Western Science

University Of Western Ontario Canada

Alpha Path-Moves v1.0

with Hedgehog Prior support

December 1, 2017

Contents

1 Legal Disclaimer

2 Download Link and Minimum Requirements

The latest version of the library could be found on Github **here???**, please download the latest *release*.
Minimum Requirements:

- 64-bit machine
- C++11 compiler (the library was only test on Windows System)
- GPU: 2Gb and CUDA architecture sm_30 or higher
- Cmake v3.9.4 or higher

3 Library Wrappers

A Matlab wrapper is currently in progress. A Python wrapper (hopefully in near future).

4 Notation

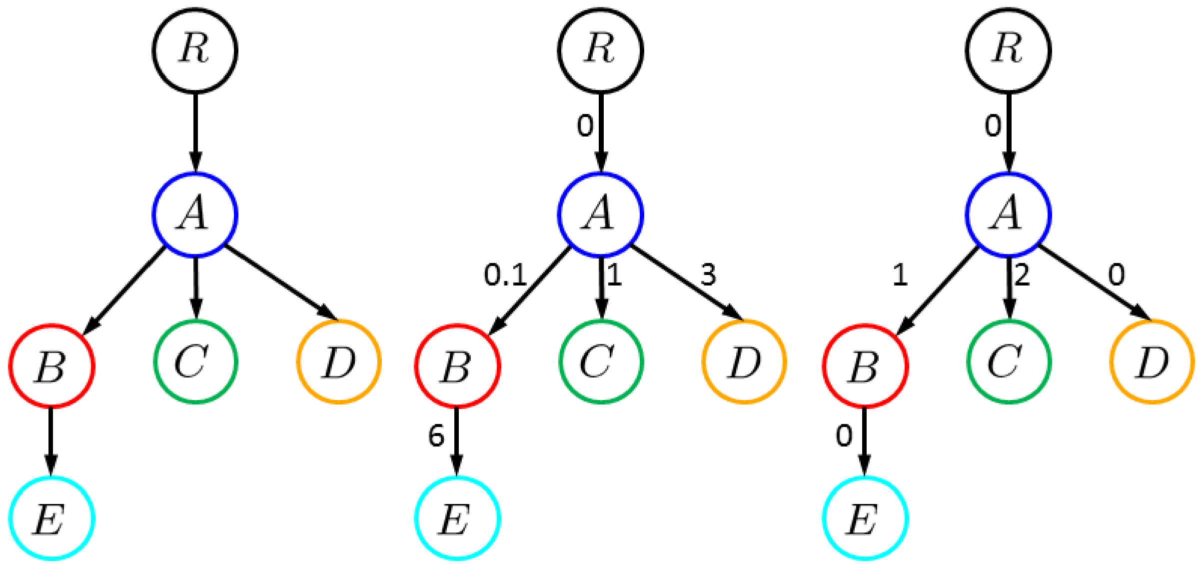
In order to be able to use this Library the user should have basic understanding of [1] and [2] (if needed) which introduced `AlphaPathMoves` and Hedgehog shape prior. Table 1 shows some of the notation used in [1] as reference to simplify the tie between the library and [1].

Symbol	Definition
Ω	set of voxels in the volume to be segmented.
$\mathcal{L}$	set of labels which correspond to objects of interest, e.g. liver, kidney etc.
f_p	label assigned to voxel p, i.e. $f_p \in \mathcal{L}$.
$\mathbf{f}$	a labeling, i.e. $f = \{f_p \mid \forall p \in \Omega\}$.
$D_p(f_p)$	cost of assigning voxel p to label f_p, usually this is the *-log-likelihood* of voxel p belonging to label f_p. If label f_p is modeled as a Gaussian $(\mu_{f_p}, \sigma_{f_p})$ then its corresponding data term will be $$D_p(f_p) = -log\left(\frac{1}{\sqrt{2\pi\sigma^2_{f_p}}} x^{\frac{(I_p-\mu_{f_p})^2}{\sigma^2_{f_p}}}\right) \quad \forall p \in \Omega$$ where I_p is the volume intensity at voxel p.
λ	a normalization constant between the data and smoothness terms.
$\mathcal{N}$	set of all neighboring pixels, used in penalizing spatial discontinuities.
w_{pq}	discontinuity cost of assigning neighboring pixels p and q to two different labels.
$V(f_p, f_q)$	discontinuity cost of a pair of neighboring pixels to labels f_p and f_q.
$w_{pq}V(f_p, f_q)$	composite cost of assigning neighboring pixels p and q to labels f_p and f_q.
w_∞	a prohibitively expensive cost. In theory, $w_\infty = \infty$.
$\mathcal{T}$	hierarchical tree defined over the set of the set of labels $\mathcal{L}$.
$\mathcal{T}(\ell)$	subtree of $\mathcal{T}$ rotated at label ℓ.
$\mathcal{P}(\ell)$	parent of ℓ in tree $\mathcal{T}$.
δ_ℓ	minimum margin constraint around label ℓ.
$\mathcal{S}_\ell$	a set of pairs of ordered voxels used to approximate Hedgehog constraints of label ℓ.

Table 1: notations of [2, 1] provided as a reference.

Recall, the HINTS objective [1] is to minimize the following energy

$$
\begin{aligned}
E(\mathbf{f}) = \overbrace{\sum_{p\in\Omega} D_p(f_p)}^{\textit{data}} + \overbrace{\lambda \sum_{pq\in\mathcal{N}} w_{pq} V(f_p, f_q)}^{\textit{smoothness}} + \overbrace{w_\infty \sum_{\ell\in\mathcal{L}} \sum_{\substack{p\in\Omega \\ f_p\in\mathcal{T}(\ell)}} \sum_{\substack{q\in\Omega \\ \|p-q\|<\delta_\ell}} [f_q \notin \{\mathcal{T}(\ell) \cup \mathcal{P}(\ell)\}]}^{\textit{interaction constraints}} \\
+ \overbrace{w_\infty \sum_{\ell\in\mathcal{L}} \sum_{\substack{p\in\Omega \\ f_p\in\mathcal{T}(\ell)}} \sum_{pq\in\mathcal{S}_\ell} [f_q \notin \mathcal{T}(\ell)]}^{\textit{hedgehog constraints}} .
\end{aligned} \tag{1}
$$



(a) hierarchical tree $\mathcal{T}$ (b) tree-metric smoothness V (c) min. margin constraints δ_ℓ

Figure 1: (a) show a sample hierarchical tree $\mathcal{T}$ that we will be using through out this guide. In (b) the tree-metric smoothness function is shown as weights on $\mathcal{T}$, e.g. $V(B, A) = 0.1$ while $V(R, A) = 0$. (c) shows the min-margins constraints for each label, e.g. $\delta_E = 0$ and $\delta_C = 2$.

5 HINTS Example

Now we will describe how to use the `AlphaPathMoves` library. Let us assume that we are trying to segment a volume of size $6 \times 5 \times 4 = 120$. Also, let the hierarchical tree $\mathcal{T}$ be the one shown in Fig. 1(a). The smoothness function V and min. margin constraints are shown in Fig.1(b) and (c), respectively. `AlphaPathMoves` accepts most of its input as arrays. Those arrays are passed via array wrappers, i.e. `Array2D` and `Array3D`, which includes a pointer to the array and the array's dimensions. Please read Appendix A to know how to create and use `Array2D` and `Array3D` wrappers.

6 C++ Library

6.1 Template Types

```
template <class captype, class tcaptype, class flowtype>
AlphaPathMoves<captype, tcaptype, flowtype>
```

`AlphaPathMoves` is a templated class and you are required to specify the following data types

- `captype` : data type used to hold pairwise potentials, e.g. $w_{pq}V(f_p, f_q)$
- `tcaptype` : data type used to hold unary potentials, e.g. $D_p(f_p)$
- `flowtype` : data type used to hold the overall flow value, i.e. $E(\mathbf{f})$.

> `flowtype` can not be smaller than `captype` or `tcaptype`. It is best to set them all to be `int64_t`.

6.2 Constructor

```
AlphaPathMoves( int64_t in_dims[3],  uint32_t in_nlabels)
```

- `in_dims` : the dimensions of the volume to be segmented.
- `in_nlabels` : the number of labels in the hierarchical tree $\mathcal{T}$.

For the HINTS example in Fig. 1, `in_dims` $\leftarrow \{6, 5, 4\}$ and `in_nlabels` $\leftarrow$ `6.`

> `AlphaPathMoves` only accepts 3D volumes. To segment 2D image simply pass it as a 3D volume consisting of one slice.

6.3 Data Terms

```
void setDataTerms(const Array2D<tcaptype> * in_dataterms)
```

- `in_dataterms` : is a pointer to 2D array wrapper, i.e. `Array2D`, see Appendix **X** on how to create 2D (or 3D) array wrappers. `in_dataterms->data` is a pointer to an array of size `n_lables`$\times v$ where v is total number of voxels. For voxel p and label index ℓ

$$\texttt{in\_dataterms->data}[v \times \ell + p] \leftarrow D_p(\ell)$$

For the HINTS example in Fig. 1, Fig. 2 shows the `in_dataterms->data` 2D array layout.

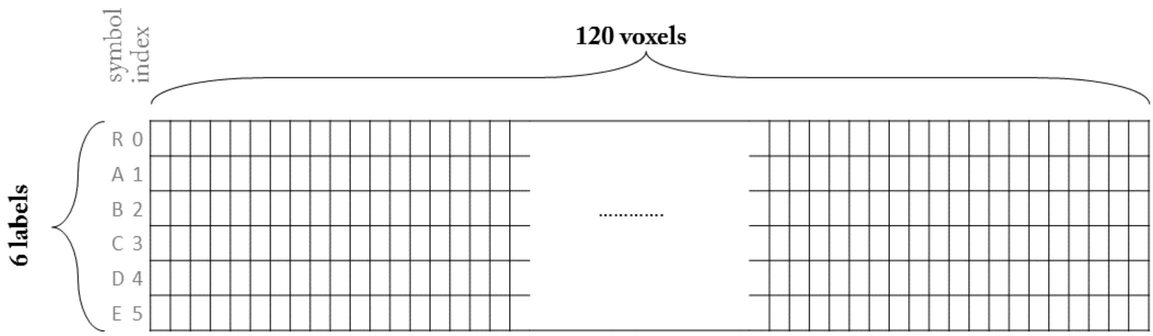


Figure 2: shows the row-major `in_dataterms->data` array that contains data terms for the example shown in Fig. 1.

> Note the label ordering/indexing, e.g. label R is indexed by 0 and D is indexed by 4. Regardless of the chosen label ordering, you must adhere to the same ordering when calling `AlphaPathMoves` functions.

6.4 Smoothness Prior

```
void setTreeWeights(double lambda,
    const Array2D<double> * in_partialTreemetric)
```

- `lambda` : is the normalization constant λ in (1)
- `in_partialTreemetric` : is a pointer to a 2D array wrapper. `in_partialTreemetric ->data` is a pointer to an array of size `n_lables×n_lables` that encodes the hierarchical tree $\mathcal{T}$ and the non-negative V weights.

For the HINTS example shown in Fig. 1, `in_partialTreemetric->data` will be as shown in Fig. 3.

child

parent	R	A	B	C	D	E
R	-1	0	-1	-1	-1	-1
A	-1	-1	0.1	1	3	-1
B	-1	-1	-1	-1	-1	6
C	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	-1	-1
E	-1	-1	-1	-1	-1	-1

Figure 3: shows the expected `in_partialTreemetric->data` array which encodes (1) the adjacency matrix of the hierarchical tree $\mathcal{T}$ where -1 means not connected, and (2) the smoothness weights of V which are given in Fig. 1(b). Notice that array uses the same label ordering/indexing used in Section 6.3.

```
void setSmoothnessNeighbourhoodWindowSize(uint32_t in_winsize)
```

- `in_winsize` : is the window size used in generating the neighborhood set $\mathcal{N}$. This must be an odd number. For example, if `in_winsize` was set to 3 then any voxel within the $3 \times 3 \times 3$ cube around voxel p will be considered its neighbour.

```
void setWpqFunction(
    double(*in_getWpq)(int64_t, int64_t, const void * ),
    const void * wpq_extra_data);
```

- `in_getWpq` : is a pointer to function that will be called to computer w_{pq}. Indices p and q will be sent to this function in addition to `wpq_extra_data`.
- `wpq_extra_data` : is pointer to any extra data you would like to have access to when computing w_{pq}. Note, the caller owns the `wpq_extra_data` memory resource.

Calling `setWpqFunction` is *optional*, by default w_{pq} is 1. Listing 1 shows a w_{pq} function that would always return 1, which is equivalent to the default behavior when `setWpqFunction` is not called. Listing 2 shows a w_{pq} function that would prefer discontinuities to occur along canny edges (which are passed as extra data).

Listing 1: Constant w_{pq}

```
double getWpq_Euclidean(int64_t p, int64_t q,
    const void * data)
{
  return 1;
}
```

Listing 2: w_{pq} as a function of Canny Edges

```
double getWpq_Canny( int64_t p, int64_t q, const void * data)
{
  const uint32_t * cannydata = reinterpret_cast<const uint32_t
      *>(data);
  if (!cannydata)
    return 1; //return 1 if cannydata is nullptr
  if ( (cannydata[p] == 1 && cannydata[q] == 1) || \
       (cannydata[p] == 0 && cannydata[q] == 0)))
    return 1;
  return 0.125;
};
```

6.5 Initial Labeling

```
void setInitialLabeling(const Array3D<uint32_t> * in_labeling)
```

- `in_labeling` : is a pointer a 3D array wrapper. `in_labeling->data` is a pointer to an array containing the initial labeling, such that `in_labeling->data[p]` is initial label of voxel p.

```
void setInitialLabeling(uint32_t lbl_id) //overloaded
```

- `lbl_id` : all voxels' initial label will be set to `lbl_id`.

`AlphaPathMoves` *requires a valid initial solution*. Thus, if using Hedgehogs then the initial labeling can not violate the Hedgehog scribbles passed along in the Hedgehog mask, see Section 6.7 for more details. `AlphaPathMoves` could easily recover from solutions with invalid min. margins.

6.6 Min. Margin Constraints

Calling this function is *optional*. If this function was not called it will be assumed that there are no min-margins for all labels.

```
void setMinimumMargins(const Array2D<uint32_t> *
   in_minmargins_radii, MinMarginMethod in_mm_method)
```

- `in_minmargins_radii` : is a pointer to a 2D array wrapper. `in_minmargins_radii->data` is a pointer to an array of size `n_labels`. `in_minmargins_radii->data[`ℓ`]` is the min-margin constraint radius for label ℓ.

- `in_mm_method` : is the method used to compute min-margin constraints. There are three options

   1. `IDX_BASED`: fast but not memory efficient (not recommended).
   2. `BIT_BASED`: slow but memory efficient, uses CPU (not recommended).
   3. `CUDA_BASED`: fast and memory efficient, uses GPU (recommended), this is the default setting.

For the HINTS example Fig.1, `in_minmargins_radii->data` $\leftarrow \{0, 0, 1, 2, 0, 0\}$ assuming that the labels ordering is (R, A, B, C, D, E) as illustrated in Section 6.3. Furthermore, a 0 min. margin means no min. margin.

6.7 Hedgehog Shape Prior

Calling `setHedgehogAttributes` is *optional*—use it only when you want to enforce Hedgehog shape prior [2].

```
void setHedgehogAttributes(uint32_t hhog_windowsize
                        , const Array3D<int32_t> * mask
                        , double theta)
```

The following parameters are used to generate the set of pairs of ordered voxels used to approximate Hedgehog constraints, i.e. $\mathcal{S}_\ell$

- `hhog_windowsize`: neighborhood window size (odd number) used in approximating the Hedgehog shape prior. A good choice would be 5 (recommended for 3D) or 7 (recommended for 2D) but ultimately the choice should depend on the volume's resolution. The lower the resolution the smaller `hhog_windowsize` should be.

- `mask`: is a pointer to a 3D array wrapper. `mask->data` is a pointer to an array with the same size as the volume being segmented. `mask->data` is a label-map of the user-scribbles that will used to generate the Hedgehog vector fields such that

$$\texttt{mask->data[p]} = \begin{cases} \ell & \text{voxel } p \text{ is part of label } \ell \text{ scribble} \\ -1 & \text{voxel } p \text{ is not part of any label's scribble.} \end{cases}$$

- `theta`: the shape tightness parameter θ in (1). Due to discretization artifacts discussed in [3], θ should be $45° \pm 20°$.

AlphaPathMoves *requires a valid initial solution*. Thus, if using Hedgehogs then the initial labeling can not violate the Hedgehog scribbles in mask. Simply make sure that the dataterms encodes the Hedgehog scribbles as hard constraints. That is for voxel p, if mask->data[p] is ℓ then

$$\texttt{dataterms->data[}v \times k + p\texttt{]} = \begin{cases} 0 & \text{if } k = \ell \\ w_\infty & \text{otherwise.} \end{cases}$$

where v is the total number of voxels, $k \in [0, \texttt{n\_labels} - 1]$ and $w_\infty >> E(\mathbf{f})$. AlphaPathMoves computes w_∞ based on the declared captype, call getLargePenalty to retrieve the automatically calculated w_∞.

6.8 Expansion Ordering

Calling setExpansionOrdering is *optional*. This function sets the order in which labels are allowed to expanded. This is useful in two cases. First, to be able to replicate the results since AlphaPathMoves is an iterative approximate algorithm and the final result depends on the order in which labels are allowed to expand. Second, when $\mathcal{T}$ consists of a single path where it is possible to compute the global optimal solution. If you called setExpansionOrdering multiple times only the last call be taken into consideration.

```
void setExpansionOrdering(int32_t seed) //overloaded
```

- seed : the random seed that will be used to generate a random expansion ordering of the labels.

```
void setExpansionOrdering(const Array2D<uint32_t>* lbls_order)
```

- lbls_order : is a pointer to a array wrapper. lbls_order->data is a pointer to a an array of size n_labels such that

  lbls_order->data[0] $\leftarrow$ index of the first label allowed to expand

  lbls_order->data[1] $\leftarrow$ index of the second label allowed to expand

  .... etc.

For the HINTS example shown in Fig. 1, we could set the expansion ordering to (R, B, D, A, C, E) by setting lbls_order->data to be $(0, 2, 4, 1, 3, 5)$. Label indices are covered in the note in Section 6.3.

```
void setExpansionOrdering(int32_t str_id, int32_t end_id)
```

- str_id: the index of the label at the start of the chain
- end_id: the index of the label at the end of the chain

When $\mathcal{T}$ is chain AlphaPathMoves can find the global optimal solution. Currently, AlphaPathMoves has no internal mechanism to automatically detect that tree $\mathcal{T}$ is a chain. As such, you simply need to pass the indices of the labels at the start and end of the chain.

6.9 Optimization

```
Array3D<uint32_t> * runPathMoves(
    MaxflowSolver::SolverName solvername,
    flowtype & final_energy)
```

- `solvername`: there are three available solvers
    1. `BK`: Boykov-Kolomogrov maxflow algorithm [4]. Memory efficient but slower than IBFS.
    2. `IBFS`: Incremental Breadth First Search maxflow algorithm [5]. Usually, faster than BK but not memory efficient (recommended solver).
    3. `QPBO_SLVR`: Quadratic Pseudo Boolean Optimization solver [6], similar performance to BK.
- `final_energy`: energy $E(\mathbf{f})$ of the final labeling $\mathbf{f}$ that AlphaPathMove converged to.
- `returned value`: a poitner to a 3D array wrapper say `Output`. `Output->data` is pointer to an array that carries the final labeling $\mathbf{f}$ that `AlphaPathMoves` converged to, such that `Output->data[p]` is the index of label f_p, see note in Section 6.3 regarding label indices/ordering.

> After calling `runPathMoves` it is possible to call `setInitialLabeling` and/or `setDataTerms` to replace the current initial labeling and/or data terms, respectively. It is advisable to reuse an AlphaPathMove instance to avoid (de)allocating memory.

6.10 Thread Safe Logging

`AlphaPathMoves` is thread safe and it uses a native thread safe logger to log time and memory information, errors, and warnings encountered during runtime. To enable logging include `TS_Logger.h` in your main and call `TS_Logger::LogAllEvents();`. To disable logging call `TS_Logger::UnlogAllEvents();`. Logging is enabled by default.

In addition to the displayed output, logged events are dumped in a file called `ts_log.dat`. Finally, the logged events are prefixed by the thread id. It is *highly recommended* to enable all logging.

7 Matlab Wrapper

7.1 Compiling MEX file

7.2 Wrapped C++ Functions

7.3 Limitations

A Array Wrappers

A.1 Array2D

`Array2D` is a C++ templated structure to wrap 2D arrays. The wrapper provides access to the array pointer directly, and other useful functions, e.g. you could load/save 2D arrays from/to .mat [1] files which are native to Matlab.

Members:

- `X`: the size of the X dimension (number of rows)
- `Y`: the size of the Y dimension (number of columns)
- `total_size`: the number of elements in the array, i.e. `X*Y`
- `data`: a pointer to the row-major array
- `constX`: is the row-step size, used to access elements.

Creating A Wrapper:

```
Array2D(T * in_array, int64_t X, int64_t Y)
```

- `in_array`: a pointer to the wrapped array of type `T`. The created `Array2D` instance assumes ownership of `in_array` memory resource and it will be responsible for freeing that resource
- `X`: the size of the X dimension (number of rows)
- `Y`: the size of the Y dimension (number of columns).

```
Array2D(void);
```

This will create an empty array. Later on you could allocate an array by calling `allocate`.

```
void allocate(int64_t X, int64_t Y)
```

This functions allocates memory of size `X*Y`. This function also acts as resize, calling it will clear the previously wrapped array.

Loading .mat Array:

```
Array2D(std::string filepath)
```

- `filepath`: the path to the file where the array is stored.

For example,

```
Array2D<uint64_t> my_array("C:\\saved_array.mat");
```

would load the array saved in `"C:\saved_array.mat"` into `my_array`.

> The .mat file should contain only one 2D array of the *same type* as declared template type, otherwise the wrapper will throw an exception. If there is more than one array in the .mat file, only the first array will be considered.

[1] Supports Matlab Level 5 only, i.e. save in Matlab using '-v6'.

Saving .mat Array:

```
void saveToFile(std::string filepath)
```

- `filepath`: the path to the file where the array will be stored.

Accessing Elements:
The fastest way to access an element is via the `data` pointer. For example if `ar` is an `Array2D` then element `(x,y)` could be accessed through `ar.data[x+y*ar.constX]`. Alternatively, you could use the `[]` operators, as such `ar[x][y]`. However, accessing array elements in this way is slow.

A.2 Array3D

`Array3D` is a C++ templated structure to wrap 3D arrays. The wrapper provides access to the array pointer directly, and other useful functions, e.g. you could load/save 3D arrays from/to .mat [2] files which are native to Matlab.

Members:

- `X`: the size of the X dimension (number of rows)
- `Y`: the size of the Y dimension (number of columns)
- `Z`: the size of the Z dimension (number of slices)
- `total_size`: the number of elements in the array, i.e. `X*Y*Z`
- `data`: a pointer to the row-major array.
- `constX`: is the row-step size, used to access elements
- `constX`: is the slice-step size, used to access elements.

Creating A Wrapper:

```
$Array2D(T * in_array, int64_t X, int64_t Y, int64_t Z)
```

- `in_array`: a pointer to the wrapped array of type `T`. The created `Array3D` instance assumes ownership of `in_array` memory resource and it will be responsible for freeing that resource.
- `X`: the size of the X dimension (number of rows)
- `Y`: the size of the Y dimension (number of columns)
- `Z`: the size of the Z dimension (number of slices).

```
Array3D(void);
```

This will create an empty array. Later on you could allocate an array by calling `allocate`.

```
void allocate(int64_t X, int64_t Y, int64_t Z)
```

This functions allocates memory of size `X*Y*Z`. This function also acts as resize, calling it will clear the previously wrapped array.

[2] Supports Matlab Level 5 only, i.e. save in Matlab using '-v6'.

Loading .mat Array:

```
Array3D(std::string filepath)
```

- `filepath`: the path to the file where the array is stored.

For example,

```
Array3D("C:\saved_array.mat");
```

would load the array saved in `"C:\saved_array.mat"` into `my_array`.

> The .mat file should contain only one 3D array of the *same type* as declared template type, otherwise the wrapper will throw an exception. If there is more than one array in the .mat file, only the first array will be considered.

```
void saveToFile(std::string filepath)
```

- `filepath`: the path to the file where the array will be stored.

Accessing Elements:
The fastest way to access an element is via the `data` pointer. For example if `ar` is an `Array3D` then element `(x,y,z)` could be accessed as `ar.data[x+y*ar.constX+z*ar.constXY]`. Alternatively, you could use the `[]` operators, as such `ar[x][y][z]`. However, accessing array elements in this way is slow.

B Smoothness: Potts Model to Tree-Metric

`AlphaPathMoves` could only be used for tree-metric smoothness functions. However, one of the most widely used smoothness functions in Computer Vision is Potts model, which is not tree-metric. Nonetheless, it is still possible to use `AlphaPathMoves` with (or without) Hedgehog priors for Potts model smoothness. The main idea is to create a tree-metric function V_t that is equivalent to Potts model.

Let $\mathcal{L}$ denote the set of labels. For Potts model smoothness,

$$V(\ell, k) = \begin{cases} 1 & \text{if } \ell = k \\ 0 & \text{otherwise} \end{cases}$$

where ℓ and $k \in \mathcal{L}$. We could use `AlphaPathMoves` on $\mathcal{L}_t = \{O, \mathcal{L}\}$ where O is an auxillary forbidden[3] label, and

$$V_t(\ell, k) = \begin{cases} \frac{1}{2} & \text{if } k = O \text{ and } \ell \in \mathcal{L} \\ \frac{1}{2} & \text{if } k \in \mathcal{L} \text{ and } \ell = O \\ 1 & \text{if } k, \ell \in \mathcal{L} \text{ and } k \neq \ell \\ 0 & \text{if } k, \ell \in \mathcal{L} \text{ and } k = \ell. \end{cases}$$

As you can see, V and V_t are equivalent over $\mathcal{L}$ yet V_t is tree-metric over $\mathcal{L}_t$. Futhermore, no voxel could be assinged to O since it is a forbidden label. The hierarchical tree and weights of V_t is shown in Fig.4.

[3] $D_p(O) = w_\infty$

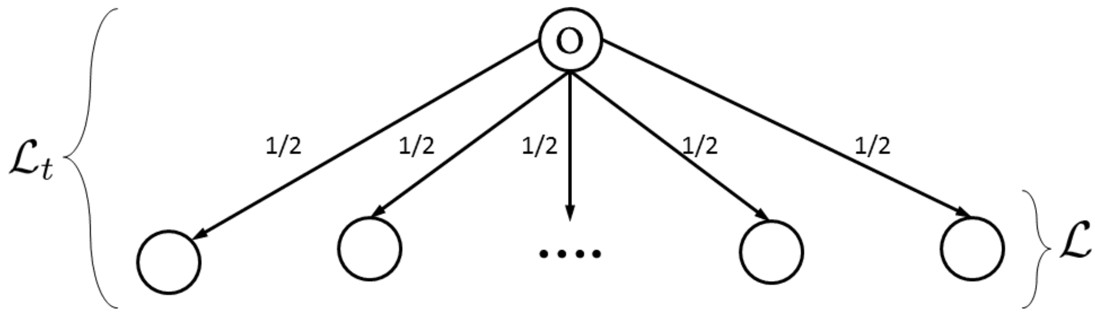


Figure 4: shows the hierarchical tree and smoothness V_t weights over labels $\mathcal{L}_t$.

References

[1] Hossam Isack, Olga Veksler, Ipek Oguz, Milan Sonka, and Yuri Boykov. Efficient optimization for hierarchically-structured interacting segments (HINTS). In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[2] Hossam Isack, Olga Veksler, Milan Sonka, and Yuri Boykov. Hedgehog shape priors for multi-object segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[3] Hossam Isack, Olga Veksler, Ipek Oguz, Milan Sonka, and Yuri Boykov. Efficient optimization for hierarchically-structured interacting segments (HINTS). Technical report, 2017.

[4] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[5] Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert E. Tarjan, and Renato F. Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015: 23rd Annual European Symposium, Proceedings*, 2015.

[6] Carsten Rother, Vladimir Kolmogorov, Victor Lempitsky, and Martin Szummer. Optimizing binary mrfs via extended roof duality. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.