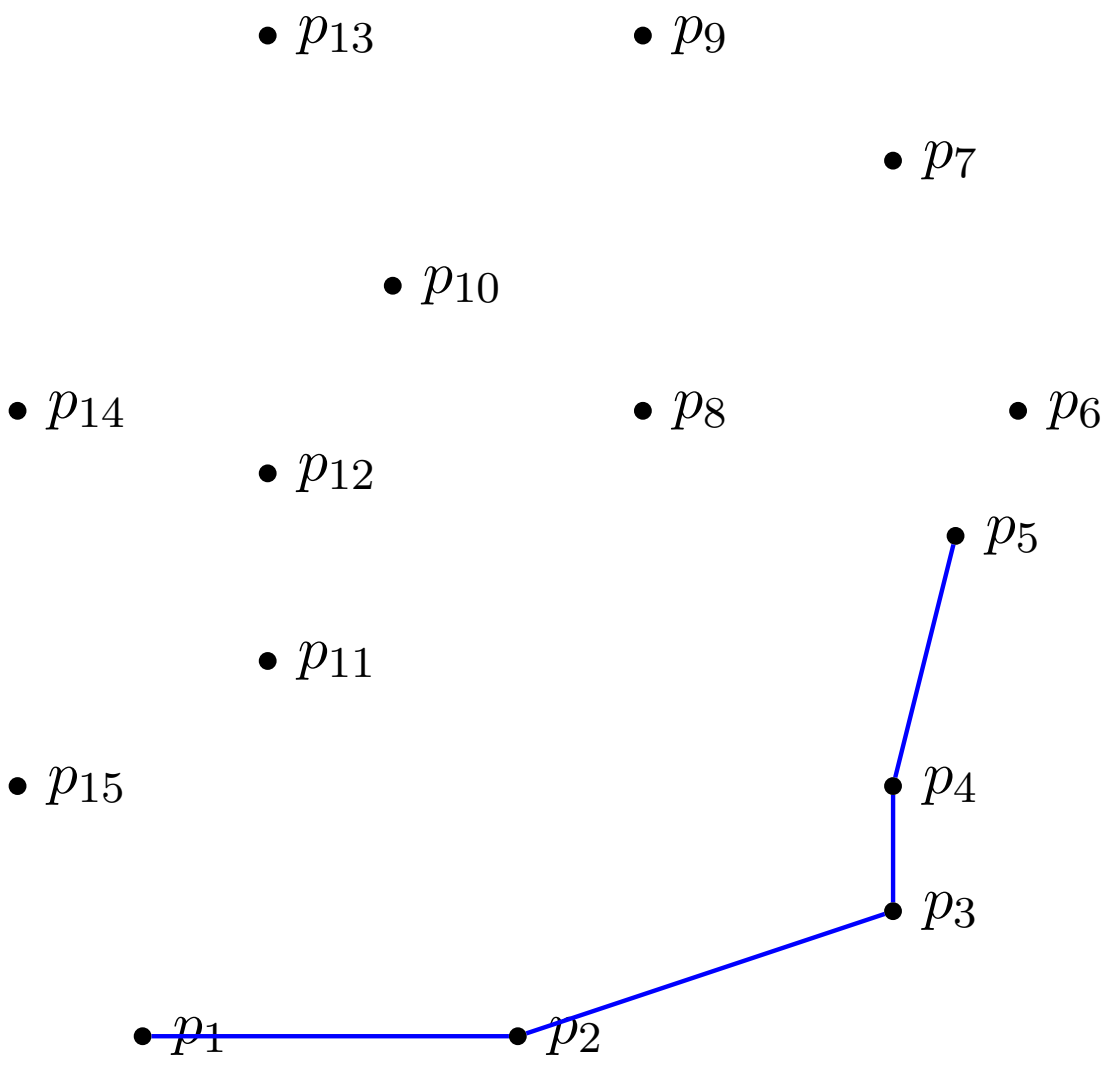
Primer



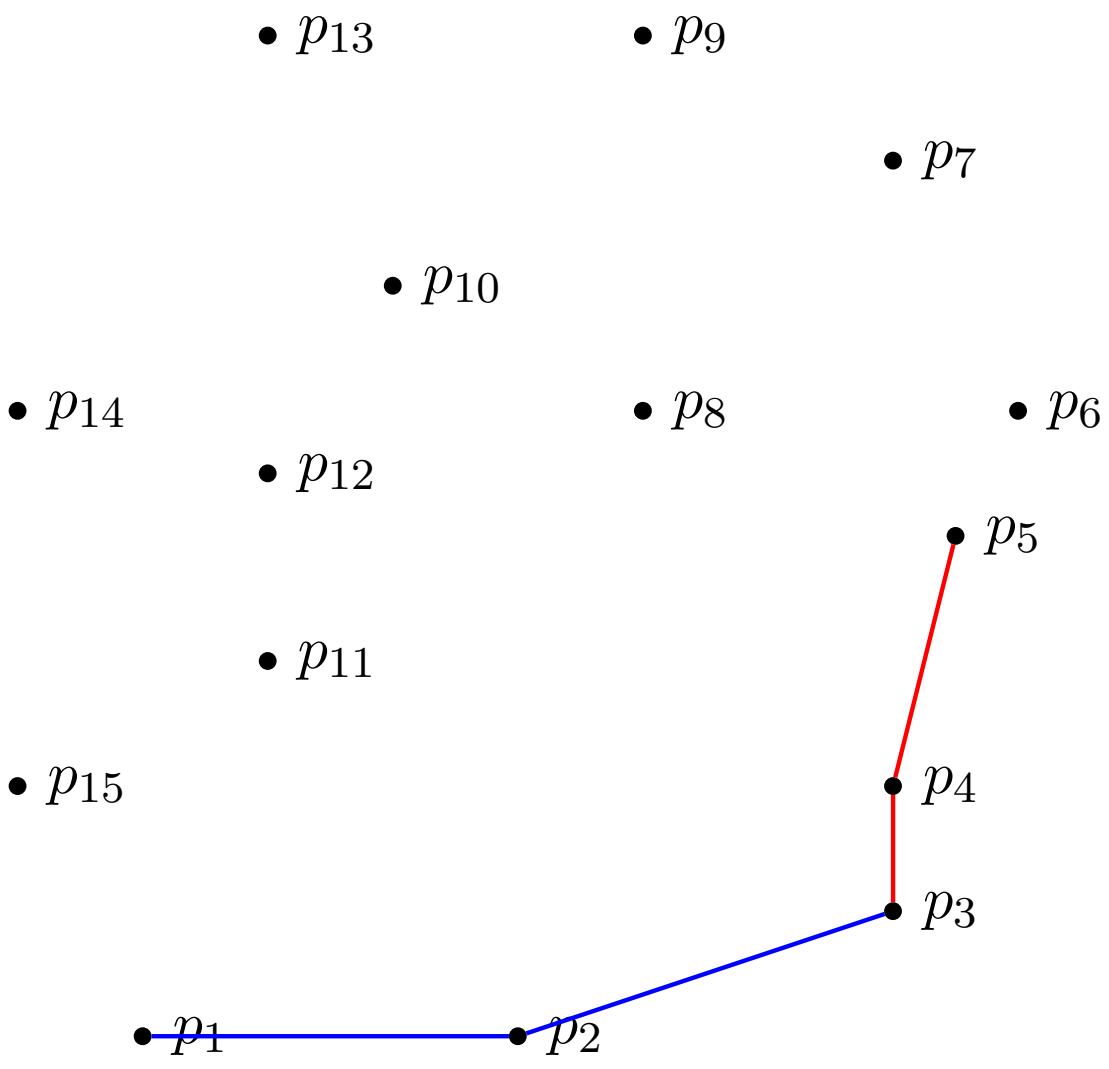
Primer



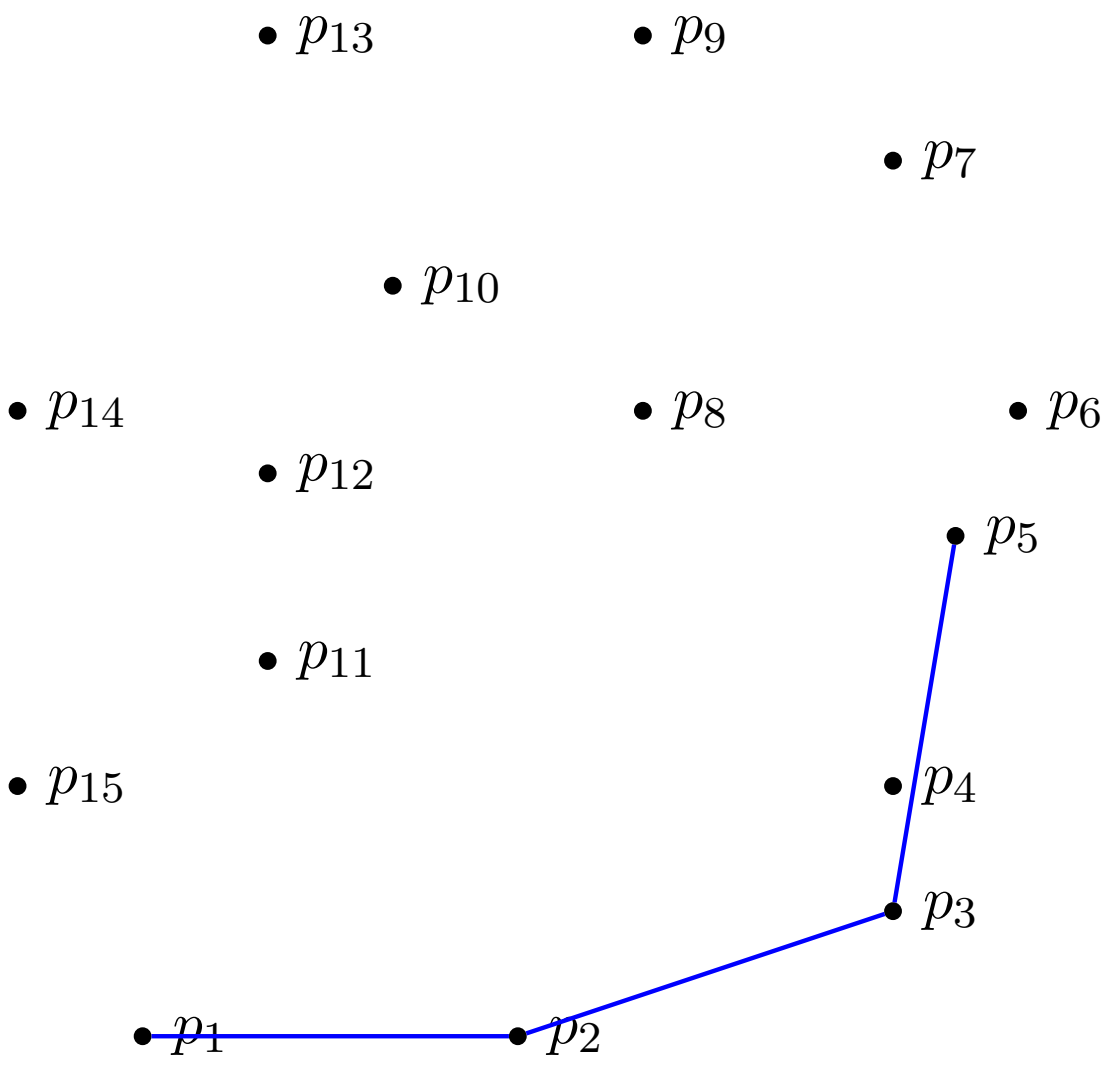
Primer



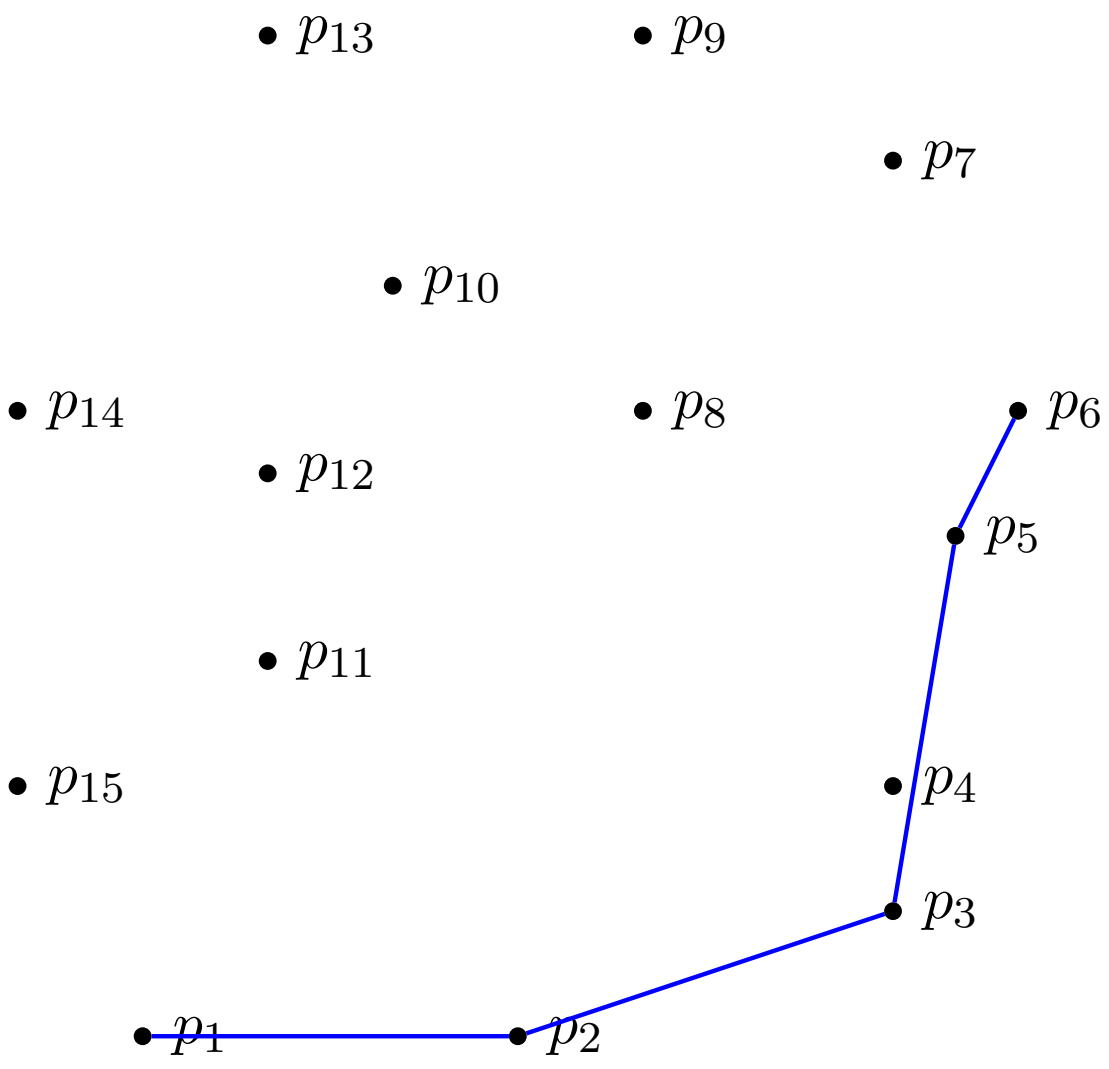
Primer



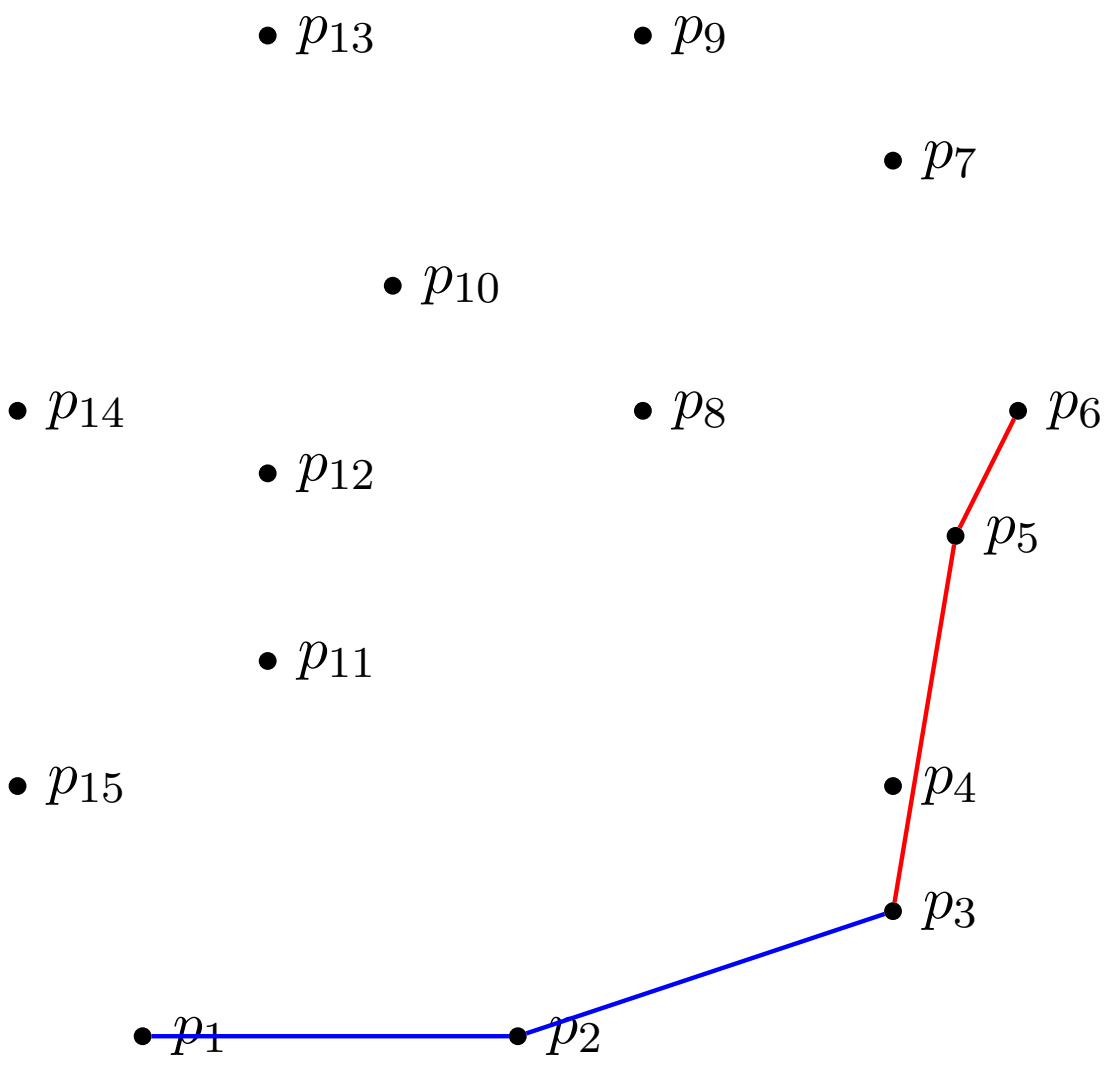
Primer



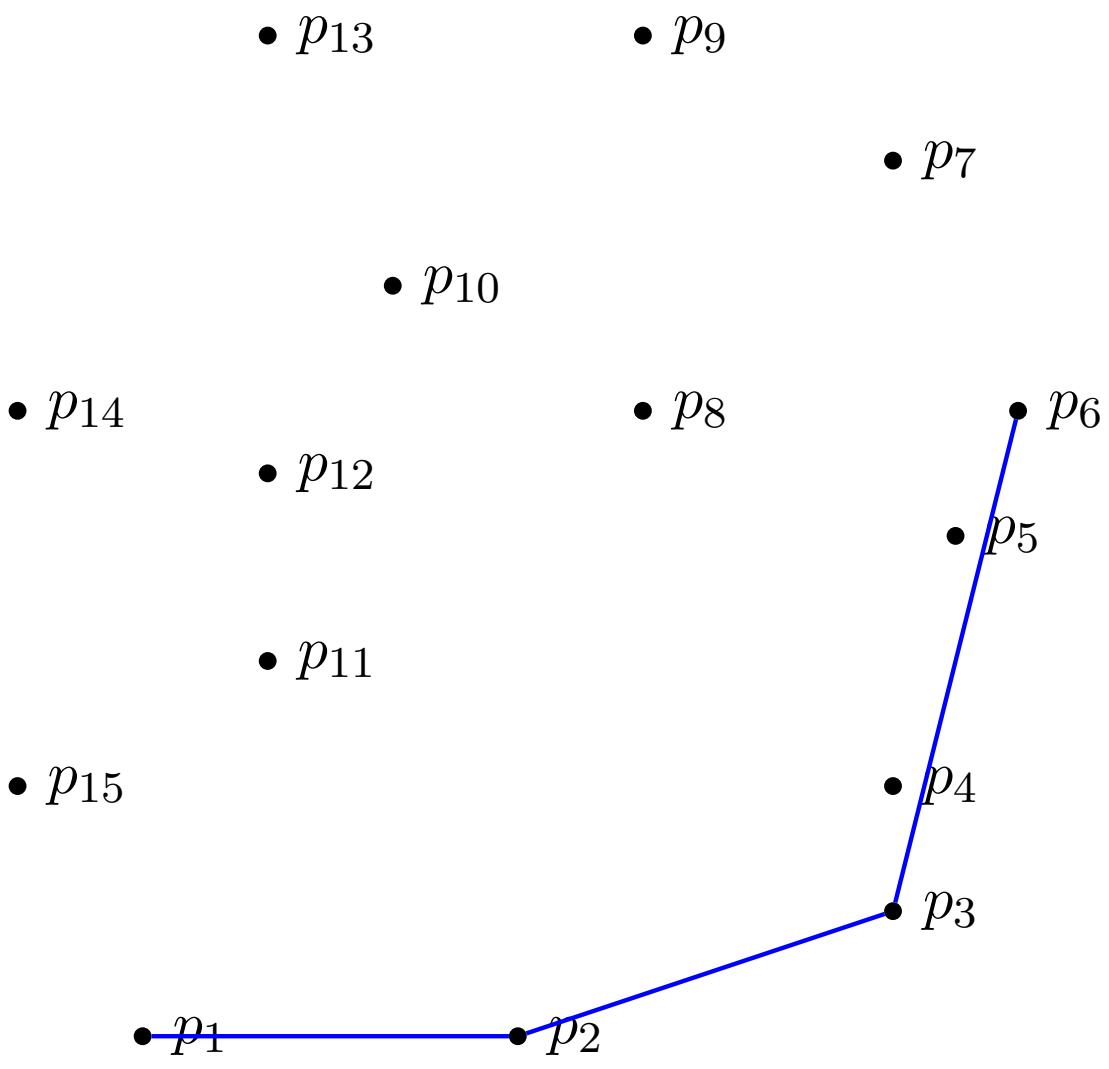
Primer



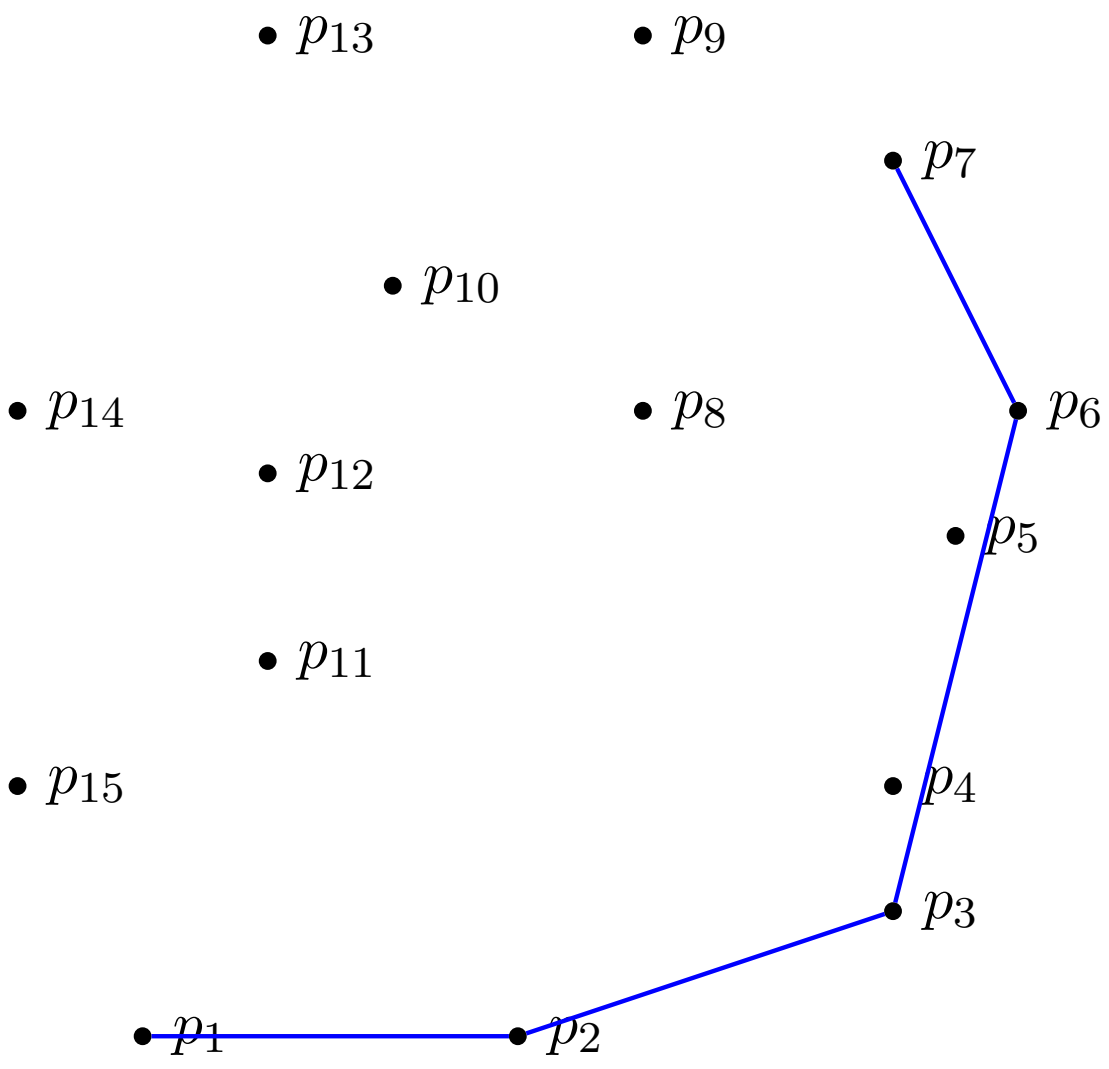
Primer



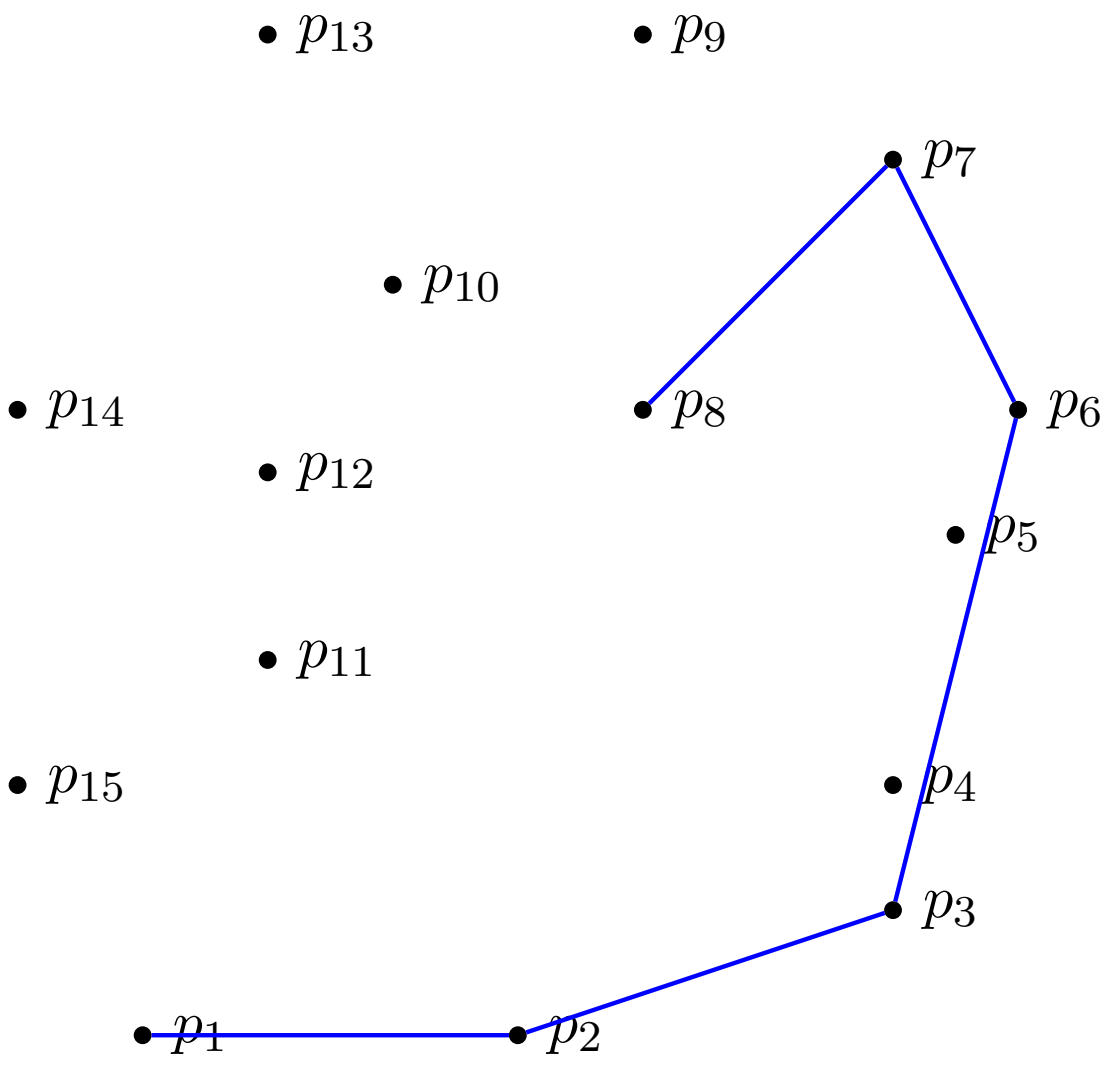
Primer



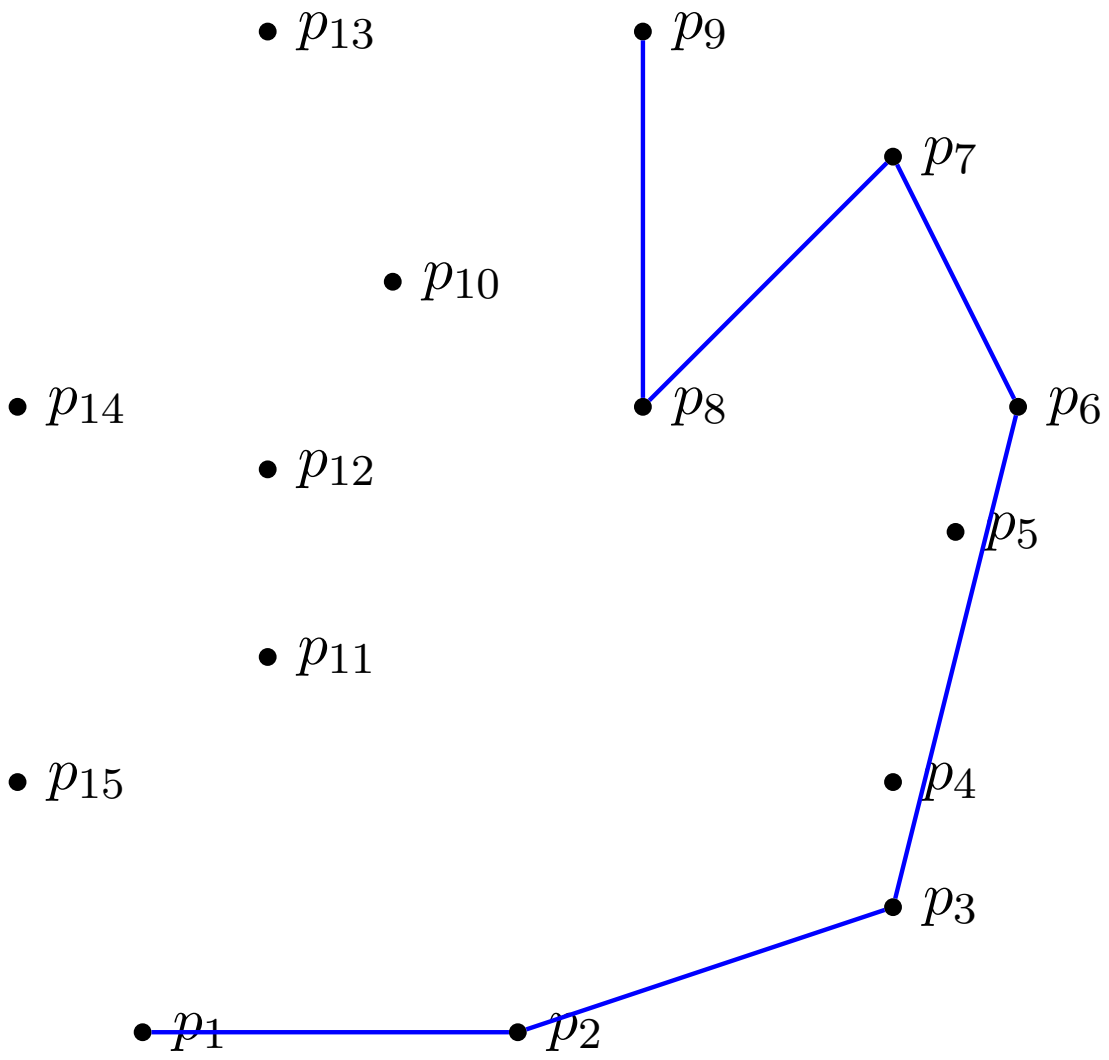
Primer



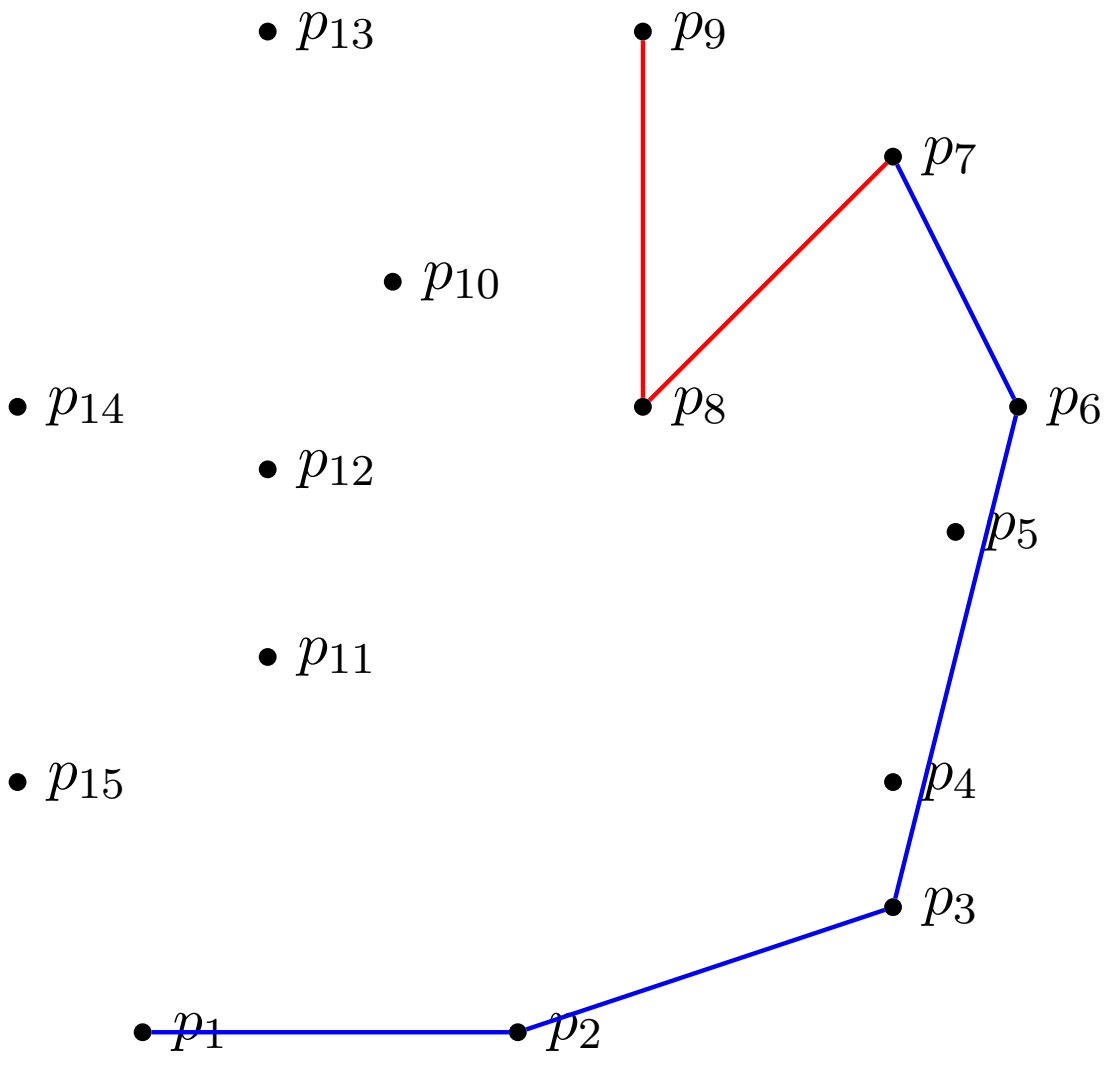
Primer



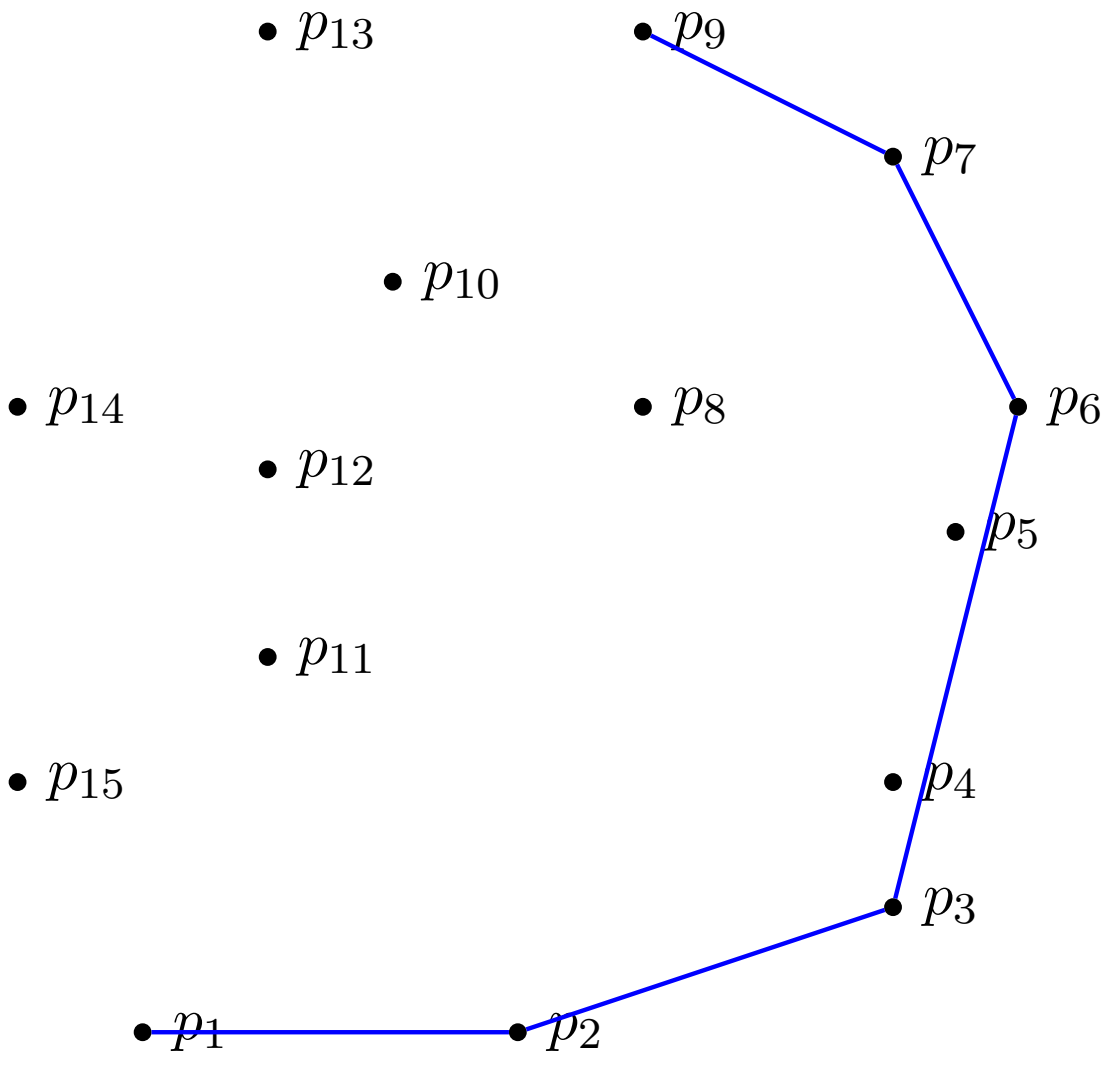
Primer



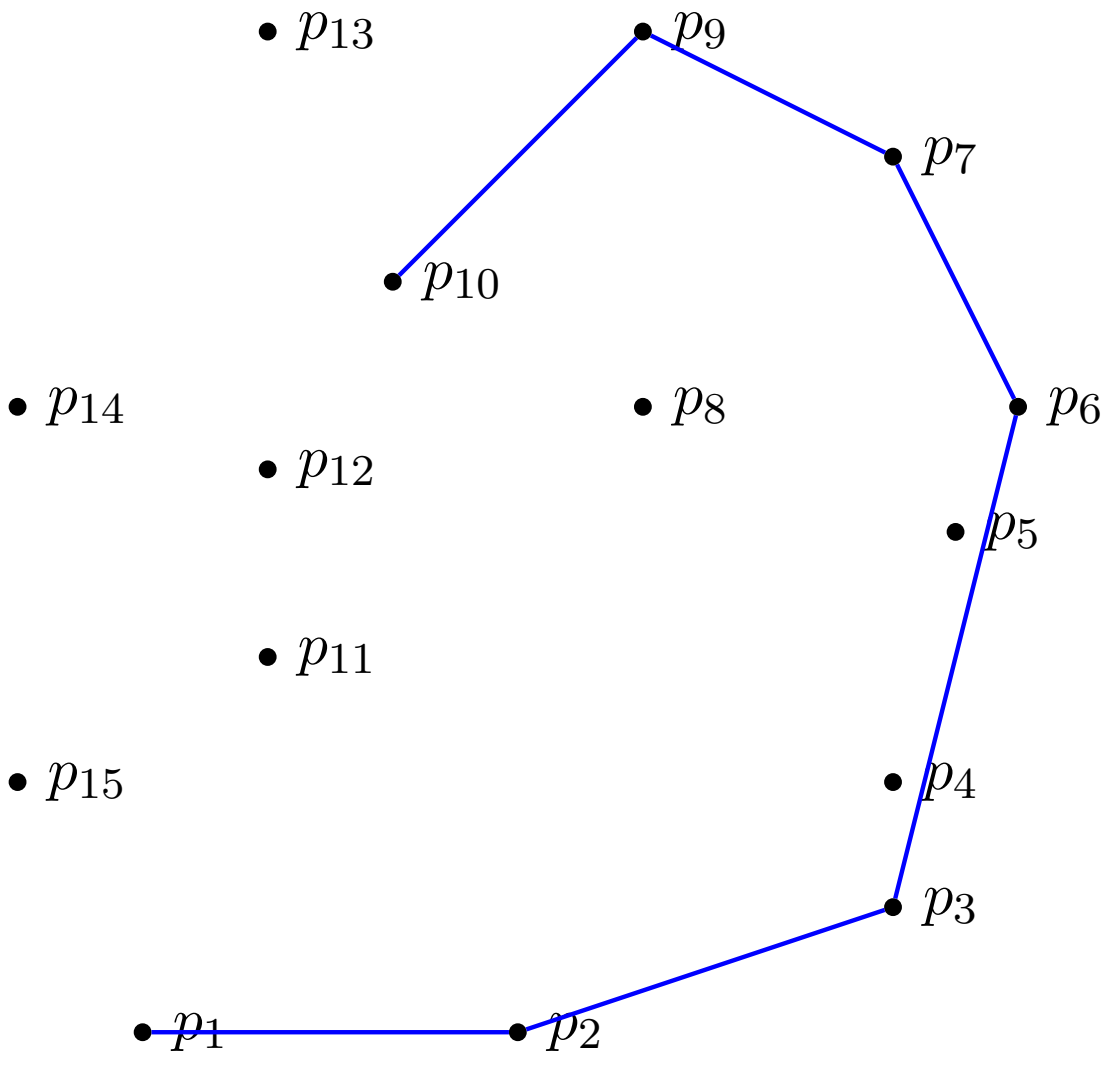
Primer



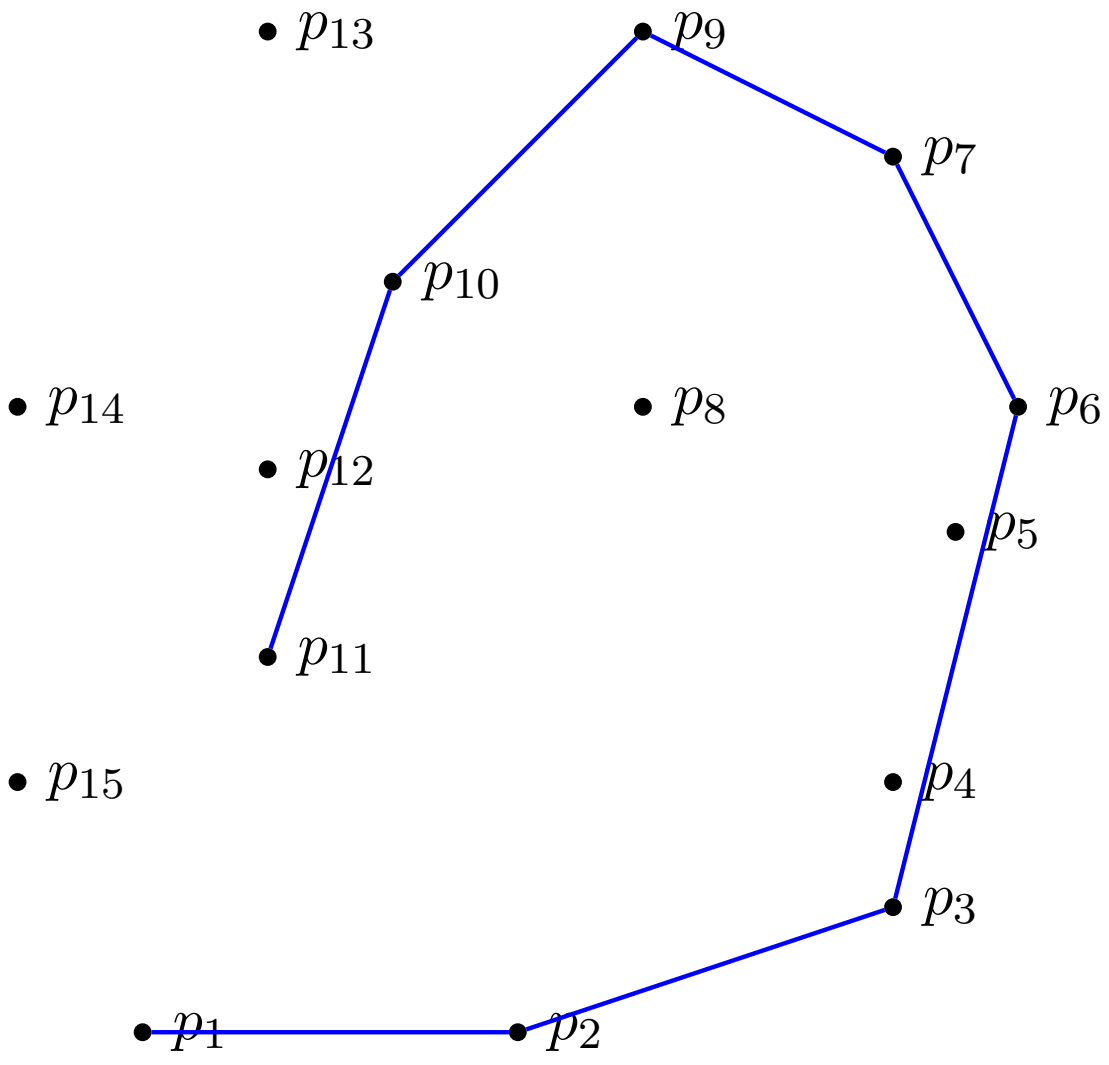
Primer



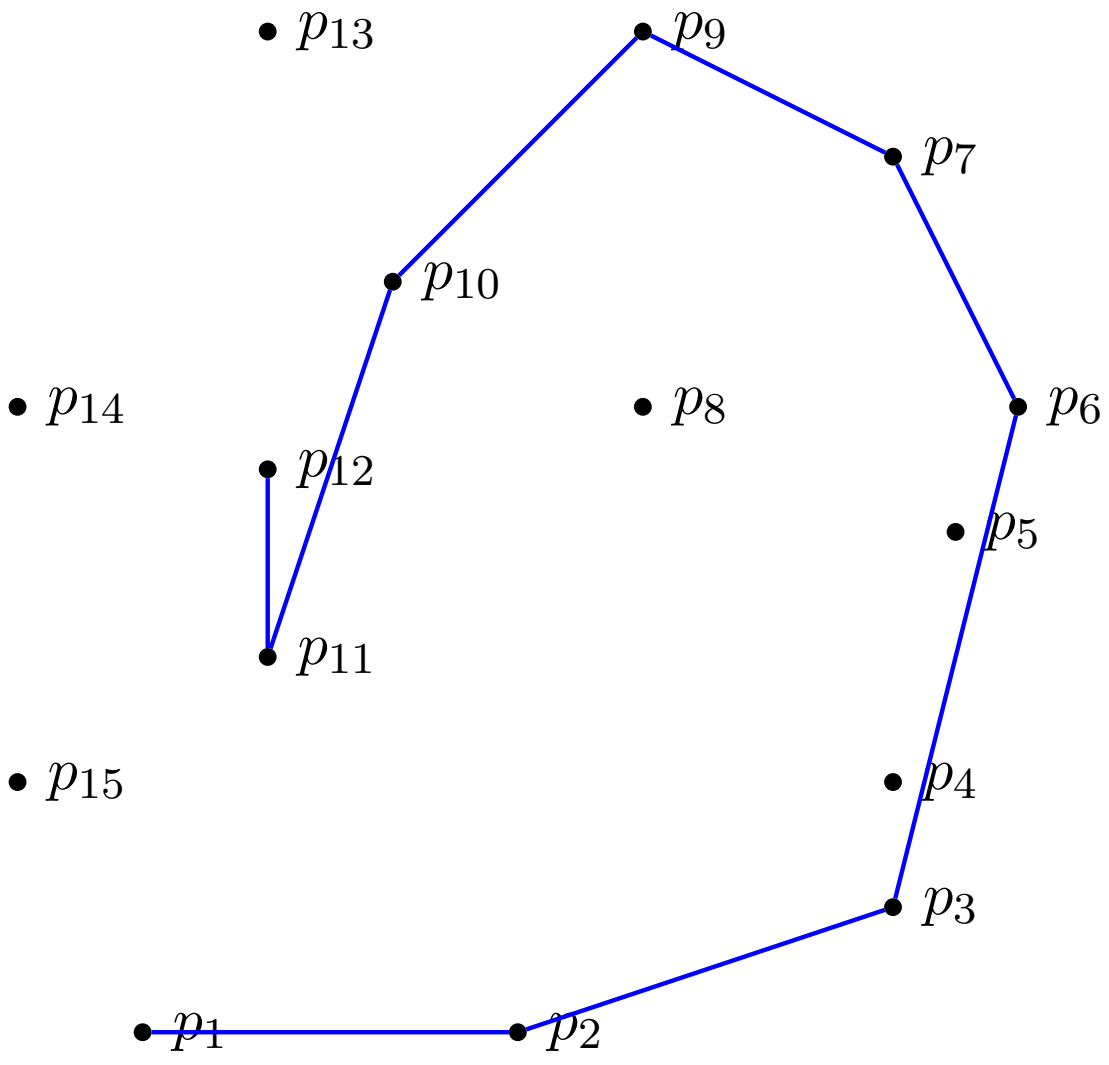
Primer



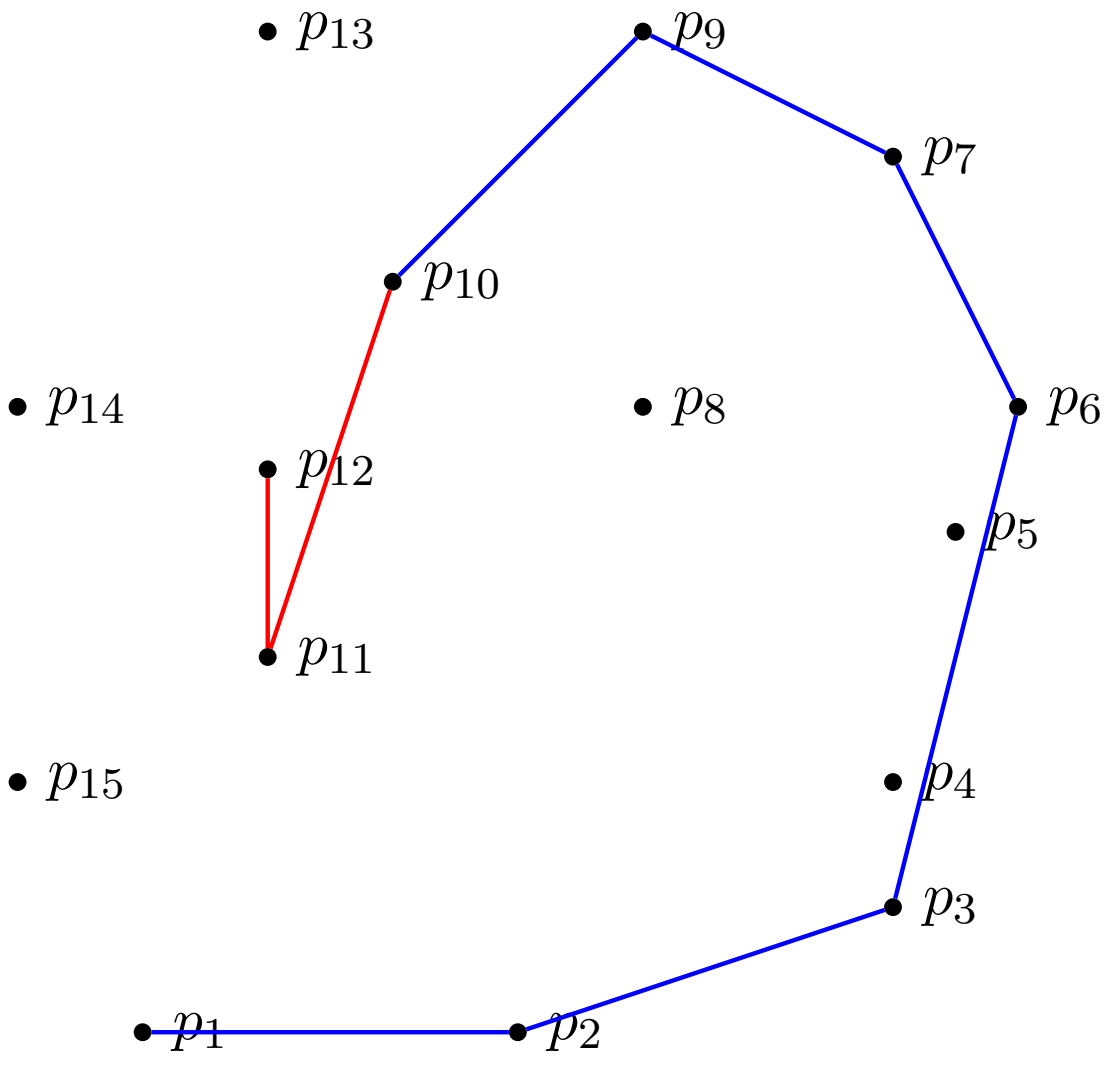
Primer



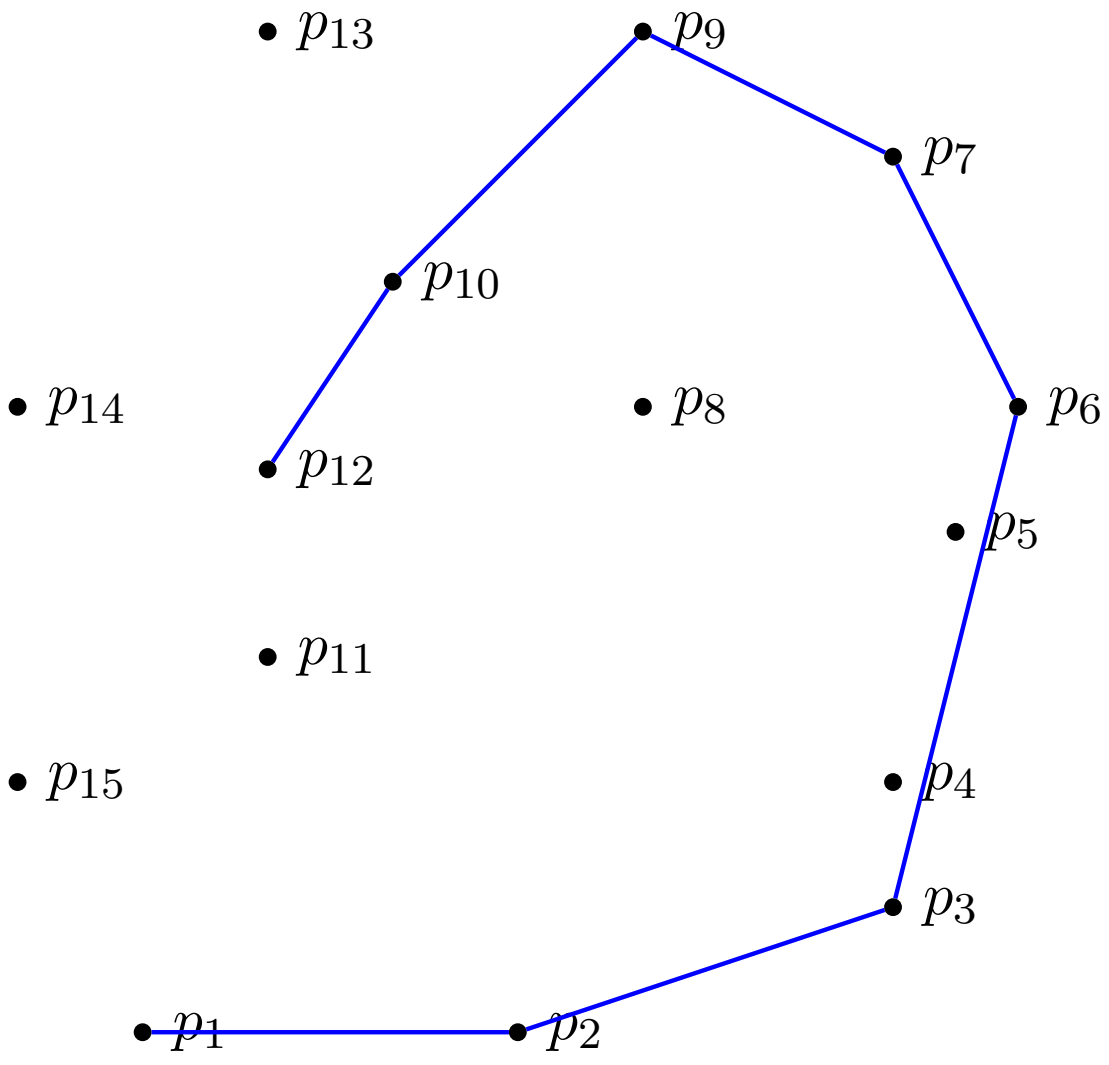
Primer



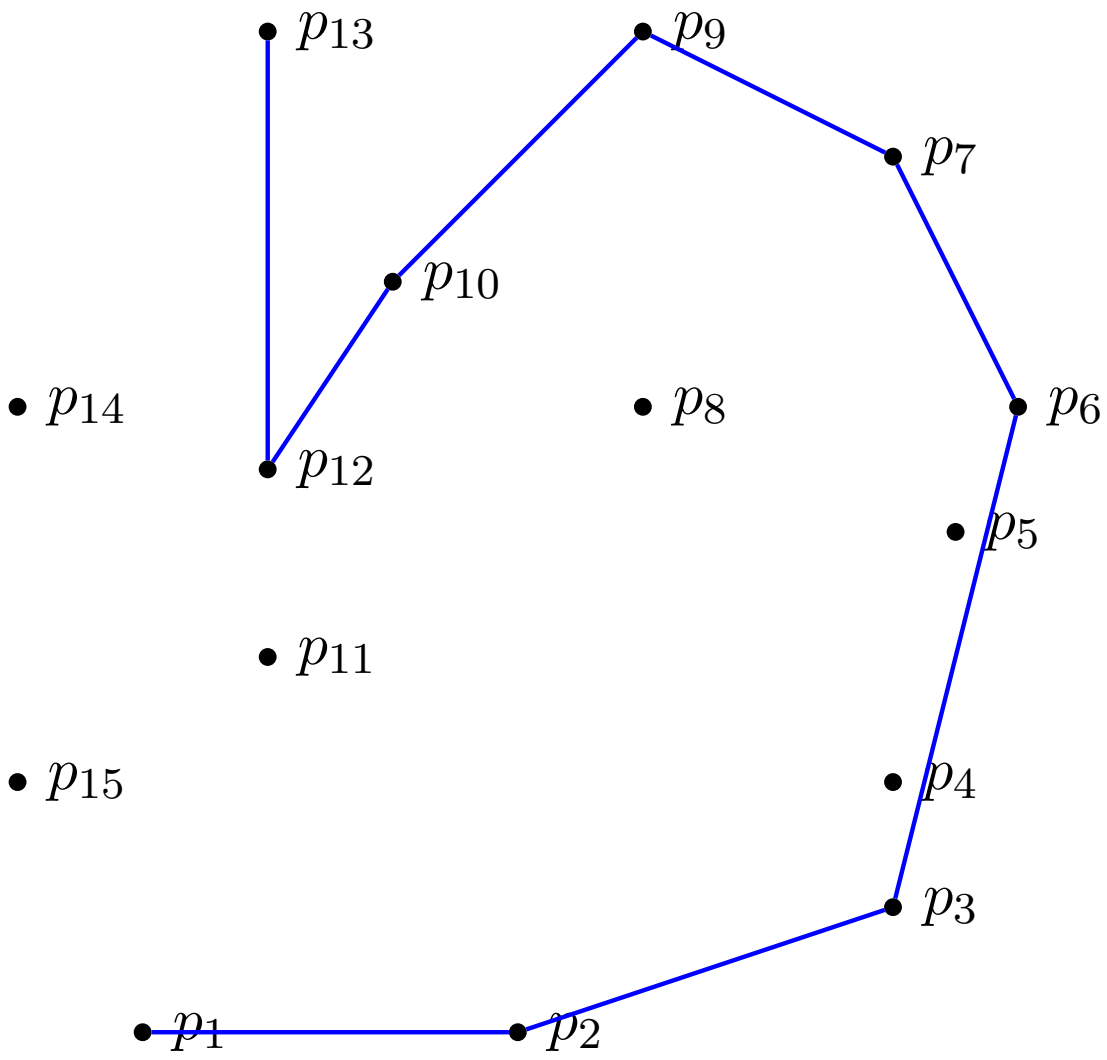
Primer



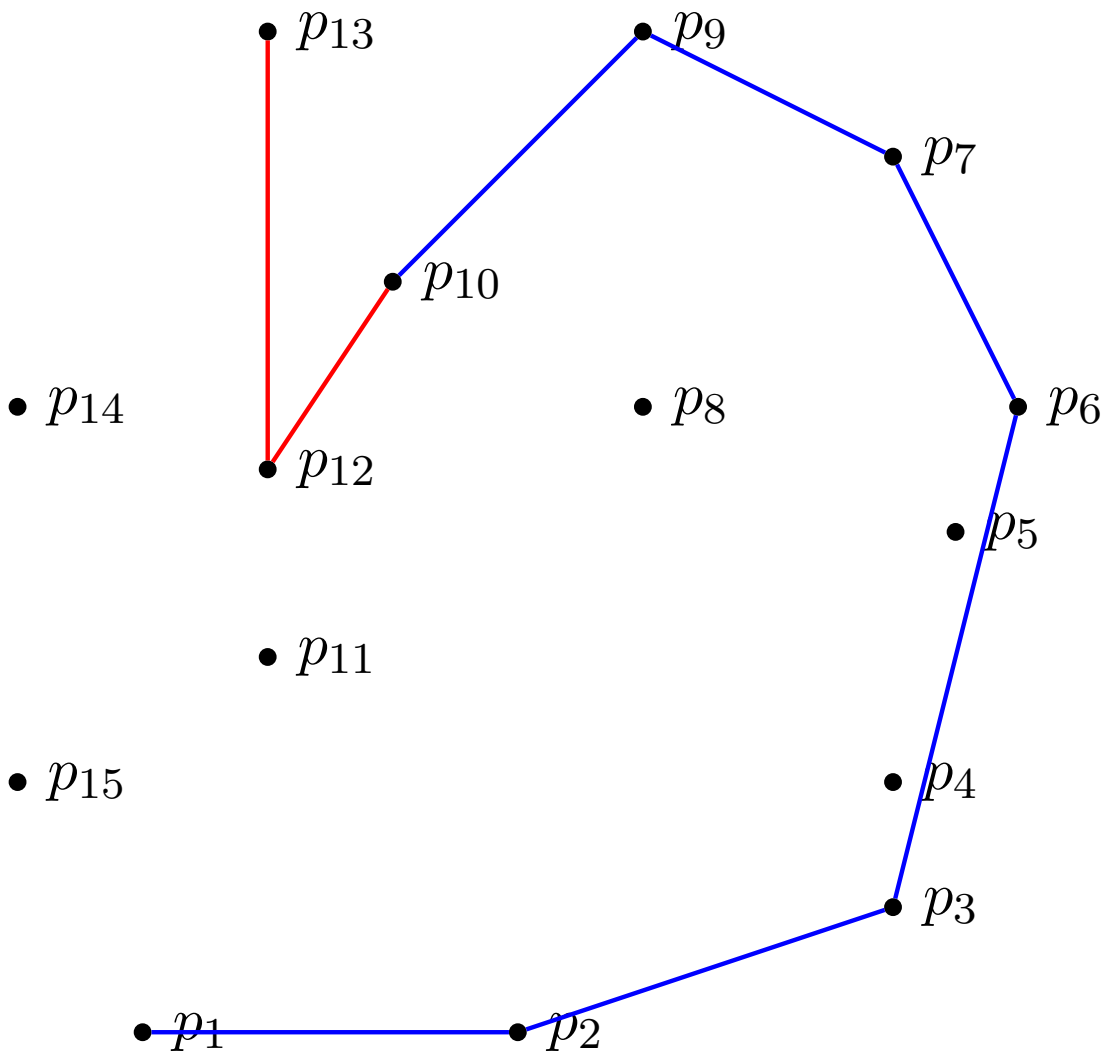
Primer



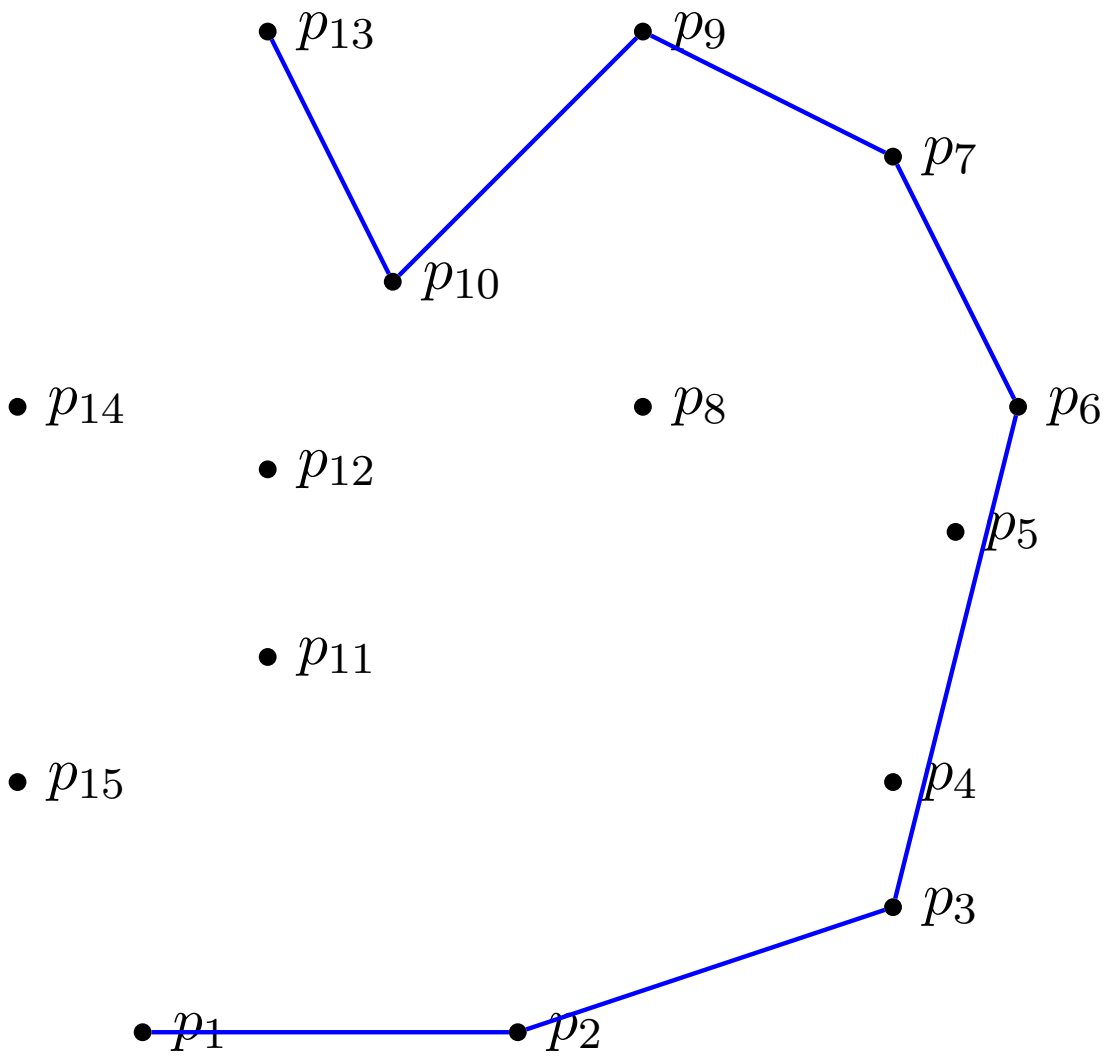
Primer



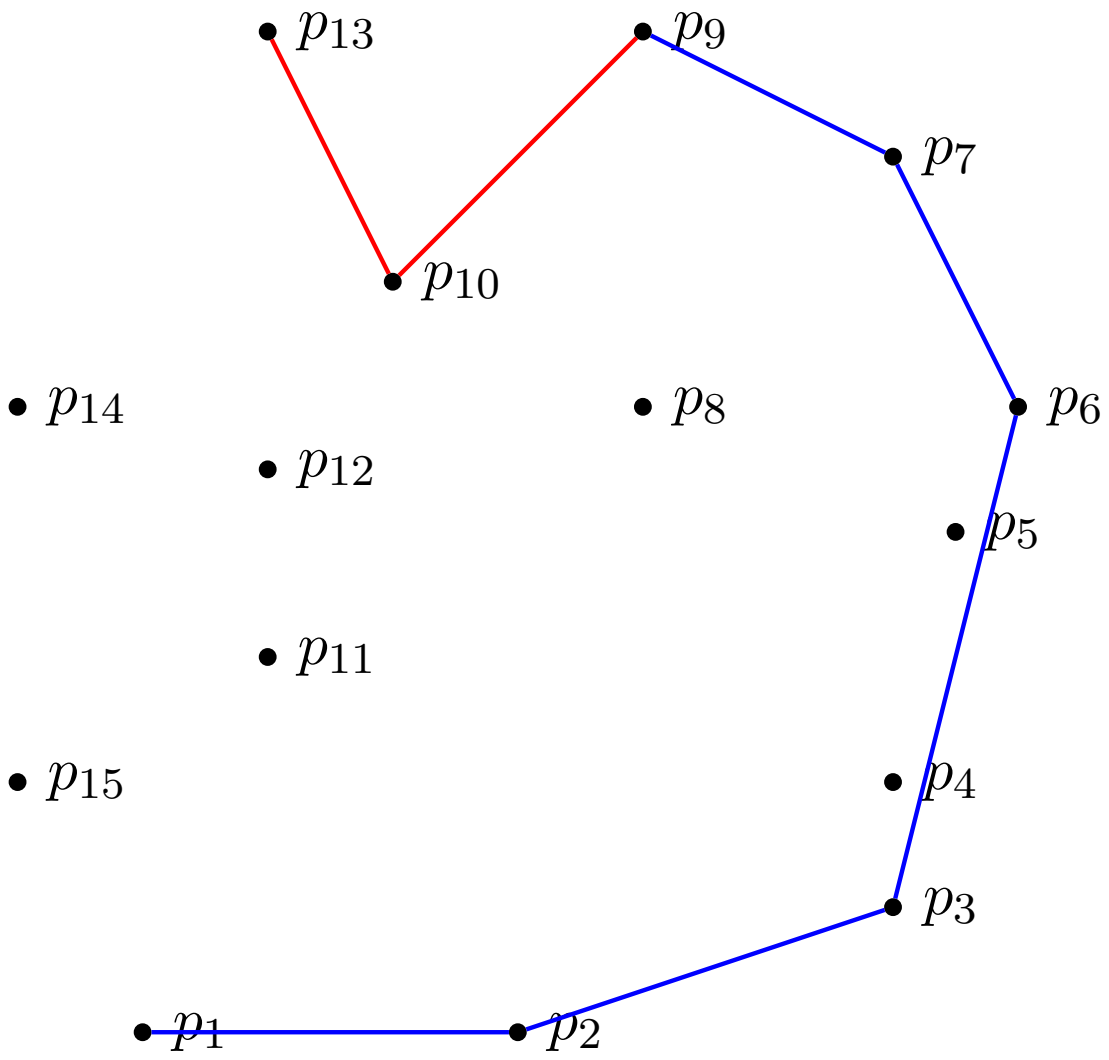
Primer



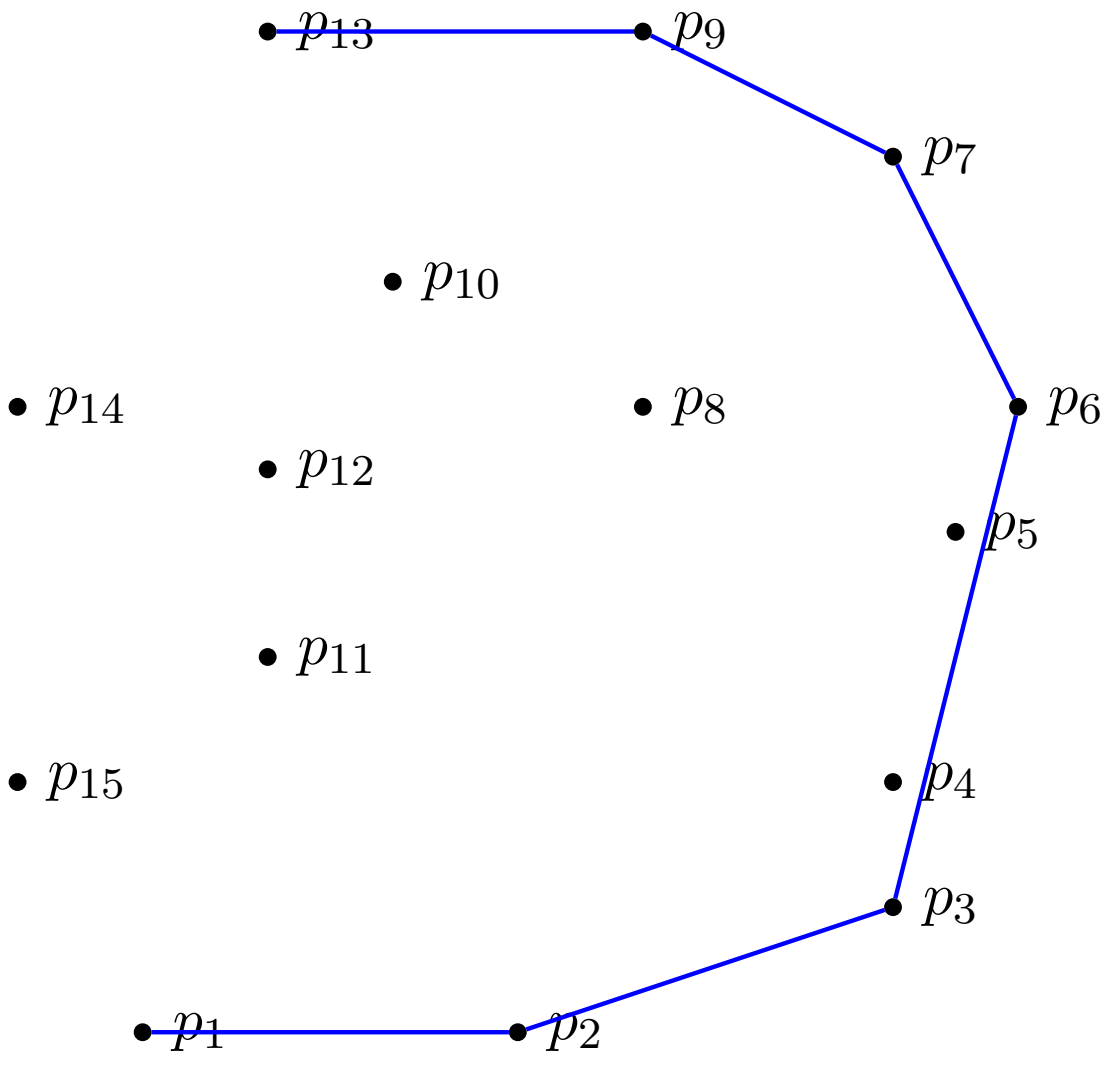
Primer



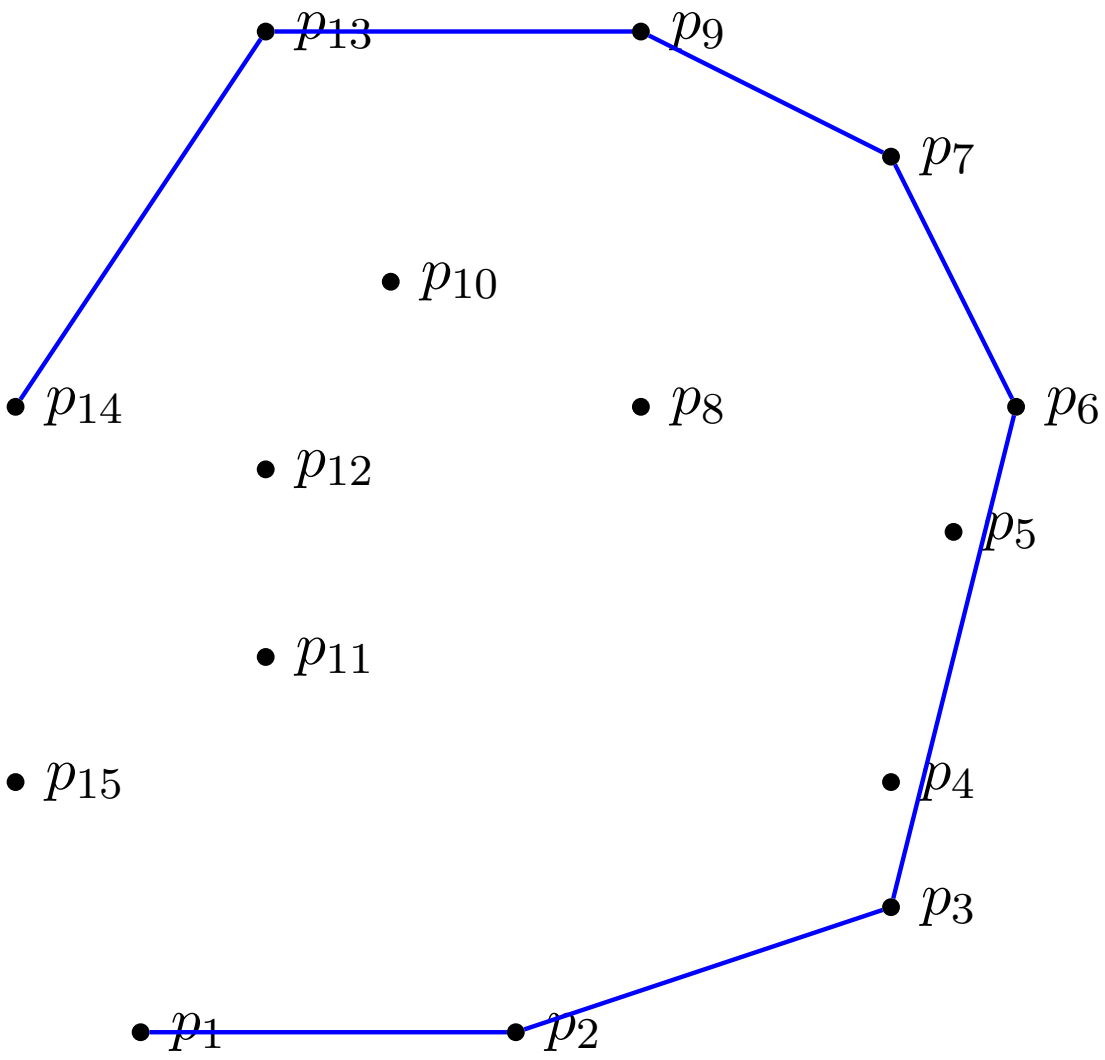
Primer



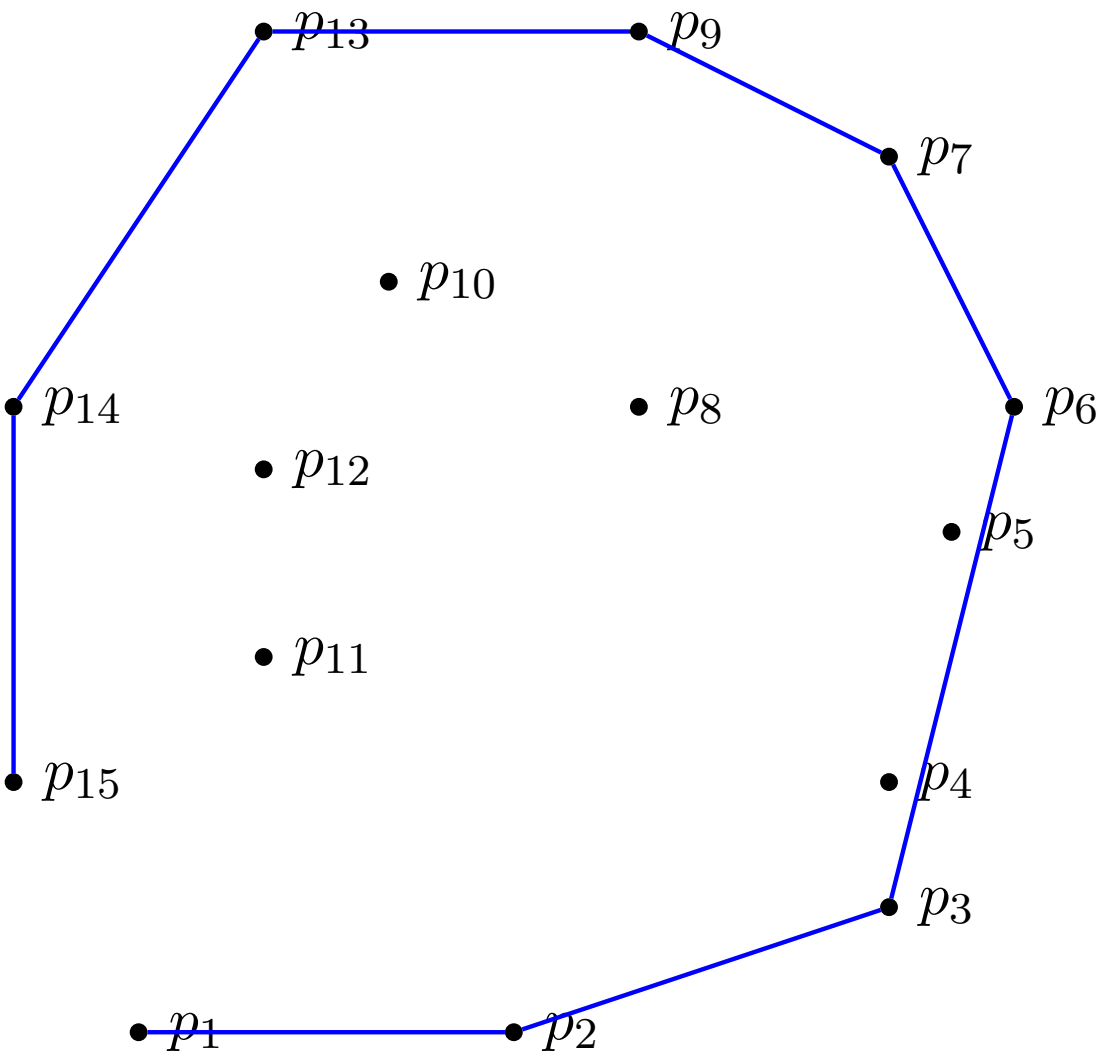
Primer



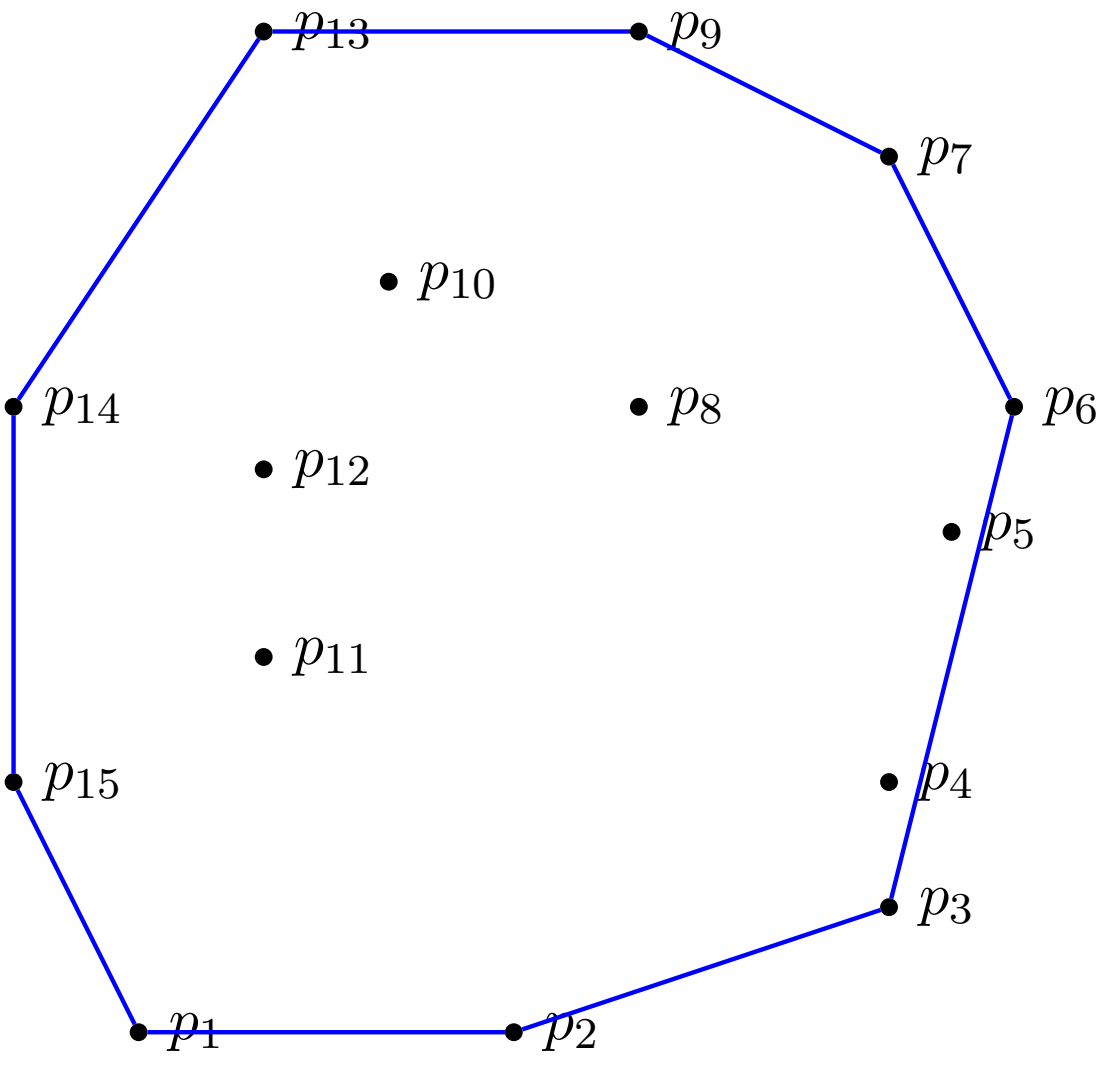
Primer



Primer



Primer



Pravilnost

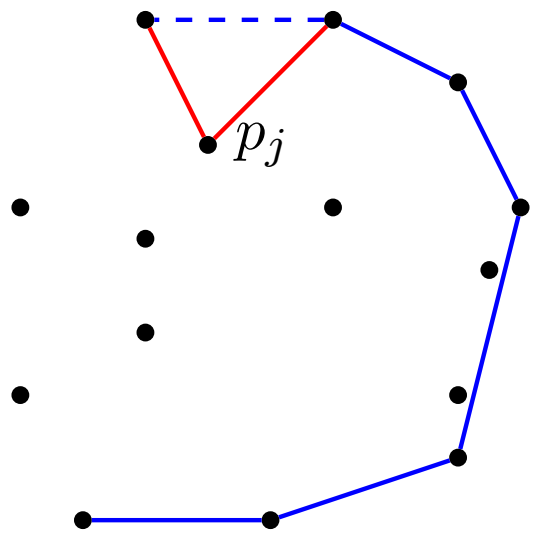
Trditev

Po obravnavi točke p_i ($i \geq 3$) so na skladu točke, ki tvorijo konveksno ovojnico točk p_1, p_2, ..., p_i. Točke na skladu so razvrščene v nasprotni smeri urinega kazalca.

- Za $i = 3$ trditev drži
- Predpostavimo, da drži za $i - 1$, in preverimo, ali drži tudi za i
- Pri obravnavi točke p_i s sklada odstranimo vrhnjih k točk (k je lahko 0) in na sklad dodamo točko p_i

Pravilnost

- Če točke na skladu tvorijo konveksen večkotnik, potem z dodatkom točke p_i ohranjamo konveksnost, saj so vsi zasuki levi
  - desne zasuke in kolinearnosti izločimo z odstranjevanjem točk s sklada
- Točka p_j, ki jo odstranimo, gotovo ne pripada konveksni ovojnici, saj leži v notranjosti ali na robu »vdrtega« trikotnika

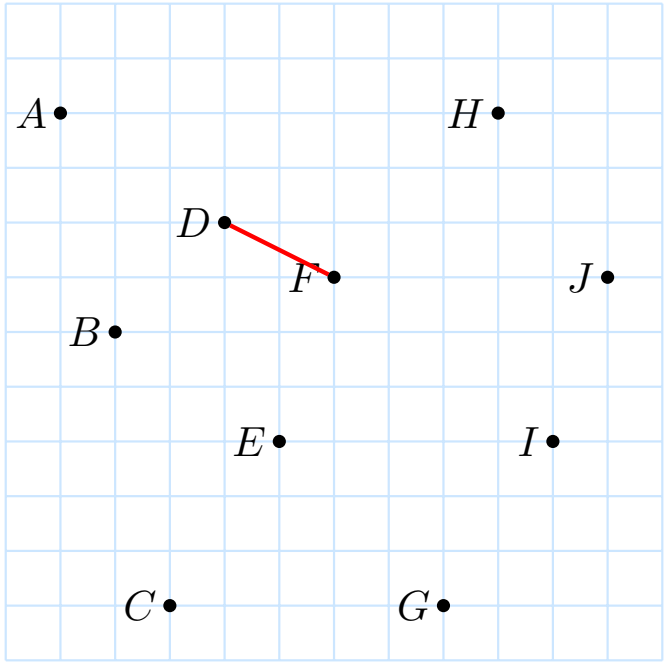


Časovna zahtevnost

- $O(n \log n)$ za urejanje točk glede na kot
- $O(n)$ za iterativno obravnavo točk
  - vsako točko natanko enkrat dodamo na sklad in največ enkrat odstranimo
- $O(n \log n)$

Najbližji par točk

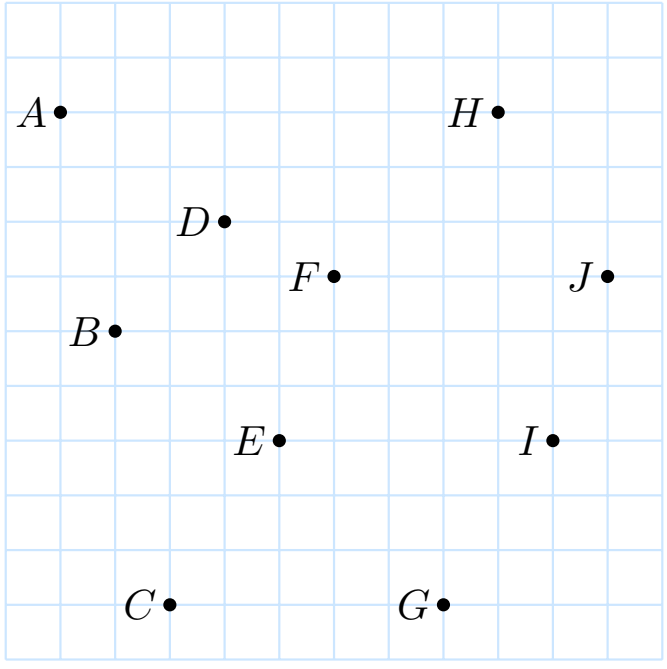
- Podana je množica T z $n \geq 2$ točkami
- Poišči točki z minimalno medsebojno evklidsko razdaljo



Najbližji par točk

- Z naivnim pristopom lahko problem rešimo v $O(n^2)$
- Obstaja pa tudi učinkovitejši algoritem po metodi deli in vladaj
- Na začetku (enkrat za vselej) izdelamo tabeli X in Y
- Tabela X vsebuje vse točke, urejene po naraščajoči koordinati x
- Tabela Y vsebuje vse točke, urejene po naraščajoči koordinati y
- Predpostavili bomo, da so indeksi točk urejeni glede na koordinate x, v primeru enakih koordinat x pa glede na koordinate y
  - $i < j \implies x_i \leq x_j \vee (x_i = x_j \wedge y_i \leq y_j)$

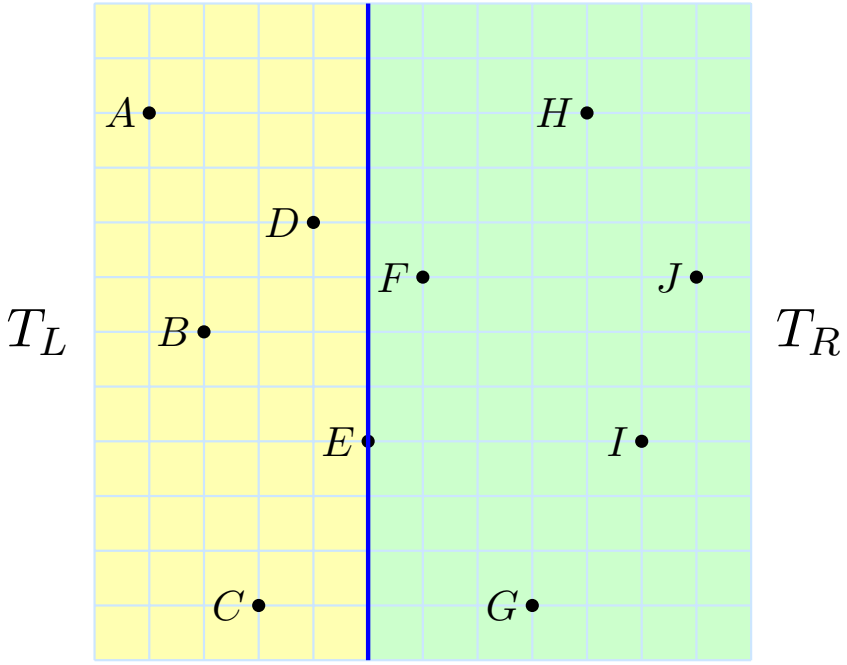
Začetek



- $X = [A, B, C, D, E, F, G, H, I, J]$
- $Y = [C, G, E, I, B, F, J, D, A, H]$

Faza »deli«

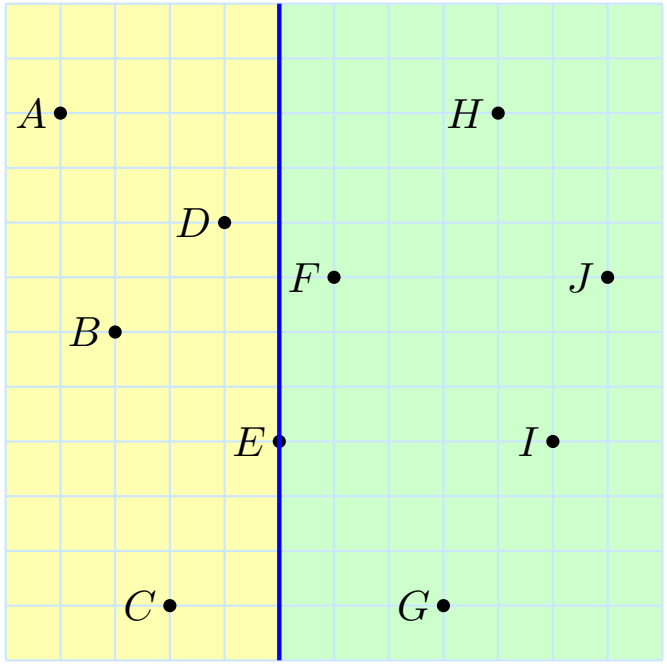
- Če je $n \leq 3$, rešimo problem z obravnavo vseh $\binom{n}{2}$ parov
- Sicer poiščemo navpično premico p, tako da je $\lceil n/2 \rceil$ točk na premici ali levo od nje, $\lfloor n/2 \rfloor$ točk pa na premici ali desno od nje
- Dobljeni množici točk označimo s T_L in T_R



Faza »deli«

- Ustvarimo tabele X_L, Y_L, X_R in Y_R
- X_L: točke v T_L, urejene po koordinati x
- Y_L: točke v T_L, urejene po koordinati y
- X_R: točke v T_R, urejene po koordinati x
- Y_R: točke v T_R, urejene po koordinati y

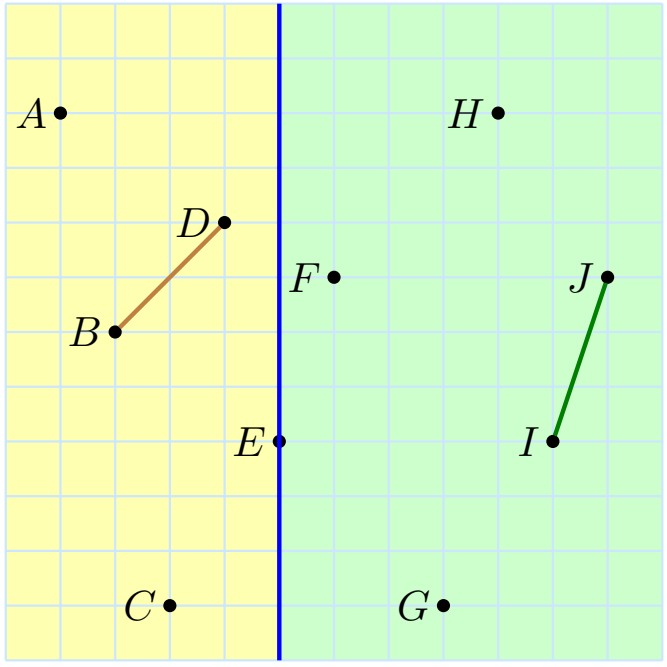
Faza »deli«



- $X_L = \{A, B, C, D, E\}$
- $X_R = \{F, G, H, I, J\}$
- $Y_L = \{C, E, B, D, A\}$
- $Y_R = \{G, I, F, J, H\}$

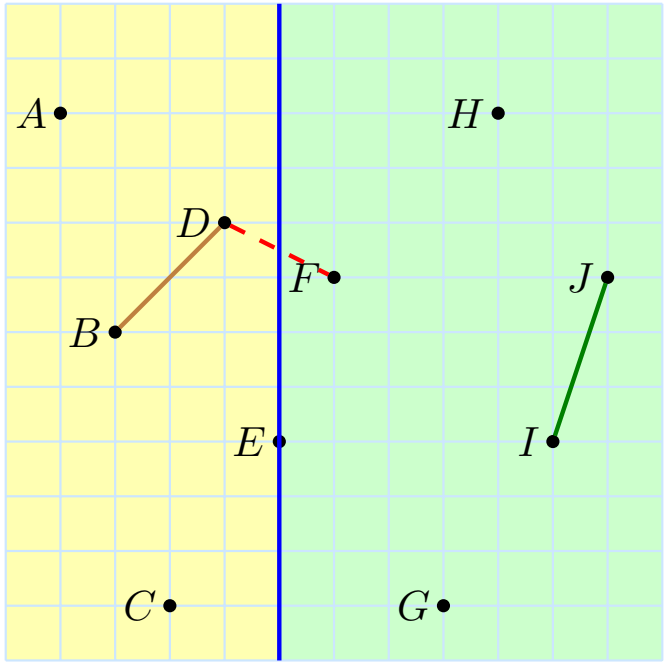
Faza »vladaj«

- Rekurzivno poišči medsebojno najbližji par točk v množicah T_L in T_R



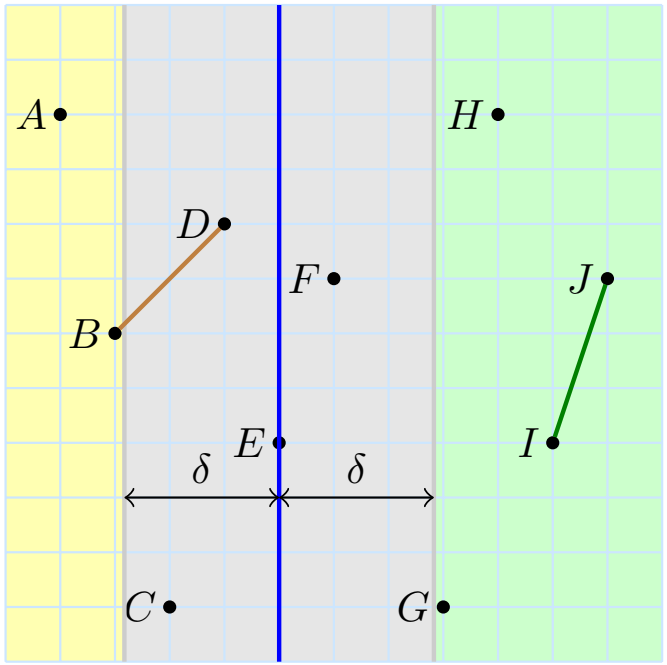
Faza »združi«

- Najbližji par v množici T je eden od sledečih:
  - najbližji par v množici T_L
  - najbližji par v množici T_R
  - najbližji par med pari (A, B), kjer je $A \in T_L$ in $B \in T_R$



Faza »združi«

- Naj bo δ_L najmanjša medsebojna razdalja v T_L
- Naj bo δ_R najmanjša medsebojna razdalja v T_R
- Naj bo $\delta = \min(\delta_L, \delta_R)$
- Med pari točk $A \in T_L$ in $B \in T_R$ pridejo v poštev samo tisti, ki so največ za δ v vsako smer oddaljeni od premice p



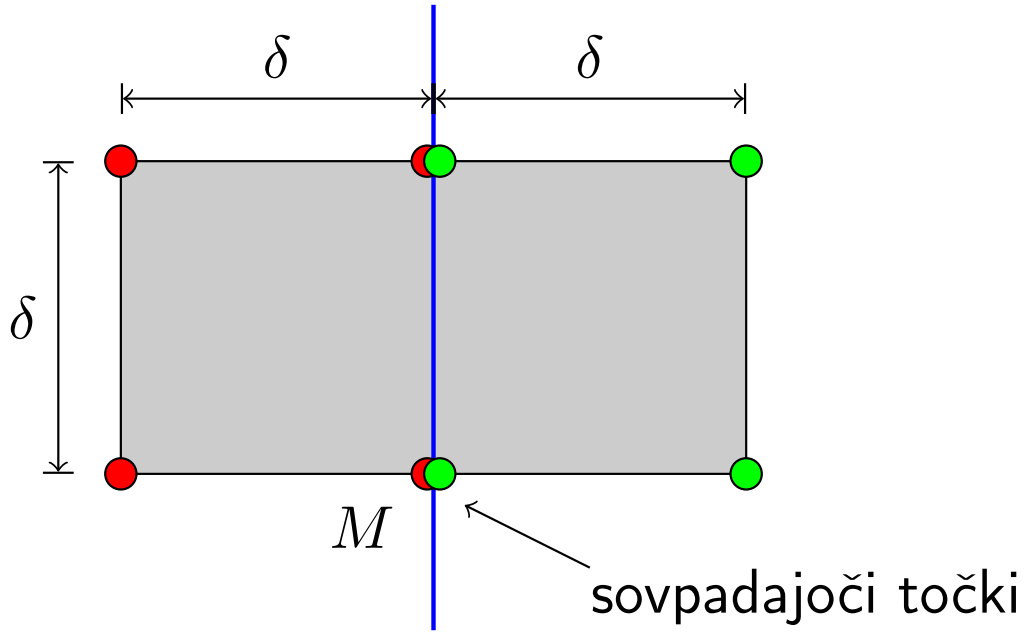
$$\delta_L = \sqrt{8}$$
$$\delta_R = \sqrt{10}$$
$$\delta = \sqrt{8}$$

Faza »združi«

- V pasu $P = [x_p - \delta, x_p + \delta] \times (-\infty, \infty)$ je še vedno lahko $\Theta(n)$ točk (oz. $\Theta(n^2)$ parov točk)
- K sreči nam ni treba preveriti vseh parov točk v tem pasu, saj lahko mejo δ upoštevamo tudi v smeri y
- Recimo, da točke v pasu P obravnavamo po naraščajočih koordinatah y
- Naj bo $M(x, y)$ neka točka v pasu P
- Največ koliko naslednikov točke M v tabeli Y_L oz. Y_R moramo upoštevati?

Faza »združi«

- Najmanjša razdalja med dvema točkama v T_L in T_R je δ
- V kvadratu $\delta \times \delta$ lahko imamo največ 4 točke na medsebojni razdalji najmanj δ
- V pravokotniku $2\delta \times \delta$ lahko imamo največ 8 takih točk
- Vsako točko M v pasu P torej primerjamo samo s 7 nasledniki



Časovna zahtevnost

- $O(n \log n)$ za začetno izdelavo tabel X in Y
  - to storimo samo enkrat
- $O(n)$ za iskanje premice p v fazi »deli«
  - točke v tabeli X so urejene po naraščajoči koordinati x, zato zadošča en sam sprehod po tabeli X

Časovna zahtevnost

- $O(n)$ za izdelavo tabel X_L, X_R, Y_L in Y_R
  - v X_L je prva polovica točk iz X
  - v X_R je druga polovica točk iz X
  - v Y_L so točke iz Y, ki pripadajo X_L
  - v Y_R so točke iz Y, ki pripadajo X_R
  - ker sta tabeli X in Y urejeni, bodo urejene tudi tabele X_L, X_R, Y_L in Y_R
  - pripadnost točke iz Y_L oz. Y_R tabeli X_L oz. X_R lahko preverimo v času $O(1)$, saj točke v X_L in X_R niso urejene samo po naraščajočih koordinatah x, ampak tudi po naraščajočih indeksih (gl. začetno predpostavko)
- $O(n)$ za obravnavo točk T_P
  - vsako točko primerjamo kvečjemu s sedmimi nasledniki

Časovna zahtevnost

- $O(n)$ za celoten postopek deljenja in združevanja
- Problem razdelimo na 2 podproblema polovične velikosti

$$T(n) = 2T(n/2) + \Theta(n)$$

- Krovni izrek: $a = 2$, $b = 2$, $d = 1$
- $a = b^d$
- $T(n) = \Theta(n^d \log n) = \Theta(n \log n)$
- Prištejemo še $O(n \log n)$ za začetno urejanje
- $O(n \log n)$

Presečišča med daljicami

- Podanih je n daljic
  - $d_i = (t_i, t'_i)$ za $i \in \{1, \ldots, n\}$
  - $t_i = (x_i, y_i)$
  - $t'_i = (x'_i, y'_i)$
- Naj bo t_i zgornje, t'_i pa spodnje krajišče
  - če je daljica vodoravna, naj bo t_i levo, t'_i pa desno krajišče
  - $y_i > y'_i \vee (y_i = y'_i \wedge x_i < x'_i)$
- Poišči vsa presečišča med daljicami!
- Za vsako presečišče izpiši tudi daljice, ki se sekajo v njem

Hm ...

- V skrajnem primeru imamo $\Theta(n^2)$ presečišč
  - npr. mreža $n \times n$ ima n^2 presečišč za $2n$ daljic
- Naivni algoritem preprosto za vsak par daljic izračuna njuno morebitno presečišče
- Ta pristop potrebuje $\Theta(n^2)$ časa in je torej optimalen
- Zakaj bi se potem sploh ukvarjali s tem problemom?

Motivacija

- Presečišč je lahko $o(n^2)$
- Želeli bi si algoritma, pri katerem je računska zahtevnost odvisna od dejanskega števila presečišč
- Takšni algoritmi so občutljivi na izhod (*output-sensitive*)

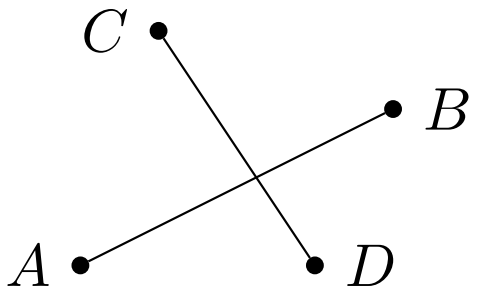
Pričnimo z dvema daljicama . . .

- Dve daljici se lahko
  (1) ne sekata
  (2) sekata v eni točki (to vključuje tudi dotikanje)
  (3) sekata v neskončno točkah

- S primerom (3) se ne bomo ukvarjali

Izračun morebitnega presečišča

- Če nas zanima samo to, ali se daljici sekata, položaj presečišča pa ni pomemben, si lahko pomagamo z vektorskim produktom



- Daljici AB in CD se sekata natanko tedaj, ko imata skupno krajišče ali pa ko sta točki C in D na različnih straneh daljice AB, točki A in B pa na različnih straneh daljice CD

$$\operatorname{sgn}(\overrightarrow{AB} \times \overrightarrow{AC}) \neq \operatorname{sgn}(\overrightarrow{AB} \times \overrightarrow{AD}) \wedge \operatorname{sgn}(\overrightarrow{CD} \times \overrightarrow{CA}) \neq \operatorname{sgn}(\overrightarrow{CD} \times \overrightarrow{CB})$$

Izračun morebitnega presečišča

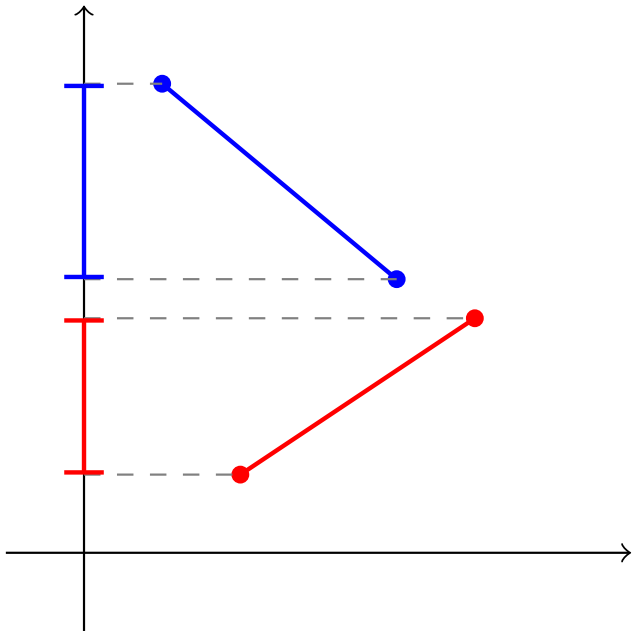
- Ker nas zanima tudi položaj presečišča, izračunamo presečišči nosilk daljic
  - sistem enačb $y = k_i x + n_i$ in $y = k_j x + n_j$
  - $k_i = (y'_i - y_i) / (x'_i - x_i)$ (in enako za j)
  - $n_i = y_i - k_i x_i$ (in enako za j)
  - navpične daljice je treba obravnavati ločeno
- Če nam test z vektorskim produktom pove, da se daljici sekata, potem je presečišče daljic istovetno s presečiščem nosilk daljic

Prehod na več daljic

- Začetni poenostavitvi
  - ni vodoravnih daljic
  - ne obstaja trojica daljic, ki se seka v isti točki
- Kasneje bomo ti poenostavitvi odpravili

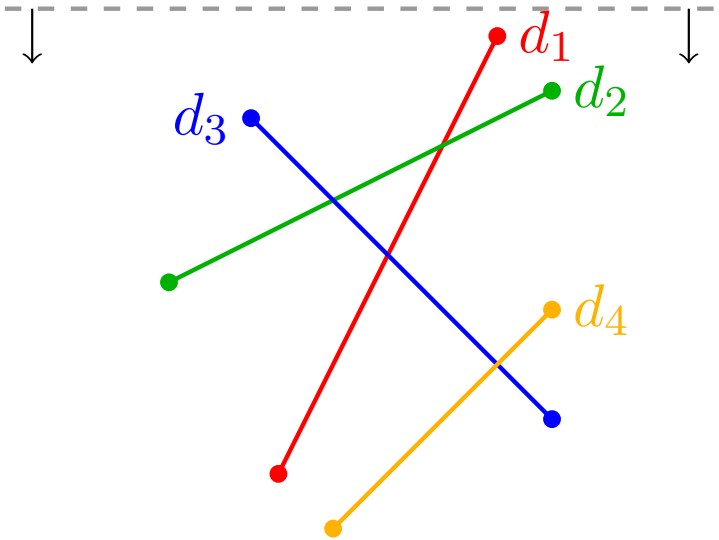
Prvo opažanje

- Če se projekciji daljic na os y ne sekata, se daljici gotovo ne sekata
- $d_1 = ((x_1, y_1), (x_1', y_1'))$
- $d_2 = ((x_2, y_2), (x_2', y_2'))$
- $y_1' > y_2 \implies$ daljici se ne sekata



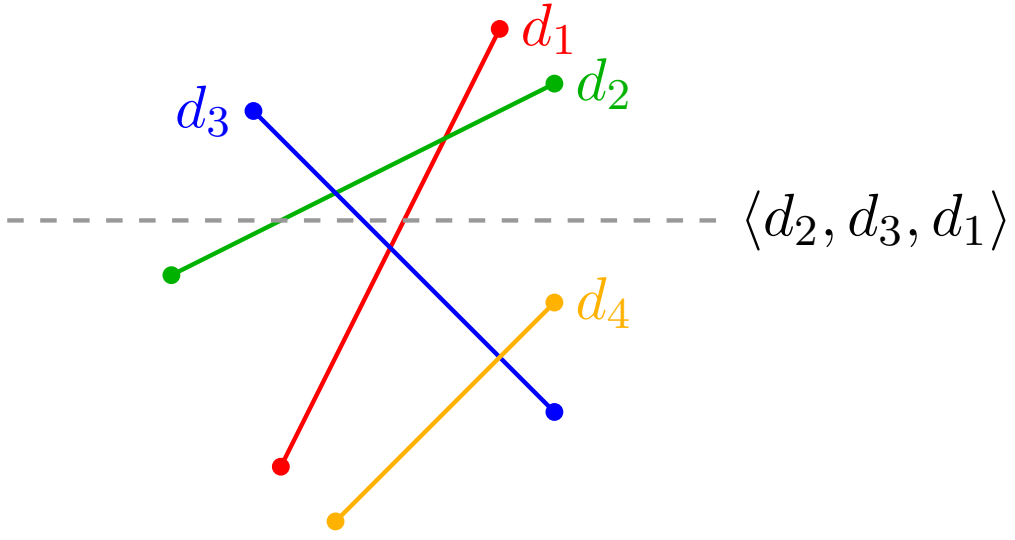
Potujoča premica (*sweep line*)

- Vodoravno premico premikamo od najvišje do najnižje ležeče točke
- Če se projekciji daljic na os y ne sekata, potem daljici ne bosta nikoli hkrati sekali potujoče premice
- Med seboj se lahko sekajo samo tiste daljice, ki sekajo potujočo premico
- Takih parov daljic je še vedno lahko $\Theta(n^2)$



Status potujoče premice

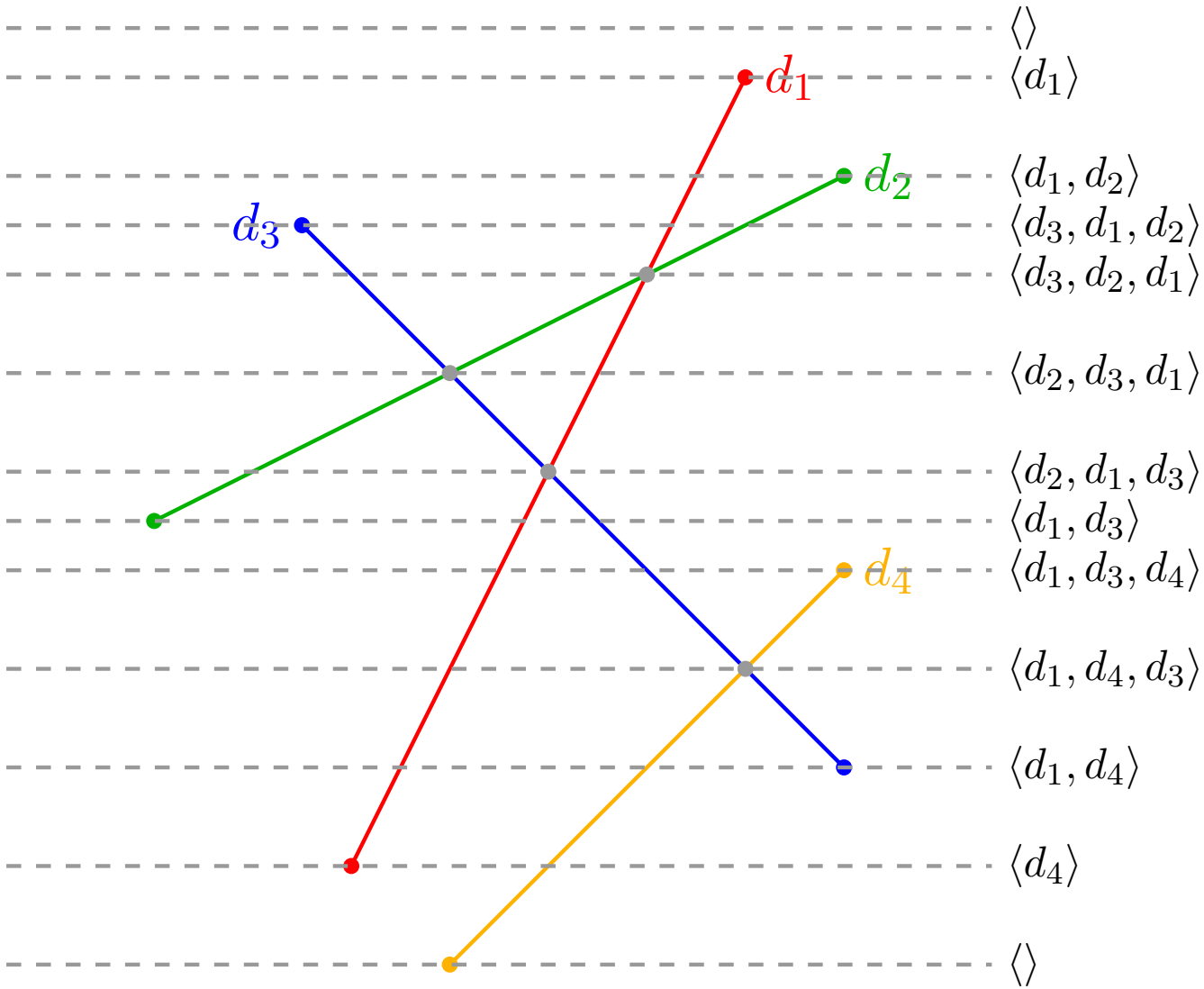
- Zaporedje daljic, ki sekajo potujočo premico
- Daljice v statusu so urejene po naraščajočih koordinatah x
- Kasneje bomo videli, zakaj je to pomembno ...



Posodabljanje statusa

- Status je na začetku prazen
- Spreminja se samo ob dogodkih
- Dogodek »zgornje krajišče«
  - naletimo na zgornje krajišče neke daljice
  - daljica vstopi v status
- Dogodek »spodnje krajišče«
  - naletimo na spodnje krajišče neke daljice
  - daljica izstopi iz statusa
- Dogodek »presečišče«
  - naletimo na presečišče daljic
  - daljici, ki se sekata, zamenjata medsebojni vrstni red v statusu

Posodabljanje statusa



Ključno opažanje 1

Trditev 1

Če se daljici d_i in d_j sekata, potem ob nekem dogodku postaneta sosedi v statusu.

- Še vedno velja predpostavka, da se nobena trojica daljic ne seka v isti točki
- Ko je potujoča premica tik nad presečiščem daljic d_i in d_j, sta d_i in d_j gotovo sosedi
- Na začetku izvajanja algoritma je status prazen
- Status se spreminja samo ob dogodkih
- Od tod sledi, da ob nekem dogodku daljici d_i in d_j postaneta sosedi

Osnovni obris algoritma

- S potujočo premico potuj od najvišje do najnižje ležečega krajišča
- Ko naletiš na dogodek »zgornje krajišče daljice d«
  - dodaj daljico d v status
  - izračunaj presečišče daljice d z njeno levo in desno sosedo
  - dodaj presečišči med bodoče dogodke

Osnovni obris algoritma

- Ko naletiš na dogodek »spodnje krajišče daljice d«
  - odstrani daljico d iz statusa
  - izračunaj presečišče leve in desne sosede daljice d
  - če se presečišče nahaja nižje od trenutnega položaja potujoče daljice, ga dodaj med bodoče dogodke

Osnovni obris algoritma

- Ko naletiš na dogodek »presečišče daljic d_i in d_j«
  - zamenjaj položaja daljic d_i in d_j v statusu
  - izračunaj presečišče daljice d_j z njeno novo levo sosedo
  - izračunaj presečišče daljice d_i z njeno novo desno sosedo
  - če se presečišči nahajata nižje od trenutnega položaja potujoče daljice, ju dodaj med bodoče dogodke

Pravilnost algoritma

Trditev 2

Algoritem najde natanko vsa presečišča.

- Po trditvi 1 vsak par daljic, ki se seka, pri nekem dogodku postane soseden
- Sosednost daljic se lahko spreminja samo ob dogodkih
- Algoritem obravnava vse dogodke in pravilno posodablja sosednost daljic

Invarianta med delovanjem algoritma

Vsa presečišča nad potujočo premico smo že odkrili in izpisali.

Podatkovna struktura za dogodke

- Dogodke (krajišča in presečišča daljic) lahko hranimo v prioritetni vrsti (kopici)
- Dogodek (x_1, y_1) ima višjo prioriteto kot dogodek (x_2, y_2) natanko tedaj, ko velja

$$y_1 > y_2 \vee (y_1 = y_2 \wedge x_1 < x_2)$$

- Vse operacije so izvedljive v $O(\log n)$

Podatkovna struktura za status potujoče daljice

- Podatkovna struktura mora podpirati učinkovito vzdrževanje urejenega zaporedja daljic
  - daljice v statusu so urejene od leve proti desni glede na njihova trenutna presečišča s potujočo premico
- Podpirati mora učinkovito iskanje levih in desnih sosedov
  - ob vsakem dogodku preverimo presečišče neke daljice z levo in/ali desno sosedo
- Vse to nam omogoča uravnoteženo dvojiško iskalno drevo (npr. rdeče-črno ali AVL) z dodanimi kazalci na starše
- Vse operacije so izvedljive v $O(\log n)$

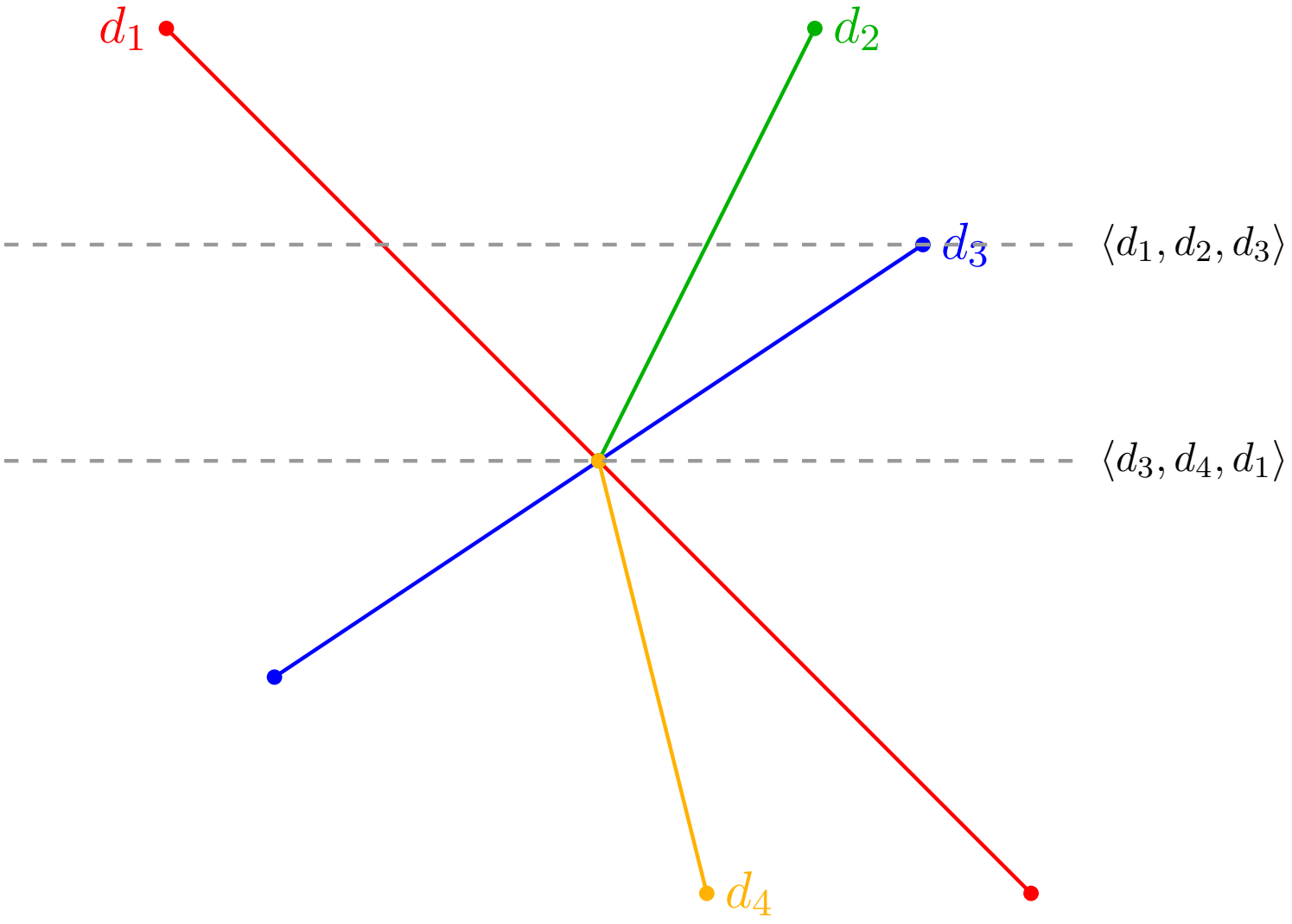
Obravnava vodoravnih daljic

- Levo krajišče obravnavamo kot »zgornje«, desno pa kot »spodnje«
- Lahko si predstavljamo, kot da potujoča premica ni povsem vodoravna, ampak je »čisto rahlo« nagnjena

Presečišče več daljic

- Lahko si predstavljamo, kot da je presečna točka spodnja točka zgornjih odsekov in zgornja točka spodnjih odsekov daljic
- Presečišče je dejansko lahko tudi stičišče
- Udeležene daljice odstranimo iz statusa, nato pa jih vanj ponovno dodamo

Presečišče več daljic

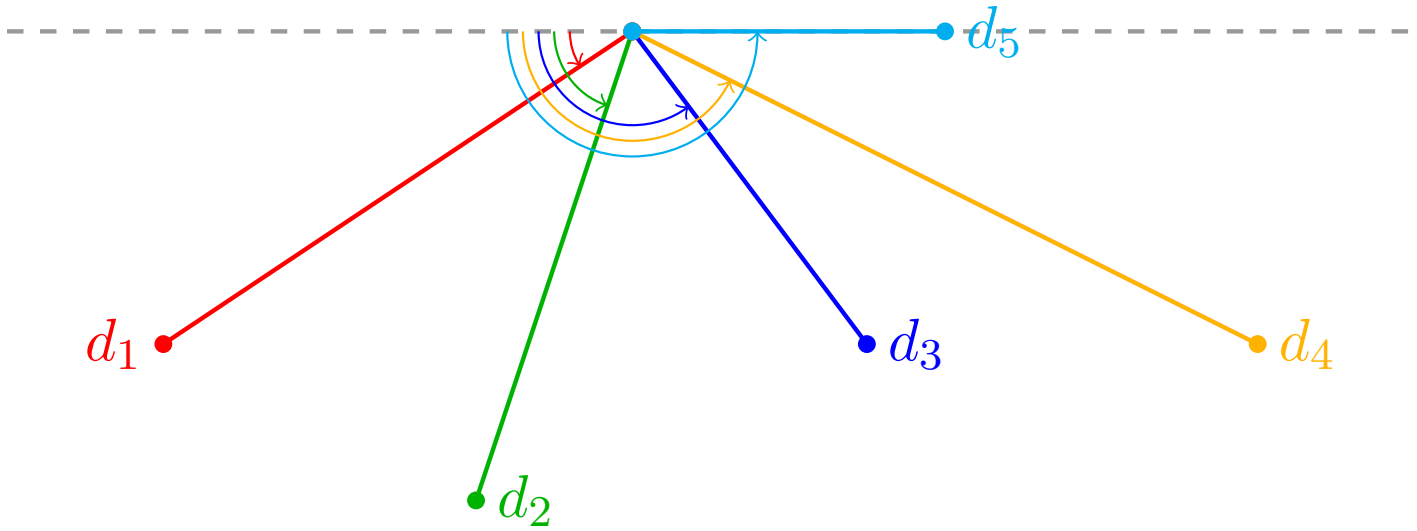


Podatkovni strukturi nekoliko podrobneje

- Vzdržujemo prioritetno vrsto dogodkov Q in statusno drevo S
- Q vsebuje elemente oblike (t, D)
  - t je zgornje ali spodnje krajišče ene ali več daljic ali presečišče dveh ali več daljic
  - $D(t)$ naj bo množica daljic s skupno točko t (to je lahko ena sama daljica)
  - če je t zgornje krajišče daljic v množici $D(t)$, nastavimo $D \leftarrow D(t)$, sicer pa $D \leftarrow \emptyset$
  - $((x, y), D) \prec ((x', y'), D') \iff y > y' \vee (y = y' \wedge x < x')$

Podatkovni strukturi nekoliko podrobneje

- Drevo S vsebuje daljice (ali njihove indekse) od leve proti desni, ki v danem trenutku sekajo potujočo premico
- Če ista točka pripada več daljicam, so daljice urejene glede na situacijo tik pod to točko
- Lahko jih uredimo po naraščajočem kotu, ki ga oklepajo s potujočo premico
  - spet nam pride prav vektorski produkt!



Celoten algoritem

function Poišči-presečišča($\langle d_1, \ldots, d_n \rangle$)
  ▷ $d_i = (t_i, t'_i)$ *za* $i \in \{1, \ldots, n\}$ ◁
  Inicializiraj prioritetno vrsto Q
  $U \leftarrow \{t_i \mid i \in \{1, \ldots, n\}\}$ ▷ *zgornja krajišča daljic*
  $L \leftarrow \{t'_i \mid i \in \{1, \ldots, n\}\}$ ▷ *spodnja krajišča daljic*
  for $t \in U$ **do**
    $D \leftarrow$ množica daljic z zgornjo točko t
    vstavi (t, D) v Q
  for $t \in L$ **do**
    vstavi $(t, \emptyset)$ v Q
  while Q ni prazna **do**
    $(t, D) \leftarrow$ prvi element vrste Q
    odstrani prvi element iz vrste Q
    Obravnavaj-dogodek(t, D)

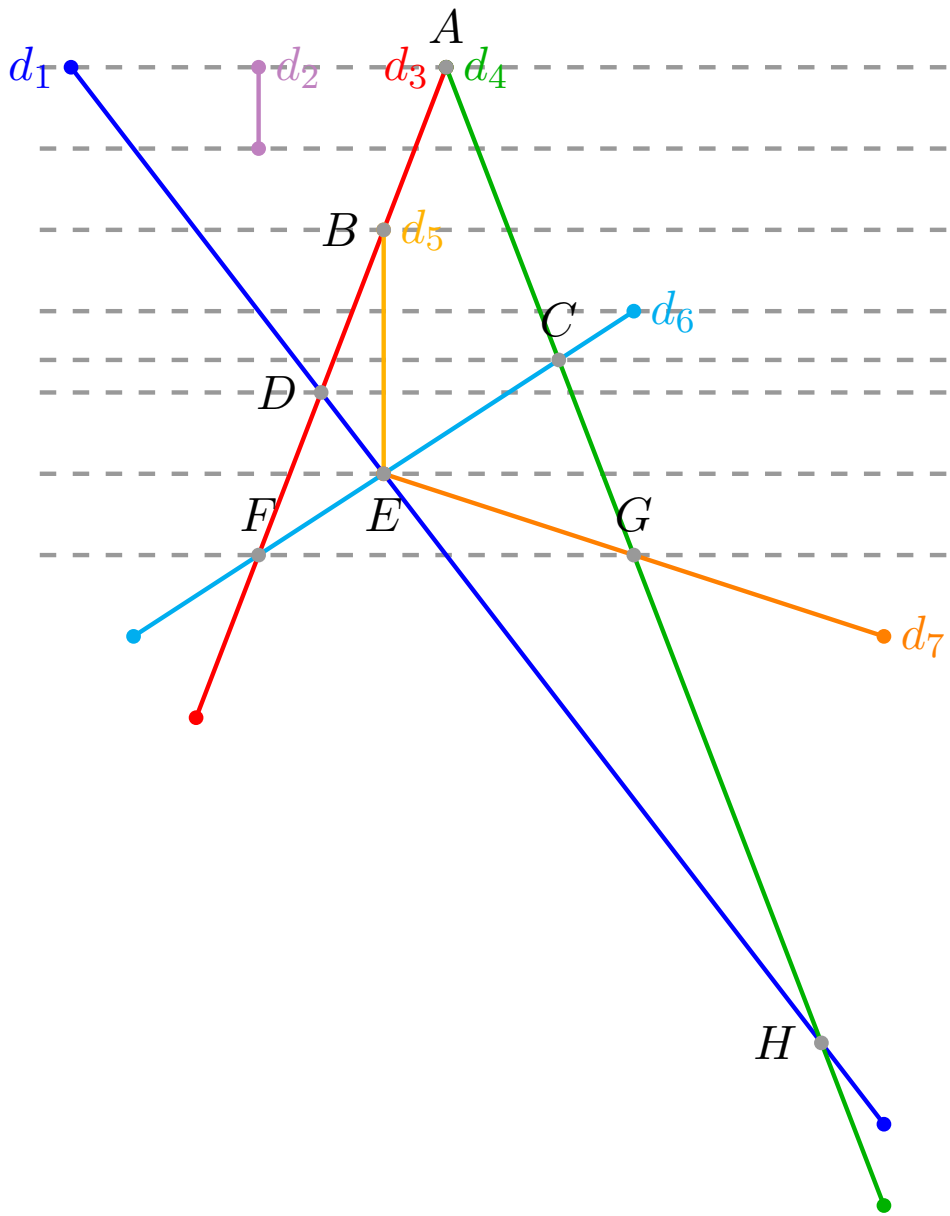
Celoten algoritem

function OBRAVNAVAJ-DOGODEK(t, D)
  $U(t) \leftarrow D$ ▷ *daljice, pri katerih je t zgornje krajišče*
  V drevesu S poišči vse daljice, ki vsebujejo točko t (sosede so!)
  $L(t) \leftarrow$ daljice v drevesu S, pri katerih je t spodnje krajišče
  $C(t) \leftarrow$ daljice v drevesu S, pri katerih je t vmesna točka
  if $|U(t) \cup L(t) \cup C(t)| \geq 2$ **then**
    izpiši t in množico $U(t) \cup L(t) \cup C(t)$
  Odstrani daljice v $L(t) \cup C(t)$ iz drevesa S
  Vstavi daljice v $U(t) \cup C(t)$ v drevo S
  if $U(t) \cup C(t) = \emptyset$ **then**
    $l \leftarrow$ levi sosed t v drevesu S
    $r \leftarrow$ desni sosed t v drevesu S
    POIŠČI-NOV-DOGODEK(l, r, t)
  else
    $d' \leftarrow$ skrajno leva daljica iz $U(t) \cup C(t)$ v drevesu S
    $l \leftarrow$ levi sosed daljice d' v drevesu S
    POIŠČI-NOV-DOGODEK(l, d', t)
    $d'' \leftarrow$ skrajno desna daljica iz $U(t) \cup C(t)$ v drevesu S
    $r \leftarrow$ desni sosed daljice d'' v drevesu S
    POIŠČI-NOV-DOGODEK(d'', r, t)

Celoten algoritem

function POIŠČI-NOV-DOGODEK(l, r, t)
    if l in r se sekata pod potujočo premico ali
            na potujoči premici, a desno od točke t **then**
        $p \leftarrow$ presečišče daljic l in r
        if $p \notin Q$ **then**
            vstavi $(p, \emptyset)$ v Q

Primer



$\langle d_1, d_2, d_3, d_4 \rangle \mid d_3 \cap d_4 = A$

$\langle d_1, d_3, d_4 \rangle \mid d_1 \cap d_3 = D$

$\langle d_1, d_3, d_5, d_4 \rangle \mid d_3 \cap d_5 = B$

$\langle d_1, d_3, d_5, d_4, d_6 \rangle \mid d_4 \cap d_6 = C$

$\langle d_1, d_3, d_5, d_6, d_4 \rangle \mid d_5 \cap d_6 = E$

$\langle d_3, d_1, d_5, d_6, d_4 \rangle \mid d_1 \cap d_5 = E$

$\langle d_3, d_6, d_1, d_7, d_4 \rangle \mid d_3 \cap d_6 = F \mid d_7 \cap d_4 = G$

$\langle d_6, d_3, d_1, d_7, d_4 \rangle$

$\langle d_6, d_3, d_1, d_4, d_7 \rangle \mid d_1 \cap d_4 = H$

$\vdots$

Časovna zahtevnost

- $O(n \log n)$ za začetno izgradnjo prioritetne vrste Q
- $O(\log n)$ za vsako operacijo nad Q in drevesom S
- Obravnava dogodkovne točke t
  - $m(t) = |U(t) \cup L(t) \cup C(t)|$ naj bo število daljic, ki se sekajo v točki m
  - $O(m(t) \log n)$ za obravnavo teh daljic

Časovna zahtevnost

- Naj bo $m = \sum_{t \in M} m(t)$
  - M je množica vseh dogodkovnih točk (krajišč in presečišč)
- $m = O(n + k)$
  - n daljic pomeni vsaj $n + 1$ dogodkov
  - k je velikost izhoda
- Skupna časovna zahtevnost: $O(m \log n) = O((n + k) \log n)$
- V resnici celo $O((n + p) \log n)$, kjer je p število presečišč
  - daljice in dogodkovne točke lahko obravnavamo kot ravninski graf (V, E)
  - za ravninske grafe velja $E = O(V)$
- Algoritem je torej res občutljiv na izhod

Implementacija v C++: prioritetna vrsta Q

- Načeloma `priority_queue`, vendar pa je koristno imeti tudi možnost preverjanja, ali je dogodek že v vrsti
- `priority_queue` tega ne ponuja, zato raje izberemo `set`
- Podatkovni tip `set` je implementiran kot uravnoteženo dvojiško iskalno drevo, ki omogoča tudi sprehajanje od najmanjšega (skrajno levega) do največjega (skrajno desnega) elementa
- `s.begin()` je torej iterator na prvi element prioritetne vrste

Implementacija v C++: statusno drevo S

- Takisto `set` (ni potrebe po programiranju rdeče-črnega ali AVL-drevesa)
- S klicem `s.find()` dobimo iterator na iskano daljico, z zmanjševanjem in povečevanjem tega iteratorja pa pa pridobimo leve in desne sosede te daljice
- Potencialna težava: kriterij za urejenost ni fiksen, ampak je vezan na trenutni dogodek
  - k sreči to ni težava, saj prehod na naslednji dogodek vpliva samo na daljice, ki so v njem udeležene
  - vrstni red ostalih daljic je konsistenten s predhodnim in trenutnim kriterijem