

Prioritní fronta a příklad použití v úloze hledání nejkratších cest

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 12

B0B36PRP – Procedurální programování

Jan Faigl, 2018 B0B36PRP – Přednáška 12: Halda a hledání nejkratších cest 1 / 50

Prioritní fronta polem

Halda

Prioritní fronta polem – rozhraní

- V případě implementace prioritní fronty polem můžeme využít jedno pole pro hodnoty a druhé pole pro uložení priority daného prvku

Implementace vychází z lec11/queue_array.h.

```
typedef struct {  
    void **queue; // Pole ukazatelů na jednotlivé prvky  
    int *priorities; // Pole hodnot priorit jednotlivých prvků  
    int count;  
    int head;  
    int tail;  
} queue_t;
```

- Další rozhraní (jména a argumenty funkcí) mohou zůstat identické jako u implementace spojovým seznamem

Viz předchozí přednáška

```
void queue_init(queue_t **queue); int queue_push(void *value, int priority,  
void queue_delete(queue_t **queue); queue_t *queue);  
void queue_free(queue_t *queue); void* queue_pop(queue_t *queue);  
void* queue_peek(const queue_t *queue);  
  
_Bool queue_is_empty(const queue_t *queue);
```

Jan Faigl, 2018 B0B36PRP – Přednáška 12: Halda a hledání nejkratších cest 5 / 50

Prioritní fronta polem

Halda

Prioritní fronta polem 2/3 – peek() a pop()

- Funkce peek() využívá lokální (static) funkce getEntry()

```
void* queue_peek(const queue_t *queue)  
{  
    return queue_is_empty(queue) ? NULL : queue->queue[getEntry(queue)];  
}
```

- Ve funkci pop() musíme zajistit zaplnění místa, pokud je vyjmut prvek z prostředka fronty (pole).

Případnou mezeru zaplníme prvkem ze startu

```
void* queue_pop(queue_t *queue) Případnou mezeru zaplníme prvkem ze startu  
{  
    void *ret = NULL;  
    int bestEntry = getEntry(queue);  
    if (bestEntry >= 0) { // entry has been found  
        ret = queue->queue[bestEntry];  
        if (bestEntry != queue->head) { //replace the bestEntry by head  
            queue->queue[bestEntry] = queue->queue[queue->head];  
            queue->priorities[bestEntry] = queue->priorities[queue->head];  
        }  
        queue->head = (queue->head + 1) % MAX_QUEUE_SIZE;  
        queue->count -= 1;  
    }  
    return ret;  
}
```

Jan Faigl, 2018 B0B36PRP – Přednáška 12: Halda a hledání nejkratších cest 8 / 50

Přehled témat

- Část 1 – Prioritní fronta polem a haldou
Prioritní fronta polem
Halda
- Část 2 – Příklad využití prioritní fronty v úloze hledání nejkratší cesty v grafu
Popis úlohy
Návrh řešení
Příklad naivní implementace prioritní fronty polem
Implementace pq haldou s push() a update()
- Část 3 – Zadání 10. domácího úkolu (HW10)

Jan Faigl, 2018 B0B36PRP – Přednáška 12: Halda a hledání nejkratších cest 2 / 50

Prioritní fronta polem

Halda

Prioritní fronta polem 1/3 – push()

- Funkce push() je až na uložení priority identická s verzí bez priorit

```
int queue_push(void *value, int priority, queue_t *queue)  
{  
    int ret = QUEUE_OK; // by default we assume push will be OK  
    if (queue->count < MAX_QUEUE_SIZE) {  
        queue->queue[queue->tail] = value;  
  
        // store priority of the new value entry  
        queue->priorities[queue->tail] = priority;  
  
        queue->tail = (queue->tail + 1) % MAX_QUEUE_SIZE;  
        queue->count += 1;  
    } else {  
        ret = QUEUE_MEMFAIL;  
    }  
    return ret;  
}
```

- Funkce peek() a pop() potřebují prvek s nejnižší (nejvyšší) prioritou

- Nalezení prvku z „čela“ fronty realizujeme funkcí getEntry(), kterou následně využijeme jak v peek(), tak v pop()

Jan Faigl, 2018 B0B36PRP – Přednáška 12: Halda a hledání nejkratších cest 6 / 50

Prioritní fronta polem

Halda

Prioritní fronta polem – příklad použití

- Použití je identické s implementací spojovým seznamem

```
make && ./demo-priority_queue_array  
ccache clang -c priority_queue_array.c -O2 -o priority_queue-  
array.o  
ccache clang priority_queue_array.o demo-priority_queue_array.o  
-o demo-priority_queue_array  
Add 0 entry '2nd' with priority '2' to the queue  
Add 1 entry '4th' with priority '4' to the queue  
Add 2 entry '1st' with priority '1' to the queue  
Add 3 entry '5th' with priority '5' to the queue  
Add 4 entry '3rd' with priority '3' to the queue  
  
Pop the entries from the queue  
1st  
2nd  
3rd  
4th  
5th  
  
lec12/priority_queue_array/priority_queue_array.h  
lec12/priority_queue_array/priority_queue_array.c  
lec12/priority_queue_array/demo-priority_queue_array.c
```

Jan Faigl, 2018 B0B36PRP – Přednáška 12: Halda a hledání nejkratších cest 9 / 50

Prioritní fronta polem

Halda

Část I

Část 1 – Prioritní fronta (Halda)

Jan Faigl, 2018 B0B36PRP – Přednáška 12: Halda a hledání nejkratších cest 3 / 50

Prioritní fronta polem

Halda

Prioritní fronta polem 2/3 – getEntry()

- Nalezení nejmenšího (největšího) prvku provedeme lineárním prohledáním aktuálních prvků uložených ve frontě (poli)

```
static int getEntry(const queue_t *const queue)  
{  
    int ret = -1;  
    if (queue->count > 0) {  
        for (int cur = queue->head, i = 0; i < queue->count; ++i) {  
            if (  
                ret == -1 ||  
                (queue->priorities[ret] > queue->priorities[cur])  
            ) {  
                ret = cur;  
            }  
            cur = (cur + 1) % MAX_QUEUE_SIZE;  
        }  
    }  
    return ret;  
}
```

Jan Faigl, 2018 B0B36PRP – Přednáška 12: Halda a hledání nejkratších cest 7 / 50

Prioritní fronta polem

Halda

Prioritní fronta spojovým seznamem nebo polem a výpočetní náročnost

- V naivní implementaci prioritní fronty jsme zohlednění priority „odložili“ až do doby, kdy potřebujeme odebrat prvek z fronty
- Při odebrání (nebo vrácení) nejmenšího prvku v nejnepříznivějším případě musíme projít všechny položky
- Tu může být v případě mnoha prvků **výpočetně náročné** a raději bychom chtěli „udržovat“ prvek připravený
 - Můžeme to například udělat zavedením položky **head**, ve které bude aktuálně nejnižší (nejvyšší) vložený prvek do fronty
 - Prvek **head** aktualizujeme v metodě **push()** porovnáním hodnoty aktuálně vkládaného prvku
 - Tím zefektivníme operaci **peek()**
 - V případě odebrání prvku, však musíme frontu znovu projít a najít nový prvek

Alternativně můžeme použít sofistikovanější datovou strukturu, která nám umožní efektivně udržovat hodnotu nejmenšího prvku a to jak při operaci vložení **push()** tak při operaci vyjmutí **pop()** prvku z prioritní fronty.

Jan Faigl, 2018 B0B36PRP – Přednáška 12: Halda a hledání nejkratších cest 10 / 50

Halda

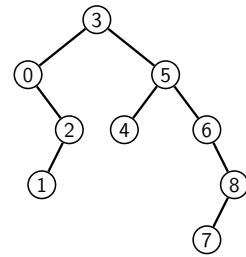
- Halda je dynamická datová struktura, která má „tvar“ binárního stromu a uspořádání prioritní fronty
- Každý prvek haldy obsahuje hodnotu a dva potomky, podobně jako binární strom
- Vlastnosti haldy – „Heap property“**
 - Hodnota každého prvku je menší než hodnota libovolného potomka
 - Každá úroveň binárního stromu haldy je plná, kromě poslední úrovně, která je zaplněna zleva doprava **Binární plný strom**
 - Prvky mohou být odebrány pouze přes kořenový uzel
- Vlastnost haldy zajišťuje, že **kořen je vždy prvek s nejnižším/nevyšším ohodnocením**

V případě binárního plného stromu je složitost procházení úměrná hloubce stromu, která je v případě n prvků úměrná $\log_2(n)$. Složitost operací `push()`, `pop()`, `peek()` tak můžeme očekávat nikoliv $O(n)$ (jako v případě předchozí implementace prioritní fronty polem a spojovým seznamem), ale $O(\log n)$.

Binární vyhledávací strom vs halda

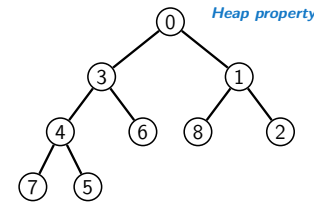
Binární vyhledávací strom

- Může obsahovat prázdná místa
- Hloubka stromu se může měnit
 - Přestože jsme raději, pokud je strom vyvážený. To je však implementačně náročnější než implementace haldy.



Halda

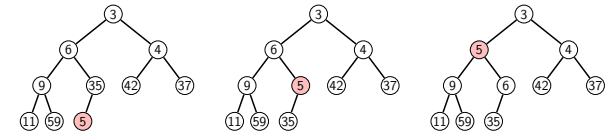
- Binární plný strom
 - Hloubka stromu vždy $\lfloor \log_2(n) \rfloor$
- Kořen stromu je vždy prvek s nejnižší (nejvyšší) hodnotou
- Každý podstrom splňuje vlastnost haldy



Halda – přidání prvku `push()`

- Po každém provedení operace `push()` musí být splněny vlastnosti haldy
- Prvek přidáme na konec haldy, tj. na první volnou pozici (vlevo) na nejnižší úrovni haldy
- Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s nadřazeným prvkem (předkem)

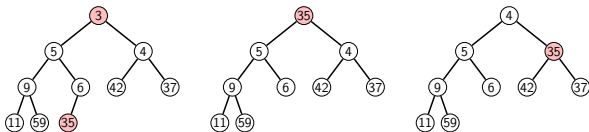
V nejnepříznivějším případě prvek „probublá“ až do kořene stromu



Halda – odebrání prvku `pop()`

- Při operaci `pop()` odebereme kořen stromu
- Prázdné místo nahradíme nejpravějším listem
- Zkontrolujeme, zdali je splněna podmínka haldy, pokud ne, zaměníme prvek s potomkem a postup opakujeme

V nejnepříznivějším případě prvek „probublá“ až do listu stromu



- Jak zjistit nejpravější list
 - V případě implementace spojovou strukturou (nelineární) můžeme explicitně udržovat odkaz
 - Binární plný strom můžeme efektivně reprezentovat polem** – pak nejpravější list je poslední prvek v poli

Prioritní fronta haldou

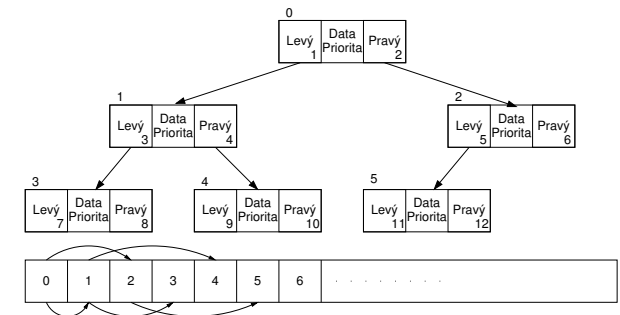
- Prvky ukládáme do haldy a při každém vložení / odebrání zajišťujeme, aby platily vlastnosti **haldy**
- Operace `peek()` má konstantní složitost a nezáleží na počtu prvků ve frontě, nejnižší prvek je vždy kořen
- Operace `push()` a `pop()` udržují vlastnost haldy záměnami prvku až do hloubky stromu

Asymptotická složitost v notaci velké O je $O(1)$.

Pro binární plný strom je hloubka stromu $\log_2(n)$, kde n je aktuální počet prvků ve stromu, odtud složitost operace $O(\log(n))$.

Reprezentace binárního stromu polem

- Binární plný strom můžeme reprezentovat lineární strukturou
- V případě známého maximálního počtu prvků v haldě, pak jednoduše předalokovaným polem položek

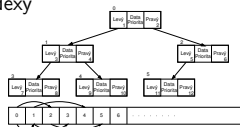


Halda jako binární plný strom reprezentovaný polem

- Pro definovaný maximální počet prvků v haldě, si předalokujeme pole o daném počtu prvků
- Binární **plný strom** má všechny vrcholy na úrovni rovné hloubce stromu co nejvíce vlevo
- Kořen stromu je první prvek s indexem 0, následníky prvku na pozici i lze v poli určit jako prvky s indexy

- levý následník: $i_{levy} = 2i + 1$
- pravý následník: $i_{pravý} = 2i + 2$

Podobně lze odvodit vztah pro předchůdce



- Kořen stromu reprezentuje nejprioritnější prvek (např. s nejmenší hodnotou nebo maximální prioritou)

Operace vkládání a odebírání prvků

- I v případě reprezentace polem pracují operace vkládání a odebírání identicky
 - Funkce `push()` přidá prvek jako další prvek v poli a následně propaguje prvek směrem nahoru až je splněna vlastnost haldy
 - Při odebrání prvku funkcí `pop()` je poslední prvek v poli umístěn na začátek pole (tj. kořen stromu) a propagován směrem dolů až je splněna vlastnost haldy
- Dochází pouze k vzájemnému zaměňování hodnot na pozicích v poli (haldě)
 - Z indexu prvku v poli vždy můžeme určit jak levého a pravého následníka, tak i předcházející prvek (rodič) v pohledu na haldu jako binární strom.
- Hlavní výhodou reprezentace polem je přístup do předem alokovaného bloku paměti
 - Všechny prvky můžeme jednoduše projít v jedné smyčce, například při výpisu
- Ověření zdali implementace operací `push()` a `pop()` zachovává podmínku haldy můžeme realizovat ověřující funkcí `is_heap()`

Příklad implementace `pq_is_heap()`

```

Pro každý prvek haldy musí platit, že jeho hodnota je menší než hodnota levého a pravého následníka
typedef struct {
    int size; // the maximal number of entries
    int len; // the current number of entries
    int *cost; // array with costs - lowest cost is highest priority
    int *label; // array with labels (each label has cost/priority)
} pq_heap_s;

_Bool pq_is_heap(pq_heap_s *pq, int n)
{
    _Bool ret = true;
    int l = 2 * n + 1; // left successor
    int r = l + 1; // right successor
    if (l < pq->len) {
        ret = (pq->cost[l] < pq->cost[n]) ? false : pq_is_heap(pq, l);
    }
    if (r < pq->len) {
        ret = ret // if ret is false, further test is not performed
        && ((pq->cost[r] < pq->cost[n]) ? false : pq_is_heap(pq, r));
    }
    return ret;
}
    
```

Příklad implementace push()

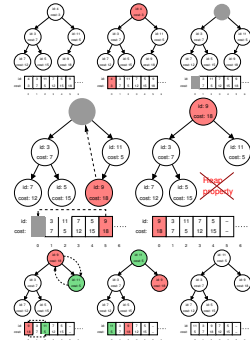
- Prvek přidáme na konec pole a iterativně kontrolujeme, zdali je splněna vlastnost haldy. Pokud ne, prvek zaměníme s předchůdcem.

```
#define GET_PARENT(i) ((i-1) >> 1) // parent is (i-1)/2
_Bool pq_push(pq_heap_s *pq, int label, int cost)
{
    _Bool ret = false;
    if (pq && pq->len < pq->size && label >= 0 && label < pq->size) {
        pq->cost[pq->len] = cost; //add the cost to the next free slot
        pq->label[pq->len] = label; //add label of new entry

        int cur = pq->len; // index of the entry added to the heap
        int parent = GET_PARENT(cur);
        while (cur >= 1 && pq->cost[parent] > pq->cost[cur]) {
            pq_swap(pq, parent, cur); // swap parent<->cur
            cur = parent;
            parent = GET_PARENT(cur);
        }
        pq->len += 1;
        ret = true;
    }
    // assert(pq_is_heap(pq, 0)); // testing the implementation
    return ret;
}
```

Příklad volání pop()

- Halda je reprezentovaná binárním polem
- Nejmenší prvek je kořenem stromu
- Voláním pop() odebíráme kořen stromu
- Na jeho místo umístíme poslední prvek
- Strom však nesplňuje podmínku haldy
- Proto provedeme záměnu s následníky
- A strom opět splňuje vlastnost haldy
- Záměny provádíme v poli a využíváme vlastnosti plného binárního stromu



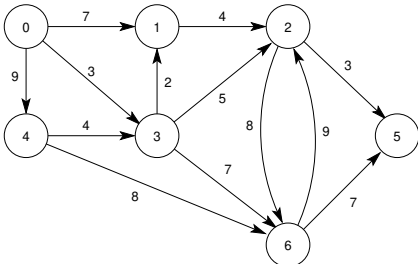
Levý potomek prvku haldy na pozici i je 2i + 1, pravý potomek je na pozici 2i + 2

Část II

Část 2 – Příklad využití prioritní fronty v úloze hledání nejkratší cesty v grafu

Hledání nejkratší cesty v grafu

- Uzly grafu mohou reprezentovat jednotlivá místa
- Hrany pak reprezentují cestu jak se mezi místy pohybovat
- Ohodnocení (cena) hrany pak může například odpovídat náročnosti pohybu mezi dvě sousedními uzly
- Cílem je nalézt nejkratší (nejlevnější) cestu z nějakého konkrétního uzlu (0) do všech ostatních uzlů

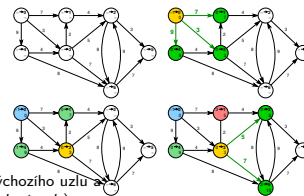


Příklad řešení úlohy hledání nejkratších cest v grafu

- Řešení úlohy se skládá z
 - Vstupních dat (grafu) – paměťová reprezentace a načtení hodnot
 - Vstupní graf je zadán jako seznam hran
 - Dalším vstupem je výchozí uzel
 - Výstupních dat (nejkratší cesty) – paměťová reprezentace a uložení (zápis)
 - Všechny nejkratší cesty vypíšeme jako seznam vrcholů s cenou (délkou) nejkratší cesty a bezprostředním předchůdcem (indexem) uzlu na nejkratší cestě z výchozího uzlu (uzel 0)
 - Algoritmu hledání cest – Dijkstrův algoritmus
 - Algoritmus je relativně přímočarý v každém kroku expandujeme uzel s aktuálně nejkratší cestou z výchozího uzlu

Dijkstrův algoritmus

- Nechť graf má pouze kladné ohodnocení hran, pak pro každý uzel
 - nastavíme aktuální cenu nejkratší cesty z výchozího uzlu
 - dále udržujeme odkaz na bezprostředního předchůdce na nejkratší cestě ze startovního uzlu
- Hledání cesty je postupná aktualizace ceny nejkratší cesty do jednotlivých uzlů
 - Začneme z výchozího uzlu (cena 0) a aktualizujeme ceny následníků
 - Následně vybereme takový uzel,
 - do kterého již existuje nějaká cesta z výchozího uzlu a zároveň
 - má aktuálně nejmenší ohodnocení
 - Postup opakujeme dokud existuje nějaký dosažitelný uzel.
 - Tj. uzel, do kterého vede cesta z výchozího uzlu
 - má již ohodnocení a předchůdce (zelené uzly).



Ohodnocení uzlů se může pouze snižovat, cena hran je nezáporná. Proto pro uzel s aktuálně nejkratší cestou již nemůže existovat cesta kratší.

Vstupní graf, reprezentace grafu a řešení

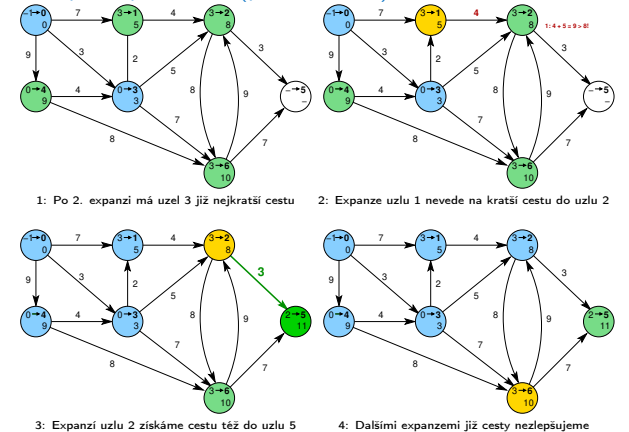
- Graf je zadán jako seznam hran v souboru, který můžeme načíst funkcí load_graph_simple() z lec10/*load_simple.c
- Graf je seznam hran


```
typedef struct {
    int from;
    int to;
    int cost;
} edge_t;

typedef struct {
    edge_t *edges;
    int num_edges;
    int capacity;
} graph_t;
```
- Navíc využijeme toho, že jsou hrany uspořádané


```
typedef struct {
    int edge_start;
    int edge_count;
    int parent;
    int cost;
} node_t;
```
- Dále potřebujeme pro vlastní řešení v každém uzlu uložit cenu nejkratší cesty cost a předcházející uzel na nejkratší cestě parent

Příklad postupu řešení (pokračování)



Datová reprezentace

- Řešení implementujeme v modulu dijkstra
- Všechny potřebné datové struktury zahrneme do jediné struktury dijkstra_t reprezentující všechny data řešení úlohy


```
typedef struct {
    graph_t *graph;
    node_t *nodes;
    int num_nodes;
    int start_node;
} dijkstra_t;
```
- Pro alokaci použijeme malloc(), allocate_graph() a inicializujeme položky struktury na výchozí hodnoty


```
void* dijkstra_init(void)
{
    dijkstra_t *dij = (dijkstra_t*)malloc(sizeof(dijkstra_t));
    if (dij) {
        dij->nodes = NULL;
        dij->num_nodes = 0;
        dij->start_node = -1;
        dij->graph = allocate_graph();
    }
    return dij;
}
```

Načtení grafu a inicializace uzlů 1/2

- Hrany načteme např. `load_graph_simple()` nebo impl. HW09.
Pro jednoduchost a lepší přehlednost zde předpokládáme bezchybné načtení
- Dále potřebujeme zjistit počet vrcholů
Lze implementovat přímo do načítání
- Alokujeme paměť pro uzly a nastavíme (bezpečně) výchozí hodnoty

```
load_graph_simple(filename, dij->graph);
int m = -1;
for (int i = 0; i < dij->graph->num_edges; ++i) {
    const edge_t *const e = &(dij->graph->edges[i]);
    m = m < e->from ? e->from : m;
    m = m < e->to ? e->to : m;
} // smyčka pro určení maximálního počtu vrcholů
dij->num_nodes = m + 1; // m je index a začíná od 0 proto +1
dij->nodes = (node_t*)malloc(sizeof(node_t) * dij->num_nodes);
for (int i = 0; i < dij->num_nodes; ++i) {
    dij->nodes[i].edge_start = -1;
    dij->nodes[i].edge_count = 0;
    dij->nodes[i].parent = -1; // pokud neexistuje indukujeme -1
    // pro cenu volíme -1 ve výpise bude kratší než např. MAX_INT
    dij->nodes[i].cost = -1;
} // nastavení výchozích hodnot uzlů
```

Inicializace uzlů 2/2

- Nastavíme indexy hran jednotlivým uzlům

```
for (int i = 0; i < dij->graph->num_edges; ++i) {
    int cur = dij->graph->edges[i].from;
    if (dij->nodes[cur].edge_start == -1) { // first edge
        // mark the first edge in the array of edges
        dij->nodes[cur].edge_start = i;
    }
    dij->nodes[cur].edge_count += 1; // increase no. of edges
}
```

Uložení řešení do souboru

- Po nalezení všech nejkratších cest (z uzlu 0) má každý uzel nastavenou hodnotu `cost` s délkou cesty a v `parent` index bezprostředního předchůdce na nejkratší cestě
Případně -1 pokud cesta neexistuje.

```
typedef struct {
    int edge_start;
    int edge_count;
    int parent;
    int cost;
} node_t;
```

Zápis řešení do souboru můžeme implementovat jednoduchým výpisem do souboru nebo implementací HW09.

```
_Bool dijkstra_save_path(void *dijkstra, const char *filename)
{
    _Bool ret = false;
    const dijkstra_t *const dij = (dijkstra_t*)dijkstra;
    if (dij) {
        FILE *f = fopen(filename, "w");
        if (f) {
            for (int i = 0; i < dij->num_nodes; ++i) {
                const node_t *const node = &(dij->nodes[i]);
                fprintf(f, "%i %i %i\n", i, node->cost, node->parent);
            } // end all nodes
            ret = fclose(f) == 0; // indicate eventual error in saving
        }
    }
    return ret;
} // lec12/dijkstra.c
```

Prioritní fronta pro Dijkstraův algoritmus

- Součástí balíku `lec12/graph_search_array` je rozhraní `pq.h` pro implementaci prioritní fronty s funkcí `update()`
- ```
void *pq_alloc(int size);
void pq_free(void *pq);
_Bool pq_is_empty(const void *pq);
_Bool pq_push(void *pq, int label, int cost);
_Bool pq_update(void *pq, int label, int cost);
_Bool pq_pop(void *pq, int *oLabel);
```
- `lec12/graph_search_array/pq.h`
- Jedná se o relativně obecný předpis, který neklade zvláštní požadavky na vnitřní strukturu  
V balíku je rozhraní implementované v modulu `pq_array-linear.c`, který obsahuje implementaci prioritní fronty polem s lineární složitostí funkcí `push()` a `pop()`
  - `lec12/graph_search_array` základní funkční řešení hledání nejkratší cesty, prioritní fronta implementována polem

## Prioritní fronta (polem) s push() a update()

- Při expanzi uzlu, můžeme do prioritní fronty vkládat uzly s cenou pro každou hranu vycházející z uzlu
- Obecně může být hran výrazně více než počet uzlů  
*Pro plný graf o n uzlech až n<sup>2</sup> hran*
- Proto pro prioritní frontu implementujeme funkci `update()` a tím zaručíme, že ve frontě bude nejvýše tolik prvků, kolik je vrcholů
- V prioritní frontě tak můžeme předalokovat maximální počet položek
- Při volání `update()` však potřebujeme získat pozici daného uzlu v prioritní frontě a změnit jeho
  - Prvek v poli najdeme lineárním průchodem prvků ve frontě  
*Budeme však mít lineární složitost*
  - Pozici prvku v prioritní frontě uložíme do dalšího pole a získáme tak okamžitý přístup za cenu mírně složitějšího vkládání prvků a vyšších paměťových nároků.  
*Operace update() bude mít výhodnou konstantní složitost.*

## Hledání nejkratších cest

- Využijeme implementaci prioritní fronty s `push()` a `update()`
- ```
dij->nodes[dij->start_node].cost = 0; // inicializace
void *pq = pq_alloc(dij->num_nodes); // prioritní fronta
int cur_label;
pq_push(pq, dij->start_node, 0);
while (!pq_is_empty(pq) && pq_pop(pq, &cur_label)) {
    node_t *cur = &(dij->nodes[cur_label]); // pro snažší použití
    for (int i = 0; i < cur->edge_count; ++i) { // všechny hrany z uzlu
        edge_t *edge = &(dij->graph->edges[cur->edge_start + i]);
        node_t *to = &(dij->nodes[edge->to]);
        const int cost = cur->cost + edge->cost;
        if (to->cost == -1) { // uzel to nebyl dosud navštíven
            to->cost = cost;
            to->parent = cur_label;
            pq_push(pq, edge->to, cost); // vložení vrcholu do fronty
        } else if (cost < to->cost) { // uzel již v pq, proto
            to->cost = cost; // testujeme cost
            to->parent = cur_label; // a aktualizujeme odkaz (parent)
            pq_update(pq, edge->to, cost); // a prioritní frontu pq
        } // smyčka přes všechny hrany z uzlu cur_label
    } // prioritní fronta je prázdná
    pq_free(pq); // uvolníme paměť
} // lec12/dijkstra.c
```

Příklad použití

- Základní implementace hledání cest s prioritní frontou implementovanou polem je dostupná v `lec12/graph_search_array`
- Vytvoříme graf `g` programem `tdijkstra` např. o max 1000 vrcholů
`./tdijkstra -c 1000 g`
- Program zkompilujeme a spustíme např.
`./tgraph_search g s`
- Programem `tdijkstra` můžeme vygenerovat referenční řešení např.
`./tdijkstra g s.ref`
- a naše řešení pak můžeme porovnat např.
`diff s s.ref`

Lineární prioritní fronta vs efektivní implementace

- Ukázková implementace v `lec12/graph_search_array`, je sice funkční, pro velké grafy je však výpočet pomalý
 - Například pro graf s 1 mil. vrcholů trvá načtení, nalezení všech nejkratších cest a uložení výsledku přibližně 120 sekund
`./tdijkstra -c 1000000 g` *Intel Skylake@3.3GHz*
`/usr/bin/time ./tgraph_search g s`
Load graph from g
Find all shortest paths from the node 0
Save solution to s
Free allocated memory
120.53 real 115.92 user 0.07 sys
 - Referenčnímu programu `tdijkstra` pouze cca 1 sekundou
Těž k dispozici jako `tdijkstra.Linux` a `tdijkstra.exe`
`/usr/bin/time ./tdijkstra g s.ref`
1.03 real 0.94 user 0.07 sys
 - Oba programy vracejí identické výsledky
`md5sum s s.ref`
MD5 (s) = 8cc5ec1c65c92ca38a8dadf83f56e08b
MD5 (s.ref) = 8cc5ec1c65c92ca38a8dadf83f56e08b
- V základní verzi řešení HW10 nesmí být hledání nejkratší cesty více než 2x pomalejší než referenční program.

Prioritní fronta haldou s push() a update()

- Prioritní frontu implementujeme haldou reprezentovanou v poli
Maximální počet prvků dopředu známe.
- Halda zaručí složitost operací `push()` a `pop()` $O(\log n)$
Oproti $O(n)$ u jednoduché implementace prioritní fronty polem.
- Je nutné udržovat vlastnost haldy. Pro kontrolu zachování „heap property“ implementujeme rozhraní `pq_is_heap()`
`_Bool pq_is_heap(void *heap, int n);`
`lec12/graph_search/pq_heap.h`
- Pro zachování složitosti operací práce s haldou potřebujeme efektivně implementovat také funkci `update()`, tj. $O(\log n)$.
 - Potřebujeme znát pozici daného uzlu v haldě
Zavedeme pomocné pole s index `heapIDX`
 - Při hledání nejkratších cest se délka cesty pouze snižuje
 - Proto se aktualizovaný „uzel“ může v haldě pohybovat pouze směrem nahoru
Jedná se tak o identický postup jako při přidání nového prvku funkci `push()`. V tomto případě však prvek může startovat z prostředka stromu.

Příklad reprezentace haldy v poli a aktualizace ceny cesty

V haldě jsou uloženy délky dosud známých nejkratších cest pro vrcholy označené: 3, 4, 5, 7, 9, a 11.

- Při expanzi dalšího uzlu jsme našli kratší cestu do uzlu 7 s délkou 5.

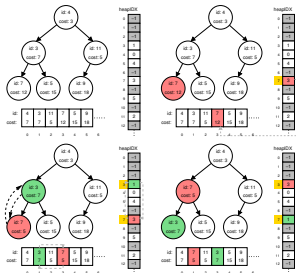
Zavoláme `update(id=7, cost=5)`

- Abychom mohli aktualizovat cenu v haldě, potřebujeme znát pozici uzlu v poli haldy.

- Proto vedle samotné haldy udržujeme pole, které je indexované číslem uzlu.

- Po aktualizaci ceny, není splněna vlastnost haldy. Provedeme záměnu.

- Při záměně udržujeme nejen prvky v samotné haldě, ale také pole `heapIDX` s pozicemi vrcholů v poli haldy.



Další možnosti urychlení programu

- Kromě efektivní implementace prioritní fronty haldou, která je zásadní, lze běh programu dále urychlit
 - efektivnějším načítáním grafu
 - a ukládáním řešení do souboru.

<code>tgraph_search s.tgs</code>	<code>tdijkstra -v g s.ref</code>	<code>dijkstra-pv g s.pv</code>
<code>lec11/tgraph_search</code>	Dijkstra ver. 2.3.4	HW10 Reference solution
Load time ...1252ms	Load time ...223ms	Load time ...235ms
Solve time ...625 ms	Solve time ...715ms	Solve time ...610 ms
Save time ...431 ms	Save time ...106ms	Save time ...87 ms
Total time ...2308ms	Total time ...1044ms	Total time ...932ms

- HW10 – Soutěž v rychlosti programu – extra body navíc
 - Na odevzdání stačí opravit funkci `update()` případně využít binární načítání a ukládání z HW09.
 - Dalšího urychlení lze dosáhnout lepší organizací paměti a datovými strukturami

Jediný zásadní požadavek je implementace rozhraní dle `lec12/dijkstra.h`

Shrnutí přednášky

Příklad implementace

- V `lec12/graph_search` je uveden příklad implementace hledání nejkratších cest s prioritní frontou realizovanou haldou
- Implementace funkce `update()` využívá pole `heapIDX` pro získání pozice prvku v haldě, záměrně je však splnění vlastnosti haldy realizováno vytvořením nové haldy s aktualizovanou cenou uzlu.

```

_Bool pq_update(void *_pq, int label, int cost)
{
    _Bool ret = false;
    pq_heap_s *pq = (pq_heap_s*)_pq;
    pq->cost[pq->heapIDX[label]] = cost; // update the cost, but heap property is not satisfied
    // assert(pq_is_heap(pq, 0));

    pq_heap_s *pqBackup = (pq_heap_s*)pq_alloc(pq->size); //create backup of the heap
    pqBackup->len = pq->len;
    for (int i = 0; i < pq->len; ++i) { // backup the heap
        pqBackup->cost[i] = pq->cost[i]; //just cost and labels
        pqBackup->label[i] = pq->label[i];
    }
    pq->len = 0; //clear all vertices in the current heap
    for (int i = 0; i < pqBackup->len; ++i) { //create new heap from the backup
        pq_push(pq, pqBackup->label[i], pqBackup->cost[i]);
    }
    pq_free(pqBackup); // release the queue
    ret = true;
    return ret;
}
    
```

Součástí řešení 10. domácího úkolu je správná implementace funkce `update()`!

Část III

Část 3 – Zadání 10. domácího úkolu (HW10)

Diskutovaná témata

- Prioritní fronta
 - Příklad implementace spojovým seznamem `lec12/priority_queue-linked_list`
 - Příklad implementace polem `lec12/priority_queue-array`
- Halda - definice, vlastnosti a základní operace
- Reprezentace binárního plného stromu polem
- Prioritní fronta s haldou
- Hledání nejkratší cesty v grafu – využití prioritní fronty (resp. haldy)

Příklad řešení a rychlost výpočtu

- Po úpravě funkce `update()` získáme prioritní frontu se složitostí operací $O(\log n)$ a vlastní výpočet bude relativně rychlý.
- Pro získání představy rychlosti výpočtu je v souboru `tgraph_search-time.c` volání dílčích funkcí modulu `dijkstra` s měřením reálného času (`make time`). `lec12/graph_search-time.c`

Alternativně lze řešit nástrojem `time` nebo pro Win platformu `lec12/bin/timeexec.exe`

- Vytvoříme graf o 1 mil. uzlů (a cca 3 mil. hran) v souboru `/tmp/g`
`./bin/tdijkstra -c 1000000 /tmp/g`

Verze s naivním <code>update()</code>	Upravená funkce <code>update()</code>
<code>tgraph_search-time /tmp/g /tmp/s1</code>	<code>tgraph_search-time /tmp/g /tmp/s2</code>
Load graph from /tmp/g	Load graph from /tmp/g
Load time ...1179ms	Load time ...1201ms
Save solution to /tmp/s1	Save solution to /tmp/s2
Solve time ...965875 ms	Solve time ...620 ms
Save time ...273 ms	Save time ...279 ms
Total time ...967327ms	Total time ...2100ms

- Správnost řešení lze zkontrolovat programem `tdijkstra`, např.
`./bin/tdijkstra -t /tmp/g /tmp/s`

Zadání 10. domácího úkolu HW10

Téma: Integrace načítání grafu a prioritní fronta v úloze hledání nejkratších cest

Povinné zadání: **3b**; Volitelné zadání: **3b**; Bonusové zadání: *Soutěž o body*

- **Motivace:** Větší programový celek, využití existujícího kódu a efektivní implementace programu
- **Cíl:** Osvojit si integraci existujícího kódu do funkčního celku složeného z více souborů.
- **Zadání:** <https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw10>
 - Funkce `update()` pro efektivní použití prioritní fronty implementované haldou v úloze hledání nejkratší cest v grafu.
 - **Volitelné zadání** rozšiřuje binární načítání/ukládání grafu o specifikovaný binární formát, tj. rozšíření HW 09.
 - **Bonusové zadání** spočívá v efektivnosti implementace tak, aby byl výsledný kód co možná nejrychlejší.
- **Termín odevzdání:** **05.01.2019, 23:59:59 PST**