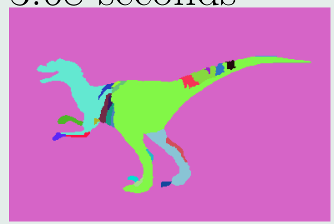
Home Page
Title Page
Contents
Page 1 of 19
Go Back
Full Screen
Close
Quit

Overview

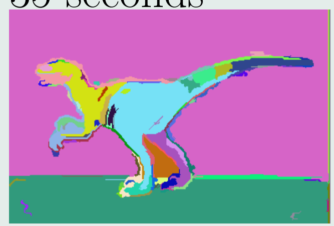
- The context of the problem
- Nearest related work
- Our contributions
- The intuition behind the algorithm
- Some details
- Qualtitative, quantitative results and proofs
- Conclusion

Original

3.68 seconds



35 seconds



Context of Segmentation

- We want to take an image as input and produces regions of which are homogeneous
  - A *good* segmentation should result.
  - Algorithm should run fast
  - Regions should reflect global properties

Good Segmentation

- Given V, find a partition $S = \{C_1, C_2, \ldots, C_n\}$
- Let $D(C_i, C_j)$ denote a pairwise comparision Boolean function that is true if there is an evidence that the pair belongs to different components
- A segmentations T is a *refinement* of S when $\forall C \in T$, $\exists C' \in S$ such that $C \subset C'$
- A segmentation S
  - Is *too fine* where there is some pair of regions C_k, C_l for which D is false.
  - Is *too coarse* when there exists a proper refinement of S that is **not** too fine.
  - Is *good* if it is not too fine nor too coarse
- For any set V, there exists some good segmentation S

Graph Segmentation

- Let $G = (V, E)$ be a weighted undirected grid graph corresponding to the image.
- Each edge $(v_i, v_j) \in E$ has a corresponding weight w which is a non negative measure of the difference between neighbouring elements
- To define D use the difference along the boundary of two components *relative* to the difference between neighbouring elements internal to each component
  - Define $Int(C) = \max_{e \in MST(C,E')} w(e)$.
  - Define $Dif(C_1, C_2) = \min_{v_i \in C_1, v_j \in C_2, (v_i, v_j) \in E} w((v_i, v_j))$
  - $D(C_1, C_2) = 1$ if $Dif(C_1, C_2) > MInt(C_1, C_2)$ where
  - $MInt(C_1, C_2) = \min(Int(C_1 + \tau(C_1)), Int(C_2) + \tau(C_2)$ and
  - $\tau(C) = k/\|C\|$ (k is a constant)
- $\tau(C)$ can be any non-negative function of C
- Repeatedly merge components C_1 and C_2 if

Algorithm K

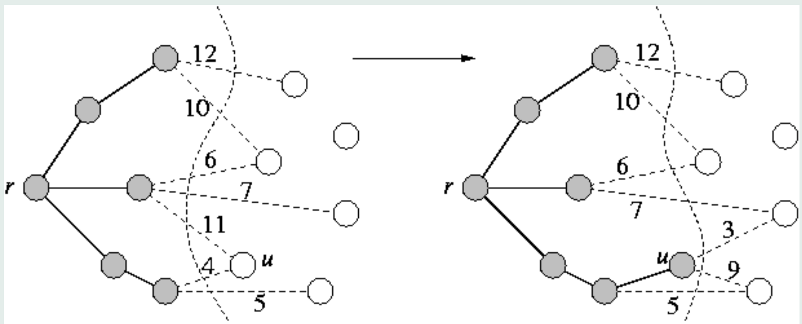
1. Sort E into $\pi = (o_1, o_2, \ldots, o_k)$ by non-decreasing edge weight.
2. Start with F^0 where each vertex is its own component.
3. Construct F^q given F^{q-1}
   - Let edge o_q connects vertices v_i and v_j, and let $v_i \in C_p$ and $v_j \in C_q$
   - Verify $C_p \neq C_q$. If equal proceed to next edge.
   - If $w(o_q) \leq Mint(C_p, C_q)$ (is small compared to the internal variation) then $F^q = F^{q-1} \cup \{o_q\}$ else $F^q = F^{q-1}$
   - Repeat above step for all edges
4. Return $S = F^k$

Our contributions

1. Uses a notion of a seed point and grows a region based on the seed. The seed is normally automatically chosen; however, when necessary, it supports segmenting only a part of a large image.
2. Uses identical parameters to those in Algorithm K. Since regions are grown sequentially, 'what if' analysis by varying the parameters is easier, and a segmentation can be abandoned earlier.
3. Algorithm K runs in $O(E \log E)$ time if there are E edges. In modeling non-grid graphs, the algorithm requires E to be $O(n)$ so that the overall algorithm runs in "almost" linear time. By using the Prim variation on MST and Fibonacci heaps, alternate algorithm has a theoretical running time of $O(E + n \log n)$ time. Therefore, there is no linearity requirement if the segmentation is to be performed in feature space.
4. Even without using Fibonacci heaps, implementation shows a faster running time in 82 out of 100 cases in images of size 768×768.

Intuition Behind Algorithm

- Start creating components by choosing a seed point
- Keep candidates for seed points in a priority queue Q2
- Decide to grow a component based on "how it interfaces with the outside world" using light edges
  - If chosen edge is too strong compared to the internal strength, stop the growth and pick another seed from Q2
  - Otherwise, update light edges (using queue Q1)
  - Don't forget to delete candidate seed points from Q2



- Algorithm P1 uses only one queue

Algorithm P2

```
overall () {
 initQ₂();
 for v ∈ V do {
  key[v] = ∞;
  insertQ₁(v, key[v]);
 }
 i = 0;
 while (Q₂ ≠ { }) {
  s = findMin (Q₂);
  Q₁.dec(s, 0);
  grow (s, i);
  i = i+1;
 }
}
```

```
initQ₂ () {
 for v ∈ V do {
  x = minAdjacent(v);
  insertQ₂ (v,x);
 }
}
```

Algorithm P2

```
grow (s, i) {
  done = false;
  C_i = makeSet();
  while not done do {
    u = findMin(Q_1);
    if (causesMerge(u, i)) {
      C_i = C_i ⋃ u;
      updateAdj(u);
      delete(Q_1,u);
      delete(Q_2,u);
    }
    else done = true;
  }
}
```

```
causesMerge(u, i) {
  if (key[u]<int(C_i) + τ)
    return TRUE;
  else return FALSE;
}
```

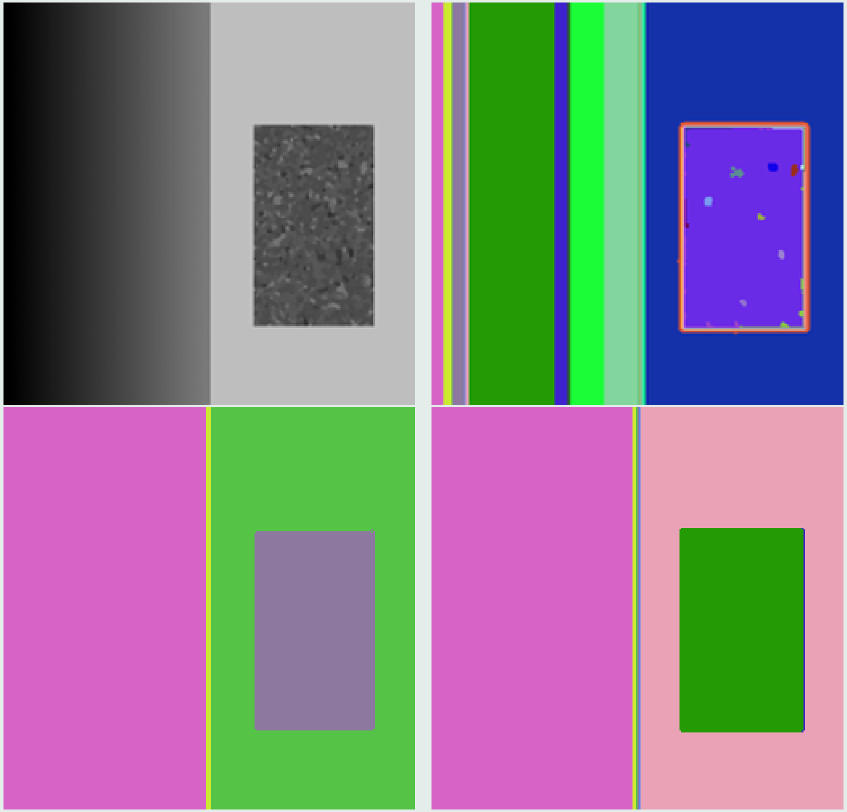
```
updateAdj(u) {
  for each v ∈ adj(u) {
    if (w ∈ Q_1 and
      w(u,v) < key[v]) {
      key[v] = w(u,v);
      Q_1.dec(v, key[v]);
    }
  }
}
```

Worst Case Asymptotic Time Complexity

- The overall algorithm picks items for growth from the priority queue Q2 and runs `grow()`
- The time complexity of `grow()` depends on
  - Time to perform an arbitrary delete in Q2 ($O(\log s_i)$ if the number of times `causesMerge()` is true is s_i)
  - Time to update keys in Q1 (degree(u)$O(1)$ if Fibonacci heaps are used, otherwise degree(u)$O(\log n)$)
  - Time to implement the addition of u in C_i (which can be performed as part of the `updateAdj()`)
- Total time for `grow()` is $\sum_{i \in C_i} degree(i) + s_i \log s_i$
- Overall time is less than $O(E) + \sum_{k \in C_k} s_k \log s_k$ which is $O(E + n \log n)$ when there are E edges in the graph and n pixels

Qualitative Results: 3 Component Image

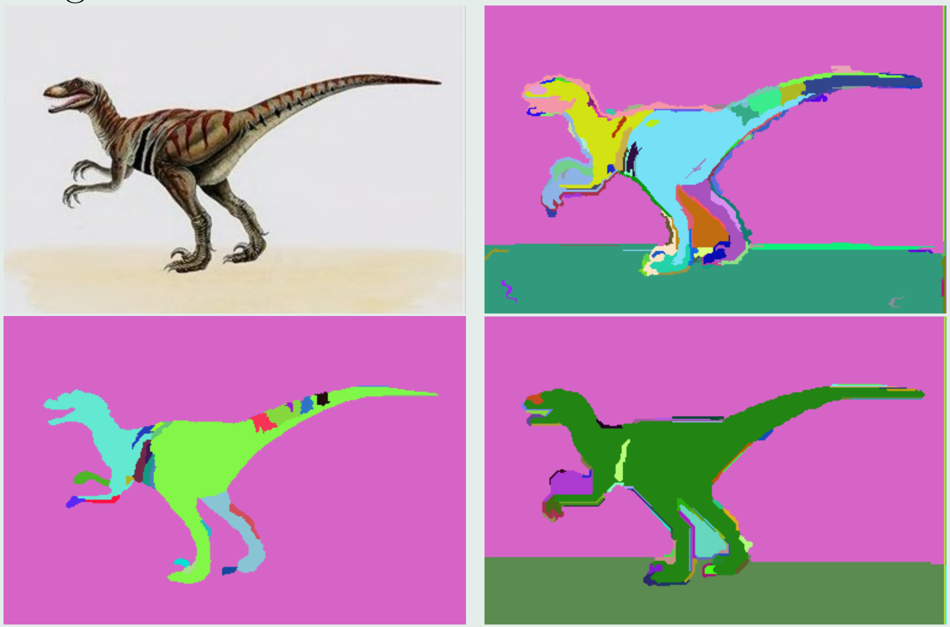
- Synthetic gray image (448x438) with 3 perceptually different regions
- Algorithm K known to work well (4.98s)
- Algorithm P2 (7.86s)
- Algorithm P1 (4.95s)



Qualitative Results: Dinasours

- Want only one background component
- Algorithm K (34.98s)
- Algorithm P2 (3.6s)
- Algorithm P1 (2.4s)

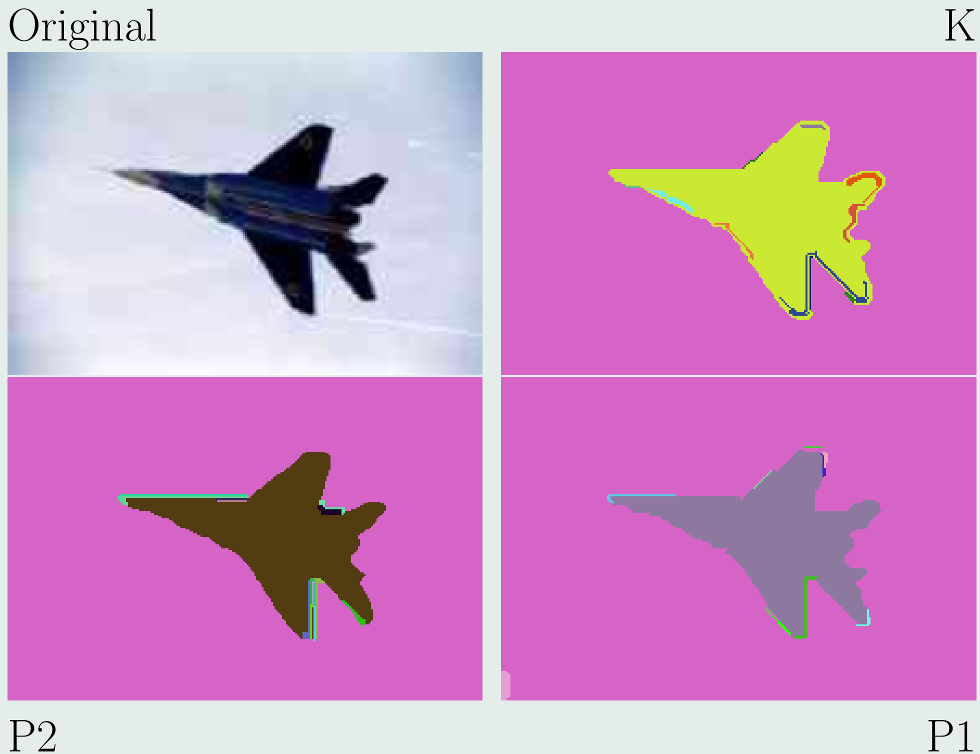
Original K



P2 P1

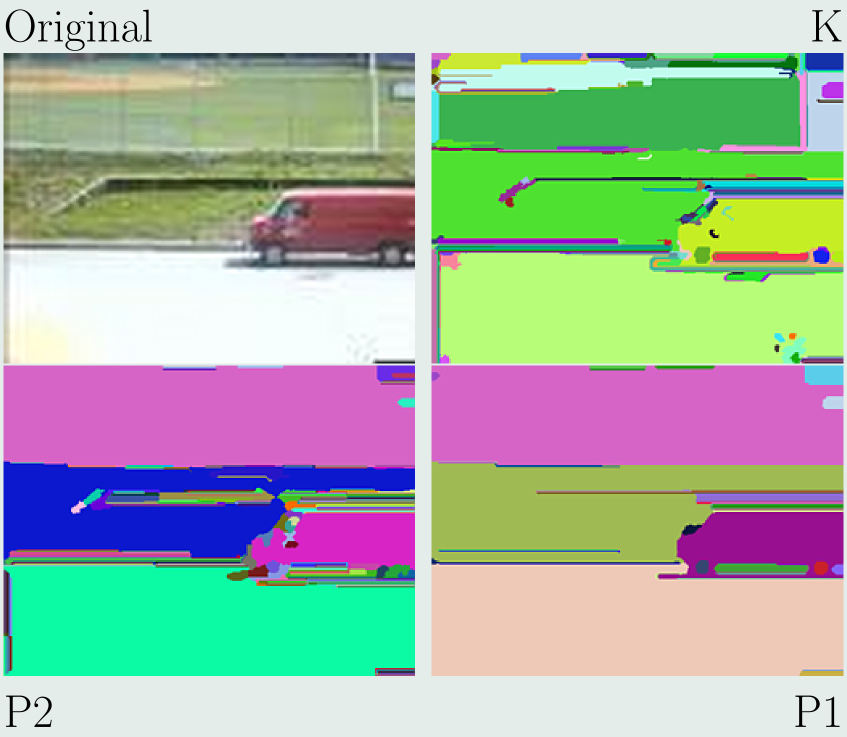
Qualitative Results: Plane

- Mainly two components
- Algorithm K (23.29s)
- Algorithm P2 (10.94s)
- Algorithm P1 (6.8s)



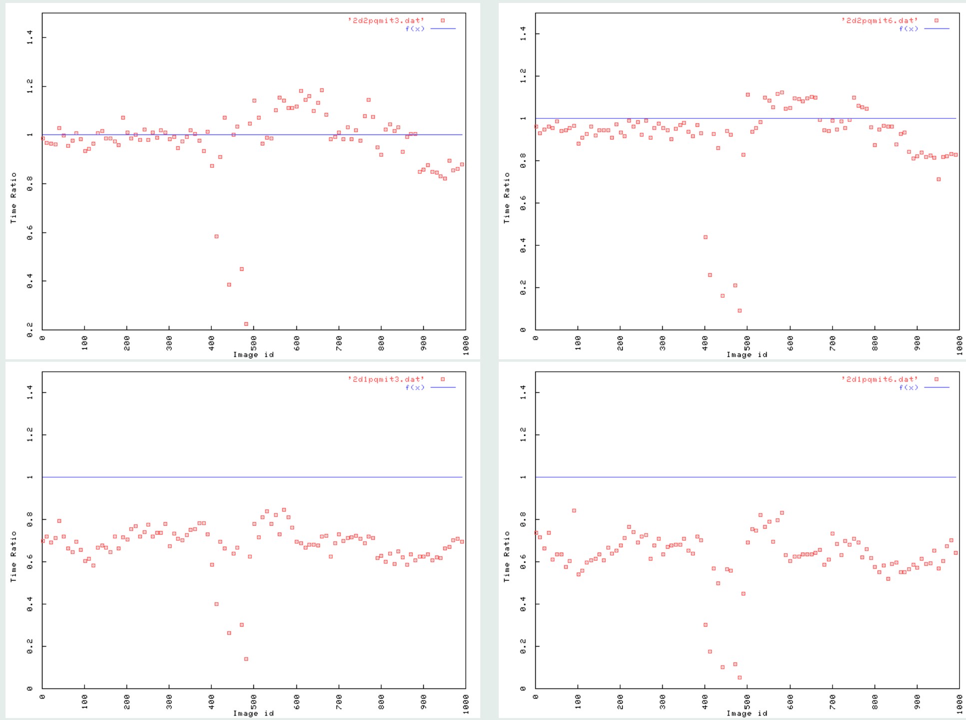
Qualitative Results: Street

- Many components, difficult to segment
- Algorithm K (5.14s)
- Algorithm P2 (5.12s)
- Algorithm P1 (3.6s)

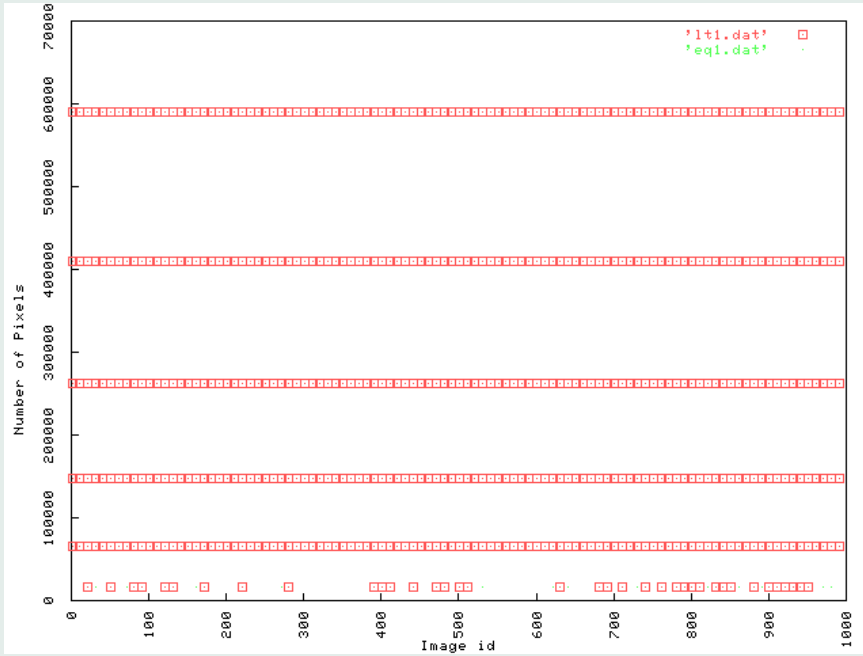


Quantitative Results: Ratio of Running Time

- Algorithm P2 (Image sizes 384x384 and 768x768)
- Algorithm P1 (Image sizes 384x384 and 768x768)



Quantitative Results: Varying Image Size



The ratio of time taken by Algorithm P1 to algorithm K for 100 images (with random images on the x-axis) of increasing sizes (y-axis). A box indicates that P1 is faster, a dot without a box indicates that P1 takes about the same time.

Segmentation S is not too fine

- In order for S to be too fine, there is some edge e between component C_i and $V - C_i$ for which D returns false.
- Some edge e such that $w(e) < Int(C_i) + \tau$.
- But in this case, `causesMerge()` would have succeeded
- This contradicts the non existence of edge e in C_i.

Segmentation S is not too coarse

- Suppose there is a proper refinement T that is not too fine.
- Thus some component $C \in S$ must be split into two or more distinct components A and B, both $\in S$.
- Of all the edges, consider the minimum weight edge e that is internal to C but connects A and B
- Since T is not too fine, let $w(e) > Int(A) + \tau(A)$
- By construction, any edge connecting A to another subcomponent of C must have weight as large as $w(e)$
- Weights of edges in A is smaller than that of e.
- Source must have been selected from A in the method `overall()`.
- Algorithm must have formed A before forming C.
- Existence of e would then have prevented the growth of A into C which happened

Conclusions

- Algorithm is faster
- Algorithm produces good quality
- Proves that the algorithm produces a good segmentation
- Did not change the nature of how components to be broken (tweaking this function results in a NP-hard problem)